Institute of Parallel and Distributed Systems
Department of Distributed Systems

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Master Thesis

In-Memory Check Pointing in Speculative Parallel Complex Event Processing

Muhammad Usman Sardar

Course of Study:	Master of Science Information Technology
Examiner:	Prof. Dr. Kurt Rothermal
Supervisor:	M. Sc. Ahmad Slo
Commenced:	01-08-2017
Completed:	15-02-2018

Table of Contents

Table of Figures

Table of Algorithms

Abstract

Parallel Complex Event Processing can be used to process high rate streams to get better latency. In order to handle out of order events, buffering and speculative processing techniques are discussed and analyzed. Another technique that merges both of them and adapts to the situation at runtime sounds promising. However, there are some inherent limitations of the technique due to which it doesn’t perform very well for parallel CEP systems. Our implementation is mainly based on this technique but we have done some optimizations to enhance the performance for parallel CEP systems.

There are two major optimizations done in this technique. First of all, the internal recovery process is introduced which allows us to reduce communication overhead because we don’t forward the events to the operator instances again after recovery. Instead, we just send some messages containing commands to amend the windows, as required. This can help a lot, especially in the cases when the events contain some heavy data like high quality pictures or video clip etc.

Secondly, we have divided the tasks to all the operator components, so that they can take checkpoints as well as recover back when required, irrespective of the other operator components. This allows them to run at their own pace instead of the consistency manager asking all of them to take checkpoint and send their respective states to the consistency manager. In this way, the waiting time of the consistency manager is avoided. Moreover, we have also introduced some minor optimizations like we can access the queues directly and add the out of order events to them, which avoids the recovery of the system, if possible. We have also introduced a high priority queue in the operator instance to allow the out of order events to bypass the normal queue and avoid recovery of the operator instance, if possible. Evaluations are done using synthetic data and the results show that our optimizations have increased the throughput as well as improved the latency in most of the cases.

This page is intentionally left blank.

1 Introduction

The number of sensors and smart devices are increasing day by day realizing the concepts of smart cities, smart homes and environment monitoring etc. For real time applications, the rate of events generated by these sensors is very high in most of the cases [BBD+02]. The event handling frameworks need to be very performant to process the events with minimum latency and maximum throughput, so that the actions are taken near to real time. To react timely to the specific situation, prompt decisions are required. In order to achieve this goal, Complex Event Processing paradigm is used to divide the whole detection process in different levels, so that performance can be enhanced. In this technique, low level sensor data is transformed to the complex events step by step in multiple levels.

In real life cases, it is not possible to know the exact delay in receiving events from all the sources in advance. There can be different kinds of delays involved and most importantly, these delays can vary at runtime. When the rate of events is very high, these random delays can cause the events from different sources to be received by the processing unit in wrong order. So, in order to ensure consistency of the complex events generated by the system, it is necessary to handle these out of order events correctly.

Buffering of events is one of the techniques used to tackle delayed events. In this technique, the events are buffered for some time, so that any delayed event is also received before passing the events for processing. Determination of the best size of buffer is the biggest problem in this technique because if the buffer is too small, it will not be able to tackle the delayed events correctly. On the other hand, if the size of buffer is too big, it will affect the overall latency of the system.

Another technique to handle the delayed events is processing of events speculatively and in case of the receipt of a delayed event, recovery of the whole system to a previous stable state and starting the processing again. If there are no delayed events, this technique can optimize the latency but if delayed events are received frequently, this technique can be worst because taking checkpoints to save the state of the system and to recover back to the state is very costly in many cases [KMR+13]. Moreover, the memory used to save all the states of the system can also be a problem. If the states are not saved frequently, system will have to recover back to very old state and then re-process a lot of events.

An optimized technique proposed by Mutschler et al. [MP13b] has merged both of the techniques introduced above. It uses K-slack algorithm (explained in section 2.4.2) to define the buffer time before every event is forwarded to the processing unit. In addition to the K-slack, another factor is used to define the speculation level such that if the speculation factor is “0”, the speculation becomes 100% and the system follows pure speculative technique and if the speculation factor is “1”, the speculation becomes 0% and the system follows pure buffering technique. The speculation factor is proposed to be based on the current CPU load to exploit the system resources for the best results at runtime. [MP13b] has proposed the generic technique that can be applied to any event processing system, so when applying this technique to Parallel Complex Event Processing, it does not give the best possible efficiency because of its generic nature. Further optimizations are done by us in this technique for complex event processing to improve the throughput and latency of the system in presence of out of order events.

1.1 Problem Statement

[MP13b] has proposed a novel approach to tackle with delayed events optimizing the buffering technique with the use of speculation factor in it. This technique is generic for all event processing systems. It handles the system as a black box with an interface to communicate with the system. This interface includes the following commands:

- Send Event
- Ask for Checkpoint
- Ask to Recover to the State
- Get CPU Load (for speculation)

A typical parallel complex event processing system has different components inside an operator. As the system was supposed to be a black box, so there are a number of optimizations that were not possible. So, for parallel complex event processing systems, this technique can be optimized further. In this thesis work, such optimizations are proposed and implemented in the complex event processing system generated by University of Stuttgart. Evaluations are done using the server cluster of Institute of Parallel and Distributed Systems (IPVS) in University of Stuttgart.

As an optimization, separate consistency controllers are introduced in all the components of the operator. This allows independent checkpoint of all the components, so that the operator components don't have to wait for each other and they can work asynchronously. Moreover, internal replay technique is also introduced which allows us to avoid sending all the events again from the splitting unit to the instances after recovery. In addition to this, a checkpoint model is introduced to optimize the latency based on the number of checkpoints. Last but not the least, our implementation has a little bit more room to tackle delayed events as compared to simple K-slack algorithm because we keep all the data saved at least since last checkpoint, so some deterministic data is also saved which allows to safely change the K-value. This optimization will be explained in detail later.

1.2 Outline

The remaining master thesis document is structured in the following way:

- **Section 2 – Background:**
  First of all complex event processing (CEP) is defined and explained in section 2.1. Then parallel CEP is defined in section 2.2. Moreover, K-slack algorithm for buffering and the speculation techniques are explained with example. Out of Order events and in-memory checkpoints are also explained along with all the terminology required to understand this document.
- **Section 3 – Out of Order Aware Event Processing Framework:**
  In this section, the working of our proposed framework is explained. All the noticeable modifications are described in this section which were required to enable the existing framework to handle out of order events without effecting the consistency of the results generated by the system.
- **Section 4 – Checkpoint of the System:**
  The process to take checkpoint of the system is explained in detail in this section. Moreover, there is a deep comparison of our implementation as compared to the implementation if [MP13b] is applied to the framework as a black box. Moreover, a checkpoint model is also

introduced which can further optimize the latency of the system adapting the frequency of checkpoints at real time based on a number of factors.

- **Section 5 – Recovery of the System**
  The complete process when an out of order event is received and the stable state is found and the system is recovered to the state is explained in this section. The optimizations done in our implementation are explained in detail along with explanation of all the data recovered back from the state.
- **Section 6 – Internal Replay of the Events**
  This section includes the explanation of the proposed internal replay method to avoid the un-necessary resending of the events from splitting unit to the instances when splitting unit recovers back. The algorithm of sending asynchronous commands to the instances to adapt to the changes in windows caused because of the out of order event is also described in this section.
- **Section 7 – Evaluations**
  Intensive evaluations are done on the server cluster of IPVS in University of Stuttgart. The results are shown graphically along with the descriptive explanation. In addition to this the technical specifications of the system used to do the evaluations are also included in this section.
- **Section 8 – Literature Review**
  References from the existing literature are included in this section to mention relevant work done by other researchers in this field. Some already existing techniques to handle the problem are discussed and analyzed for the pros and cons of every technique.
- **Section 9 – Conclusion**
  In the last section, the summary of the proposed technique and the results of the evaluation is given. Moreover, the possible future directions are proposed for further research in the topic.

2 Background

In this section, we will discuss the background of the topic and all the related terms in detail. This section will introduce different techniques available in the literature and the critical analysis of those techniques to show the importance of our research work.

2.1 Complex Event Processing

Complex Event Processing (CEP) is formally defined by David Luckham as:

"Complex Event Processing (CEP) is a defined set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems." [Luc08]

A sequence of ordered events is known as an event stream. However, if there are many different streams which are not necessarily generated by the same source, then it is known as the event cloud. CEP can be used in many different applications because it can process event cloud as well as event stream [Luc06]. It is one of the main benefits of CEP that it can handle out of order events received from different sources.

Information is extracted from a cloud of events via event data analysis in multiple layers of abstraction. This whole task of extracting the information which can be used to take decisions at business level, is divided over multiple lower level event detectors. Thus, a complex event is detected by combined effort of all the event detectors. Sensor events are analyzed by lower level event detectors and resulting intermediate level events are generated. These intermediate events are passed on to the next level event detectors, which will analyze these events and produce complex events that can be used to take an action or notify something. There can be multiple intermediate levels before the top level complex events are generated. Figure 2-1 shows the generic hierarchy of a complex event processing system.

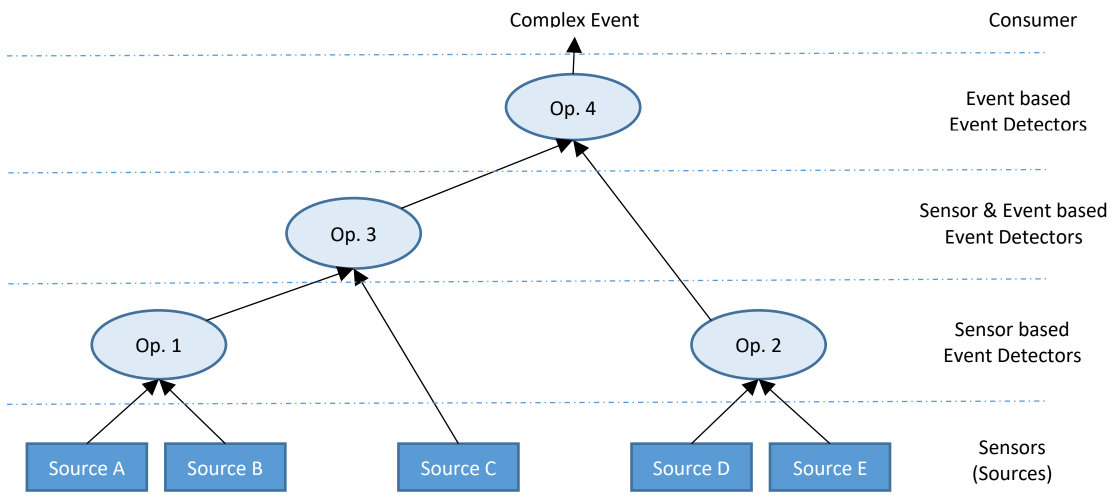

Figure 2-1 CEP Event Detection Hierarchy

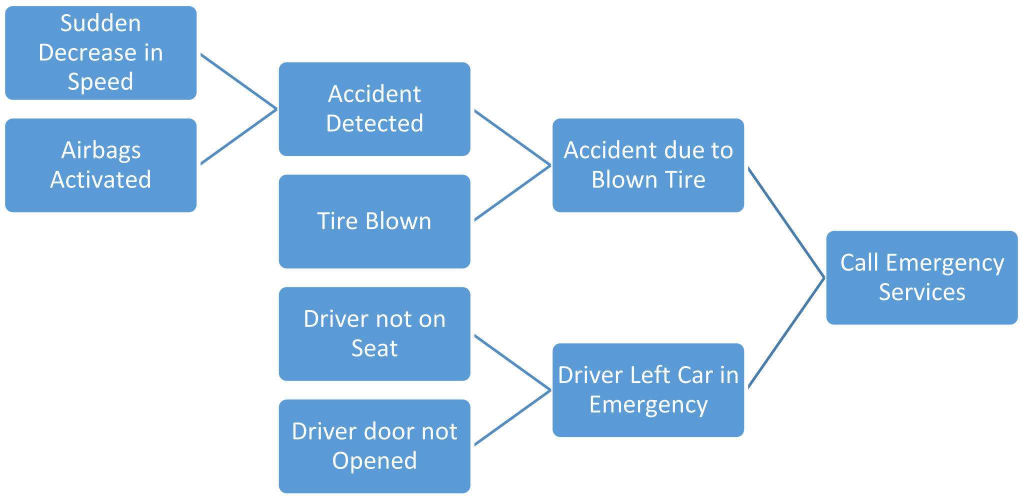

Figure 2-2 Example of Complex Event Processing

Let's take an example to show the detection of a complex event based on multiple lower level event detectors. In this example, a car will have a built-in system to detect that a severe accident has been occurred in which driver is thrown out of the car, so emergency services are automatically informed. Based on multiple sensors, lower level event detectors will detect that if there is sudden decrease in the speed of the car and if the airbags are also activated at the same time. In this situation, an intermediate event is detected that the car has faced an accident. If another event detector has detected that the tire of the car was blown before accident, based on these both events, it can be detected that the accident happened due to blown tire.

On the other hand, if a detector has detected that the driver has just left the seat but the door of the car on driver's side was not opened. It can be assumed on this information that the driver has somehow left the car in an emergency without opening the door. Moreover, if this situation has occurred at that time of accident, it can be detected that the accident has occurred and due to the accident, driver has been thrown out of the car. If this complex event is detected, the car can take an action to inform the emergency services about the situation, so that they can reach the location as soon as possible to provide rescue services. Figure 2-2 shows such a complex event processing unit.

2.2 Parallel Complex Event Processing

The performance of stream processing can be further enhanced using parallel processing paradigm [Bri$^+$11]. In order to take the advantage of parallel processing in complex event processing explained above, the data streams can be divided to different identical operators at the same level. As shown in **Figure 2-3**, the incoming data from multiple streams is received by the splitting unit. Splitting unit splits the received data to all available instances according to the splitting predicate and scheduling technique. All instances process the portion of data assigned to them and send the generated events (complex / intermediate) to the merger. Merger combines the results from all the instances and forwards them to the next level. Figure 2-3 depicts the basic architecture of an operator of parallel CEP system.

Different splitting predicates can be used by the splitting unit to divide the input stream into multiple partitions. Predicates can be based on the number of events, time, or any other basic logic. Scheduler is responsible to allocate these partitions to available instances based on some defined rule. This rule can be as simple as round robin scheduling or it can also be a complex one like predictive scheduling.

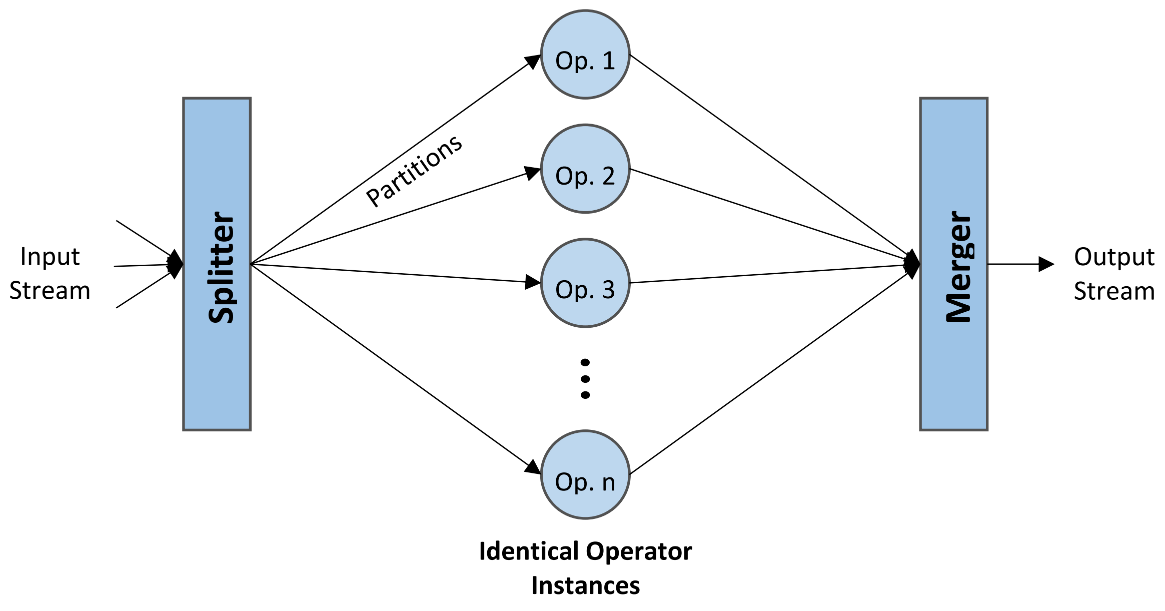

Figure 2-3 Architecture of Parallelization Framework

2.3 Out of Order Events

As discussed in the sub-section 2.1, complex event processing can handle input streams of events from multiple sources. These multiple sources can be of different types and they can be located at different locations. Moreover, there is a possibility that the mode of communication between these sources and the event detectors is also different e.g. a source having wired connected with the event detector and other sources communicating to it via wireless connection.

All these factors account for the difference in time taken by the generated event to arrive at the CEP system. The events that are received late than the normal stream are called out-of-order events in this document. In order to detect that the event is out of order or not, the system stores the maximum timestamp of the received events. If the timestamp of the newly received event is less than this maximum timestamp, then the event is an out of order event. Equation 2.1 shows the inequality to detect if the event is out of order or not. ‘e.ts’ is the timestamp of all events received by the CEP system till now. If the inequality is not true, then the received event is called as out of order event.

$$e_{received}.ts > \max[e.ts] \qquad \text{Equation 2.1}$$

In some applications, the order of events while processing is very important to detect complex events out of the stream. So, the operator should process the events in the correct order, otherwise it can give false negatives or false positives. A false negative means that there should be a complex event detected by the system but by processing the events in wrong order, a complex event is missed. In contrast to it, a false positive means that there shouldn’t be a complex event detected by the system but still it has generated a complex event due to wrong processing order.

A simple example is explained below to show the consequences of processing the events in wrong order. Let’s take a sequence detector that will generate a complex event if it receives event ‘a’, ‘b’ and ‘c’ in a row. State diagram of such an event detector is shown in Figure 2-4.

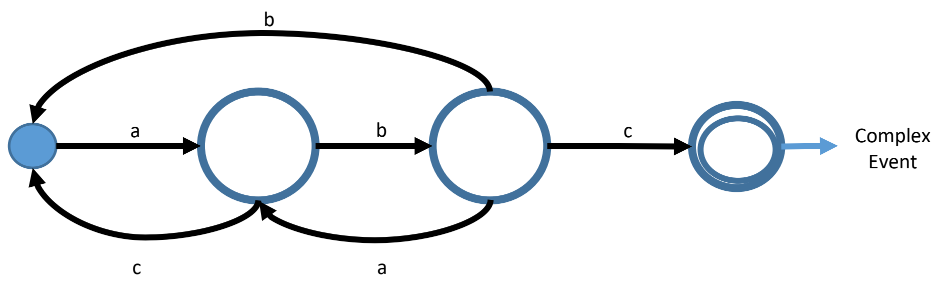

Figure 2-4 State Diagram of Sequence Detector

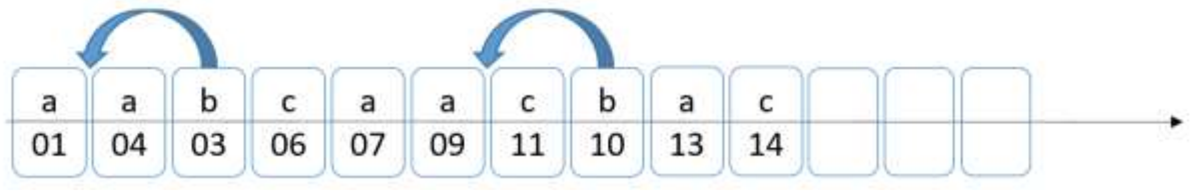

Figure 2-5 An Example of Out-of-Order Events Stream

The synthetic stream of events shown in Figure 2-5 is merged from three different sources A, B & C. It shows that the events coming from source B are delayed for 2 seconds, so if the sequence detector processes the stream as it is, it will produce wrong complex events and miss the actual complex event that should have been generated if the stream was being processed in correct order.

If the stream is processed as it is, a complex event (false positive) will be generated from “a-4, b-3 and c-6”. However, correct order of these events will be “b-3, a-4 and c-6”, so there should not be any complex event generated from these 3 events. Moreover, event detector will not be able to detect complex event (false negative) from “a-9, b-10 and c-11” because in the received stream, they are present in the wrong order. So, in this way, processing the events as they are received from the sources, without re-ordering them can have disastrous effects on the working of the system.

2.4 Buffering Technique

Buffering is one of the techniques to handle the out of order events. In this technique, all the events are buffered for some time before processing them. In this way, if an event faces a communication delay for some time, it can also be added to the buffer at the correct position and when the events are forwarded from the buffer to the processing system, they will be in the correct order. In this technique, any expensive processing like taking checkpoint of the system or recovery from the state is not required. However, latency of the system is effected by using this technique because of the buffer time. [MP13b]

2.4.1 Importance of Buffer Time

The buffer time has prime importance in this technique. It should be optimum for good results as well as good performance. If the buffer time is not long enough, the necessary purpose of buffering will not be fulfilled. Events will be forwarded to the processing system before the delayed event is received. So, the order of the events forwarded will still be wrong. However, if the buffer time is too long it will affect the latency of the system. Moreover, it can also take a lot of memory if the incoming events have large data and/or frequency is very high. [MP13b]

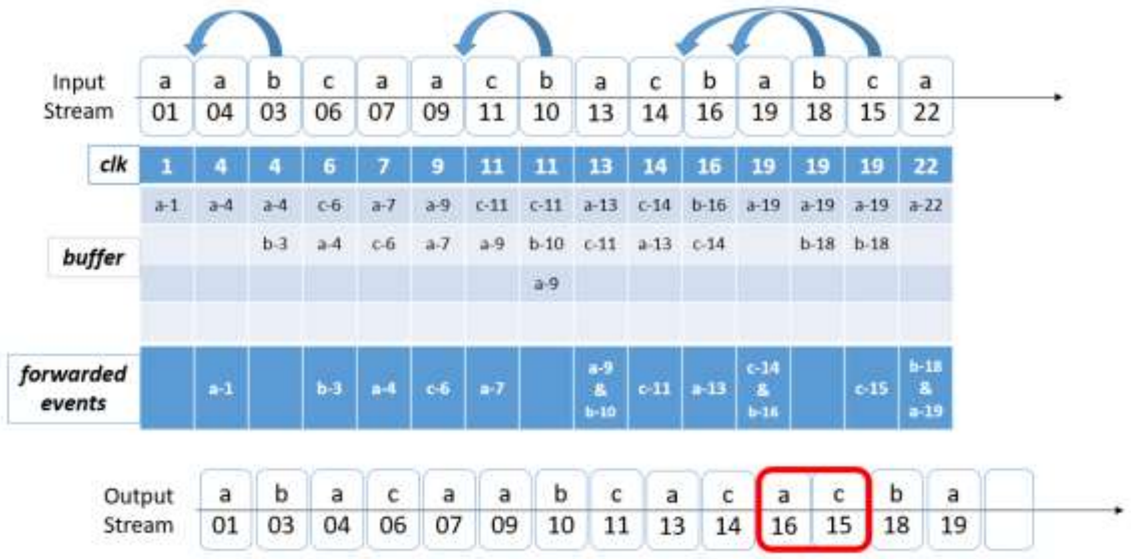

Figure 2-6 Example of K-Slack Algorithm

2.4.2 K-Slack Algorithm

An algorithm based on buffering technique for out-of-order events is K-slack Algorithm [BSW04]. K-value defines the buffer time which needs to be fixed offline. All the events will wait in the buffer for at most K time units. To use this algorithm in distributed environment, Li et al. [LLD+07] proposed to use the local clock of every system and whenever Equation 2.2 is satisfied for an event, it is forwarded for processing.

$$\boldsymbol{e_i.ts + K \leq clk} \qquad \text{Equation 2.2}$$

In Equation 2.2, '$e_i.ts$' is the timestamp of the event, '*K*' is the delay and '*clk*' is the local clock defined by the largest timestamp seen so far. Figure 2-6 shows the example of this algorithm to explain the working in detail.

An example was shown in the previous sub-section to highlight the problem of out of order events, we will use the same example to show that K-slack algorithm has solved the problem. In this example, we take fixed value of K=3. All the events are kept in the buffer for at most 3 time units. Whenever, they fulfill Equation 2.2, they will be forwarded to the system, shown as output stream in the **Figure 2-6**. This technique needs a-priori knowledge of the system, so the problem of events coming from source B (which are always 2 time units delayed) is solved in this example because we are buffering all the events for K = 3.

However, in this example, we have also extended the previous stream with the case if the events from one of the sources get delayed for a short time interval that cannot be predicted before starting the system. So in the example, we have an event from source C that is delayed for 6 time units. So, the K-slack algorithm with fixed value of K=3 is failed here and 'b-16' was forwarded before 'c-15' as shown in Figure 2-6. So, the events forwarded to the system for processing are still out of order.

Thus the example leads us to the conclusion also mentioned in [MP13a] that it's not possible to decide the K-value beforehand for distributed hierarchical event detector. So, [MP13a] has proposed a technique to start the system with K= 0 and change the value of K at runtime. In this technique, one of the stream sources is responsible for maintaining the local clock. This avoids any abrupt changes in the clock value. In the beginning, some events will be forwarded in the wrong pattern, but once the K-value is stable, the output stream will be well-ordered. Further detail of the working of this technique will be discussed in section 3 along with its implementation in our framework.

2.5 Speculation Technique

The basic idea behind speculative processing is that the events are directly forwarded to the system for processing. Meanwhile, system takes checkpoints periodically to save its states. Later on, if any delayed event is received, the system will be recovered back to a stable state and the events will be processed again from that point. So, in this technique, the events are not buffered before sending them to the processing unit which decreases the latency of the system. However, if out of order events are very frequent, then this technique can be worse because it can overload the CPU and system failure in the worst case scenario. [MP13b]

As discussed above, in this technique, the system might have to recover back to some previous state if an out of order event is received. So, the system will have to take checkpoints beforehand so that in case of recovery, it will find the nearest stable state, recover back to this state and start processing events again. While taking a checkpoint, all the necessary data needs to be deep cloned so that the system can come back to exactly the same state as it was at the time of taking checkpoint. It is necessary to mention here that the process of checkpoint and recovery are very expensive processes because it includes deep cloning of data structures. Moreover, the frequency of checkpoints is also of prime importance because if we take checkpoints very frequently, it will waste a lot of time in doing so [Qua01]. On the other hand, if the frequency is very low, it will have to recover back to very old state and then process all the events again in correct order.

2.6 Merging Speculation with Buffering Technique

Both techniques: speculation and merging have their pros and cons, but Mutschler et al. [MP13b] has proposed a way to merge both of these techniques to get benefit of the pros of both techniques and minimizing the effect of the cons on the performance of the system. They modified the equation of K-slack algorithm with another factor i.e. speculation factor 'α'. This factor defines the level of speculation and it proposed to be dependent on the CPU load to exploit the resources in a best possible manner. The modified equation is as follows:

$$\boldsymbol{e_i.ts + \alpha \cdot K \leq clk} \qquad \textit{Equation 2.3}$$

As shown in the Equation 2.3, speculation factor is multiplied with the K-value of the K-slack algorithm. So, in this way the actual buffering delay is also dependent on the speculation factor. The speculation factor can has its value ranging between 0 and 1. If the speculation percentage is 100% (speculation factor = 0), the system follows purely speculative technique and if the speculation factor is 1, the system follows purely buffering technique. And if the speculation factor is in between, the system follows both techniques partly.

[MP13b] says that whenever the modified equation of K- Slack is satisfied but the original equation of K-slack is not satisfied, the event is forwarded to the system speculatively but before forwarding this event, system is asked to send back its current state. This state of the system is saved in the buffer before sending the event for processing. If an out order event is received, a stable state is sent is sent to the system first, so that it can recover back to that state and then events are replayed in the correct order.

In [MP13b], this technique is explained for generic event detectors but if we apply this technique to parallel complex event processing units, its performance is not optimum. First of all, the system is black box for the technique proposed by [MP13b], so if an event is forwarded to the system and an out of order event is received, it will have to do recovery. It cannot avoid recovery even if the event is still waiting in the buffer of the operator or splitting unit to be processed. Secondly, after recovery, it will have to send all the events again to the splitting unit and then splitting unit will send all the events to the instances again, however, we have proposed an alternative approach to avoid sending events again, instead send the commands to the instances to adapt to the changes in the windows caused by this out of order event. Last but not least, we have proposed to make all the operator components independent to checkpoint and recover, so the splitting unit doesn't have to ask all the instances to checkpoint and wait for their response. This waiting time can highly affect the throughput of the system. Our implementation will be explained in detail in the section 3.

2.7 Introduction of Terms

From now onwards, we will be using these terms as defined below:

- State
  A state of the system or a component of the system is a container class which consists of all the data required for the system or a component of the system to recover back.
- Checkpoint
  A process in which system or a component of the system saves all of its current data as a state so that when it recovers back to this state, the system or a component of the system is exactly in the same condition as it was at the time of checkpoint.
- Recovery
  A process in which in which a stable state of the system or a component of the system is found and then all the data structures are cloned to that state, so bring system back to the state.
- Stable State
  A stable state is nearest state older than the out of order event received. This is the state to which the system can recover back, so that it can start processing events again.
- Internal Replay
  The process in which splitting unit checks the predicate for the events after recovery. However, it doesn't forward the events again to the instances, unless the event was not sent to an instance previously and is required by it now. Splitting unit sends the message to tell instances, how to adapt to the changes in the windows caused by the out of order event.
- Message
  A message contains a command along with some information. Splitting unit sends different messages to the instances with some commands to adapt to the changes in the windows.

- Command
  Command is a part of the message. It is actually a keyword, which tells the instance how to react to the specific message received from the splitting unit.

- Intermediate Events
  The events generated by the lower level event detectors that are supposed to be used by higher level event detectors to detect complex events.

- Pseudo-Events
  The events manually generated by the framework to inform the operator instances about the start point and the end point.

3 Out of Order Aware Event Processing Framework

A framework is developed in this Master's thesis work which is capable of handling the out of order events using a merge of buffering technique and speculation technique, as discussed briefly in the previous section. This framework is implemented as an extension to a framework developed by University of Stuttgart [MKR15]. The already existing framework was not having capability to handle out of order events. So, we have extended it with necessary components to add this capability. The complete framework will be explained in this section.

As, the framework [MKR15] was designed for a special situation previously where there cannot be any out of order events from any of the sources, so it was relying on the serial number assigned to the events as their identity. Similarly the serial number was also assigned to the windows to identify them. But in our case, when there is possibility of out of order events, it is not possible to rely on the serial numbers. We considered different options to use serial numbers so that we don't have to modify the complete framework but there were many problems. So it is not possible to use serial numbers to identify events in presence of out of order events with some basic modifications [BBD+02]. So, we had to make the timestamps as the identity of events and windows

A CEP system consists of multiple level event detectors, all of them have the duty to detect some specific events and send it to the higher level detectors. The higher level detectors receive intermediate level events generated by the lower level detectors and generate complex events or intermediate level events for the next higher level detectors. In this way, a complex event is generated step by step. However, every single event detector (operator) can contain multiple components in the case of parallel CEP. But all the operators have the same basic architecture as explained in sub-section 2.2. In this section, we will explain our approach to handle out of order events within a single operator. This framework can be easily replicated for all the operators to have it completely embedded in a parallel CEP system.

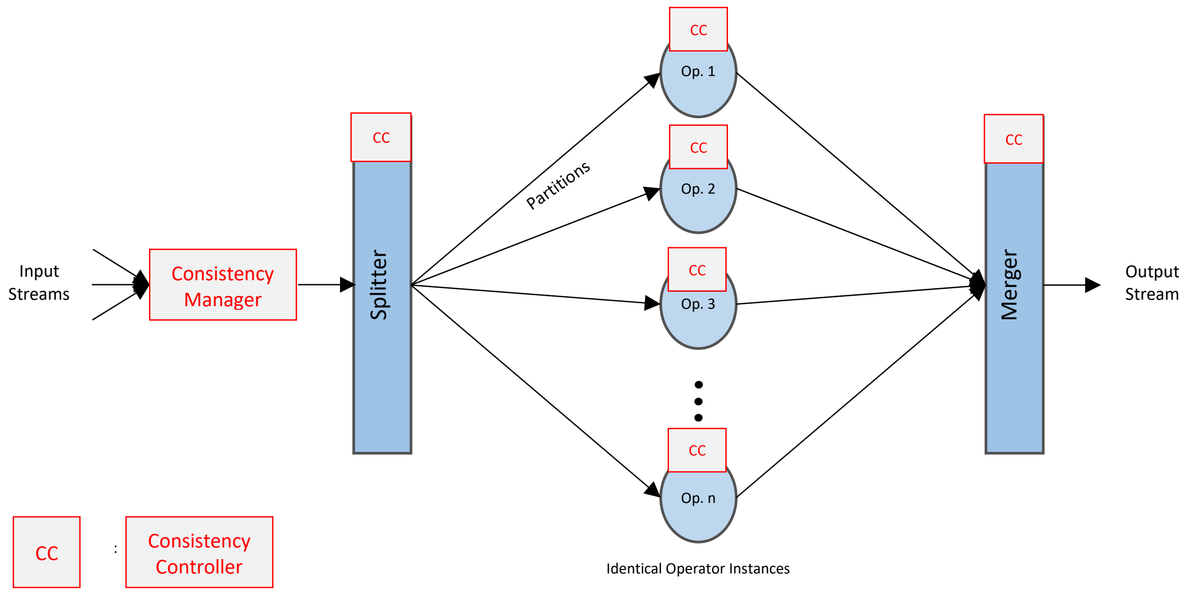

Figure 3-1 Architecture of an Operator in our Framework

The architecture of our framework consists of operator components along with a consistency component used for buffering purposes. The components are as follows:

- Consistency Manager
- Splitting Unit
- Operator Instances
- Merging Unit

Figure 3-1 shows the architecture of extended parallel CEP in our implementation which can handle out of order events also. The input stream shown in Figure 3-1 is received from a source and the output stream shown is sent to the consumer. All the components are explained below:

3.1 Source

In the context of a specific operator, a source can be anything which is sending a stream to the operator. It can either be a sensor which is generating events by observing its surroundings according to its sensing capabilities. Or a source can also be a lower level event detector which is detecting some specific events and sending those events to this operator, so that this operator can generate complex events based on those events along with the streams coming from other sources.

3.2 Consistency Manager

Consistency Manager is a newly introduced component of the operator which is responsible for buffering and speculation at run time. All the events received from sources are first directed towards consistency manager, then consistency manager forwards the events further to the splitting unit, according to the current K-value for buffering and current speculation factor, so that they can be processed. In our current implementation, consistency manager is a part of splitting unit running on a separate thread. However, it doesn’t have any dependency on the splitting unit, so it can be easily modified to be a completely separate process. If we make the consistency manager a separate process, then it can either run on the same machine as splitting unit or on a different machine. It will be connected to the splitting unit via TCP connection.

Consistency Manager is responsible for 2 main tasks. Firstly, it is responsible to buffer the events according to modified K-slack Algorithm as in [MP13a]. For this it also has to keep modifying K-value and the local clock value. It will be explained in detail below. Second task of the consistency manager is to get CPU load from all connected instances and according to the aggregate CPU load of the system, speculation factor is calculated as in [MP13b].

However, it should be noted here that our consistency manager is not responsible to ask the system to take checkpoint and get the state back from the system, nor it is responsible to ask the system to recover back to a previous state. In contrast to [MP13b], our framework has consistency controllers in all the operator components which are responsible to take care of checkpoints and recovery according to the needs of an individual component.

Here, we discuss the buffering algorithm used in our implementation. Then, we will introduce the speculation model that we used in it and then we will give the pseudo-code of overall working of the consistency manager in our implementation.

3.2.1 Buffering Algorithm

The buffering algorithm used in our framework is based on K-slack algorithm [BSW04]. However, some modifications are adapted from [MP13a]. As discussed in sub-section 2.4.2, constant value of K is not enough to handle the possible inconsistencies in case of distributed CEP systems. Thus, we need to change the value of K at run-time.

The main idea is to have one of the sources to be the one which defines the clock value. It makes the clock more stable and avoids the clock to go back in time. All the events received since last event from clock defining source are kept in a list. When the event from clock defining source is received, the timestamp of this event is compared with all the events in the list. In the start, the K-value is initialized to be 0. After that, Equation 3.1 and Equation 3.2 are used to calculate the current K-value.

$$K = max_j\ [\delta(e_j)] \qquad \text{Equation 3.1}$$

where,

$$\delta(e_j) = e_k.ts - e_j.ts \qquad \text{Equation 3.2}$$

In Equation 3.2, e_k.ts is the timestamp of the current event received from the source which defines the clock and e_j.ts is the timestamp of the events in the list of event since last event from clock defining source. It will loop over all the events in the list and find the maxima of difference between timestamps. This value will be updated as new K-value. However, if the maxima is less than the previous K-value, then the K-value remains unchanged. This means that the K-value can only increase, it cannot decrease.

We will use the same example and apply this algorithm to explain the working of this algorithm and to show how this algorithm is able to handle the problem, which was not solved by the simple K-slack algorithm.

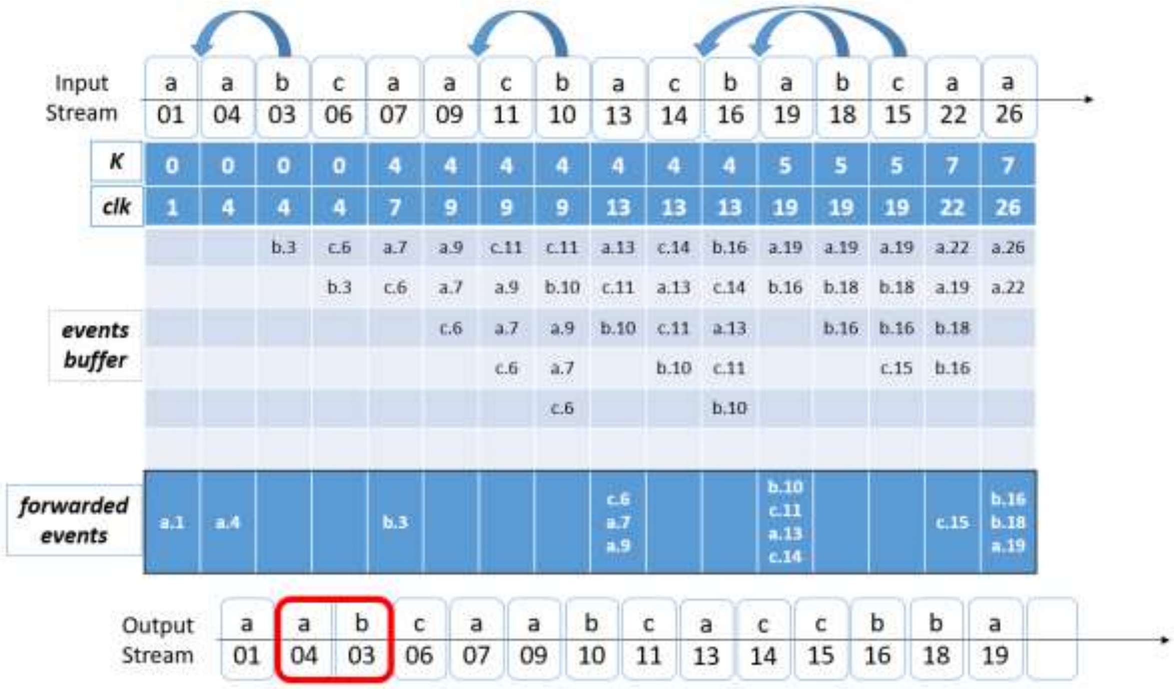
Input Stream
a 01, a 04, b 03, c 06, a 07, a 09, c 11, b 10, a 13, c 14, b 16, a 19, b 18, c 15, a 22, a 26
K
0 0 0 0 4 4 4 4 4 4 4 5 5 5 7 7
clk
1 4 4 4 7 9 9 9 13 13 13 19 19 19 22 26
events buffer
b.3 c.6 a.7 a.9 c.11 c.11 a.13 c.14 b.16 a.19 a.19 a.19 a.22 a.26
b.3 c.6 a.7 a.9 b.10 c.11 a.13 c.14 b.16 b.18 b.18 a.19 a.22
c.6 a.7 a.9 b.10 c.11 a.13 b.16 b.16 b.18
c.6 a.7 b.10 c.11 c.15 b.16
c.6 b.10
forwarded events
a.1 a.4 b.3 c.6 a.7 a.9 b.10 c.11 a.13 c.14 c.15 b.16 b.18 a.19
Output Stream
a 01, a 04, b 03, c 06, a 07, a 09, b 10, c 11, a 13, c 14, c 15, b 16, b 18, a 19

Figure 3-2 Example of Modified K-Slack Algorithm

Input stream is the same as in the previous example. Previously, we explained this stream with constant K=3. In that example, we were able to resolve the inconsistencies of source ‘B’ but when the event from source ‘C’ was delayed more than 3 time units, the algorithm failed. Thus, we concluded that we cannot use that algorithm as it is.

In this example, we don’t fix the value of K, rather we initialize with 0 and keep increasing it if required. Initially, it might forward events in the wrong order as in the case of ‘a-4’ forwarded before ‘b-3’ because when ‘a-4’ was forwarded, we were not having any calibrations. When ‘a-7’ is received, ‘b-3’ and ‘c-6’ will be in the list (not shown in the figure). Equation 3.2 gives 4 & 3 respectively, so If we apply Equation 3.1, we get K=4. So, only event ‘b-3’ satisfies Equation 2.2 and other events are still in the buffer.

When ‘a-9’ is received, there are no events in the list, so K-value remains unchanged. None of the events from buffer satisfies Equation 2.2, so we don’t forward any event. By applying Equation 3.1 & Equation 3.2 on the arrival of ‘a-13’, we get K=3, but it is still lower than current K, so actual K-value still remains unchanged. K-value is updated to K=5 on the arrival of ‘a-19’. And this update of K-value, clock and events forwarding continues like this.

As shown in the output stream of Figure 2-1, except for the initial problem, this algorithm has arranged all the events in perfectly correct order. The initial problem is not very crucial in our case because we are dealing with continuous streams, so our main focus is in the performance of the algorithm during run-time.

3.2.2 Speculation

Consistency Manager is also responsible to take aggregate of the CPU load of all the operator instances and according to the CPU load, speculation factor is defined as in [MP13b]. The runtime adaptation of speculation factor is inspired from congestion control mechanism in Transmission Control Protocol (TCP).

As adapted from [MP13b], a target zone of the CPU load is defined. If the CPU load is lower than the lower limit of target zone, speculation factor is halved to increase the level of speculation. When the CPU load is increased from the higher limit of target zone, the current speculation factor is saved as α_{best} before it is set back to 1 to avoid system failure. Then again, speculation factor is halved until its value is just below the bisection line $(1 – \alpha_{best})/2$. From that point, speculation factor is decreased in small steps to reach the target zone.

The value of speculation factor defines the level of speculation. It ranges between 0 and 1. If the value is equal to 0, system becomes purely speculatively processing system. However, if the value is 1, it works on purely buffering technique. If the value is between 0 and 1, system is buffering for some time but forwarding events pre-maturely.

If the events are forwarded prematurely because they satisfy Equation 2.3 but they don’t satisfy Equation 2.2, then the event is flagged as a speculative event, so that the system can handle it accordingly. This information might also be required by the checkpoint model to decide if it needs to take checkpoint or not.

To explain the concept of speculation with an example, let’s take the same input stream and apply constant speculation factor 0.5. So, the output stream will be as shown in Figure 3-3.

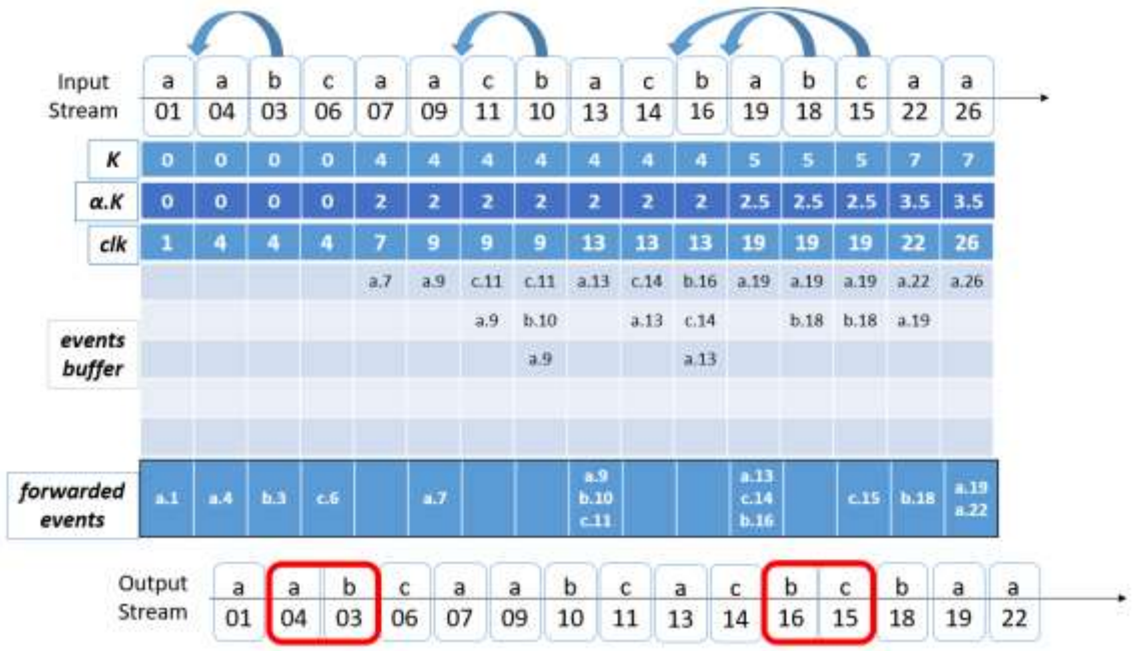

Figure 3-3 Speculation with α=0.5

As, we can clearly see that the buffer time is simply decreased by the factor of 2 which helps in decreasing the latency of the system. But, it can have some consequences if out of order events are received. Output stream in the Figure 3-3 shows that events 'b-16' and 'c-15' are again forwarded to the splitting unit in wrong order. But in our implementation, we have consistency controllers available in all components of the operator to handle such cases. All the components take checkpoints according to the checkpoint model and recover back if required to avoid any false positives or false negatives to ensure consistency in the results. Functionality of consistency controllers is explained in respective operator components.

3.2.3 Pseudo-Code

Algorithm.1 shows the pseudo-code to handle a received event in Consistency Manager. As shown in line.4, it saves all the events received since last clock event in the '*eventsList*'. This list is used to calculate the K-value according to the adaptive K-value algorithm by [MP13a]. Moreover, it saves all the events to '*sortedEventsBuffer'* so that they can be forwarded to the splitting unit, once they satisfy the equation. Whenever, the clock event is received, speculation factor, clock, k-value and delay is updated (line. 5-11).

On arrival of every event, we check if any of the event is ready to be passed to splitting unit speculatively. Then it checks if the event is already deterministic or not and saves this information in the meta-data of the event (line. 14-18). Current value of k and clock is also added to the meta-data of the event, so that consistency controllers can also update their local k-value and clock. The local values of clock and K are used by the consistency controllers to delete deterministic data.

The events are purged from the buffer of consistency manager right after they are forwarded to the splitting unit. It doesn't save any older data because it doesn't play any role in the recovery. Recovery of operator components is responsibility of the respective consistency controller.

Algorithm.1. Pseudo-Code of Event Handling Routine of Consistency Manager

```
1.  Data: InputEvent event, DelayCalculationList eventsList, sortedEventsBuffer, clkSources,
2.  begin
3.          add to sortedEventsBuffer                          // Events Buffer
4.          add to eventsList                                  // list to calculate K-Value
5.          IF e.sourceId ϵ {clkSources} THEN                  // if clock event
6.                  updateSpeculationFactor()
7.                  clock ← event.ts
8.                  updateKvalue()
9.                  delay ← α × K
10.                 eventsList.clear()
11.         ENDIF
12.         FOR Event e : sortedEventsBuffer DO                // loop for all events in
    buffer
13.                 IF e.ts + delay ≤ clock THEN
14.                         IF e.ts + K ≤ clock THEN
15.                                 e.consistent ← true        // deterministic event
16.                         ELSE
17.                                 e.consistent ← false       // speculative event
18.                         ENDIF
19.                         e.kValue ← K
20.                         e.clock ← clock
21.                         forward e to Splitting Unit
22.                         remove e from sortedEventsBuffer
23.                 ELSE
24.                         RETURN                             // break loop and return
25.                 ENDIF
26.         ENDFOR
27. end                                                                            .
```

3.3 Splitting Unit

Next in the hierarchy after consistency manager is the splitting unit. Events are forwarded to it after buffering in the consistency manager. As discussed previously, splitting unit doesn't have any dependencies on consistency manager, so it can be a completely independent process having consistency manager connected to it via TCP connection. However, in our implementation, consistency manager is also initiated by the splitting unit as a separate thread. The main task of the splitting unit is to split the continuous stream of events into smaller windows which are then allocated to different identical operator instances, so that they can be processed in parallel and then the results of the working of these operator instances is forwarded to the merger.

Figure 3-4 shows the basic architecture of the consistency manager and the splitting unit. The consistency manager is connected to a number of sources via source connectors. As source connector is a separate thread for each source which is responsible to receive the events from the source and pass it to the consistency manager. Consistency manager buffers the events for some time according to the K-slack algorithm and controls the speculation factor based on the CPU loads reported by the operator instances.

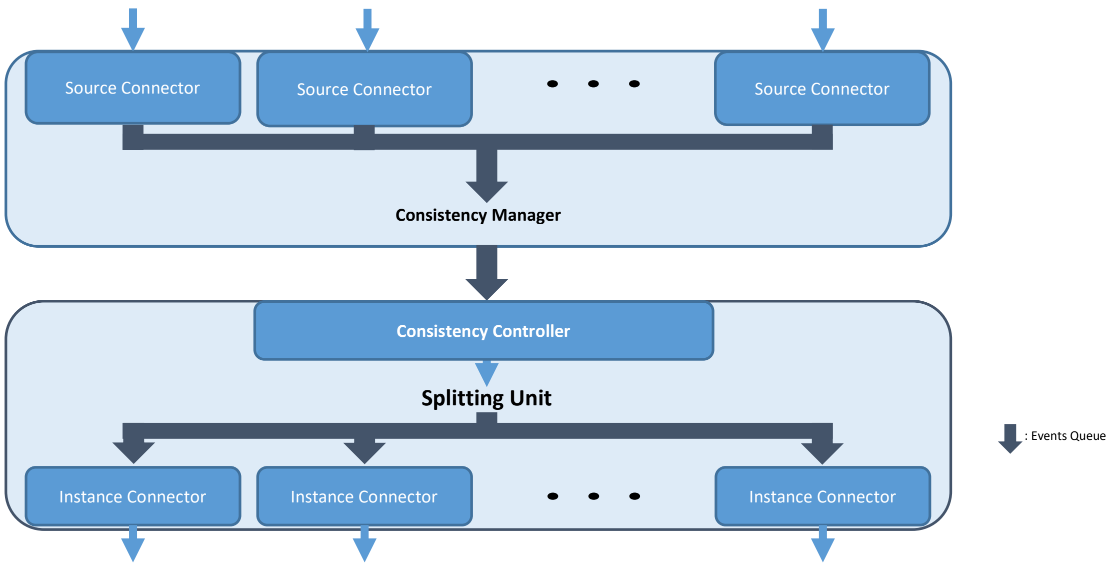

Figure 3-4 Architecture of Consistency Manager & Splitting Unit

The events from the consistency manager are passed to the splitting unit via a queue as shown in the Figure 3-4. The splitting unit is connected to the operator instances through separate threads instance connectors. The instance connectors are responsible for all the communication between splitting unit and the operator instance. All the events, messages and windows to be sent to the instance are passed to it and it forwards further to the operator instance. All the acknowledgements and the load information are also received by the instance connector and then passed to the splitting unit.

To ensure that the main tasks of the splitting unit are not affected by the arrival of an out of order event, a consistency controller thread is present as a part of splitting unit, which is responsible to maintain the consistency of the working of the splitting unit. Consistency Controller is responsible to take checkpoint of splitting unit and to recover it back to previous state and start internal recovery. Firstly, we will explain how the splitting unit performs its normal task. Then we will give the detailed in sight of the consistency controller's contribution to enable the splitting unit to work correctly in the presence of out of order events. Figure 3-4 shows the basic architecture of the splitting unit along with consistency manager and shows the queues in between different components that allow all the components to run on their own pace without interfering each other.

3.3.1 Splitting Predicate

In normal processing, when an event is received by the splitting unit, it checks for the predicate to check if a new window needs to be started from this received event. If the predicate returns false for window start check, no new window is generated. However, if the predicate returns true for window start check, a new window is generated. The metadata of the event from which window is started, is copied to the window for the identification purposes.

Similarly, to check the closing point of a window, the received event and a window is passed to the predicate to check if this event is the end point of the window. If the predicate responds with true, it means the window needs to be terminated at this point. The meta-data of the event is saved in the window as closing point and a pseudo-event is sent to the operator instance to which this window was assigned to be processed. This pseudo-event has the necessary information, so that the operator instance can identify the window and mark it as closed with the correct information of the closing event. For every event, predicate needs to be checked for every open window to check if any of them needs to be closed at the specific event.

3.3.2 Scheduling of Windows

A newly generated window needs to be allocated to one of the connected operator instances. This is the responsibility of the scheduler to do so. Different scheduling strategies can be used to divide windows. Moreover, if the number of the windows is greater than the number of instances, batching is done so that more than one windows can be allocated to operator instances at a time. In case of batching, an event is sent just once to an operator instance for all windows as proposed by [MTR16] and [MTR17]. Our framework is a generic middleware which can handle any scheduling strategy defined by the user to schedule the windows to operator instances. Our framework calls a function of scheduler every time it has to allocate a new window, and allocates the window to the operator instance as told by the scheduler.

3.3.3 Consistency Controller

All the events forwarded by the consistency manager to the splitting unit are first received by the consistency controller of the splitting unit. Consistency controller saves the event in a sorted map to ensure that it has a copy of the event in case it needs to replay events after recovery. Consistency Controller has 4 main tasks to be done when necessary.

- Recovery
- Internal Replay of the Events
- Checkpoint
- Delete Deterministic Data

Algorithm.2 shows the handling of an event received from the consistency manager. After updating values of clock and K, the consistency controller checks the timestamp of the last event checked for predicate and forwarded to the instances (line 7). If the timestamp of the received event is smaller than that, the received event is marked as out of order event and broadcasted to all the instances, so that all the instances can handle this event before other events to avoid recovery, if possible. After broadcasting the event, the list of all active windows is copied to be used by the internal recovery routine (line 10). Then a stable state is found from the sorted map of states of the splitting unit and recovery of the splitting unit is initiated (line 11-14).

Once the recovery is completed, the events from the sorted events map are replayed by the consistency controller and every event is checked for the predicate but before reacting to the response from the predicate, first results are compared with the copy of active windows before recovery (line 15). If the same action was taken before recovery, nothing is sent to the operator instances. But if previously something wrong was done as compared to current response of the predicate, then a message is sent to the respective operator instance to modify the said window as required. Detailed discussion of internal recovery along with all the messages which are sent from the splitting unit to the operator instances is in the section 6.

If the received event is not out of order, recovery and internal replay will not be required. So, it will ask the checkpoint model if checkpoint is required before checking predicate for this event. If the response is true, a checkpoint is taken as explained in section 4 (line 17-19).

Algorithm.2. Pseudo-Code of Event Handling Routine of Consistency Controller of Splitting Unit

```
1.  Data: InputEvent event, sortedEventsMap, sortedStatesMap, windowsList,
2.  lastProcessedTimestamp
3.  begin
4.          add to sortedEventsMap
5.          clock ← event.clock                               // update clock
6.          K ← event.kValue                                  // update k value
7.          IF event.ts < lastProcessedTimestamp THEN         // if event is out of order
8.                  event.outOfOrder ← true
9.                  broadcast(event)
10.                 windowsCopy ← windowsList.clone()         // save a copy of windows
11.                 state ← sortedStatesMap.getLowerEntry(event.ts)
12.                 recover(state)
13.                 deleteHigherStates(state.ts)
14.                 lastProcessedTimestamp ← state.ts
15.                 internalReplay(windowsCopy)
16.         ELSE
17.                 IF checkpointRequired() = true THEN
18.                         takeCheckpoint()
19.                 ENDIF
20.                 splitAndForward(event)
21.                 lastProcessedTimestamp ← event.ts
22.                 deleteDeterministicData()
23.         ENDIF
24. end
```

In order to purge events, states and windows on time, updated K-value and clock value is sent along with every event. Consistency controller maintains local K-value and clock based on the values received from the consistency manager. These values help the consistency controller to delete the deterministic events and states. Moreover, it is made sure that there is at least 1 state available every time to which the system can recover back even if all the received events are deterministic according to the K-slack algorithm. This allows to handle some out of order events that are not handled by the K-slack algorithm as discussed in sub-section 4.5.

3.4 Operator Instances

Within an operator, there can be multiple identical operator instances in order to process windows in parallel. These operators first receive pseudo-event to get information about the window that is going to be started. After this pseudo-event, actual events that belong to that window are received by the operator instance. Operator Instance is mainly responsible to process the events with user-defined function and generate complex / intermediate events from them.

Every operator instance has its own consistency controller to handle the out of order events and to respond to the messages sent by splitting unit during its internal recovery. Consistency Controller is running on a separate thread in each operator instance to ensure that it doesn't affect the efficiency.

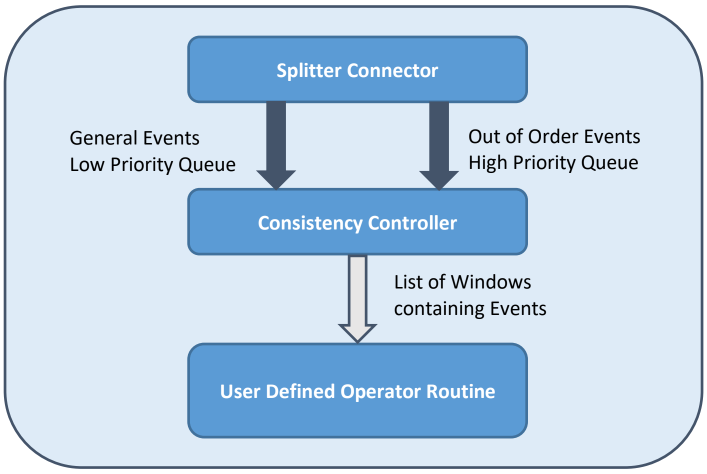

Figure 3-5 Architecture of an Operator Instance

Figure 3-5 shows the basic architecture of an operator instance in our implementation. All the events received from the splitting unit are added to the low priority queue but if an out of order event is received, it is added to the high priority queue as an attempt to avoid recovery (explained in next section). Consistency Controller picks up the events from the queues and add them to the respective windows, so that the user defined operator routine can process them. User defined operator routine is the actual processing that needs to be done on every event of the incoming stream to generate complex events out of it.

3.4.1 Sequence of Event Processing

There were 2 different options considered to process events when a batch of windows is assigned to an operator instance. First option we considered was to process all the events in a window first and then move to the next window. This sequence of processing events had a benefit that only a single object of the data required by the operator instance to process events in a window was saved in the memory. However, it has some drawbacks like the overall latency of the event will be increased in the case when the event is not being processed for the next windows because this window is still being processed. Once this window will be processed completely, then the next window will be processed, so the overall latency to process a single event for all windows is increased. Moreover. The operator can also be blocked if the current processing window is waiting for next events to be received even if we have some events in the other windows to be processed.

Another problem is how to identify states saved after checkpoint. For example, if an operator instance is allocated with 2 windows with 10 events each. Window A having events 0-10 and window B having events 5-15. Let’s say, window A is being processed first, and during processing, we take a checkpoint at event 7. Once this window is completely processed, we start processing window B and now we take a checkpoint at event 7 again. Then during recovery, if we have to recover back to state 7, then it will be difficult to decide which state should be recovered back. This is just a simple example which can be handled with some extra effort but there can also be some complex cases which will be very difficult to handle.

So, as an alternative approach, we take an event, take checkpoint if state needs to be saved before this event, process the event for all windows containing this event and then move to the next event and do the same. In this way, processing of the events as well as the process of taking checkpoint and recovery are performed smoothly. However, it needs an object of data for every window, so if there are a lot of windows being processed at a time by an operator instance, it can take a lot of memory and it will also take longer to checkpoint and recover the data.

3.4.2 Consistency Controller

Consistency Controller of the operator instances also works in a similar fashion as of splitting unit. All the events forwarded by the splitting unit are first received by the consistency controller of the operator instance. It saves the events to the sorted events map so that events can be retrieved back from the map if it needs to re-process events after recovery. In contrast with the consistency controller of splitting unit, it is not required to perform internal replay. Instead, it just recovers back and starts processing events again. However, it has to handle the messages received from the splitting unit during the internal replay in splitting unit. So, the main tasks of this consistency controller are as follows:

- Recovery
- Checkpoint
- Delete Deterministic Data
- Handle Messages from Splitting unit

All the events coming from the splitting unit are received by the main thread of operator instance. It checks whether the event is marked as out of order or not. If the event is marked as out of order, it is added to a special high priority queue, so that consistency controller can handle this event before normal events waiting in the lower priority queue. Consistency Controller thread picks up the event from high priority queue, if there is any out of order event present in it. It checks the timestamp of the last processed event by this operator instance as shown in line 5-6 in algorithm.3. If the timestamp of out of order event is higher than that timestamp, the out of order event is simply added to the sorted events buffer, so that the operator instance can process this event in the correct order. In this way, recovery of this instance can be avoided. However, if the timestamp of last processed event is higher than the timestamp of the out of order event, then recovery process is started for this operator instance (line 18-23). Important thing to note here is that all this process is independent of splitting unit and all other operator instances. So, if one of the operator instances recovers back, others are not required to recover back.

Before passing the event to the user-defined function to process this event, checkpoint model is asked if checkpoint is required before processing this event (not shown in the algorithm.3). If the checkpoint is required, a checkpoint function is called and the state is saved in the sorted states map. In order to delete deterministic data, K-value and the clock value of the consistency manager is retrieved from the events received. In this way, stable timestamp is determined from K-value and the clock value. Stable state is determined by finding the higher state than this stable timestamp. All the events and states and windows older than this stable state are deleted.

As the splitting unit is independent of operator instances to recover back and start internal replay, so it can start sending messages any time. So, when a message is received from the splitting unit with a command, operator instance has to respond to this command as soon as possible to increase the chances to avoid recovery by adapting the changes before operator instance has processed the events of concern (explained in algorithm.5). Further detail of handling messages received from the splitting unit is discussed in sub-section 6.2.

<u>Algorithm.3. Pseudo-Code of Consistency Controller Thread (Run Function) of Operator Instance</u>

```
1.  Data: OutOfOrderEventsQueue q1, ReceviedEventsQueue q2, sortedEventsMap,
2.  sortedStatesMap windowsList, lastProcessedTimestamp
3.  begin
4.      WHILE (true) DO
5.          IF q1.empty = false THEN                  // if out of order event is waiting
6.              event ← q1.poll()
7.          ELSEIF q2.empty = false THEN
8.              event ← q2.poll()
9.          ELSE
10.             CONTINUE;                             // Loop until new event is received
11.         ENDIF
12.         add event to sortedEventsMap
13.         clock ← event.clock
14.         K ← event.kValue
15.         addToWindows(event)                       // add event to all open windows
16.         IF event.outOfOrder = true THEN
17.             waitingPhase ← true
18.             IF event.ts < lastProcessedTimestamp THEN
19.                 state ← sortedStatesMap.getLowerEntry(event.ts)
20.                 recover(state)
21.                 deleteHigherStates(state.ts)
22.                 lastProcessedTimestamp ← state.ts
23.             ENDIF
24.         ELSE
25.             deleteDeterministicData()
26.         ENDIF
27.     ENDWHILE
28. end
```

3.5 Merging Unit

Merging Unit is the last component of an operator. It is connected to all the operator instances. Whenever, a complex event is generated by any of the operator instances, it is forwarded to the merger. Merger takes all the complex events, checks if there are any duplicate events generated by the instances because of the overlap of windows. It removes the duplicate complex events and sorts them before it emits these events out of the operator. [MST+17]

Just like all other operator components, merger can also have consistency controller in it, so that it can take care of out of order complex events generated by the operator instances. Moreover, there is also a possibility that wrong complex event was generated by the operator instance which needs to be retracted. This information can be sent via messages from operator instance to the merger to ask it to remove the specific complex event that was generated speculatively.

3.6 Consumer

Consumer is the one who receives the output stream of complex events generated by this operator. It can be another event detector that depends on the events generated by this operator. In this case, complex events generated by this operator are actually the intermediate events. Moreover, in this case, this operator will be called as the source of the events for the consumer. On the other hand, if this operator is at higher level in the hierarchy of operators and it generates the complex event on which an action can be performed, then the consumer will be an actuator or a software that will activate a routine to perform the specified action. [Luc06], [Luc08]

4 Checkpoint of the System

Checkpoint is a term used for the snapshot of the system. While taking checkpoint, we do a deep copy of the important information of the system at a particular time and save it as a state of the system at that time. So, in case of any out of order event, our system can recover back to the required state and start processing again from that point of time.

As discussed in section.3, the process of taking checkpoints is not the same as proposed by [MP13b]. The basic idea of taking checkpoint of the whole system is changed to take checkpoint of every component independently in our implementation. Moreover, the frequency of taking the checkpoints also varies because it depends on a number of factors in our approach but in [MP13b] it was based on just one factor i.e. if the event is speculative or not. Here we will discuss the differences of both approaches in detail.

4.1 Open Box Implementation

Our implementation is not for a generic black box system. It is specific for parallel complex event processing. So, it is an open box implementation with some optimizations which were surely not possible with black box approach. Consistency Manager, lying outside the splitting unit is only responsible for maintaining the K-slack algorithm and it also takes care of the speculation according to the CPU load reported by the instances. Consistency Controllers, which are part of splitting unit as well as all of the instances, are actually responsible to take checkpoints and to recover the state if required.

However, the approach described by [MP13b] supposes that the system is black box. So it means that the consistency manager doesn't know anything about the system. Such an approach that supposes the system to be black box has a big advantage of being generic for any system. But obviously, nothing comes for free, performance and efficiency is compromised to keep the approach generic, so that it can be applied to any event processing system.

4.2 Checkpoint Independently

An open box approach for parallel CEP system has a great advantage of independence of all components. All operator components have their own consistency controllers which will work independently of other consistency controllers as well as consistency manager. Each consistency controller observes the performance of its own operator component and has complete authority to decide whether to take checkpoint or not. This decision can be based on a number of different factors along with the status of the event whether it is speculative or deterministic. Different factors that can be important for the decision are discussed in sub-section 4.7. Independent decision to take checkpoint allows all the instances to work at their own pace. Moreover, it also ensures that no instance is bound to take checkpoint if it is not processing any window at the moment, as in the Mutschler's approach [MP13b].

One of the main problems that affects the throughput of the system in Mutschler's approach [MP13b] is that whenever a checkpoint is required, checkpoint is taken by the whole system. Before an event is forwarded speculatively to the system, consistency manager will first have to ask the system to send its current state. This current state should be the state of the whole splitting unit as well as all the operator instances connected to the splitting unit, which can cause a lot of overhead because even if an instance is not processing any window, it will have to take a checkpoint of its current state. In contrast to this, our approach has a consistency controller in every operator component to independently decide whether to take checkpoint or not.

4.3 Save States Locally

As discussed before, the consistency controller decides whether to take checkpoint or not. So, if the consistency controller decides to take checkpoint, it will take the checkpoint and save the state in its own memory. In parallel CEP system, the consistency controller of splitting unit will be responsible for taking checkpoint of splitting unit if required and save the state in its own memory. Similarly, the consistency controller of every operator instance will be responsible to take checkpoint and keep the state saved. As, it doesn't need to send the state back to consistency manager, so the communication overhead is reduced.

On the other hand, according to [MP13b] system will have to send the complete state to consistency manager. Because consistency manager will save all the states in its memory and whenever it detects that the recovery is required, it will find the correct state and send the state back to the system. The system will receive the state from the consistency manager and recover the splitting unit and the operator instances back to the state. In case of parallel complex event processing, it causes communication overhead because all the processes; Consistency Manager, Splitting unit and Operator Instances can be running on different machines, so they will have to send the state over the cable which can be a large overhead.

4.4 No Need to Wait for State

As, all the operator components are independent in our approach, so none of them has to wait for others while taking checkpoint. However, in the approach proposed by [MP13b], the consistency manager has to ask the system to take a checkpoint and send its state back. Meanwhile, consistency manager is waiting idle to receive the state back from the system. Furthermore, in case of parallel complex event processing, splitting unit will receive the request from consistency manager to take checkpoint, so it will take checkpoint of its current state and send the request to all the operator instances connected to it. Meanwhile, splitting unit will be waiting idle to receive the states back from all the operator instances. The operator instances will receive the request from splitting unit to take checkpoint. They will save their state and send it back to the splitting unit. Splitting unit will populate all the states of operator instances in the system state. It will also include its own state in the system state and send it back to the consistency manager. After consistency manager has received the system state back from the splitting unit, it will proceed further and send the speculative event to the system, so that it can be processed. So, it includes a lot of waiting time for consistency manager because before sending every speculative event, it is waiting for the state to be received from the system. This waiting time is further increased for parallel complex event processing because splitting unit has to get states from all the operator instances, so there is a waiting time in it also. Figure 4-1 shows the complete loop graphically that needs to be completed in the approach of [MP13b] before forwarding speculative event.

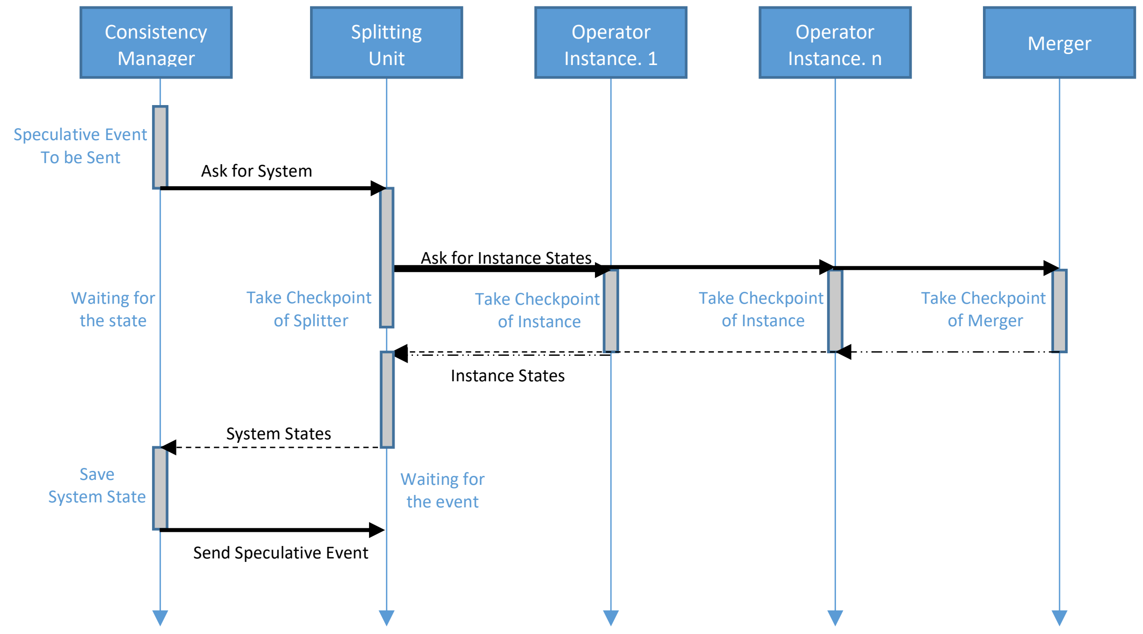

Figure 4-1 Taking Checkpoint in Mutschler's Approach

However, a little optimization is also possible in this approach. When the consistency manager asks the splitting unit to take a checkpoint, it has to do 3 tasks; take checkpoint of its own state, ask operator instances to send their states and wait for states from the operator instances. It can be optimized by first asking all the operator instances to take checkpoint, then while waiting to receive their states back, take checkpoint of splitting unit's own state. It will reduce the waiting time of consistency manager a little bit. Instead of first taking checkpoint of its own state and then asking operator instances to take checkpoint. This optimization is already done in the implementation used for all the evaluations in section 7.

4.5 Handle Out-of-Order Events

In real world applications having multiple sources, there is quite a possibility that we receive a number of out of order events. Moreover, the delay can also vary at runtime because of network problems or any unexpected conditions. [MP13a] and [MP13b] determine the K-value at run-time by increasing the K-value if an event was delayed more than the previous K-value and strictly follow the K-slack algorithm. So, there is a chance that the event that changed the K-value cannot be processed in the correct order because the events having higher timestamps are already processed deterministically because K-value was less at that time. As, [MP13b] proposes to delete the older states and events if a deterministic event is being passed to the system. So, in this case, the event will have to be passed to the system in wrong order and the system will fail to recover back because there will be no state available in the memory to which it can be recovered back.

Our approach always keeps at least one last state to which the system can be recovered back and all the events received after that state, so that we can do internal replay from those events after recovery. This allows us to give some more time for the system to adapt to the correct K-value without any failure.

Here, we give an example to show such a condition when the approach of [MP13b] will fail, however, our approach will still be able to handle the inconsistency correctly. To show this, we have slightly modified the example we used in the previous sections. Figure 4-2 shows speculatively forwarded events shaded gray in output stream. Whenever an event is forwarded deterministically, all the older states are deleted by [MP13b]. So, in the very start, when 'b-3' is received after forwarding 'a-4', it doesn't has any state to recover back. This is the case when the K-slack is not having any information yet, so it can be ignored in continuous CEP systems. Moreover, this can also be solved simply by sending all the events speculatively until we have some value for K.

However, the second case in this example has more importance when 'c-15' is received out of order. Before the arrival of this event, K-value was 4, so when the clock was updated to 20, it allowed to forward 'b-16' deterministically because it also satisfies the Equation 2.2. In this way, when 'c-15' is received, it should recover the system back to some state older than timestamp 15. However, it doesn't has any state of the system at that time because last event was forwarded deterministically. In this way, this technique will fail to handle this out of order event.

On the other hand, our implementation will be able to handle this case also, we always keep at least one state, so that the system can recover in such cases. And it works perfectly fine even if we use the same checkpoint model as proposed by [MP13b]. Our implementation will have the state saved before processing 'c-11' speculatively. It will recover back to this state and process the events again in the correct order.

4.6 Data included in State of System

In our implementation, there is no need of something like a "system state" that includes the whole state of the complete system including the state of the splitting unit as well as the states of all the instances. Rather, the states are saved locally in this approach. That means, the splitting unit will take the checkpoint of its required data and save it as a state of splitting unit within its memory. Similarly, all the instances take checkpoint when required and save the state of operator instance in their own memory.

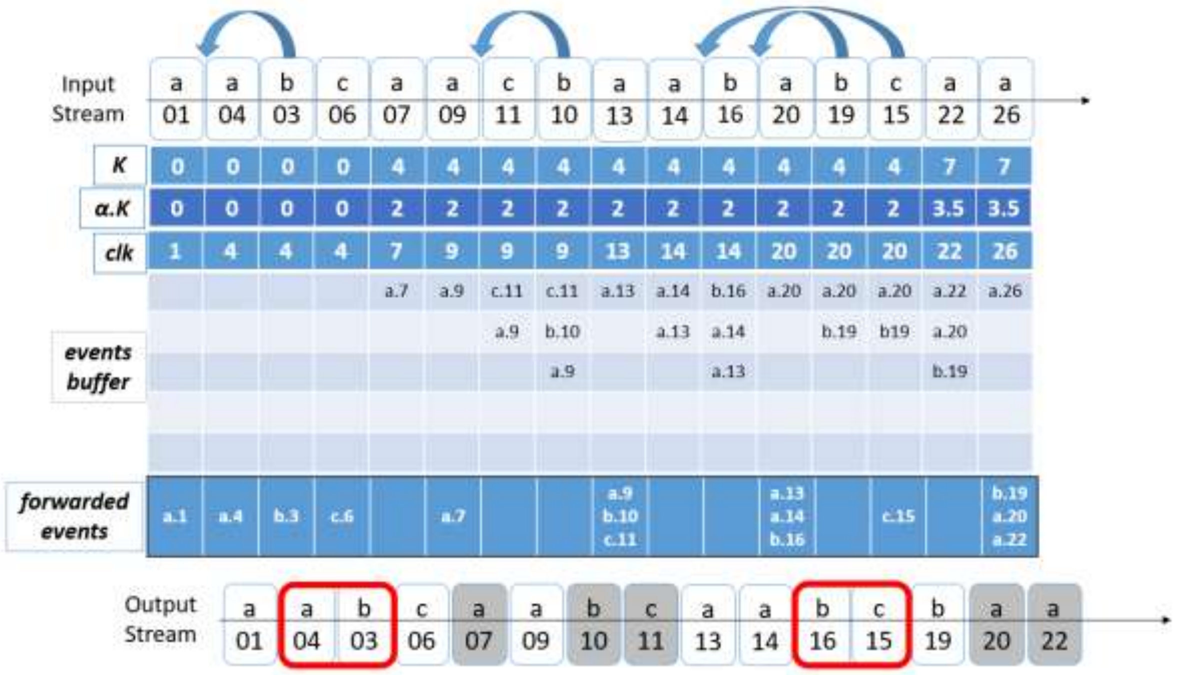
Input Stream
a 01, a 04, b 03, c 06, a 07, a 09, c 11, b 10, a 13, a 14, b 16, a 20, b 19, c 15, a 22, a 26
K
0 0 0 0 4 4 4 4 4 4 4 4 4 4 7 7
α.K
0 0 0 0 2 2 2 2 2 2 2 2 2 2 3.5 3.5
clk
1 4 4 4 7 9 9 9 13 14 14 20 20 20 22 26
events buffer
a.7 a.9 c.11 c.11 a.13 a.14 b.16 a.20 a.20 a.20 a.22 a.26
a.9 b.10 a.13 a.14 b.19 b19 a.20
a.9 a.13 b.19
forwarded events
a.1 a.4 b.3 c.6 a.7 a.9 b.10 c.11 a.13 a.14 b.16 c.15 b.19 a.20 a.22
Output Stream
a 01, a 04, b 03, c 06, a 07, a 09, b 10, c 11, a 13, a 14, b 16, c 15, b 19, a 20, a 22

Figure 4-2 Example of K-slack Algorithm Fail

However, in [MP13b], there should be a “System state” in the case of parallel complex event processing. A system state consists of a state of splitting unit as well as a list of states of operator instances. When the system state is saved by the consistency manager, it is marked with the current timestamp also.

4.6.1 State of Splitting unit

While taking checkpoint of the splitting unit, all the following data needs to be cloned to keep a copy of the current state. This state will be required by the splitting unit later on to recover back. The state of splitting unit includes the following information for our framework:

- <u>Index of last processed event</u>
  The index of the event that was last processed before taking the checkpoint is saved, because it will be required to replay the events after recovery.
- <u>Number of events acknowledged</u>
  As instance, acknowledges the event after it has completely processed the event. This count needs to be saved because events will be replayed by splitting unit after recovery, they will be acknowledged again.
- <u>Linked List containing all the events that are not acknowledged yet</u>
  A linked list is used to detect if the event is acknowledged after processing by all the instances it was assigned to. It also needs to be cloned because the events will be acknowledged again after processing them again after recovery.
- <u>ID assigned to the last started window</u>
  ID that is assigned to every window at the time of start is assigned by the window factory, which needs to be recovered back because it will start windows again.
- <u>Array List of Windows</u>
  The array list used to keep windows is also cloned to recover back the windows that were opened at the time when checkpoint was taken. It consists of all the windows that were open at the time of checkpoint.
- <u>Deep Copy of Predicate Data</u>
  As, this is a generic middleware, so predicate can be anything implemented by the user. Furthermore, the data being used by the predicate will also be user defined. But it can be important for the user to recover its data back if system recovers back. So, the user has to implement the clone function in its predicate data class to save the data in the state.
- <u>Event for New Window</u>
  During internal replay after recovery, there is an optimization that if an event wants to start a new window which was not started previously, we save that event as event for new window. Then we check the next event, if this event wants to close the window that it has mistakenly opened because it was speculatively processed. If that’s the case, we ignore the start new window request for the previous event and close window request from this event and just add previous event to the window that was supposed to be closed. The detailed explanation of this optimization is in the section 6.

4.6.2 State of operator instance

Similarly, all the instances also save the necessary data that needs to recovered back at the time of recovery. In our approach, the state of operator instance will be saved locally by each instance, however, according to [MP13b], every instance should send its state to the splitting unit. Every state is tagged with the instance ID before sending it to the splitting unit, so that it can be identified at the time of recovery. The data saved in the state of operator instance that is common in both approaches is as follows:

- Timestamp of Last Processed Event:
  The timestamp of last event that was processed by the operator instance needs to be saved in the state because operator instance will start processing the events again after recovery, so it should know, from where to start processing again.
- Timestamp of Start Event of Next Window:
  As the window is identified by the timestamp of start event, so this needs to be saved because it is required to add the next window to current processing windows.
- Timestamp of Last Event of First Window in Current Windows:
  To remove the first window from current windows list and mark it processed, we need the timestamp of its last event, so that when that event it processed, the window can be removed from the list. So, this timestamp is also necessary to be saved in the state.
- Flag of Availability of Next Window:
  If next window is not available at the time of recovery, this flag is marked high, so that start event timestamp of next window can be updated as soon as a new window is received by the instance.
- Flag of Availability of Close Point of First Window in Current Windows:
  If the close point of first window is not available at the time of taking checkpoint. This flag will be kept high, so that it can be checked at the time of recovery, if the close point is received.
- Linked list of Current Windows:
  List of current windows also needs to be cloned, because the processing will resume from the very same point, so the current list of windows should correspond to the next event. But this is not the deep clone of the windows, it is just a list of pointers to the windows that is deep cloned.
- Operator Data:
  To keep this framework generic, the operator can be implemented by the user according to his requirements. So, the data required by the specific operator will be saved in operator data which implements clone-able interface. The user has to implement the mandatory “clone” function in which he can deep clone the data that he wants to recover at the time of recovery.

In addition to this data, [MP13b]’s approach also needs the following data, so that it can recover back properly based on this state:

- Timestamp of Last Received Event:
  The timestamp of the last received event needs to be saved in the state because the events will be replayed by the consistency manager, so it needs to be recovered back to the previous state upon recovery.

- Map of Assigned Windows:
  As, the events will be replayed by the consistency manager, after recovery, so we need to take a clone of the windows assigned to the operator instance at the time of taking checkpoint because it must be recovered back.

Important thing to note here is that the state of operator instance in our case doesn't include the data structure that contains all the assigned windows because the events will not be replayed by the splitting unit on recovery, so there is no danger of having replicated events. Secondly, if the assigned windows data is recovered back, the system will lose all the windows that were received in between the checkpoint and recovery point. This optimization is explained further in section 6.

4.7 Checkpoint Model

In our implementation, the decision to take checkpoint is not just dependent on whether the event going to be processed is speculative or deterministic. There can be a number of different factors that affect the latency or throughput of the system.

Before forwarding event from the splitting unit, a function "checkpointRequired" is called which returns Boolean output. Based on the output of this function, splitting unit will decide to take checkpoint of its current state before forwarding the event or not. Same is the case with operator instances, before processing every event, its local "checkpointRequired" function is called and decision is taken based on the output of the function.

Within each "checkpointRequired" function, a checkpoint model can be implemented that will return true or false based on a number of factors. Moreover, splitting unit and operator instances can have different checkpoint model based on different factors which are best suitable to optimize the performance.

As discussed before, in our implementation, windows and events received since last state are saved until we have another state which has all the deterministic events behind it. So, the frequency of the checkpoints needs to be optimized because, if frequency of taking checkpoints is very low, the data structures containing events and windows can grow and as a result it can affect the efficiency of the system. And if the frequency is very high, a lot of time might be wasted just to take checkpoints of the system which can have a bad impact on the throughput of the system.

We have currently implemented 2 different models in our framework. However, we have also proposed an adaptive checkpoint model that adapts the checkpoint interval depending on the time consumed to take checkpoint and the time required to re-process the events after recovery. First model is as simple as taking checkpoint after a fixed specific number events. In section 7, the impact of changing frequency of checkpoints in this model is discussed.

Second model is based on the speculative event. In this model, checkpoint is taken after a fixed specific number of speculative events. If the frequency is equal to "1" in this model, it becomes exactly the same as proposed by [MP13b]. As, we save all the events and windows since last checkpoint, so this model might eat up all the memory of the system if there are no speculative events forwarded by the consistency manager. In order to handle this problem, in addition to the checkpoints proposed by the model, we also take checkpoint whenever the number of events since last checkpoint is increased from 1000.

4.7.1 Adaptive Checkpoint Model

Another proposed checkpoint model adapts the frequency of checkpoint at runtime based on average time consumed to take a checkpoint and the average time consumed to recover back. In addition to this, the average time consumed to process an event by the operator instance is also taken into consideration to take more calculated decision if the checkpoint is required or not. This model focuses on the fact that if the frequency of checkpoint is high, it will consume so much time in taking checkpoints and on the other hand, if the checkpoints are not taken frequently, then after recovery, a lot of events will have to be processed again.

In order to optimize this frequency at run time, we can consider both of these factors and determine at runtime if it will be better to take a checkpoint or not. First factor is the time required to re-process the events if an out of order event is received. This factor further depends on two variables; number of events processed since last checkpoint and the average time required to process an event for all the windows. This will give us the total time required to re-process all the events after recovery which can be compared with the time required to take checkpoint. If the number of events to be re-processed is increasing, it means we need to take a checkpoint because it will take longer to re-processing a lot of events (processing of the events can be very expensive). On the other hand, if the checkpoint takes a lot of time as compared to processing time of events, it’s better to re-process the events in case an out of order event is received. Equation 4.1 can be used to determine if the checkpoint is required before processing this event or not.

$$n \times t_{proc} > t_{cp} \qquad \textit{Equation 4.1}$$

In Equation 4.1, ‘n’ is the number of events processed since last checkpoint, ‘t_{proc}’ is the time consumed by the operator instance to process an event and ‘t_{cp}’ is the time consumed by the consistency controller to take checkpoint and save the state.

5 Recovery of the System

The process of recovery means that the system is restored to a previous state that was saved while taking checkpoint. The recovery process can also be very expensive with respect to time taken during the recovery. Main reason behind expensiveness of this process is that the deep cloning involved in this process is very expensive. After recovery of the operator instance, we need to check if there are any messages that have modified any of the windows in current windows list after the checkpoint. If that's the case, we will have to modify the window again after recovery. The main features of our implementation are explained below:

5.1 Open Box Implementation

As, our implementation is optimized for parallel complex events and we have consistency controllers in all the operator components, so recovery process can also be handled by the consistency controllers themselves. Every operator component is independent to decide to recover or not.

In our implementation, consistency manager will deal with the out of order event just like any other event and forwards it. This event will be received by the consistency controller of the splitting unit. Consistency Controller of the splitting unit will check the time stamp of the last event forwarded to an instance and compare it with the timestamp of the received out of order event. If necessary, it will recover the splitting unit back to some previous state. It will also broadcast the event to all the instances, so that they can start recovery (if required) as soon as possible.

Figure **5-1** shows how an out of order event is handled in our approach.

An instance will receive the event and pass it to its own consistency controller. The consistency controller of the instance will check the timestamp of the last event that was processed by the operator instance and recover itself if required. The states are saved locally in our approach, so if any operator component needs to recover back, it will find nearest possible stable state and recover back to that state and start processing again.

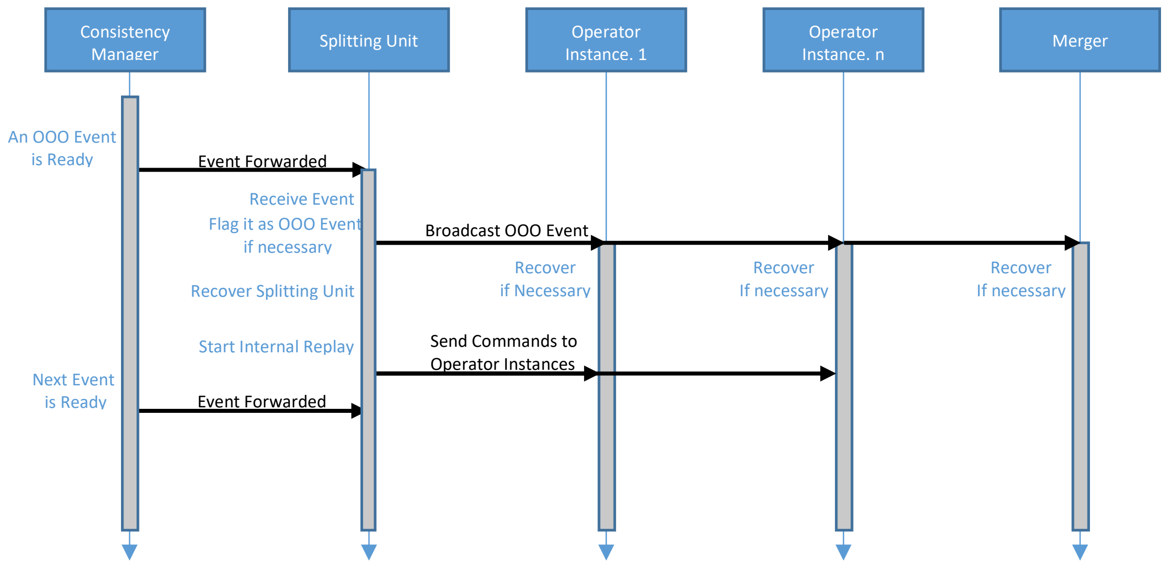

Figure 5-1 Handling of Out of Order (OOO) Event in Our Approach

However, [MP13b] is generic for all the systems because the consistency manager doesn’t need to know any details about the state of the system. It will just ask for the state when required, keep it saved as it is, in a data structure marked with the timestamp when this state was received. Later on, when recovery of the system is required, consistency manager will search for the nearest state to the out of order event received by it. The state is accessed from the data structure and forwarded to the system as it is without knowing the internal details of the state. However, it has some performance issues that are optimized for parallel complex event processing in our implementation.

5.2 Recover Independently

The decision of recovery depends on the current state of every operator component. Consistency controller of every processing unit is responsible to check whether recovery is required for this component or not. It’s quite possible that on arrival of an out of order event, some of the processing units recover and some don’t. However, it was not possible in [MP13b].

In [MP13b], when the consistency manager detects that the wrong speculation was done and the system needs to be recovered back, it will find the required state from the data structure containing all the states and forward this state to the system. The system; in this special case parallel CEP system will receive the state in the splitting unit. The splitting unit will retrieve the state of splitting unit from the system state and the states of operator instances will be forwarded to the operator instances. The splitting unit will recover itself to the state retrieved from the system state and wait for the new events to be replayed by consistency manager.

All of the instances will receive their states from the splitting unit, they will make sure from the instance ID that the state received is actually the state of this instance and recover back to the state. In this way the whole system is recovered back to the previous state including some of the operator instances even if they were not processing any window at the time of recovery or at the time of checkpoint.

5.3 High Priority Queue for Out of Order Events

When an operator instance receives an event, it first checks if the event is out of order or not, if the event is out of order, it will be added to a higher priority queue, so that it can be handled as early as possible, even if there are more events waiting in the normal queue.

In the normal processing, there is possibility of having a number of events in the queue in operator instances. To avoid waiting in the queue, the higher priority queue is solely used for the out of order events. In this way, the possibility to avoid the need of recovery is increased. We will explain in next sub-section that how we can avoid the recovery.

5.4 Avoid Recovery by Accessing Events Queue

All the operator components have queues to receive the events and handle them one by one. There is a possibility that the out of order event received by the consistency manager can be simply added to the queues of the splitting unit without any problem because splitting unit has not forwarded these events yet, so it will forward them in the proper sequence.

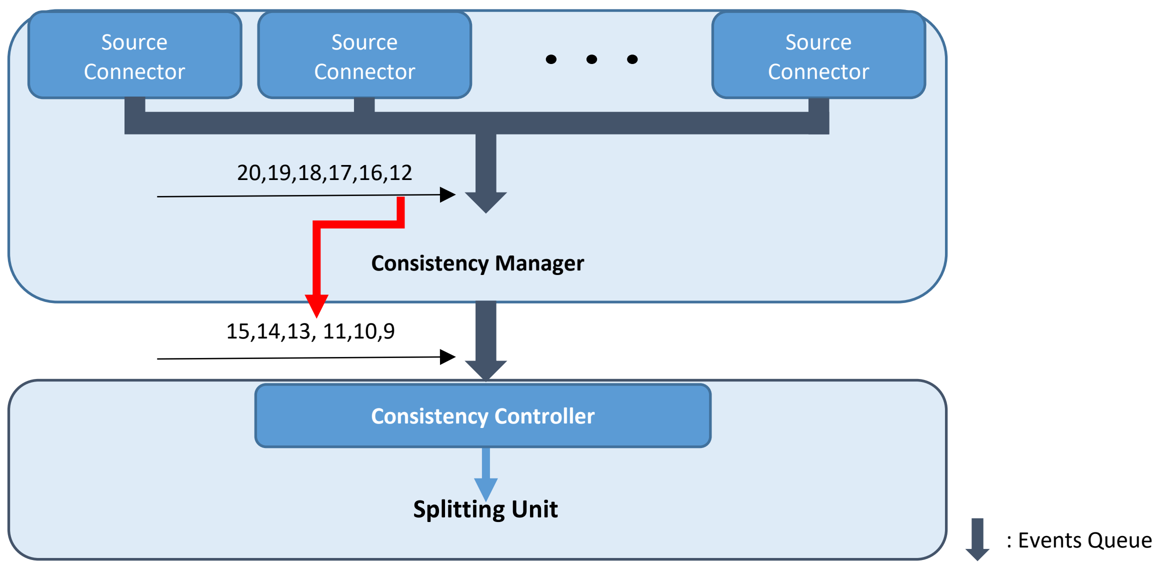

Figure 5-2 Example to show that Recovery can be avoided by adding events to the queues in Splitting Unit

As, our approach is an open box implementation, and we have our consistency controllers within every operator component, so we can access the queues of the events directly. If the event forwarded by the consistency manager was actually out of order event, it will not yet be marked as out of order here. First of all consistency controller of the splitting unit will check if it has processed any event that makes this new event out of order, if no, the event is simply added to the queue, it will be processed further as a normal event. In this way, recovery can be avoided from the very first stage. Figure 5-2 shows that an event with timestamp '12' is out of order for Consistency Manager but it can be added to the queue in front of consistency controller of the splitting unit to avoid recovery.

Even if the event was marked out of order by splitting unit and it had to recover, we broadcast the event and use high priority queue for out of order events in the operator instances, so that they can be processed as soon as possible. If we are successful to bypass all the events that must be processed after this out of order event in the operator, we can simply put this event to the queue, so that it can be processed normally without any recovery in the operator instance. An example is Figure 5-3, which shows that the out of order event with timestamp '3' will be added directly to the windows to avoid recovery. Similarly, the out of order event with timestamp '5' is handled in the correct order due to the presence of high priority queue. Although this event was received after events '6', '7' and '8' but due to high priority queue, it will be processed before them to bring it to the correct order.

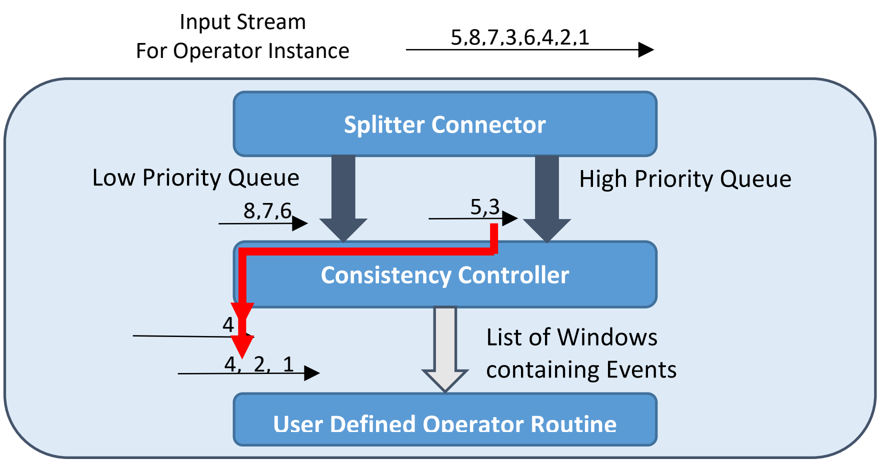

Figure 5-3 Example to show that Recovery can be avoided by adding events in queues in Operator Instance

In [MP13b], the system is supposed to be a black box, so consistency manager does not know the current status of the system. So, once the event is forwarded by the consistency manager to the system, it has no control over the event anymore. It will have to ask the complete system to recover back if something goes wrong. In the examples shown in Figure 5-2 and Figure 5-3, it will have to do recovery which is avoided by our implementation.

5.5 Recover from Locally Saved State

When the consistency controller of a processing unit detects that the recovery is required, it finds the required state in the tree-map of saved states of its unit. The timestamp of the event is used to find nearest lower state in the tree-map. Then, the recovery process is started once it gets the semaphore to ensure that no other thread of the instance is doing something during recovery. It is necessary to make sure that the data is not corrupted by some other thread during recovery.

Contrary to our approach, in [MP13b], states are not saved locally. As discussed in the section 4.2, all the states received from the system are saved by the consistency manager. At the time of recovery, consistency manager finds the state to be recovered and sends the complete state of the system to splitting unit and further to the operator instances. These operator components might be running on different CPUs or different machines connected via cable. Sending the complete state over the network is always expensive in sense of communication overhead.

An optimization can be done to some extent by the user in [MP13b]. Instead of sending the complete state on the network, it can be optimized by sending just the pointer to the state. So, in this way, all the states will be saved locally but just an identifier of the state will be sent to the splitting unit by all instances and similarly just a pointer to system state will be sent by the splitting unit to consistency manager. At the time of recovery, just the pointer will be sent back by consistency manager, and the operator components will be able to identify the state to be recovered.

5.6 Data Recovered from System State

As discussed in the sub-section 4.6, there is no system state in our implementation because all the states need to be saved locally, so, we don’t need to combine them together to make the system state. We have a state of splitting unit and a state of operator instance separately.

However, in Mutschler’s approach, there is a system state that consists of a state of splitting unit as well as a list of the states of the operator instances. When the recovery request is received by the splitting unit, it retrieves the state of splitting unit for itself and sends the states of operator instances to them, so that they can retrieve their required data and recover back. The data included in the state of splitting unit and state of operator instance and how this data is used to recover is explained below:

5.6.1 State of Splitting unit

The state of splitting unit includes all the necessary data that is required for the splitting unit to work correctly after the recovery. An important point here is that the implementation of this project for evaluation is done in Java. In java, if the data structures are not cloned, they point to the same objects. So during recovery all the data structures are cloned back from the state to make sure that if the system has to recover back to the same state twice, the state should not be edited after first recovery.

It needs to be the original state, so that the system can recover to this state again. The information recovered from the state of splitting unit is as follows:

- Index of last processed event:
  The index of the last processed event is actually used as the "pointer" to the last processed event. This needs to be recovered back from the state because it defines that which event needs to be forwarded next. So, it is an essential recovery to ensure correct replay of the events.

- Number of events acknowledged:
  All the instances, acknowledge the events when the event is processed completely. It needs to be recovered because events will be replayed again by the consistency manager, they will be processed again by the instances, and acknowledged again, so to keep the count correct, this value needs to be recovered back.

- Linked list containing all the events that are not acknowledged yet:
  The linked list is used to keep a copy of events that are forwarded to instances but not acknowledged yet. So, due to the same reason as last point, we need to recover this list because we will receive all the events again from the consistency controller that will be added to this list and then removed from the list when acknowledgements are received.

- ID assigned to the last started window:
  New windows are started by the window factory as described in the section 3.3. It assigns an ID to each window, which is defined by a variable that keeps track of the ID of last started window. So, in order to keep the IDs of windows correct, we need to recover the value of this variable from the state.

- Array list of windows:
  An array list is used to keep track of all the windows that are currently open. So, we need to recover it, so that after recovery it has the windows that were open at the time of checkpoint. It is actually used to check that which windows are assigned to which operator instance, so that all the events of that window should be sent to that operator instance, and when it's time to close the window, pseudo-event for window close can be sent to the operator instance.

- Deep Copy of Predicate Data:
  To keep this framework generic for any type of predicate to be used to split the events to different windows, predicate is to be defined by the user, similarly, data that will be required by the predicate is also defined by the user. So, in case it has some data that needs to be recovered back while recovery process, the clone function can be defined by the user, so that the process of predicate is not affected if the system recovers back to some previous state.

- Event for New Window:
  This event is used in the internal replay, it is used for the optimization that if an event wants to start a new window during internal replay and the next event wants to close its window because it was opened mistakenly, then that event for new window will be added to the previously started window to avoid communication overhead. So, this event needs to be recovered back because it will be used to start new window.

5.6.2 State of operator instance:

A state of operator instance consists of all the data that it needs to recover back to the previous state and behave just the same as if it has not processed any event since the checkpoint was taken. The data that needs to be recovered to ensure this important condition is as follows:

- Timestamp of Last Processed Event:
  As, the system is recovered back because the wrong event was processed by the system at wrong time, so some events have to be processed again. So, to make sure, the operator instance processes the events again, this variable needs to be recovered back.

- Timestamp of Start Event of Next Window:
  Out of assigned windows, the current windows are the ones that are being processed right now, so in order to add the next window to the current windows, the start time is saved in this variable. As, all the windows are also recovered back to previous state, we need to recover this variable to old value.

- Flag of Availability of Next Window:
  There can also be a possibility that the next window was not available at the time of checkpoint, in that case this flag will be high, and it will be checked when the new window is received from the splitting unit to react accordingly. This flag must be updated, otherwise the algorithm will fail.

- Timestamp of Last Event of First Window in Current Windows:
  To remove the first window in current windows list and mark it processed, we need the timestamp of its last event, so that when that event it processed, the window can be removed from the list. This variable needs to be recovered after recovery to properly remove the windows from current windows list when all the events of that window are processed.

- Flag of Availability of Close Point of First Window in Current Windows:
  Just like the next window availability flag, this flag is also very important to be recovered to make sure that the proper actions are taken when the closing of a window is received.

- Linked List of Current Windows:
  As discussed before, current windows are the ones that are being processed by the operator at that specific time of checkpoint. So, it needs to be recovered back to the same state.

- Operator Data:
  As, the operator can be implemented by the user according to the needs. So, the data required by the specific operator is cloned according to the needs of the user at the time of checkpoint. At the time of recovery, data is cloned back to the instance operator, so that it can work flawlessly after the recovery of the system.

In addition to this data, Mutschler's approach also needs the following data:

- Timestamp of Last Received Event:
  The timestamp of the last received event needs to be recovered back to the previous state, because right after the recovery, some of the events will not be there in the windows because windows are also recovered back. The last received event timestamp is used by the operator to check if there is new event available to be processed, so if we do not recover this value, the operator will start trying to process the event that is not available now.

- Map of Assigned Windows:
  All the events will be replayed by the consistency manager after recovery, they will be processed by the splitting unit and new windows will be sent to the operator instances. So, the tree map containing all the assigned windows should be recovered to the state at the time of checkpoint to avoid any clashes.

Important thing to note here is that the state of operator instance in our implementation doesn't include the data structure that contains all the assigned windows because the events will not be replayed by the splitting unit on recovery, so there is no danger of replicated events. Secondly, if the assigned windows data is recovered back, the system will lose all the windows that were received in between the checkpoint and recovery point. So, we do not recover the events received from the splitting unit during recovery, so we don't need to recover the timestamp of the last received event also. This optimization is further explained in section 6.

5.7 Replay Events after Recovery

Once the recovery of the system is completed, the replay of the events should start. Here, we try to avoid the communication overhead caused by sending all the events with complete data again. It also saves us some memory because we don't need to include assigned windows and events received by the instance in the state during checkpoint because we don't need to recover them back.

Instead of sending the events, the events are reprocessed by the splitting unit internally and newly formed windows are compared with the previously generated windows that were sent to the instances. The results of these comparisons are analyzed and commands are sent to the operator instances to react according to the situation. It avoids transmission of heavy data of the events. The complete detail of this process of internal replay is explained in the next section.

Contrary to this, in [MP13b], once the consistency manager has forwarded the recovery event to the system, it recovers its own "pointer" that defines which event needs to be sent next. After that, it starts to forward the events again from the recovery point. These events will be in the queue for the time when splitting unit is in the recovery phase. Once, it has finished recovery, it will start handling those events again. It will create the windows again and forward all the events again.

In this process events are forwarded from the splitting unit to the operator instances again can cause communication overhead. In some cases, if the data contained by the events is a high quality image or a video clip or any other heavy data, this communication overhead can no longer be in negligible range.

6 Internal Replay of the Events

In this section, we will explain the working of our proposed technique of internal replay in the splitting unit. It avoids un-necessarily sending the events to operator instances again. As discussed in previous sections, our implementation can handle the out of order events also while processing the stream of events. When an out of order event is detected, the splitting unit has to recover back to the previous stable state. After the process of recovery, instead of sending all the events again to all the operator instances, we have proposed another technique in which events are replayed within the splitting unit but not forwarded to the operator instances. The splitting unit checks the splitting predicate for every event again and compares it with the previous speculative windows. During comparison, all the actions required to be taken by the operator instances are sent via messages, so that operator instances can act accordingly.

The explanation of the technique in this section is arranged such that the process of internal replay in splitting unit is explained in sub-section 6.1. The actions that an operator instance takes on arrival of every message from the splitting unit is explained in sub-section 6.2. The algorithm in tabular form with situations and actions taken by splitting unit and the operator instances is shown in sub-section 6.3. Some of the advantages of using this technique are explained in sub-section 6.4.

6.1 Internal Replay in Splitting unit

As, the main task of the splitting unit is to check the predicate, if new window needs to be started and also to check the predicate, if any open window needs to be closed. There is a list of windows maintained by the splitting unit which contains both open and closed windows until they become completely deterministic. A flag is saved within the window class to show if the window is still open or closed. If the window is closed, the timestamp of the event that has closed this window is also saved. Once, the windows are deterministic and ready to be deleted, they will purged out of the list.

In order to check if the decisions taken previously to open and close the windows were correct or not, we need to copy the list before recovery and keep it saved for comparison during replay. After the recovery, the list of windows will be recovered back from the recovery state. So, now the list of windows will be having all the windows that were there in the list at the time of checkpoint and the splitting unit is in the same state as it was at the time when checkpoint was taken. Additionally, it has the copy of the list of windows that was saved before recovery. And this list also contains the windows that were created by the splitting unit till the point of time when out of order event was received and recovery was initiated.

Now, the splitting unit starts to check the predicate for all the events having timestamps greater than the timestamp of recovery state. For every event, first it checks the predicate, if new window needs to be started from this event and compares the result with the windows in the copy of the list that was saved before recovery. Then, for the whole list (not the copy of list) of windows, it checks if any of the window needs to be closed at this event. These results are again compared with the saved copy of the list of windows. During these comparisons, different situations can occur, so we have to respond to every situation and inform the operator instances via messages to adapt to the changes in windows. The response of the splitting unit in different situations is discussed here one by one.

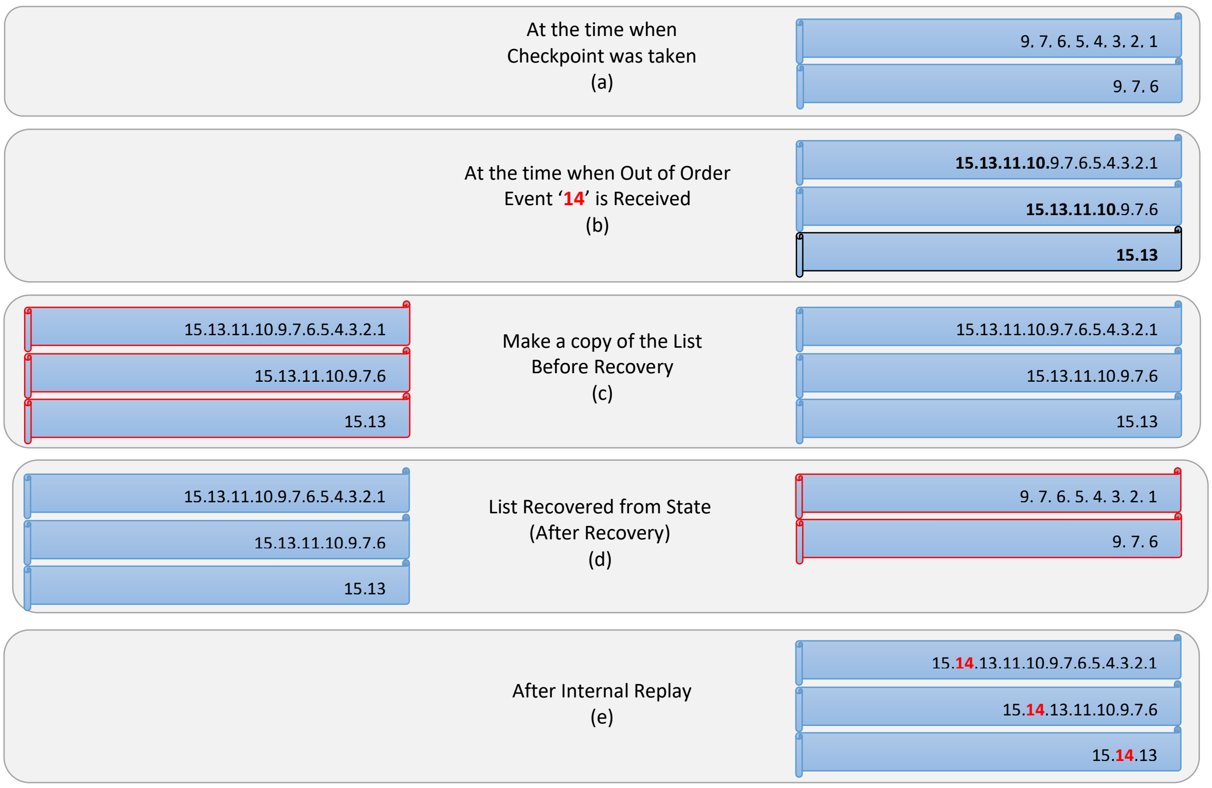

Figure 6-1 Example of Recovery due to an Out of Order Event in Splitting Unit

An example of handling an out of order event is shown in Figure 6-1. At the top, it shows the list of windows in the splitting unit at the time when checkpoint was taken. It includes 2 windows at that time and the events till event '9' are also present in the windows. While taking the checkpoint, the complete list of windows in the splitting unit will be cloned into the state, so that it can be recovered.

When an out of order event '14' is received by the splitting unit and it has already forwarded event '15' to the instances, then it has to do recovery before processing this event. The list of windows at this point of time is shown in Figure 6-1-b. Before starting recovery of the splitting unit, we need to take a copy of this list to compare the new windows with these windows. This step is shown in Figure 6-1-c with the copied list shown in red color.

Recovery is done in the next step and the list of windows is recovered back from the recovery state which was shown in Figure 6-1-a. So, now we have a copy of list that has all the windows that were there before recovery and we have a list that is recovered back to the time of taking checkpoint as shown in Figure 6-1-d. From here, internal replay will be started and all the events will be checked for the predicate and the results will be compared with the copy of the list. The changes in the windows will be reported to the operator instance via messages as explained in the next sub-section. The copy of the list of windows is deleted once the internal recovery process is completed. Figure 6-1-e shows the list of windows after the completion of internal recovery process.

6.1.1 Situations for Window Open

There can be 4 different cases based on the current reply from predicate and the availability of the respective window in the copy of the list of windows. Here, the window found means that there is a window available in the copy of the list of windows, such that the starting event timestamp of the window is same as the timestamp of this event.

- True from Predicate and Window Found
  If the window is found with same starting timestamp, then we just ignore it because the event was handled correctly previously. So the instance doesn't need to do anything with the start point of this window and continue processing this window as it is. To keep process smooth, the window is added to current list of windows maintained by the splitting unit.

- True from Predicate but Window Not Found
  If the reply from the predicate is true but we cannot find the window then we need to start the window as normal and start sending all the events to this window. No command needs to be sent to operator instances in this case also. Operator instance to which this window will be assigned will handle this window as a normal case.

- False from Predicate but Window Found
  If the reply from the predicate is false but we have found a window in the copy of the list of windows. Then we need to send a message to the operator instance to whom this window was allocated. The message will contain a key "DEL" as a command and starting time of the window to identify which window needs to be deleted.

- False from Predicate and Window Not Found
  If the reply from the predicate is false and there is no window in the list found, that means we do not need to do anything for this case. There will be no message sent to the instance also.

6.1.2 Situations for Window Close

For current event, predicate is checked for all the open windows, if any one of them needs to be closed at this event. We will discuss different cases for a specific window being checked for this event. There are 4 such cases for window close based on the reply from the predicate and the availability of window of concern in the copy of the list. Here, window found means that there is a window in the copy of the list of windows that has the same end time as the timestamp of current event.

- True from Predicate and Window Found
  If the reply from predicate is true and the window is also found, then theoretically neither we need to do anything in the splitting unit nor we need to send any message to the operator instance. However, if we send the close window event again, it will not have any effect.

- True from Predicate but Window Not Found
  If the reply from the predicate is true but we are not able to find any such window in the list, then we need to send a pseudo-event to close the window. This will inform the operator instance that the window needs to be closed at this event timestamp.

- False from Predicate but Window Found
  If the reply from the predicate is false but the window is found, then we need to send a message to the operator instance that this is not the end of the window. Operator Instance will mark this window as open, so that further events can be added to this window and processed further. The message will contain a key "NOTCLOSE" as command and the start timestamp to identify the window in operator instance.

- False from Predicate and Window Not Found
  If the reply from the predicate is false and the window is not found, then we do not need to do anything. There will no message sent to the instance in this case.

6.1.3 Optimization

An optimization is possible in the window open during internal replay. If the predicate for window open is true but there is no window found having the matching start time. Then actually we need to start new window but instead if the event was not the last event in the events to be replayed, then save this event and move to the next event to check the predicate. If predicate is false for this next event but the window is found then instead of deleting this window, send a message to the operator instance to add the previous event to this selection and update the window.

An example of such situation is shown in Figure 6-2. In the same example as discussed in the previous sub-section. If the out of order event received is having a timestamp of '12', then the initial steps will be the same to copy the list before recovery and then recover the splitting unit. Figure 6-2-a shows the recovered list of windows and the copy saved before recovery. When the out of order event '12' is checked for the predicate, we get a request to start new window, but as an optimization, we don't start the window right away if this event was not the last event in the replay. We save the request, check the closing predicate for this event and then move to the next event as shown in Figure 6-2-b.

If the next event has the situation in which it has to delete a window, we don't delete it but add the previous event to it to avoid resending all the events to the window as shown in the Figure 6-2-c. As, the start event of the window is updated with action, so we also have to update the meta-data of the window and move it to the current list of windows. Figure 6-2-d shows the list of windows in the splitting unit after completion of the internal recovery process.

This is not just the case with out of order events as shown in the example. Even if the out of order event received in this example was having timestamp '8'. We will get into such a situation for event '11'. So, we can add event '11' to the window just like we have added event '12' in the figure shown below.

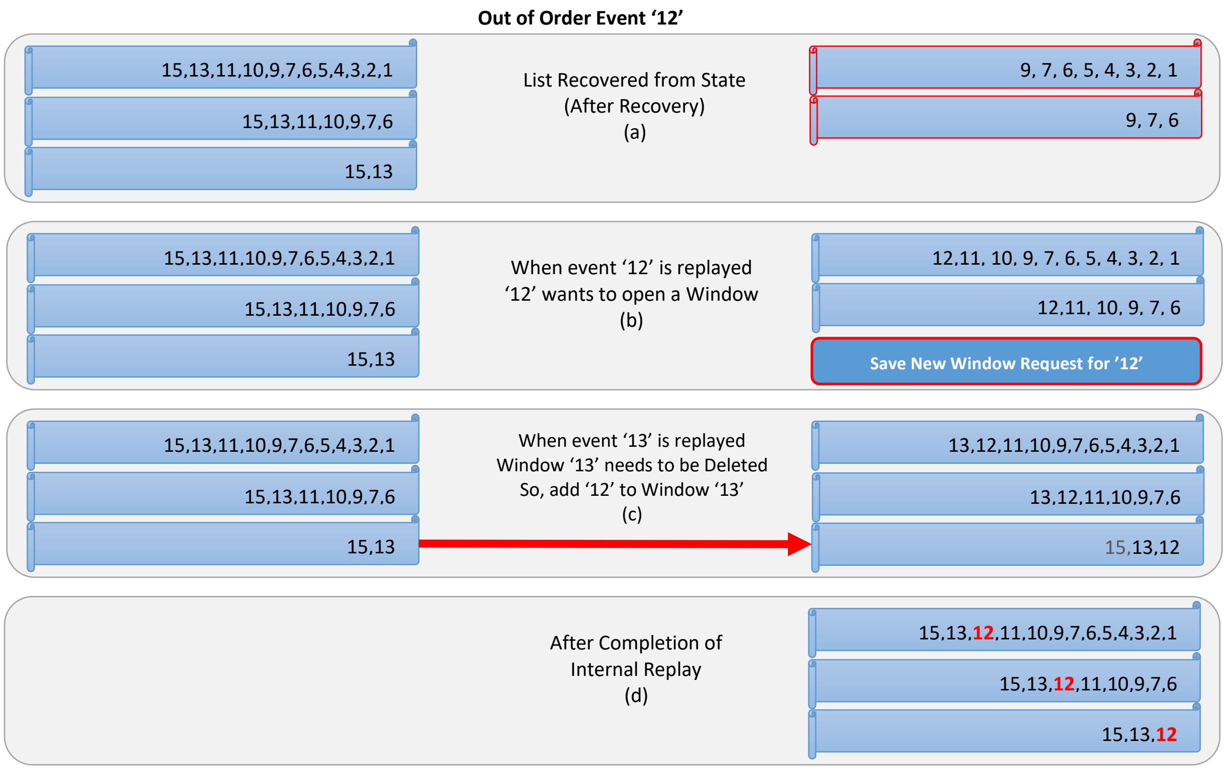
Out of Order Event '12'
15,13,11,10,9,7,6,5,4,3,2,1
15,13,11,10,9,7,6
15,13
List Recovered from State (After Recovery) (a)
9, 7, 6, 5, 4, 3, 2, 1
9, 7, 6
When event '12' is replayed '12' wants to open a Window (b)
12,11, 10, 9, 7, 6, 5, 4, 3, 2, 1
12,11, 10, 9, 7, 6
Save New Window Request for '12'
When event '13' is replayed Window '13' needs to be Deleted So, add '12' to Window '13' (c)
13,12,11,10,9,7,6,5,4,3,2,1
13,12,11,10,9,7,6
15,13,12
After Completion of Internal Replay (d)
15,13,12,11,10,9,7,6,5,4,3,2,1
15,13,12,11,10,9,7,6
15,13,12

Figure 6-2 Optimization of Internal Replay Technique in Splitting Unit

In this way, the overhead to delete this window and to start a new window and send all the events to it again can be avoided. This can be helpful in many different predicates e.g. tuple based window predicate, time based window predicate etc. However, while checking the next event, if the necessary condition for this optimization mentioned above is not met, then new window will have to be started.

Algorithm.4. Pseudo-Code of Internal Replay of Events in Splitting Unit

```
1.  Data: sortedEventsMap, windowsList, windowsCopy, lastProcessedTimestamp lpts, predicate
2.  begin
3.          WHILE sortedEventsMap.higherKey(lpts) != null DO
4.                  event = sortedEventsMap.next()
5.                  IF checkpointRequired() = true THEN
6.                          takeCheckpoint()
7.                  ENDIF
8.                  IF predicate.start() = true THEN
9.                          IF eventForNewWindow != null   THEN
10.                                 startNewWindow()
11.                                 eventForNewWindow = null
12.                         ENDIF
13.                         windowFound ← findWindowWithStartEvent(event)
14.                         If windowFound = true THEN
15.                                 windowsList.add(window)
16.                         ELSEIF sortedEventsMap.higherKey(lpts) != null THEN
17.                                 eventForNewWindow = event
18.                         ELSE
19.                                 startNewWindow ()
20.                         ENDIF
21.                 ELSE
22.                         windowFound ← findWindowWithStartEvent(event)
23.                         IF windowFound = true THEN
24.                                 IF eventForNewWindow != null THEN
25.                                         sendEvent(eventForNewWindow)
26.                                         sendMessage(ADD, event, window)
27.                                         updateWindowMetaData()
28.                                         windowsList.add(window)
29.                                         eventForNewWindow = null
30.                                 ELSE
31.                                         sendMessage(DEL, window)
32.                                 ENDIF
33.                         ELSE
34.                                 IF eventForNewWindow != null THEN
35.                                         startNewWindow()
36.                                         eventForNewWindow = null
37.                                 ENDIF
38.                         ENDIF
39.                 ENDIF
40.                 WHILE Window w : windowsList DO
41.                         IF w.open = true THEN
```

```
                    IF predicate.close() == true THEN
                        closeWindow()
                    ELSE
                        windowFound ← findWindowWithEndEvent(event)
                        IF windowFound = true THEN
                            sendMessage(NOTCLOSE, window)
                        ENDIF
                    ENDIF
                ENDIF
                deleteDeterministicData()
            ENDWHILE
            broadcastMessage(END)
        ENDWHILE
end
```

6.2 Message Handling in Operator Instance

When an out of order event is received by the operator instance, it marks a flag high. When this flag is high, no further data is deleted to make sure that the operator instance has not deleted the data before getting messages to adapt to the changes. The messages received from the splitting unit are directly processed instead of keeping them in the queue of the events to make sure the changes proposed by the splitting unit can take effect as early as possible. The action to be taken on arrival of every message from the splitting unit is explained below.

After taking the required action according to the received message, it is saved in a list. Because there can be a case that a window was in the current windows list at the time of checkpoint. Then later on it was modified by a message. So after recovery, the unmodified window will be back in the current windows list. So, we have to save the messages and check if any of the windows in the current windows list needs to be modified again after recovery. Algorithm.5 shows the pseudo-code of message handling in the operator instance.

6.2.1 Add Event

When the add message is received from the splitting unit, it has timestamp of the window with it to identify the window of interest. In addition to that, it has the timestamp of the event to be added, so that the event can be identified and added to the said window. As, this event is supposed to be the first event of the window, so the information of the start of the window is also updated to the information of this event. Moreover, in our implementation, there is a list of start timestamps of current windows, so if the old timestamp of the window was there in the list, it is also updated with the new one. The timestamp of the next window to be added to the current windows is also updated if required. Before handling this message, mutex lock needs to be acquired, to make sure recovery or event processing is not being done parallel to this message handling.

If the operator instance has already processed events with higher timestamp than the timestamp of the event, then the operator instance will have to recover back to some older state and start processing again, so that it can process this window including this new out of order event as well.

The format of the message is as follows: “ADD,<Event Timestamp>,<Window Timestamp>”

Algorithm.5. Pseudo-Code of Message Handling Routine in Operator Instance

```
1.  Data: sortedEventsMap, windowsList, lastProcessedTimestamp, currentWindowsList,
2.  waitingPhase
3.  begin
4.          CASE command OF
5.                  ADD:
6.                          event ← identifyEvent()
7.                          window ← identifyWindow()
8.                          window.add(event)
9.                          updateWindowMetaData(window)
10.                         updateCurrentWindowsList()
11.                         IF event.ts ≤ lastProcessedTimestamp THEN
12.                                 recover()
13.                         ENDIF
14.                         BREAK
15.                 DEL:
16.                         window ← identifyWindow()
17.                         windowsList.remove(window)
18.                         window.complexEvents.clear()
19.                         BREAK
20.                 NOTCLOSE:
21.                         window ← identifyWindow()
22.                         IF window.endEvent.ts ≤ lastProcessedTimestamp THEN
23.                                 recover()
24.                         ENDIF
25.                         window.open ← true
26.                         addNewEventsToWindow()
27.                         BREAK
28.                 END:
29.                         waitingPhase ← false
30.                         BREAK
31.         ENDCASE
32. end
```

6.2.2 Delete Window

When the delete window message is received, the window is identified by the timestamp received along with the message and the window is deleted along with all the complex events that are generated by this window. Here, we don't need to recover back even the events with higher timestamps are processed because we will simply delete the complex events generated by processing the events in this window.

The format of the message is as follows: "DEL,<Window Timestamp>"

6.2.3 Window Not Closed

This is a message to inform the operator instance that previously the window was marked as “closed” by mistake, so it is required to be marked as open again, so that it can receive further events. A pseudo-event to close the window will be received later. If the operator instance has already processed events with higher timestamp than the time at which this window was closed previously, then the operator instance will have to recover back to some older state and start processing again, so that it can process all the events for this window also until the new close time is not received.

The format of the message is as follows: “NOTCLOSE,<Window Timestamp>”

6.2.4 End of Internal Replay

This message informs the operator instance that the internal replay in splitting unit has been completed, so the waiting phase of the operator instance is over now. Now the deterministic data can be deleted as normal. This message doesn’t contain any further information other than the command.

The format of the message is as follows: “END”

6.3 Algorithm

The complete algorithm is divided into 2 main parts i.e. first one relating to different situations that can occur while checking ‘window open predicate’ cases and the other part dealing with the different situations that can occur while checking ‘window close predicate’ for a specific window and current event.

Figure 6-3 shows the tabular form of the algorithm that deals with the window open predicate and respective actions to be taken by the splitting unit and the operator instances.

Situations		**Splitter Action**		**Operator Instance Action**
Window Start Required	**Window Found**			
True	**True**	Move window from old copy of list of windows to current list of windows.		No action required!
True	**False**	If this is the last event in replay, then start a new window.		Receive new window and start processing as usual. If events with higher timestamp have already been processed, then recover the operator instance.
True	**False**	If this is not the last event in replay, then save the request to start new window.	If next event had started wrong window, then add this event to that window. Update the window and move it to current list. Send “ADD” message to instance.	Receive “ADD” message and add the event to the said window. If window was already being processed, then recover the operator instance.
True	**False**	If this is not the last event in replay, then save the request to start new window.	If next event had not started wrong window, then start new window.	Receive new window and start processing as usual. If events with higher timestamp have already been processed, then recover the operator instance.
False	**True**	If there is no request to start new window, then send “Delete” message to the Operator Instance.		Receive “Delete” message and delete the window along with all the complex events generated by it.
False	**False**	No action required!		No action required!

Figure 6-3 Tabular form of the Internal Recovery technique of "Window Open Predicate"

Situation			
Window End Required	**Window Found**	**Splitter Action**	**Operator Instance Action**
True	**True**	Mark the window in current list as closed. However, it's not necessary to send the pseudo-event to the operator instance to close the window.	No action required!
True	**False**	Send the pseudo-event to the operator instance to close the window.	Receive pseudo-event from the splitter and mark the specified window as closed. Remove if any extra events are there in the window. If events with higher timestamp have already been processed, then recover the operator instance.
False	**True**	Send "Not Closed" message to the operator instance to let it know that the window is still open, so keep adding and processing events for it also.	Receive "Not close" message and mark the specified window as open. Add events to it if there are any new events received with higher timestamp than the old closing time. If events with higher timestamp than the old closing time of this window have already been processed, then recover the operator instance.
False	**False**	No action required!	No action required!

Figure 6-4 Tabular form of the Internal Recovery technique of "Window Close Predicate"

Figure 6-4 shows the tabular form of the algorithm that deals with the window close predicate and the respective actions to be taken by the splitting unit and the operator instances. In some situations, operator instances have to take multiple actions, so these actions are shown in bullet marks.

6.4 Advantages

Internal replay in the splitting unit has two main advantages that are vital for throughput and latency of complex event processing system. The advantages are explained below in detail.

6.4.1 No need to send Events Again

Main idea of internal replay is to avoid sending the events to the instances again because they have already received events before the recovery of the splitting unit. So, it usually it is not necessary to send the events again. In most of the cases, just the start and end of the windows need to be updated during replay, which can be informed to the operator instances via messages instead of asking them to recover back to some state so that events can be replayed to them. In this way, communication overhead is avoided which can be more than the negligible overhead in most of the cases because the instances and splitting unit can be running on different machines, so the communication of events can take quite a while. Moreover, this overhead can be much of interest if there is a large data included in the event, for example a high quality picture or a video clip etc.

6.4.2 Save Memory in Instance Checkpoint

If there is no internal recovery, whenever the splitting unit needs to recover back, it will have to ask all the instances to recover back to the same time. Once all the instances have recovered back to that time, splitting unit will start to replay the events from that time. This means that the instances will receive the events again from the splitting unit during replay and also there is a possibility that the events received by the instance in replay doesn't include some of the events that were received by it previously. There can also be some changes in the windows that were assigned previously. As, it's not possible to guide the operator instance to amend the list of windows to match it to the new windows. So, it can cause serious problems which means that the complete container of the events and windows needs to be recovered back to the recovery state to avoid any problems. To do so, we need to clone the container containing events while taking a checkpoint and save it in the state.

An example is shown in Figure 6-5, in which before recovery, window 'A' and window 'C' were assigned to operator instance. 1 for processing. Only window 'B' was assigned to operator instance 2. If, windows lists of the operator instances were not recovered back to the previous state during recovery. So, operator instance. 1 will still have window 'C' in its list even after recovery. During replay, if new window 'C*' is assigned to operator instance. 2 instead of operator instance. 1. Then, the window 'C' in operator instance. 1 will be problematic if there is no way to communicate this operator instance that it needs to delete this window and not process it. This shows that if there is no method to pass some messages to the operator instance to guide it to modify the list of windows assigned to it, then it is necessary to clone the list of assigned windows of an operator instance in its state while taking a checkpoint and recover it back during recovery.

However, in our case, we do not send the events and windows again, instead we send messages to let operator instance amend the windows itself. In this way, the list is updated according to the instructions from the splitting unit via messages as explained in sub-section 6.2. As a result, we neither need to save the windows nor events in the state. This can save a lot of memory when checkpoints are taken frequently.

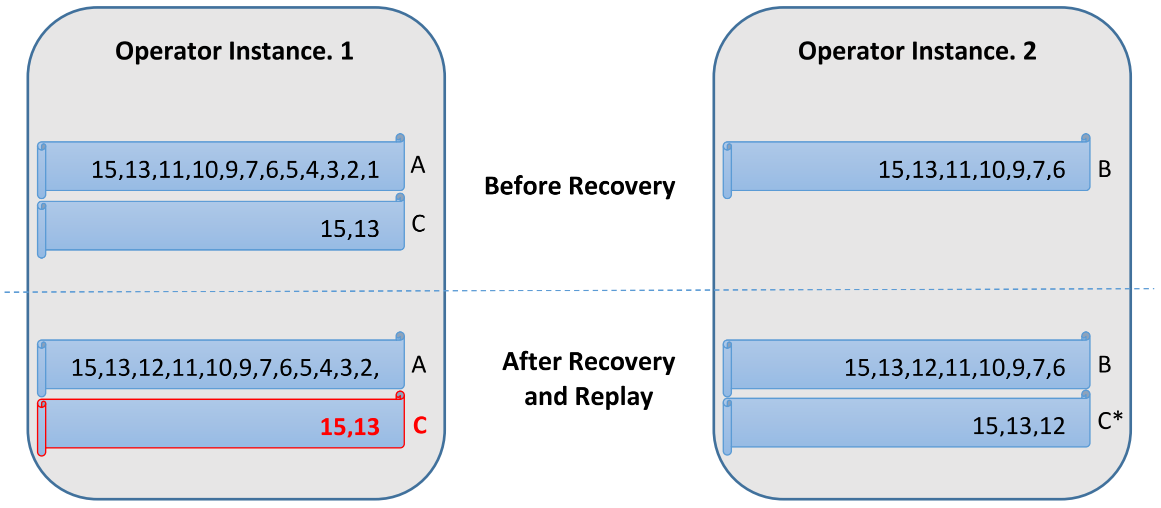

Figure 6-5 Example to show that Cloning of List of Windows is necessary without Internal Replay Technique

7 Evaluations

For the evaluation of the all the optimizations proposed in this document, we have implemented it in Java programming language. The framework acts as a middleware which can handle the out of order events also and process most of the out of order events in the correct order. The user just needs to define the function which should be performed on every event. All the remaining stuff to split the events in multiple windows based on the chosen predicate and allocating these windows to the operator instances based on the specific scheduling technique and to take checkpoints based on the checkpoint model and also to recover the system when required is all taken care by our framework. However, if the user wants to use his own splitting predicate, scheduling technique or checkpoint model, he can simply implement the logic and the framework will adapt to it. So, the framework is very much generic for parallel CEP systems.

In this section, we will first explain the conditions used for all the evaluations and then we will discuss the results. In sub-section 7.1, data generator used to generate the synthetic data is explained. Splitting predicate, scheduling technique and some specific conditions common to all evaluations are mentioned in sub-sections 7.2, 7.3 and 7.4. The specifications of server cluster used to run the framework for evaluations is discussed in sub-section 7.5. The results are shown in sub-section 7.6 and the overall performance is summarized in sub-section 7.7.

7.1 Data Generator

For the evaluations of the framework, synthetic data is used. To generate this data, we have developed a program that takes multiple inputs and generates the data randomly based on these inputs. The random data generated by this program is based on these configurable factors:

- Total Number of events
  It is the total number of events to be generated altogether. These events will be divided in the different files based on the number of sources. However, it is not guaranteed that all the source files will be having same number of events because the events are added to the files randomly.
- Number of Sources
  The number of sources defines the total number of source files to be generated. All the events of a specific source file will be having the same data, so that it can be detected by the operator instance to generate complex events when events are received in a specific sequence.
- Minimum interval between two events
  The minimum interval between two consecutive events defines the overall frequency of the events. This value is in nanoseconds, so this program can generate data having 1×10^9 events per second.

The program loops until total number of events are generated. Every time in the loop, an event is generated by incrementing the timestamp with minimum interval between 2 events. Then a random number will be generated which is limited by the number of sources. So, if the number of sources is defined to be 3, it will not generate the random variable more than this. Based on this random variable, the generated event is allocated with a source id. Sequence number is maintained for every source, which is added to the meta-data of this event based on the source id. And the data is also added to the event before writing it to the source file. All the events in a source file will have the same data. Moreover, it is guaranteed that all the events in all files will be having unique timestamps.

7.2 Events Source

A source is a program used to read the source file (generated by the data generator) and pass the events read from the file to our framework. A source mimics to be a sensor which generates events to report the environment conditions in some specific condition. As, it is quite possible in real life cases that the events from a sensor are delayed for some time due to network problems or some other factors. So, our source is also able to mimic the delays based on some command line arguments passed to it.

In our implementation, the source reads all the events from the file and save it to a sorted hash map such with the timestamp of the event as a key. We don't read events from the file every time because the frequency of the vents is very high, so file reading can become the bottleneck, which can affect the throughput of the whole system. After adding all the events to the hash map, events are played to the CEP system according to the timestamp. *System.nanotime()* java function is used to get high precision clock. Before starting to play the events, the current time of the system is saved. Then whenever Equation 7.1 is satisfied for the event, it is passed to the CEP system.

$$System.nanoTime(\,) - startTime > timestamp \qquad \text{Equation 7.1}$$

7.2.1 Generation of Out Of Order Events

As our framework is capable of handling out of order events, so we have developed our sources such that, we can have synthetic delays in the events forwarded by a specific source. When dealing with synthetic data instead of real-life data, we pass some arguments to the source to define the delay delta, the time to start delay and the duration to keep delaying events.

For example, if we define delay delta to be 3 milliseconds and time to start delay is defined to be 20 milliseconds and the duration is defined to be 50 milliseconds, then all the events having timestamps between 20 milliseconds and 70 milliseconds are delayed by 3 milliseconds.

The corner cases are handled especially when the delay is going to end, making sure that no events are forwarded to the system violating the assumption of FIFO source. Let's extend our example to explain that how do we remain consistent with the assumption. With all the arguments passed to the source as shown in the previous paragraph, if there is an event with timestamp 69 milliseconds, so it is supposed to be delayed by 3 milliseconds. Hence, it will be forwarded to the system at 72 milliseconds. However, if there is another event with timestamp 71 milliseconds, it is not supposed to be delayed, so that means it can be forwarded to the system at exact 71 milliseconds but if we forward this event at 71 and then forward the delayed event at 72, it will be against the assumption and the source will no longer be FIFO. So, to make sure all the events are forwarded in FIFO pattern, the events like the one with timestamp 71 will also be delayed by at least some time (in this case 1 millisecond), so that event with timestamp 69 is forwarded before this event.

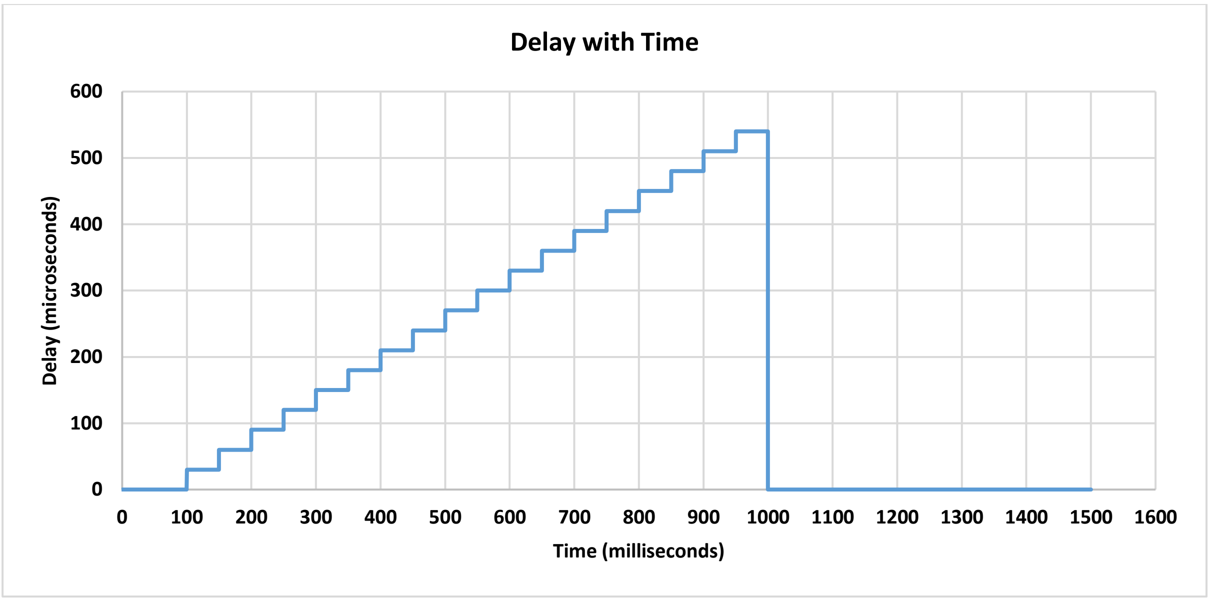

Figure 7-1 Graph showing the change in delay value with Time

In order to include the delay factor, while reading and adding the events to the sorted hash map, delay factor is added to the timestamp and saved as "hashKey". All the events are saved to the hash map using this value as a key and forwarded to the CEP system when the Equation 7.2 is satisfied. The delay value is based on the command line arguments. It receives a starting time and an ending time during which the events from a specified source will be delayed. For the remaining time, delay value will always be zero. The delay value during this time will be started form zero and incremented after a specified time unit by a specified time in nanoseconds. At the ending time of the delay, the delay value will be again zero.

$$System.nanoTime() - startTime > timestamp + delay \qquad \text{Equation 7.2}$$

Figure 7-1 shows the change in the delay value throughout our evaluations unless stated otherwise. This delay is applied only on the specified source whenever we say that the out of order events are included. If there is mentioned that there are no out of order events, then the delay remains zero for all the sources.

7.2.2 High Precision Clock

When the frequency of events is higher than 1000 events per second, then the difference between 2 consecutive events is surely be less than a millisecond, so we need a high precision clock, so that we can compare the timestamp of the events with higher precision clock to make sure events are sent on time as mentioned by their timestamp. In order to achieve this in our implementation, we have used a clock with nanoseconds precision clock to determine the correct time to forward the next event to the operator.

It is difficult to make sure that all the sources have the same *"startTime"* when different programs acts as different sources. And having a minor difference between the "startTime" of different sources can lead to unwanted out of order events because we are passing the events to the CEP system at very high frequency. So, we modified our source to read all the different source files and delay just the specified source. It will still read the source files into hash map as discussed before and forward them to CEP system.

7.3 Splitting Unit

Splitting unit is the first component of an operator which receives the events form the source and splits them into multiple windows and assign the windows to the available operator instances. We have already discussed the implementation of the splitting unit in sub-section 3.3, so here we will just introduce the specific splitting predicate and the scheduling technique used for our evaluations.

For the evaluations we have used tuple based predicate in which a window is started after a fixed number of events known as “window slide”. An every window has a fixed number of events in it known as “window size”. As our framework cannot depend on the serial numbers of the events, so we have implemented the tuple based predicate that works on the timestamps of the events. In our implementation, timestamps are the identity of an event as well as windows. So, data generator takes care of this thing that the timestamps of the events must be unique.

Scheduling technique used in our evaluations is round robin scheduling. In this technique, every window is assigned to the next operator instance, once all the operator instances are assigned a window, first one receives the next window and it goes on like this. A pointer is used in this technique that points to the next instance to which next window should be assigned. In order to let this scheduler work correctly, the pointer needs to be saved in the checkpoint and recovered back during recovery process.

7.4 Operator Instance

Operator Instance is the one which is meant to be implemented by the user, so that our framework can use the function implemented by the user and process the events in parallel. For evaluations, we have implemented the sequence operator which looks for a specific sequence in the received window. The state diagram of such an operator is shown in Figure 7-2. The number of complex events generated in our case depends on the window size used. So, we have fixed the window size to make the comparison easy.

As, the task of looking for a sequence is not much time consuming for the system, so we have simulated the synthetic load to get stable results. Busy looping is used to simulate the load. The time of busy looping depends on the command line argument (in nanoseconds). The complex events generated by the operator instance are just passed to the next level in our implementation. However, it can be extended such that the complex events are passed to the merger and then merger sorts them out and then pass to the higher hierarchies.

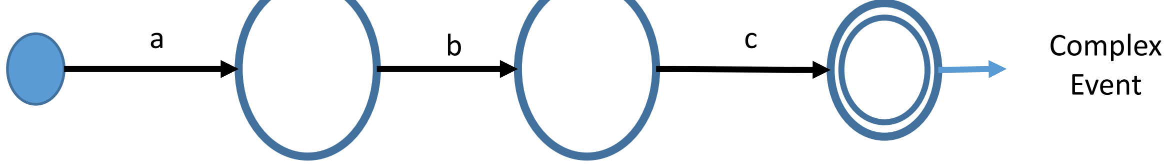

Figure 7-2 State Diagram of the Sequence Operator in Our Implementation

7.5 System Details

The evaluations of our framework are done on the server cluster named as "curium" which belongs to the Institute of Parallel and Distributed Systems (IPVS) in University of Stuttgart, Germany. The cluster is running Rocks 6.0 (Mamba) operating system based on CentOS release 6.2. It is based on NUMA architecture with 8 CPUs. There are 2 CPU sockets and 4 cores per socket in every CPU. However, every core is capable of running a single hardware thread. The clocking frequency of the CPU is 3000.278 MHz as reported by the system information. Cache memory of the system is as follows:

• Lld Cache:	32Kbytes
• Lli Cache:	32Kbytes
• L2 Cache:	6144Kbytes

7.6 Results

Here, we will discuss the comparison of throughput and latency of our implementation with that of the implementation that strictly follows the idea of [MP13b]. In this comparison, we will discuss different cases and compare the throughput and latency in all cases. After showing the comparison, we will focus on our implementation and show interesting results with some changes in the situations.

7.6.1 K-Slack Algorithm only (Baseline)

As, most of our work is to optimize the technique proposed by [MP13b] for parallel CEP systems. So, we will first compare our implementation with the implementation that strictly follows their approach with just K-slack algorithm working. In this case, the throughput as well as the latency of both frameworks perform almost the same. We have evaluated the systems for different number of operator instances i.e. 1, 2, 4, 8 and 16 operator instances.

The total number of events is 1 million with frequency of 100 thousand events per second. Moreover, the simulated load of 100 microseconds is used to get stable results. Window size used for this test is 10000 events with window slide of 2000 events. There are no out of order events as well as no speculation done by any of the systems in this case.

As shown in Figure 7-3, the throughput of both system also increases linearly with the increase of number of operator instances. The throughput increases because the windows can be processed by different operator instances in parallel, so the load is divided and the throughput is increased. Moreover, the graph shows that both systems are having almost the same throughput for all number of instances.

Similarly, Figure 7-4 shows that the latency of both systems is increasing linearly with the increase in the number of operator instances. This increase in the latency can be caused by the data structures used to sort the events or it can be due to the congestion of events to be passed between different CPUs. However, the important thing to note here is that the latency of both systems is comparable that shows that the basic implementation is same for both systems which doesn't effects the results discussed later in this section.

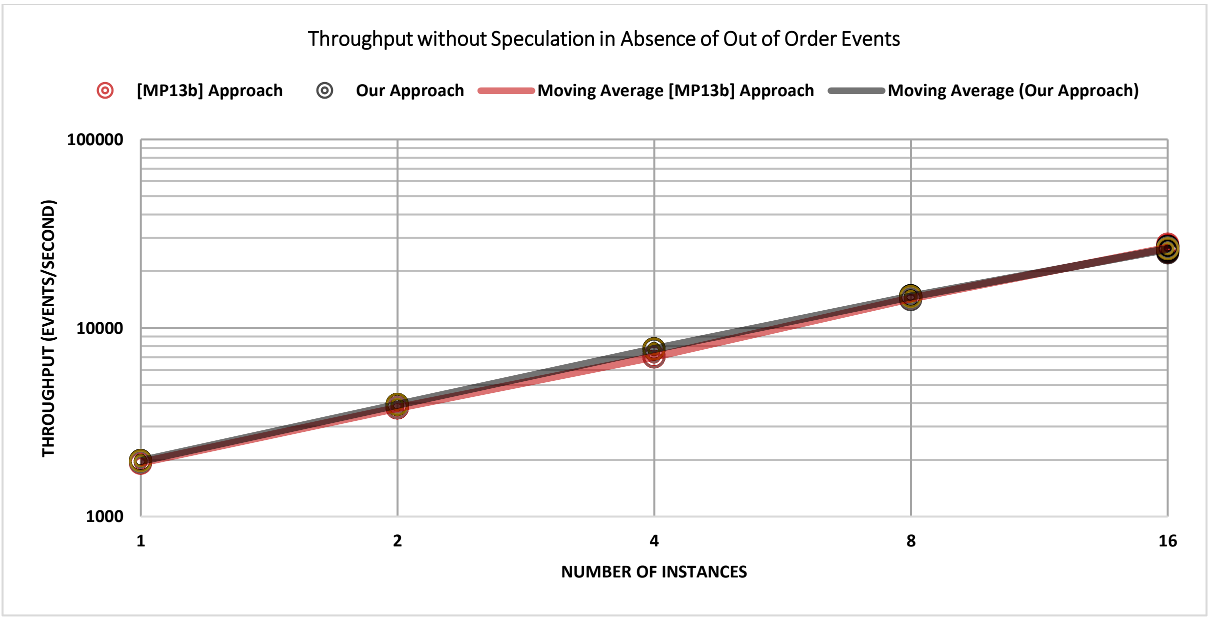

Figure 7-3 Throughput Comparison of Systems with K-Slack Buffering only

As mentioned before, the frequency of the events from the sources together is around 100 thousand events/second but both systems are not capable of processing that many events per seconds. So, we used it as it is for throughput comparison but after that we introduced a limiter in the source connectors of consistency manager. It limits the number of events forwarded to the consistency manager based on the throughput graph shown above. The latency measurement is started once the event has passed this check and forwarded to the consistency manager. In this way the waiting time of the events is included in the latency measurements. The latency measurement is stopped when the event is completely processed in all the windows and confirmation acknowledgements are received by the splitting unit from operator instances that the event is processed.

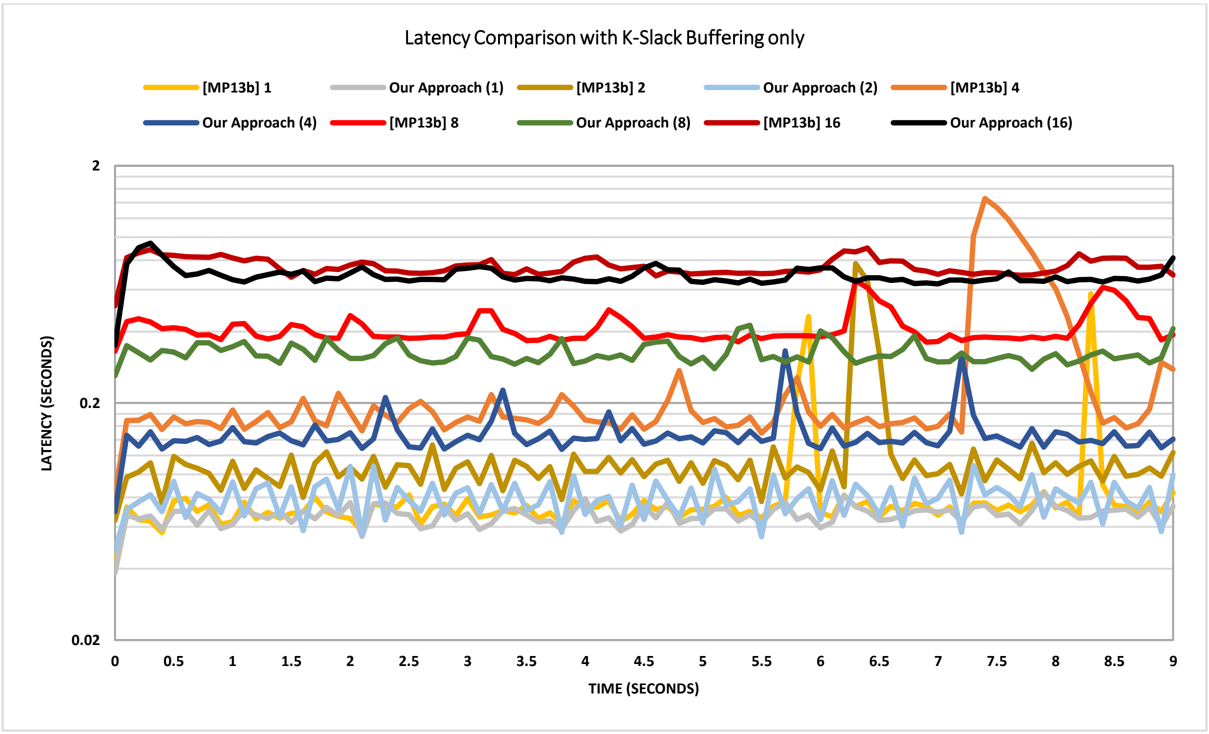

Figure 7-4 Latency Comparison of Systems with K-Slack Buffering only

7.6.2 K-Slack Algorithm with Speculation

In this test case, speculation is also activated along with the K-slack buffering in both systems. The basic speculation technique is same in both systems, our system takes checkpoints locally and in the implementation strictly following [MP13b], before forwarding the speculative event, it takes the checkpoint of the whole system as discussed before.

The test case for our implementation is having the same conditions i.e. 1 million events with the frequency of 100 thousand events per second. Moreover, the window size and the window slide are also the same. In this test, all the operator instances report their CPU load, so that the system can calculate the speculation factor at runtime. There are no out of order events in this case also. However, the throughput of the system following [MP13b] is badly affected by the speculation in the case of parallel CEP system as shown in figure. So, in order to do evaluations within time, the test case to evaluate this system is scaled down to 10 thousand events with the same simulated load but the window size of 100 events and window slide of 20 events. The frequency of the events is the same.

Figure 7-5 shows that the throughput of our implementation is almost the same as previous case but the throughput of the other implementation is affected very badly. The reason for this decrease in the throughput is mainly the speculation in that technique which is not suitable for parallel CEP systems. When, the system starts, the load on the operator instances is not much, so consistency manager tries to send the events speculatively. But before sending the event speculatively, it has to ask the splitting unit to send the state of the system. Splitting unit will further ask the operator instances to send their state. Once the state is received from all the operator instances, the event will be sent for processing. However, during all this time the operator instance was still not fully loaded, so speculation will be further increased but before sending the next event, checkpoint is necessary. The operator instance cannot take a checkpoint before finishing the processing of the current event, so the checkpoint request will be waiting. So, the efficiency of the consistency manager and the splitting unit is also affected. In short, the loops goes so that the consistency manager asks for checkpoint, instance takes checkpoint, then consistency manager sends the event and operator instance processes it. And it is repeated continuously. Moreover, the throughput doesn’t even increase with the number of operator instances because the consistency manager has to wait for the operator instance to completely process the event and then it will send back the state, so during that time, consistency manager cannot assign the event to the other instance also.

For the latency calculations in this case, number of events passed from the source connector to the consistency manager are limited according to the throughput shown in Figure 7-5. So, the latency of both systems is independent of the number of events waiting in the queue of source connector because the latency measurement is started after the events are passed from this check.

Figure 7-6 shows that the latency of our approach is a little bit decreased from the previous case but it is not much different from that one because in absence of out of order events, K-value doesn’t increase much, so there is not enough room for speculation to decrease the latency even if the system load allows to forward events speculatively. The latency of the other implementation is effected by the speculation in this case because the event has to wait for some time before the checkpoint is taken by all the operator instances and reported back to the consistency manager. Moreover, the Figure 7-6 also shows that the other implementation gives almost the same latency for all number of operator instances. This can be because the events are literally processed one by one even if there are more than one operator instances available. To show the comparison, the latency of the other implementation is expanded over the whole length of the graph.

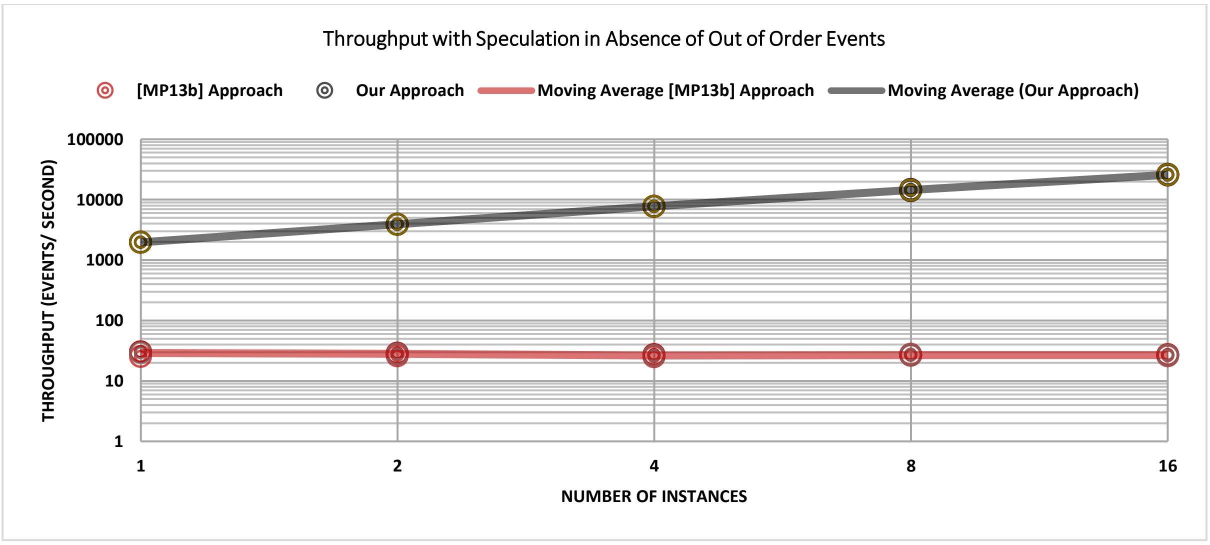

Figure 7-5 Throughput Comparison of Systems with K-Slack Buffering and Speculation

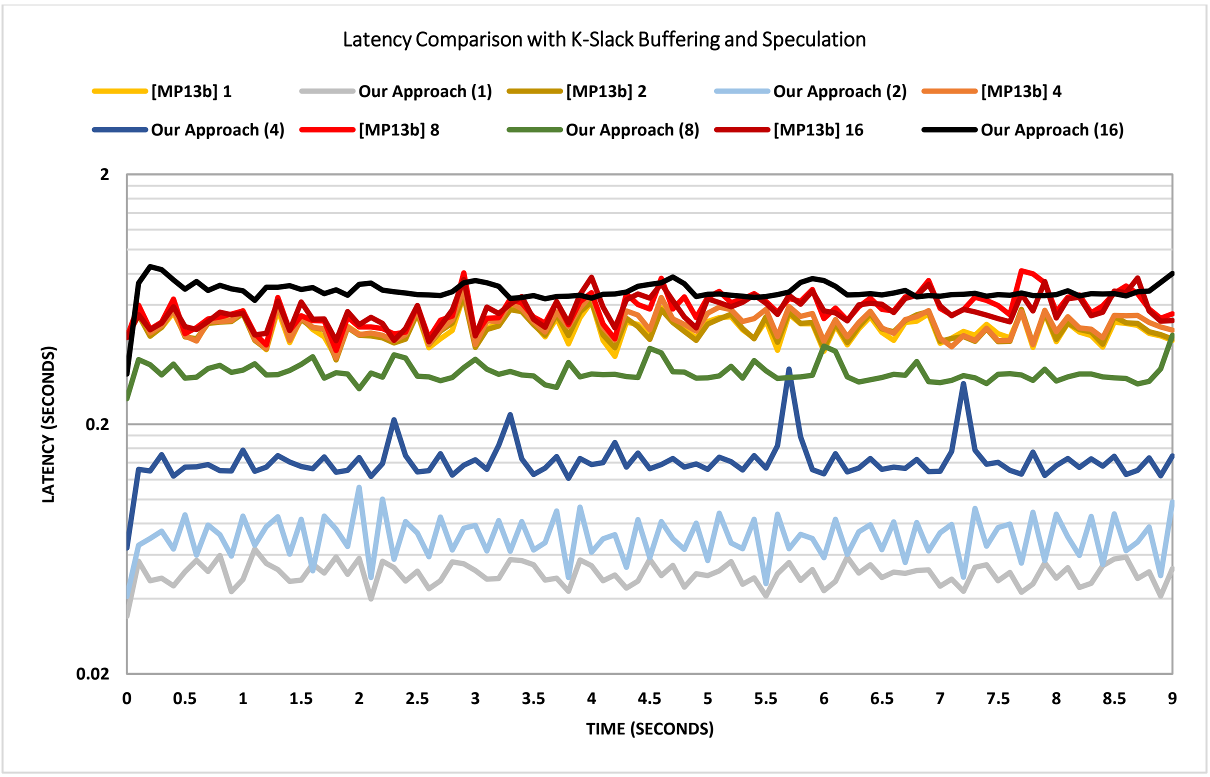

Figure 7-6 Latency Comparison of Systems with K-Slack Buffering and Speculation

7.6.3 K-Slack in Presence of Out of Order Events

This case is the modified version of the one explained in sub-section 7.6.1. It just includes the out of order events. All the remaining parameters are exactly the same. There is no speculation done by any of the systems in this case also. The throughput in this case is just similar to the one shown in Figure 7-3. So, we have omitted the specific figure for it. However, there is a noticeable change in the latency of the events. As shown in Figure 7-7, the latency is increased for both systems because the K-value is increased due to the arrival of out of order events. And after that, the K-value remains constant which increases the latency for all the events received even if there are no more out of order events.

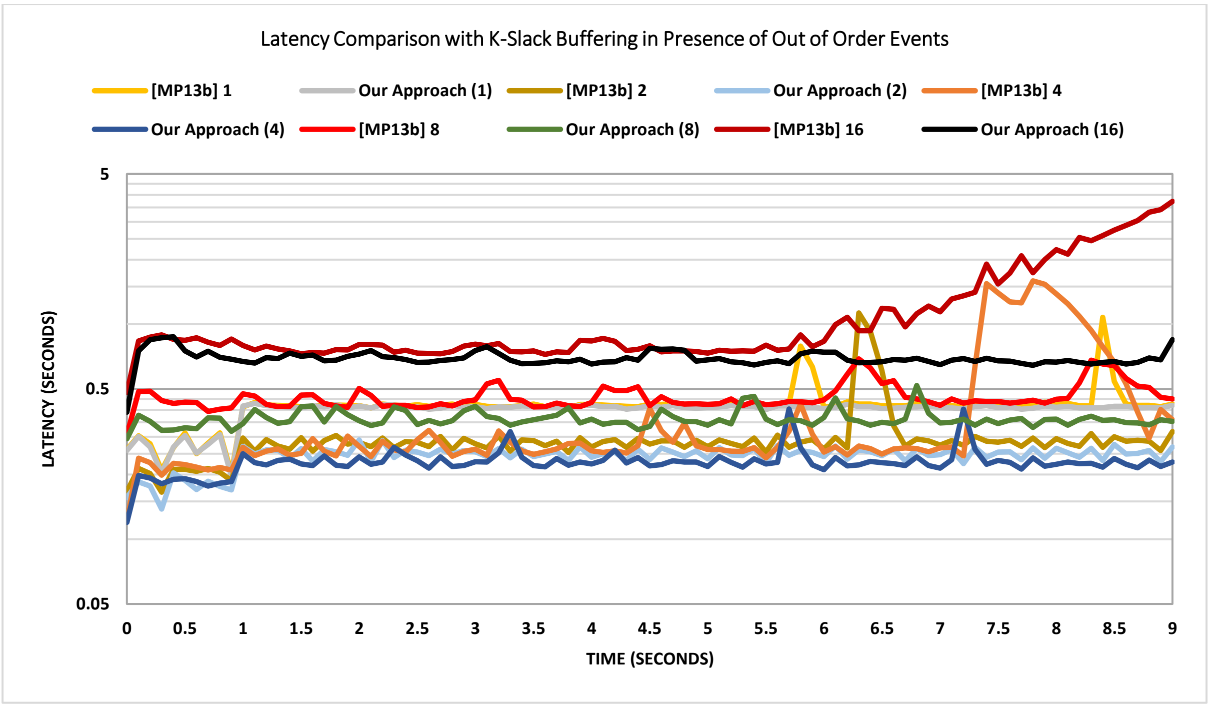

Figure 7-7 Latency Comparison of Systems with K-Slack Buffering in Presence of Out of Order Events.

Figure 7-7 shows that the increase in the latency is more visible for lower number of operator instances because their latencies were less previously, so the change in K-value has higher effect on their latencies. For the higher number of operator instances, the effect is not much visible. However, if the K-value rises further, it can have noticeable effect on the latency of higher number of operator instances also.

7.6.4 Latency Comparison with Different Speculation Factor Values

Here, we just focus on our implementation and show the effect of speculation factor on the latency of the events in presence of out of order events. The conditions are still the same with 1 million events having frequency of 100 thousand events per second. Window size is 10000 events and window slide is 2000 events. The checkpoint is taken on every 50th event. The simulated load is 100 microseconds and the delay is applied to source 'C' between timestamps 100 milliseconds and 1 second. The value of delay is incremented by 30 microseconds after every 50 milliseconds.

Here we have shown a figure for each number of instances that we evaluated. In every figure, we show the latency when there was just K-slack algorithm but no speculation or out of order events. This is almost the minimum latency we can get. Then we have different cases with different speculation factor. The speculation is shown in percentage. If the speculation factor is 0.8, speculation is supposed to be 20%, 0.6 having 40% speculation, 0.4 having 60% and so on such that 0 speculation factor lead to 100% speculation.

Figure 7-8 shows that there is a peak initially when the out of order event are being received. It is probably because of the data structures used and also because of the recovery required to be done again and again and probably because there are a lot of events cluttered in the system. This peak increases with increasing percentage of speculation. The latency then decreases almost linearly with time when there are no more out of order events received. This figure is for the case when only a single instance is connected to the splitting unit.

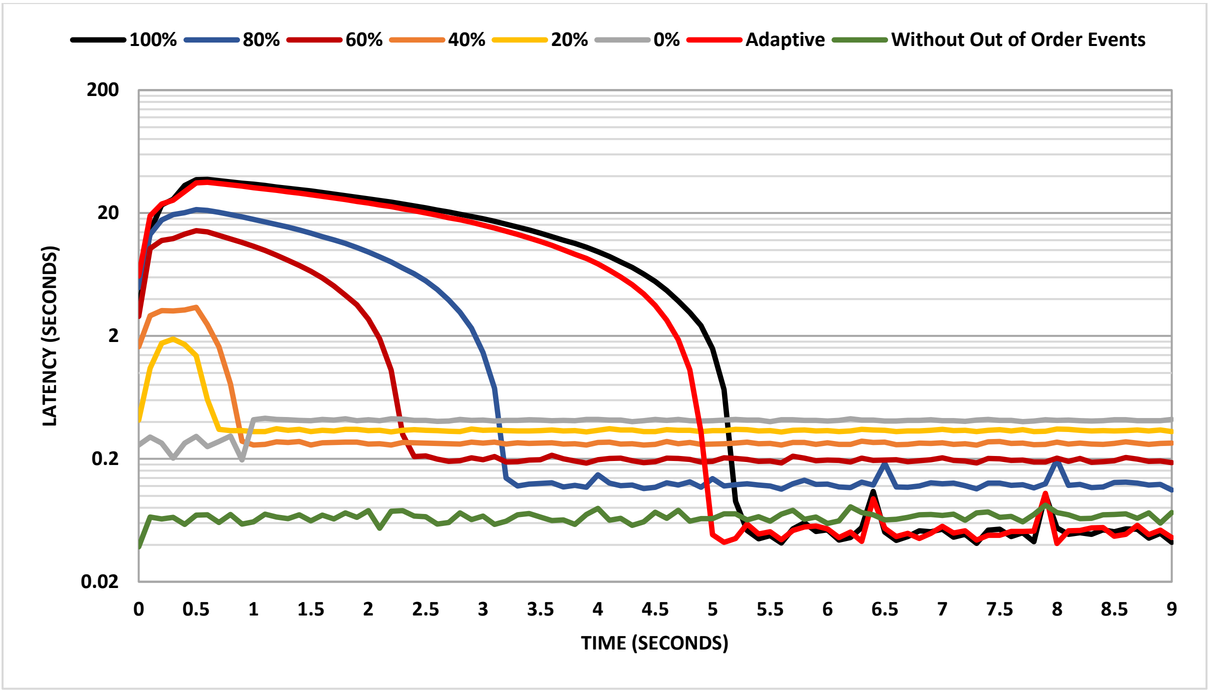

Figure 7-8 Latency Comparison of Our Implementation with different Speculation Factor (Single Operator Instance)

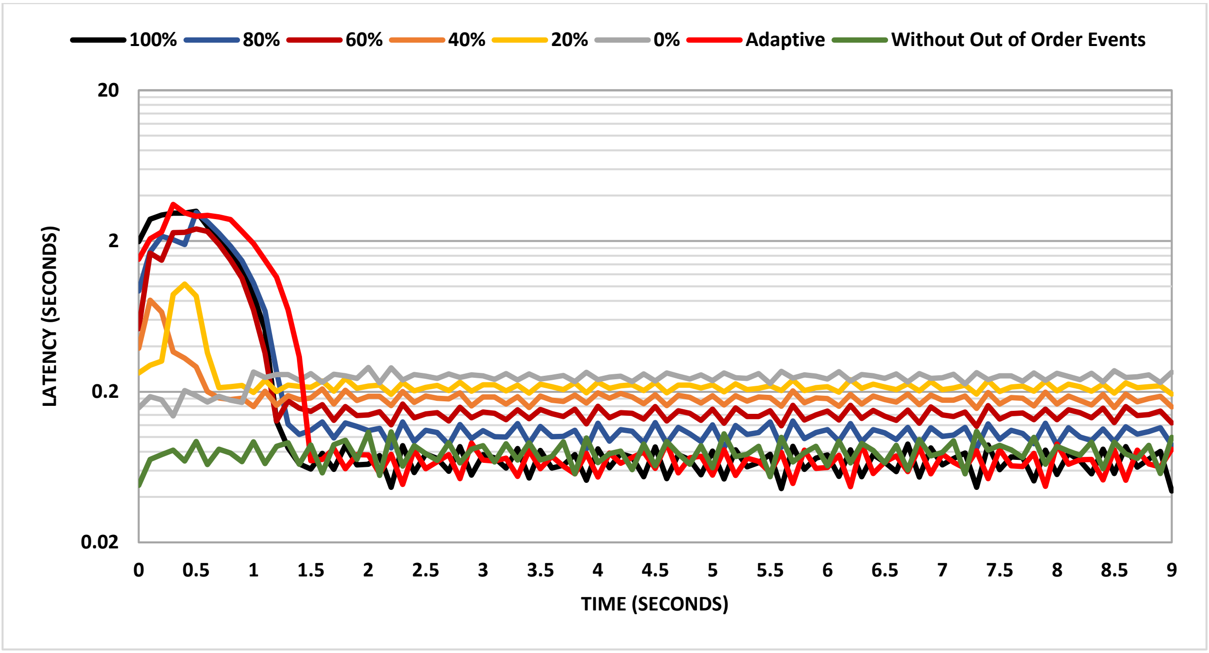

Figure 7-9 Latency Comparison of Our Implementation with different Speculation Factor (2 Operator Instances)

Once all the peaks are normalized, the speculation factor effects the latency as shown in the Figure 7-8. Latency becomes inversely proportional to the speculation percentage. We have also shown the automatically adaptive speculation results in the figure, which is very near to 100% speculation, so the latency is minimized.

Figure 7-9 shows the case when 2 operator instances are connected to the splitting unit. The results are a bit similar in trend as discussed before. But the effect of the speculation is decreased a little bit because there the latency was not affected that much due to out of order events.

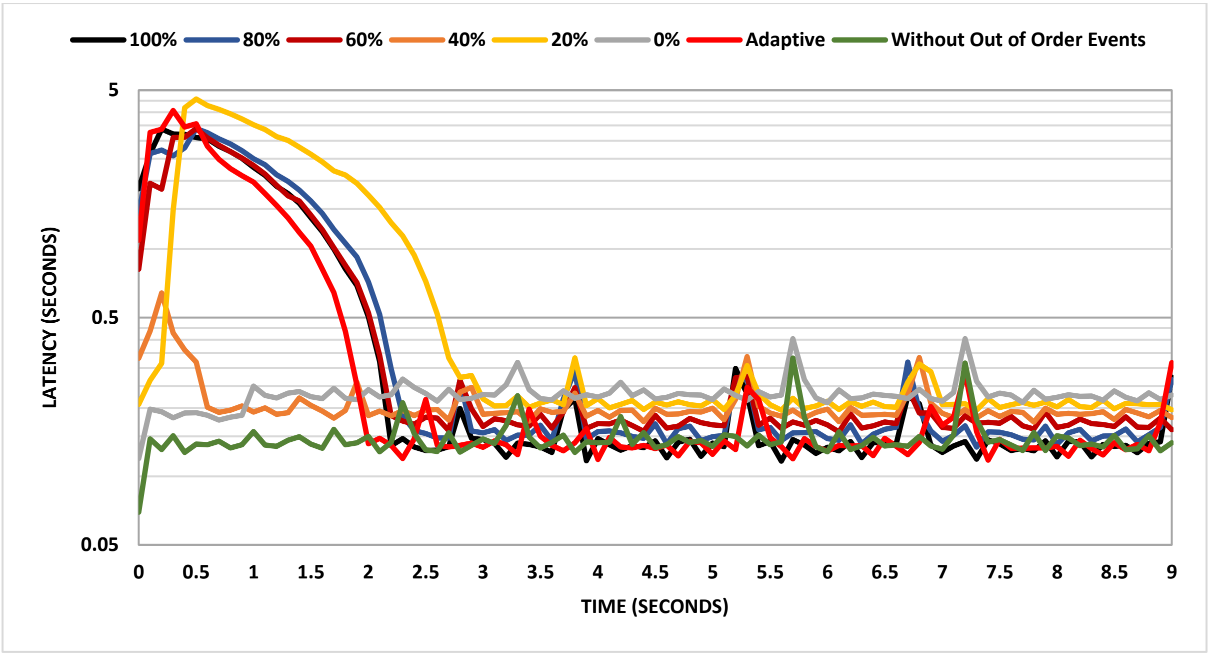

Figure 7-10 Latency Comparison of Our Implementation with different Speculation Factor (4 Operator Instances)

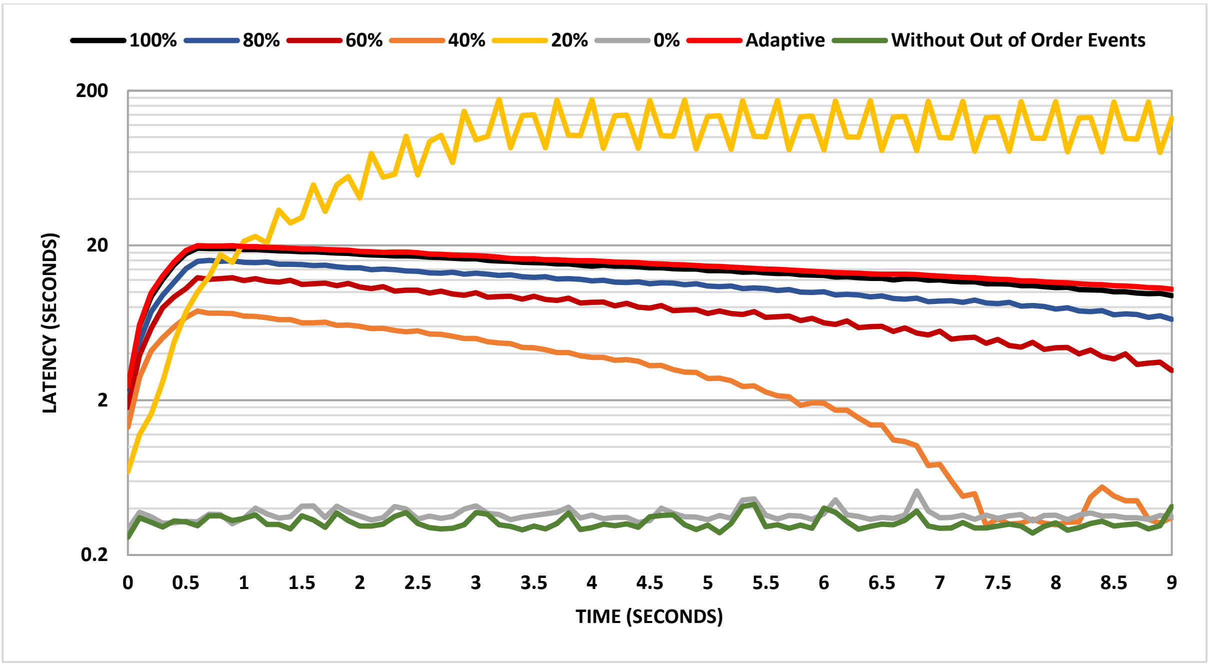

Figure 7-11 Latency Comparison of Our Implementation with different Speculation Factor (8 Operator Instances)

Figure 7-10 shows the case when there are 4 instances connected to the splitting unit. This case is also pretty much similar to the previous ones. But there is an exception of 20% speculation case which rises even much more than 100% speculation case. After the peak has once decreased to the normal stable value, the behavior is as expected. In the case of 8 operator instances (Figure 7-11), the behavior of all different lines looks as expected except for the 20% speculation line. The 20% line rises extremely high as compared to all other lines. This behavior is totally unexpected and we couldn't give a probable reason for it. Further evaluations can be done to find the reason behind it. The lines don't reach the stable value within our test case because they have risen too high but their decreasing trend is same.

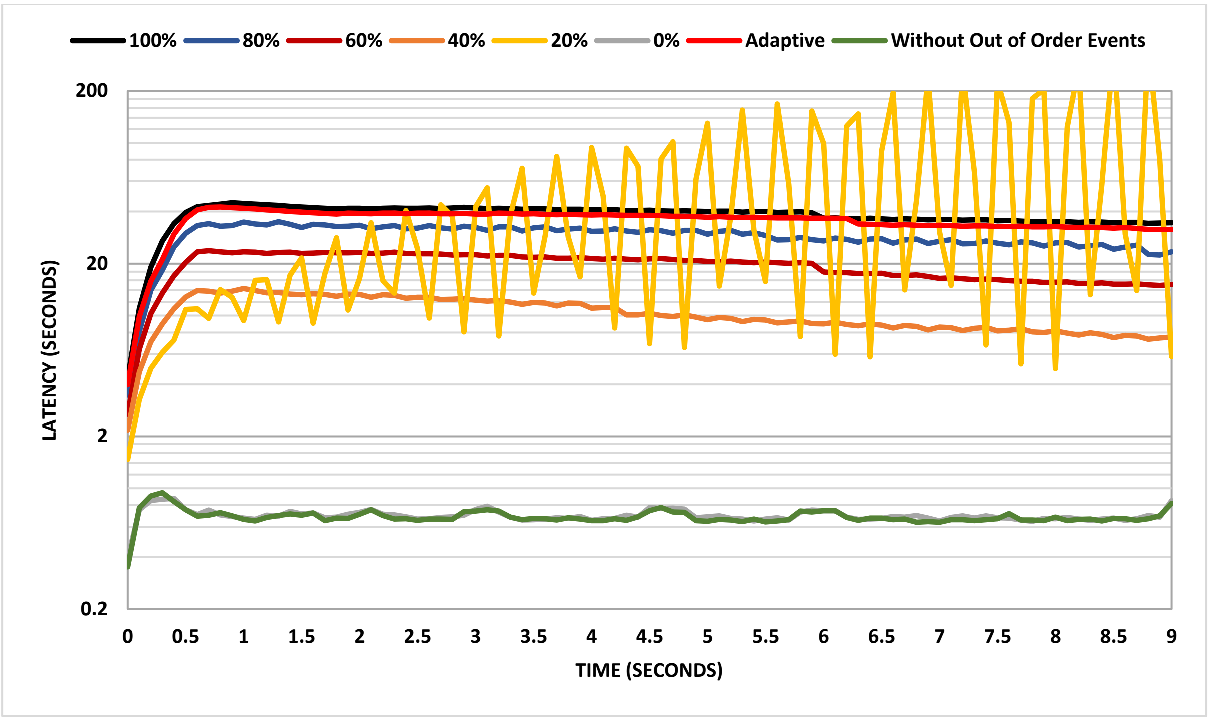

Figure 7-12 Latency Comparison of Our Implementation with different Speculation Factor (16 Operator Instances)

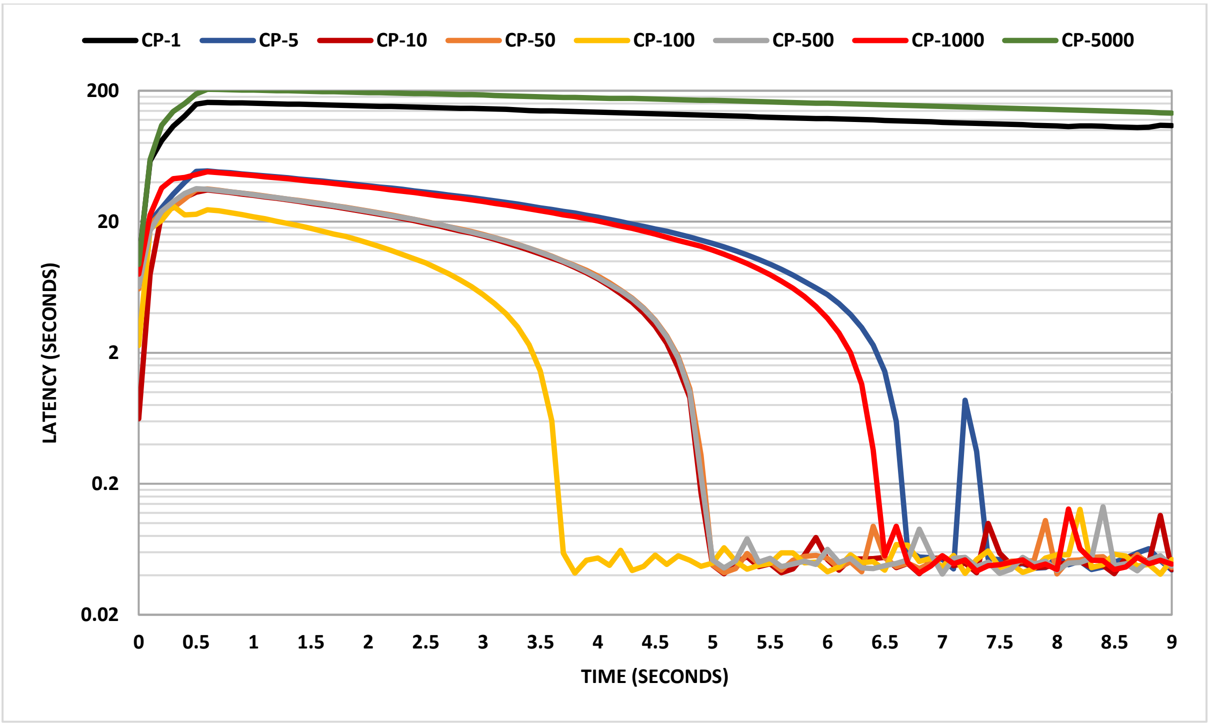

Figure 7-13 Latency in our Implementation for different checkpoints intervals with 1 Operator Instance

The unexpected behavior of 20% speculation line even increases in the Figure 7-12 shown for 16 operator instances. Also, the fluctuations have increased a lot in it. The value of this line is having the increasing trend throughout the graph, however, it is expected that it will reach its highest point and come down to the stable value then. All the remaining lines are acting as expected.

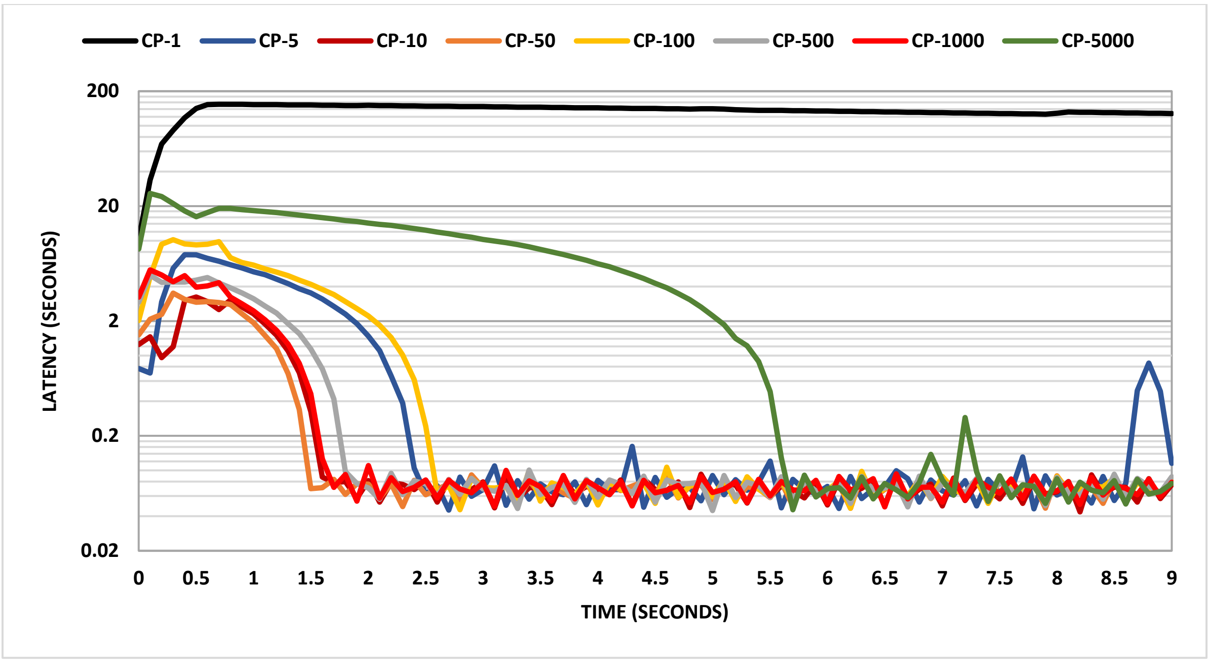

Figure 7-14 Latency in our Implementation for different checkpoints intervals with 2 Operator Instances

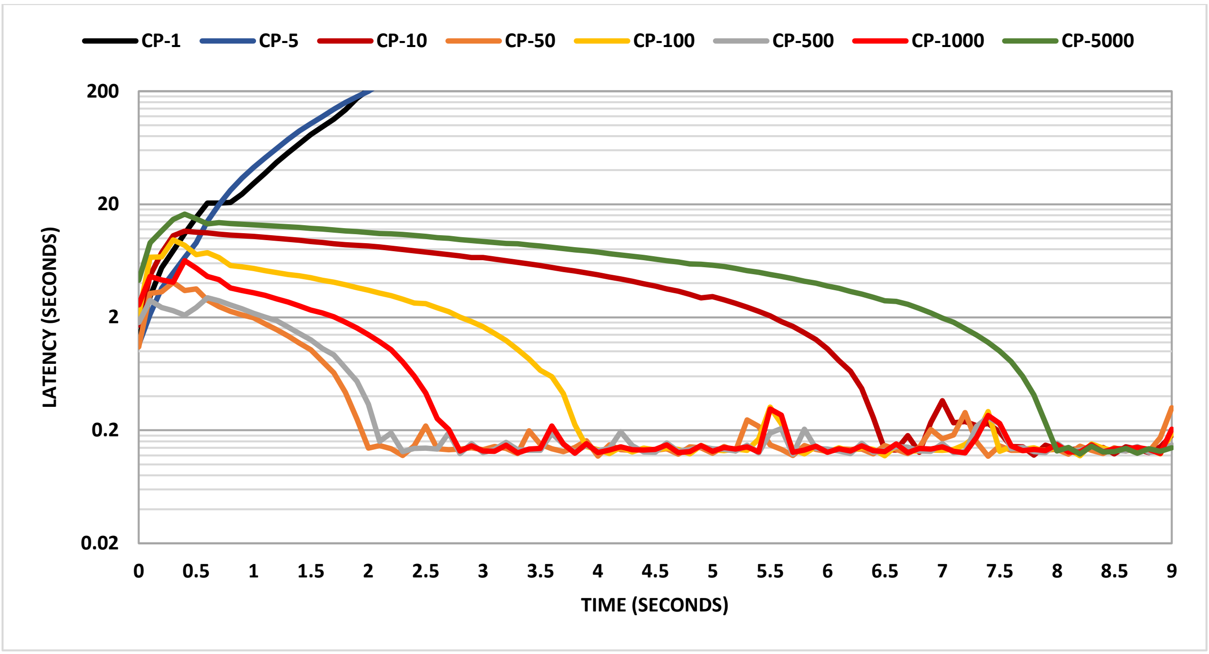

Figure 7-15 Latency in our Implementation for different checkpoints intervals with 4 Operator Instances

7.6.5 Latency Comparison with Different Checkpoint Intervals

In this test, we focus on the effect to checkpoint interval on the latency of the events. Similar conditions are used for this case with adaptive speculation. There are a total 1 million events with frequency of 100 thousand events per second. Tuple based predicate is used with window size 10000 events and window slide 2000 events. 100 microseconds simulated load is used in the operator instances. Events of the source ‘C’ with timestamps between 100 milliseconds and 1 seconds are delayed. Separate figures are shown for different number of instances to make the analysis of results easier. Results for different interval of checkpoints is shown in every figure.

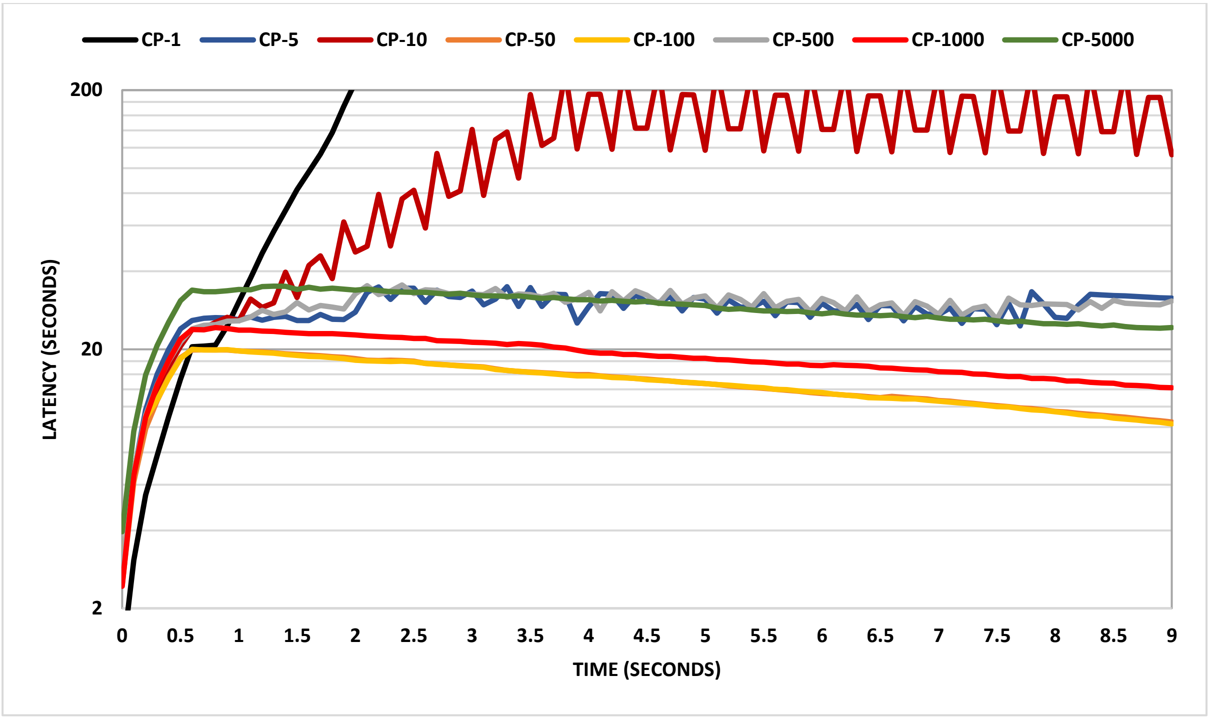

Figure 7-16 Latency in our Implementation for different intervals with 8 Operator Instances

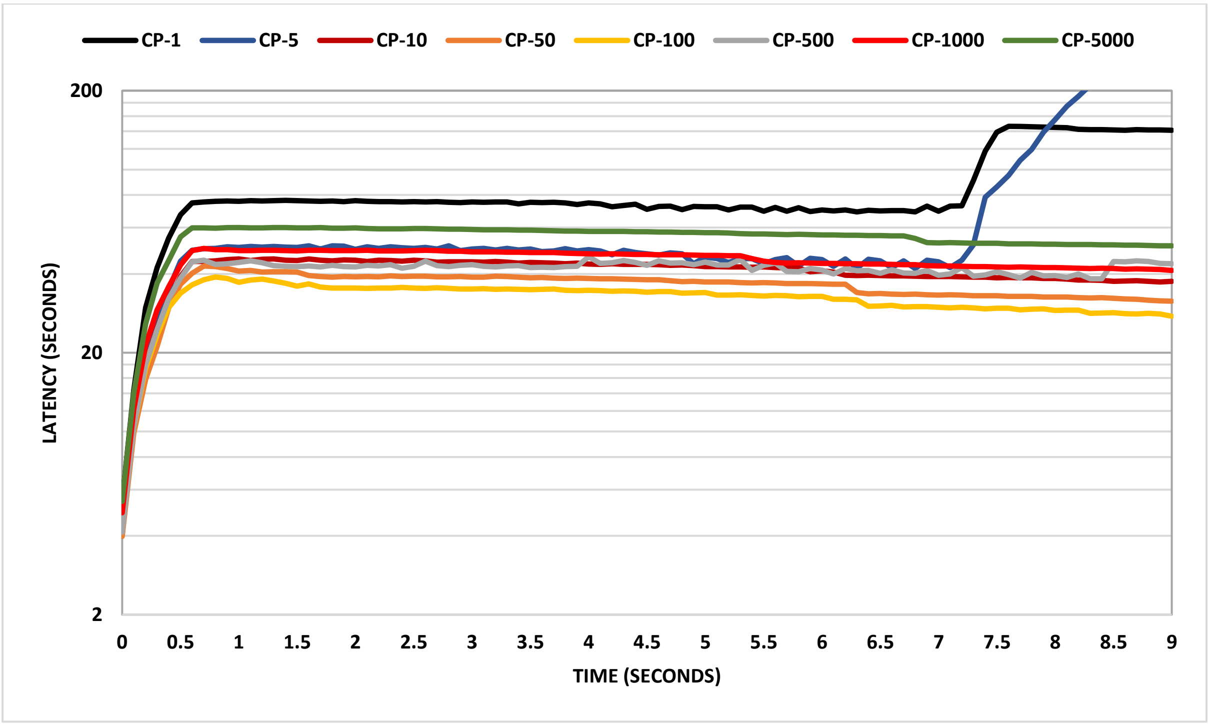

Figure 7-17 Latency in our Implementation for different intervals with 16 Instances

Figure 7-13 is the case when only single operator instance is processing the events. It shows that the latency has risen very high due to out of order events when checkpoint was taken at every event. The peak has decreased in case of checkpoint at every 5th event. It has decreased further with interval of 10 events but remained constant there for 50 events interval also. Latency peak is minimum for checkpoint interval of 100 events but after that it starts to increase again.

Figure 7-14 shows the results for the case when we have 2 operator instances. It is also showing similar results but this time minimum latency is for the case when checkpoint is taken at every 50th event. When we have 4 operator instances, the latency peaks rise too high for the case of checkpoint at every event and for the case when checkpoint is taken after 5000 events. Remaining cases also take longer to settle down after the peak. However, taking checkpoints at every 50th event is still performing best for 4 operator instances as shown in Figure 7-15.

Figure 7-16 shows the same trend in which all the figures are taking much longer to stable down in case of 8 operator instances. They are taking so long to stable that the events in this experiment were not enough to get the latencies in this case reach the stability. Checkpoint intervals of 50 and 100 events are performing best in this case. Figure 7-17 also shows the similar trend but the checkpoint intervals of 1 and 5000 are not performing that bad in this case. However, the remaining trends remain the same here.

7.7 Performance Overview

The performance of such a system can be accessed in terms of latency as well as throughput. We have analyzed both parameters in different possible situations and shown that our system is performing much well than the one proposed by [MP13b]. The technique of speculation proposed by them is performing extremely badly in the case of parallel CEP system. There are some necessary improvements proposed by us to make this technique worthy enough to be used by such a system in real world applications.

We have also proposed the internal recovery technique that helps to avoid unnecessary communication between different components of the operator. It also helps to let all the components run on their own pace, such that no one has to wait for the other one which can have a great impact on the system performance. We also save the memory by not saving the list of windows in the state of operator instance. It can save a lot of memory if the checkpoints are so frequent.

We have also proposed simple checkpoint models to use the best one that fits the situation. Moreover, the logic behind checkpoint decision is kept separate from the internal working of the framework, so that the user can also implement his own checkpoint model just like the predicate and the scheduling techniques that can also be implemented by the user himself according to the specific needs.

8 Literature Review

The literature review can be divided into two parts. First part focuses on the different techniques to handle out of order events. We discuss the shortcomings of the systems and discuss the ability of our framework to overcome those shortcomings to get performance of the system. Checkpoints are necessary for our framework to handle out of order events, so the second part focuses on the different models used for checkpoints and we compare them with our implementation.

8.1 Techniques to handle Out of Order Events

It is very important to process the incoming events in the correct order in most of the cases. [BSW04] proposes a technique to process all the events in the operator in correct order even when out of order events are being received from the sources. They propose a buffering technique in which all the events are buffered for a fixed K time units at most. So, that the events delayed for K- time units are included in the stream and forwarded to the operator in correct order. [LLD+07] extends this technique with local clock to use it in distributed systems. But the K-value in CEP systems cannot be predicted a-priori because there can be some unexpected delays. Details of the shortcomings of this technique have already been discussed above.

Speculation techniques avoid the buffering overhead in the latency and forward the events to the operator directly. In this way, at the time when there are no out of order events received, the latency will be very good but when a lot of out of order events are received, then the system will have to roll-back at a lot and it can has a very bad effect on the latency and throughput of the system [MP13b]. [Bur85] and [BFS+08] proposes such an approach.

[MP13a] proposes to adapt the K-value at runtime based on the out of order events received by the system. However, the K-value can rise very high in this technique, which can introduce a lot of latency for the events which is not good for many applications where latency is of prime importance.

[MP13b] adds speculation technique to adaptive K-slack buffering to get better latency. It can enhance the performance as compared to the adaptive K-slack algorithm in cases when out of order events are received for some time and after that all events are received in correct order. When the K-value was increased to a very high value, speculation can help to decrease the latency based on the system load as proposed by [MP13b]. However, as shown in the section 7, this system doesn’t work very well for parallel CEP systems due to a number of reasons discussed before.

[MTR16] and [MTR17] discuss different techniques to reduce the communication overhead by decreasing the communication between operator components in case of parallel CEP systems. However, they don’t focus on the replay of the events after recovery. So, we have implemented the technique to do internal recovery, so that the communication overhead can be further reduced in the presence of out of order events.

8.2 Checkpoints of the System

As, speculation is a risk, so there is a chance that the system has to be recovered back to some previous state. These states are saved while taking checkpoint of the system and to save these states in a database can be very expensive [LPW14]. Thus, we take in-memory checkpoints to avoid overhead of data transfer to and from database.

Different fixed interval checkpoint models are implemented to analyze the influence of checkpoint and recovery in our framework. Moreover, a number of existing checkpoint models are studied, which can also be implemented in the same framework to analyze adaptive checkpoints.

9 Conclusion

The work presented here extends the existing framework such that it can also handle the out of order events. A technique to merge the K-slack buffering along with speculation is used which adapts the degree to speculation at runtime based on the CPU load. No a-priori knowledge is required by the framework to handle out of order events. It adapts the buffering size at run-time according to the out of order events received by it. Some deterministic events are also kept saved sometimes to allow the system to handle abrupt changes in the buffering size.

Optimizations are done to let all the operator components run independent of each other. Moreover, a technique of internal replay is also proposed which reduces the communication overhead in case of recovery of the system due to the arrival of out of order events. Intensive evaluations prove that our framework is out-performing (in terms of latency and throughput) the previous work in the case of parallel CEP system. Synthetic data is used to evaluate the framework and it works very well.

Different basic checkpoint models are implemented to evaluate the effect of checkpoint frequency on the latency and throughput in presence of out of order events as well as in absence of out of order events. Results of the evaluations show that checkpoint frequency has a great impact on the latency. It is very important to run the system on the optimal checkpoint frequency to get the best possible performance.

9.1 Future Extensions

As, our work is generic for parallel CEP systems, so it can be extended to have other splitting predicates and scheduling techniques. It can also be extended further by the checkpoint model which adapts the checkpoint frequency at runtime. The model can consider different parameters as follows:

- Average time consumed to take a Checkpoint
- Average time consumed to process the event in Operator Instance.
- Number of events processed since last checkpoint.
- Success rate of speculation.

In addition to these factors, it can be observed that the events from a specific source are being delayed recently and the average delay value can be calculated to predict the arrival time of that event to check if the speculation of the current events will be beneficial or not. Different conditional probabilities can also be considered to take calculated risk to spend time on taking the checkpoint because the process of taking checkpoint is usually not negligible in high-rate stream processing systems. Incremental checkpoints can also be helpful to save memory used to save checkpoints, especially when the checkpoint frequency is very high.

Moreover, the future work can also be extended to analyze different data structures that can be used to save the events and the windows. Sometimes, the data structures can grow very high, so it is very important to use the data structures that can give best performance. We have used sorted hash maps mostly because every event needs to be added to the data structure in the correct order. However, there is a possibility that another data structure performs even better than sorted hash maps.

In our results, there is a sudden peak increase in latency when out of order events are received. This peak increases with increasing percentage of speculation and also with increase in the number of the operator instances. We assume that the latency is increased because the system tries to speculate events but the speculation goes wrong and the system has to recover back. This causes the events to clutter in the data structures. So, the effectiveness of data structures comes into play. However, this is just an assumption, it can be further evaluated in different cases to identify the exact reason. There are also a few anomalies in the results which can also be investigated further as a future work to get to know the exact reason of the anomalies, so that the necessary modifications can be done to get better results.

Bibliography

[Luc08] Luckham D., "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems," In: Bassiliades N., Governatori G., Paschke A. (eds) Rule Representation, Interchange and Reasoning on the Web. RuleML 2008. Lecture Notes in Computer Science, vol 5321. Springer, Berlin, Heidelberg.

[Luc06] D. Luckam, "What's the Difference Between ESP and CEP?" http://www.complexevents.com/2006/08/01/what%E2%80%99s-the-difference-between-esp-and-cep/, Real Time Intelligence and Complex Event Processing, 1 Aug. 2006. Web. 5 Sep. 2017.

[Bri+11] A. Brito et al., "Scalable and low-latency data processing with stream mapreduce," in Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci. (CloudCom'11), pp. 48–58, 2011.

[MP13a] C. Mutschler and M. Philippsen, "Distributed low-latency out-of-order event processing for high data rate sensor streams," in Proc. 27th Intl. Conf. Parallel and Distributed Processing Symposium, (Boston, MA), pp. 1133–1144, 2013.

[BSW04] S. Babu, U. Srivastava, and J. Widom, "Exploiting k-constraints to reduce memory overhead in continuous queries over data streams," ACM Trans. Database Systems, vol. 29, no. 3, pp. 545–580, 2004.

[LLD+07] M. Li, M. Liu, L. Ding, E. Rundensteiner, and M. Mani, "Event stream processing with out-of-order data arrival," in Proc. 27th Intl. Conf. Distrib. Comp. Systems Workshops, (Toronto, CAN), pp. 67–74, 2007.

[MP13b] C. Mutschler and M. Philippsen, "Reliable Speculative Processing of Out-of-Order Event Streams in Generic Publish/Subscribe Middlewares," in Proc. 7th Intl. Conf. Dist. Event-Based Systems, (Arlington, TX), 2013.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in Proc. 21th ACM Symp. Principles Database Systems, (Madison, WI), pp. 1–16, 2002

[MKR15] R. Mayer, B. Koldehofe, and K. Rothermel., "Predictable low-latency event detection with parallel complex event processing," IEEE Internet of Things Journal, 2, 2015.

[KMR+13] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel and M. Völz, "Rollback-recovery Without Checkpoints in Distributed Event Processing Systems," Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, ACM, pp. 27-38, 2013.

[MST+17] R. Mayer, A. Slo, M. A. Tariq, K. Rothermel, M. Gräber, and U. Ramachandran, "SPECTRE: Supporting Consumption Policies in Window-based Parallel Complex," Event Processing Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, ACM, pp. 161-173, 2017.

[MTR17] R. Mayer, M. A. Tariq, K. Rothermel, “Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing,” Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, ACM, pp. 54-65, 2017.

[Qua01] F. Quaglia, "A cost model for selecting checkpoint positions in time warp parallel simulation," in IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 4, pp. 346-362, Apr 2001.

[LPW14] H. Li, L. Pang, and Z. Wang, “Two-Level Incremental Checkpoint Recovery Scheme for Reducing System Total Overheads,” Ed. Matthias Dehmer. PLoS ONE 9.8, 2014.

[MTR16] R. Mayer, M. A. Tariq, and K. Rothermel, “Real-time Batch Scheduling in Data-Parallel Complex Event Processing,” Technical Report 2016/04. University of Stutgart. 14 pages

[BFS+08] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, “Speculative out-of-order event processing with software transaction memory,” in Proc. 2nd Intl. Conf. Distributed Event-Based Systems, (Rome, Italy), pp. 265–275, 2008

[Bur85] F. W. Burton, "Speculative computation, parallelism, and functional programming," in IEEE Transactions on Computers, vol. C-34, no. 12, pp. 1190-1193, Dec. 1985.

Acknowledgement

I am sincerely thankful to my mentor and supervisor Mr. Ahmad Slo from the University of Stuttgart for his help, guidance, motivation, and support during all the phases of my master thesis. I would also like to thank Prof. Dr. Kurt Rothermal for giving me this wonderful opportunity to do my master thesis at the Institute of Parallel and Distributed Systems (IPVS).

I am also thankful to my family and friends for their help and moral support during the tenure of my master thesis.

Muhammad Usman Sardar

Declaration

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

In addition, I certify that no part of this work will, in the future, be used in a submission in my name for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Stuttgart

Stuttgart, February 15, 2018 ______________________________

(Signature)