An Empirical Study of High Availability in Stream Processing Systems

Yu Gu[*]
Department of Computer Science and Engineering
University of Minnesota
yugu@cs.umn.edu

Zhe Zhang[*]
Department of Computer Science
North Carolina State University
zzhang3@ncsu.edu

Fan Ye
IBM T.J. Watson Research Center
Hawthorne, NY USA
fanye@us.ibm.com

Hao Yang
IBM T.J. Watson Research Center
Hawthorne, NY USA
haoyang@us.ibm.com

Minkyong Kim
IBM T.J. Watson Research Center
Hawthorne, NY USA
minkyong@us.ibm.com

Hui Lei
IBM T.J. Watson Research Center
Hawthorne, NY USA
hlei@us.ibm.com

Zhen Liu
Nokia Research China Lab
Beijing, China
zhnliu@yahoo.com

ABSTRACT

High availability (HA) is critical for many stream processing applications such as financial data analysis and disaster response. Existing HA schemes use either active standby or passive standby to guard the system against unexpected failures such as machine crash. Despite previous efforts of simulation-based studies that report active standby is superior, there is a lack of in-depth understanding of the tradeoff between different HA approaches under practical settings. In this paper, we propose a novel sweeping checkpointing method that can reduce the overhead by one order of magnitude. Whereas most previous work addresses single failures, we prove that the sweeping checkpointing method ensures no loss of data even against multiple concurrent failures. We then implement and compare the resulting passive standby variant against active standby using a real stream processing system. We find that passive standby presents a different tradeoff from active standby: longer recovery time, but 90% less overhead. Thus each approach has its suitable scenarios.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability

General Terms

Design, Experimentation, Reliability

Keywords

Sweeping checkpointing, high availability, stream processing systems

[*]Yu Gu and Zhe Zhang have contributed equally to this work.

1. INTRODUCTION

Stream processing systems have become increasingly popular for real-time analysis of massive and continuous data streams in a wide range of application scenarios, such as financial analysis, disaster response and network intrusion detection. In these systems, multiple stream processing jobs run on a cluster of machines interconnected with high speed networks. One job may contain many processing elements and run on multiple machines.

High availability (HA) is a critical issue for stream processing systems to provide continuous and uninterrupted operation for mission-critical applications against failures such as machine crash. There are two general HA approaches, namely *active standby* and *passive standby*. In active standby, two or more copies of the same job run independently on different machines, and the failure of one copy does not affect the other. In passive standby, a primary copy periodically checkpoints its state to another machine, and recovers from that machine once failures happen. Most stream processing systems adopt one of the approaches and apply it in their specific context [3, 6, 13, 16].

Despite the fact that AS and PS are two basic approaches, the design and performance tradeoffs between them are not fully understood. One notable exception is the work of Hwang et al [10, 11], which reported that active standby is superior to passive standby as it can achieve much shorter recovery time with a similar amount of overhead. However, that work is based on analysis and simulation, therefore it could not take into account all the delicacies of a real system. Further, it is not clear whether AS/PS variants other than the ones chosen in [10, 11] would lead to the same conclusion. These cast doubts on the generality of the findings.

In this paper, we examine the variants evaluated before and propose a novel *sweeping* checkpointing method that can reduce the checkpointing overhead by one order of magnitude. As a result, passive standby may present a different performance tradeoff compared to active standby: longer recovery delay, but less overhead. This is preferable for applications that are less sensitive to delays but demand efficiency. We then conduct an experimental study of active and passive standby using a real stream processing system. We find that while active standby provides fast recovery, it can incur significantly more processing and message overhead. This is in contrast to the findings in [10, 11] that PS incurs similar message overhead as AS and being slower in failure recovery.

We have made three contributions in this work. First, to the best of our knowledge, we are the first to propose the sweeping checkpoint method, which can reduce the checkpoint overhead by one order of magnitude. Second, we also prove that it can ensure no loss of data against *multiple concurrent failures*, whereas previous studies in [10, 11] focus on single failures only. Third, we have implemented and evaluated the sweeping checkpoint method and compared the performance of the resulting PS and AS. The discoveries make it clear that PS and AS present different performance tradeoffs and each has its suitable scenarios.

The rest of the paper is organized as follows: Section 2 presents the stream processing model and the assumptions we make for the investigation. In Section 3, we briefly review active and passive standby, then describe our sweeping checkpointing, and prove its ability to ensure no loss of data against multiple concurrent failures. We describe the system implementation in Section 4, then we provide a comprehensive evaluation of AS and PS (with different checkpointing methods) in Section 5. We summarize related work in Section 6 and conclude the paper in Section 7.

2. STREAM PROCESSING MODEL

A stream processing system contains many machines connected through a high speed network. This infrastructure can be shared by many users, each of which can submit jobs. One job contains multiple processing elements (PEs, or operators) connected in a directed graph. A PE is a software module that has possibly multiple input queues, each of which receives one input data stream, performs certain processing and produces possibly multiple output data streams, stored in multiple output queues and then sent to input queues of other PEs, and further processed by other PEs.

A PE can be classified in different ways. They can be *stateful* or *stateless*. A stateful PE has internal state that depends on previous state and input data streams; its output streams depend on the state and input. When such a PE recovers, it is critical to restore the internal state. A PE can be *deterministic* or *non-deterministic*. Given the same sequence of input data, a deterministic PE always produces the same sequence of output data. The inter-arrival times of input data do not affect the output. For a non-deterministic PE, such inter-arrival times affect the output; it may even produce different output given the same sequence of input data with the same arrival timing.

We assume that the system has a scheduling component [17] that decides how to divide a job into multiple non-overlapping *subjobs*, and decides on which machine to run each subjob, based on resource (CPU, memory, bandwidth) availability on machines and requirements from PEs. A subjob contains a subset of PEs and are deployed to a machine together, and managed as a unit.

3. A SWEEPING CHECKPOINTING METHOD

An earlier study [10] shows that AS is always superior to PS: PS has almost the same overhead as AS, while taking at least 50% more time in recovery. A careful examination shows that the study uses a specific method of checkpointing in PS, where the input and output queues, the internal state are all checkpointed, and all PEs in one machine are checkpointed together periodically. We propose a new sweeping checkpointing method, which outperforms this method in speed and overhead; it gives PS a magnitude lower overhead than AS.

Active Standby In active standby, both the primary and secondary machines are running the same subjobs that receive the same input, process them and send output data to downstream machines. Downstream subjobs need to eliminate duplicates, using mechanisms such as sequence number. When one of the machines fails, the other is not affected. Active standby has almost no recovery delay; the cost is the duplicate messages and processing.

Passive Standby In passive standby, the primary machine periodically checkpoints job states (such as input / output queue data and internal states) and sends the checkpoint message to the secondary machine. The checkpoint message includes the internal states of the PEs and also the data in the input and/or output queues. When the primary machine fails, the secondary copy is restarted using the stored states. It will reconnect the upstream and downstream subjobs, and resumes the processing.

3.1 Sweeping Checkpointing

The performance of passive standby depends heavily on the checkpointing method. There are three issues in checkpointing: 1) when to trim (i.e., remove) data from output queues, 2) which data to include in checkpoint messages, and 3) when to checkpoint each PE in multi-PE jobs.

When to trim A PE produces data in output queues and sends them to the input queues of downstream PEs. Different from reliable transport such as TCP, we do not remove data that are received at downstream input queues immediately. This is because the downstream PEs may crash with data unprocessed in their input queues. Such data must be preserved and reprocessed after the downstream recovers.

We use accumulative acknowledgment (like that of TCP) from each input queue. An output queue assigns an incremental sequence number to each newly produced data element. When a downstream PE finishes *receipt, processing* of some data and the *checkpointing* of resulting states, it sends an acknowledgment to each of the upstream output queues. If an output queue sends data to multiple downstream input queues, it removes the data element only when it receives the acknowledgments from all downstream input queues for that data element.

What to include A PE has data in input/output queues, and internal states. Note that the internal states are not the PE's memory image, which can be huge, but variables that are updated as the PE processes data. These variables affect the output and are much smaller than the complete memory image. In [10] all the three are checkpointed. we decide to include the internal state and output queues, but not input queues. Because in our case, data in the input queues are still available in the output queue of upstream PEs before they are processed and resulting states checkpointed. Should a PE crash before finishing processing some data or checkpointing resulting state, it can recover them from the output queues of upstream PEs.

Excluding input queues significantly reduces the size of checkpoint messages. Because these messages are periodic, it reduces a recurring overhead. The alternative, also checkpointing input queues, increases the message size. It does not allow output queues to remove data upon their receipt at downstream input queues either. Because the downstream PE may crash before finish processing those data, output queues have to preserve the data until the data are acknowledged by downstream input queues.

When to checkpoint Given multiple PEs in the same subjob, when to checkpoint each of them significantly affect the message size and overhead. The simplest option is *synchronous checkpointing*, which uses a timer for each subjob to periodically suspend all its PEs, checkpoints their states and then resumes all of them. This option is adopted by the PS variant in [10]. Because checkpointing happens after all PEs are suspended, this method is usually slower. Another option is *individual checkpointing*, where each PE has its own timer to drive its own checkpointing.

We propose a *sweeping checkpointing* technique that takes only half or a third of the time to do a checkpoint, and one magnitude less message overhead than the previous two options. In this method, the checkpointing is not driven by a timer, but by the acknowledgment from downstream PEs. As described earlier, whenever an output queue receives an acknowledgment that all its downstream PEs have successfully processed and checkpointed the states of some data, such data can be removed. Since the output queue size becomes much smaller after the queue trimming, we immediately checkpoint the state of this PE after queue trimming.

Given a chain of PEs that all require checkpointing, the checkpoint of the last PE can be driven by a timer since it does not have PEs further downstream. Its checkpointing will incur periodic acknowledgments upstream and trigger checkpointing. The checkpointing events "sweep" in the reverse direction of the data flow, from the last PE toward the data source. If the last PE does not require HA protection, it may send acknowledgment upon receipt of new data. Such acknowledgments can still trigger checkpointing in upstream PEs.

3.2 Proof of Consistency

A sound checkpointing design must ensure the correctness of the processing results after a job recovers from the failure of one or multiple PEs. If the checkpointing of different PEs are not properly coordinated, a recovered PE may re-process old data that has already been processed, or miss some data that it should process. In either case, incorrect results will be produced.

In what follows, we first define the consistency property of a checkpointing method, and then formally prove our sweeping checkpointing method is consistent even with multiple concurrent PE failures.

Definition 3.1. *A checkpointing method is consistent if and only if a job after recovery always produces processing results as if no failure had happened when the same sequence of input data was fed into the job.*

Several remarks are in order. First, when a deterministic PE is recovered, it should produce exactly the same output as in the case where no failure has occurred. However, for a non-deterministic PE, there is no unique output even for the same input, thus we can only ensure the output of a recovered PE is one of the possible results with the original input. Secondly, we do not consider the failure of the first PE (i.e., source PE) in a job, because the data source is typically external to the streaming processing system and there is no guarantee it can preserve data lost during failures. Thus data losses may happen when the source PE fails. Lastly, there may be duplicated, delayed and out-of-order delivery of data elements during the recovery phase. However, these issues can be easily dealt with, e.g., by using application level sequence numbers. Thus, the checkpointing method only needs to ensure all the intermediate processing results can be recovered and re-delivered to the downstream PEs.

Theorem 3.1. *Sweeping checkpointing is consistent when a PE fails but all its immediate upstream PEs are still available.*

Proof: Suppose PE_i fails and it has a set of upstream PEs, denoted by PE_i^U, and a set of downstream PEs, denoted by PE_i^D. In the last checkpoint before the failure, PE_i has processed up to data element m_i^c and its internal state is s_i^c. Clearly, the upstream PEs in PE_i^U are not affected by the failure of PE_i. Thus, it is sufficient to prove that PE_i and all its downstream PEs correctly resume the processing after the recovery.

After PE_i is recovered, it is loaded with its last checkpointed state s_i^c. Then it starts to pull the data elements after m_i^c from the upstream PEs in PE_i^U. Note that all these data elements are still in the output queues of these upstream PEs, because they have not been checkpointed by PE_i, thus there were no corresponding acknowledgments triggering their removal. As these old data elements are re-sent to PE_i, its processing resumes with the results flowing into the downstream PEs in PE_i^U. These downstream PEs will drop any data elements they have previously received and processed, and eventually start to process new data elements produced by PE_i. □

Next we prove the consistency of sweeping checkpointing in a more general setting.

THEOREM 3.2. *Sweeping checkpointing is consistent with multiple concurrent PE failures.*

Proof: When failures of multiple PEs overlap, these PEs can be either consecutive (i.e., directly connected) or nonconsecutive in the flow. When the failed PEs are nonconsecutive, the consistency of sweeping checkpointing can be derived directly from Theorem 3.1. On the other hand, when consecutive PEs fail, they will start the recovery process individually. However, the recovery of a downstream PE cannot complete until all its upstream PE are available. As such, these failed PEs naturally recover in an order from upstream to downstream.

Consider the most upstream one among the failed PEs, say PE_i. Based on Theorem 3.1, it can correctly recover and re-process those data elements since its last checkpoint. As a result, its output queue will be populated again. Now consider the downstream PE, say PE_{i+1}, that has also failed. PE_{i+1} can start its recovery process once the new PE_i has caught up with data elements in the last checkpoint of PE_{i+1}. By applying Theorem 3.1 recursively, we can show that PE_{i+1}, and similarly all failed downstream PEs, can correctly recover in a sequential manner. □

3.3 Discussions

The sweeping checkpointing reduces overhead mainly because it excludes input queue data and reduces the amount of output queue data to checkpoint. It improves the speed of checkpointing because there is no need to wait all PEs to suspend before checkpointing. It does not reduce the internal state of PEs. Although the internal states are variables that affect output data and is not the complete PE memory image, they can be significant depending on the application. In such scenarios the performance improvement of the sweeping method compared with other checkpointing methods may be reduced (compared with the results that we will show in Section 5.2). However, given the increasingly higher data rates in today's stream processing applications, we expect the efficiency and speed of sweeping checkpointing to offer significant benefits.

4. SYSTEM IMPLEMENTATION

We have implemented a fully functional stream processing system consisting of over $25K$ lines of code in Java. It reuses some of the components in our previous CLASP [4] work. For the sake of self-containment, we will briefly introduce some components but will focus on new components related to high availability.

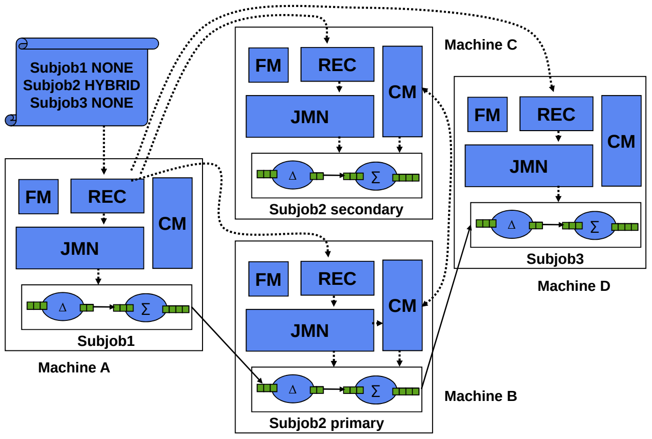

Figure 1: System architecture. A job consisting of 3 subjobs, running on 4 machines. The 2nd subjob has passive standby protection, with a primary copy running on machine B and a secondary copy deployed on demand on machine C.

Figure 1 shows the architecture of our system. Each machine will run an instance of our system, including Remote Execution Coordinator (REC), Job Management (JMN), Checkpoint Manager (CM) and Failover Manager (FM). Each instance will register itself as a distinct RMI object and they will use the RMI name to contact each other.

A job is submitted to the REC of a machine as a text file. Each line contains the file name of the Job Description Language (JDL) of a subjob. We use an XML-based language to describe which PEs are contained in a subjob, and how their queues are connected, possibly with PE queues in other subjobs. Each line also contains the high availability type, which machines are involved and their roles.

```
/home/joe/jdls/sisubjob1.jdl \
HA_TYPE_NONE host1
/home/joe/jdls/sisubjob2.jdl \
HA_TYPE_PASSIVE host2 host3 host3 host3
```

This example shows a job with 2 subjobs. The first one has no HA protection and runs on host1. The second one is configured with passive standby and involves 3 machines, host1 for running, host3 for storing checkpoint data, monitoring the aliveness of the primary (host1) and recovering the subjob upon failures. These functions can be carried by separate machines if needed. Similarly, a subjob configured with active standby would have two hosts as the primary and the secondary respectively, both to be actively running, and another host responsible for monitoring and recovery.

The REC is in charge of the execution, failure recovery and life cycle management of jobs. When a job is submitted to the REC of a machine (called the *owner* machine), the REC will render the JDL (adding machine-specific configurations), create additional JDLs needed for secondary or state machines if needed, and submit these JDLs to the REC of their respective machines. It also notifies the local FM to start a monitoring task at the monitoring machine.

The local FM notifies remote FM, which sends periodic heartbeat messages to monitor the aliveness of remote machines. There is a deadline for receiving a response for each heartbeat message. Upon misses of heartbeat responses, it notifies the owning machine's FM how many misses have occurred; when a machine comes back, it also notifies how many heartbeat responses have re-emerged. We use the conventional wisdom that 3 heartbeat misses indicate the failure of a machine. This is a simple and effective method to monitoring.

The JMN manages subjobs deployed on a machine. It receives JDLs from the local REC, parses the JDL, invokes an instance of each PE according to its parameters in the JDL. For PASSIVE type, it also starts a checkpoint task and submits it to the CM of the local machine. We will describe in detail how the CM manages the checkpoint of PEs.

4.1 PE Checkpoint

In our implementation, each PE has multiple input and output queues, each receiving and sending data to possible multiple output or input queues. Each PE has two threads, one of which is a processing thread that takes data from input queues, processes the data and puts results to output queues. When receiving a new data element, an output queue gives a unique sequence number to that element. The other is a sending thread goes through all output queues, sends data to each of the input queues connected to an output queue.

For PEs to contact each other, they register themselves as RMI objects. Sending data across PEs is implemented as an RMI call to the input queue of remote PEs. We use two threads because they can improve the throughput from about 700 data elements per second to over 25K, suitable for evaluation under high data rates.

A $PEController$ controls the checkpointing of a PE. It can be a subjob, or the CM. It calls a PE's *pause(PEControllerIfc pec)* method to suspend it, giving its own interface as the parameter for the PE to call back. When the PE has suspended, it calls the $ackPePause(PEIfc\ pe)$ method of the PE controller, also giving its own interface *pe* for the controller to perform further operation. The controller will call the $checkpoint()$ method to obtain the state of the PE. After storing the state on the secondary machine, the controller calls the $resume()$ method to resume the PE.

To allow PEs to accurately determine which data should be sent during recovery, each output queue maintains a sent sequence number and an acknowledged sequence number for each of the downstream input queues. Only data acknowledged by all downstream input queues can be removed.

To measure the performance under high data rates, we also make one important decision not to limit the size of queues. Such limits will back pressure and cause the upstream processing to stop when an input queue is full. Such slowdown is not caused by the overhead of high availability mechanisms, but the mismatch between processing speed and data rates. Not limiting the size not only allows us to measure the true overhead by high availability mechanisms, but also the queue size needs of them.

4.2 Checkpoint Manager (CM)

The CM is responsible for managing the checkpoint of subjobs and PEs. When the JMN deploys a subjob whose JDL has PASSIVE type configuration, the JMN will use $submitCheckpointJMNJobTask()$ to submit a checkpoint task to the local CM, including parameters for the ID and interface for that subjob, and the checkpoint requirements such as state machine, frequency.

For synchronous checkpointing, the CM will periodically suspend the subjob through a *pause(JMNJobControllerIfc jc)* method, using itself as the parameter for callback. After the job's callback, it obtains the job state by a $checkpoint()$ call on the job (basically the internal state and output queues of all PEs), then calls the $storeJobState(js)$ of the CM of the state machine to store the state js. Finally the CM resumes the subjob by the $resume()$ on the subjob. For individual or sweeping checkpointing, the CM will control individual PEs directly. The only difference is that for sweeping checkpointing, the call to $trim()$ that removes data in output queue will trigger a one-time checkpoint task for that PE only.

During recovery, the owner machine's REC will sends a JDL to the recovery machine's REC, which forwards it to the local JMN. The JMN parses the JDL and obtains the state by $recoverJobState(jobId)$ of the local CM, which in turn calls the $retrieveJobState(jobId)$ of the CM on the state machine to retrieve the state.

5. EVALUATION

5.1 Experimental Environment and Methodology

The experiments are performed on a cluster of RHEL 4.4 Linux workstations connected with 1 Gbps LAN. Each workstation has a 3.07 GHz 4-core Xeon processor, 4.2 GB memory and 80 GB hard drive. The stream processing job used in our experiments consists of 8 PEs connected in a chain topology. The entire job is then further divided into 4 subjobs, each consisting of 2 PEs. Each subjob is assigned to a separate primary machine. Inside the processing loop of each PE, there is code that performs some synthesized computation. The PE selectivity is 1, meaning that it produces exactly one data element for each input data element. Such a simple job topology and computation avoid the impact of job division/scheduling and application logic, so the difference in results can be exclusively attributed to different HA approaches.

5.2 Comparison of Different Checkpointing Methods

We first compare the message overhead and the time it takes to do one checkpointing for the three different checkpointing methods: synchronous, individual and sweeping. We found that sweeping checkpointing has less than 10% the total message overhead, and takes 1/4 time, compared to those of the other two methods.

Message Overhead

Figure 2 and 3 show the impact of checkpoint interval on average and total checkpoint message size, respectively. We fix the data rate to 3000 elements/s, the PE state size is

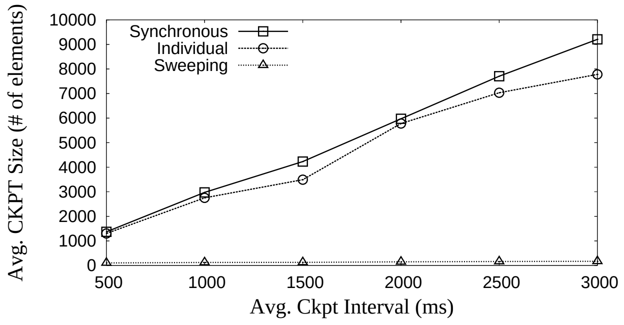

Figure 2: Average checkpoint message size vs. checkpoint interval

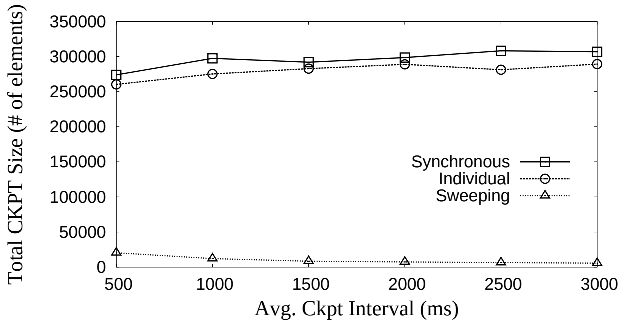

Figure 3: Total checkpoint message overhead vs. checkpoint interval

that of 20 data elements. Figure 2 shows that the average checkpoint message size of sweeping is roughly 1/30 of that of synchronous and individual checkpointing. We also observe that the size for sweeping grows more slowly than the other two when checkpoint interval increases.

Figure 3 shows that the total amount of checkpointed messages at various checkpoint intervals during 100 seconds of run. Sweeping still has the smallest total overhead, about 1/13 to 1/30 of those of the other two. For both synchronous and individual checkpointing methods, their total checkpoint message size stays relatively constant. This is because the number of checkpoint messages decreases inverse proportionally, while the average checkpoint message size increases proportionally, to the checkpoint interval. Thus the total overhead remains about the same. Interestingly, for sweeping checkpointing, the total overhead drops as checkpoint interval increases. This is because the average checkpoint size for sweeping grows sub-linearly to the checkpoint interval. As the number of checkpoint messages decreases linearly, the total overhead decreases.

Checkpoint Time We vary the input date rate from 1000 to 7000 elements/s and keep checkpoint interval at 500 ms. Figure 4 shows the amount of time needed to finish one checkpointing. Sweeping clearly outperforms the other two: It takes about 1/4 of the time of those of the other two. Synchronous has the longest time because it needs to suspend and then resume all PEs for a subjob. The difference between individual and sweeping checkpointing is caused by checkpoint message sizes. As shown in Figure 3, individual has about 30 times the data to transmit and store on the

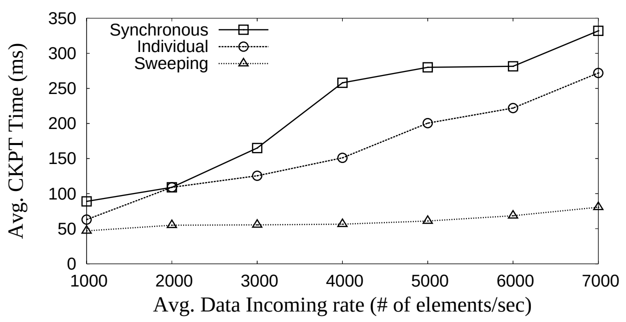

Figure 4: Checkpoint Time vs. Data Rate

state machine. Based on the above comparison, we can see that sweeping is clearly faster and more efficient.

5.3 Comparison of AS and PS

Message Overhead We compare the message overhead incurred for one subjob under four different configurations: NONE, where the subjob has only one copy; AS-AS, where both this subjob and its downstream subjob each has two copies, and both copies send to the two downstream copies; None-AS, where a subjob has two copies but its upstream and downstream subjobs have only one copy; and sweeping based PS, where the subjob has one copy and a recovery machine for deploying the secondary copy on demand. We measure both AS-AS and None-AS because they are both feasible configurations, with AS-AS protecting each, while None-AS one of the subjobs. We increase the input data rate from 1000 elements/s to 25000 elements/s. The PS has 500 ms checkpoint interval.

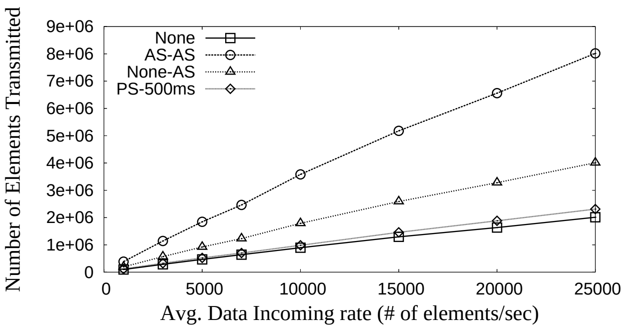

Figure 5: Overhead vs. Data Rate

Figure 5 shows that the message overhead of None-AS is around twice of that of NONE. That is consistent with the intuition since both copies received and send data. AS-AS is about four times because each copy send to two downstream copies. In PS, the message overhead increases merely about 10% to None, which is a magnitude lower than that of AS. This clearly shows that with sweeping checkpointing, PS can have much lower message overhead than AS.

Recovery Delay To demonstrate the behavior of different HA policies upon failures, we kill one of the subjobs while the system is running. Then by plotting the sequence number vs. receipt time of received data elements on downstream subjob of that killed subjob, we can see how much recovery delay an HA policy incurs. In this set of experiments, the

PS policy uses a checkpoint interval of 500 ms and heartbeat interval of 100 ms. As illustrated in Figure 6, with the AS policy, the failure is almost transparent to the downstream subjob. With the PS policy, a "silent" period is observed, during which the system does not produce any new data element and the sequence number curve is flat.

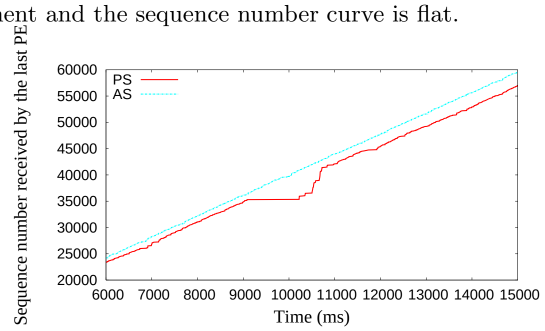

Figure 6: Sequence number received over time

We then show a decomposition of PS method's failure recovery time, defined as the time from a failure happens to the producing of the first new output data element. The recovery delay for PS consists of three parts: *failure detection*, *job redeployment*, and *data retransmission/reprocessing*, which reprocess data sent to the primary subjob but whose results were not produced due to failures.

Figure 7 shows the impact of heartbeat interval on recovery time. We fix the checkpoint interval to 500 ms, and change the heartbeat interval from 100 ms to 500 ms. We can see that with larger heartbeat intervals, the failure detection time increases about linearly, and becomes dominant in the failure recovery time.

Figure 8 shows the impact of the checkpoint interval with a fixed heartbeat interval of 100 ms. With larger checkpoint intervals there is more data to retransmit and reprocess. We observe that the retransmission / reprocessing time tends to increase when the checkpoint interval grows from 100 ms to 900 ms. However, the other two parts are greater or remain about the same. Thus the total recovery delay does not change much. The fluctuation of recovery delay is caused by the randomness in the timing of the failure, which determines the amount of retransmission data.

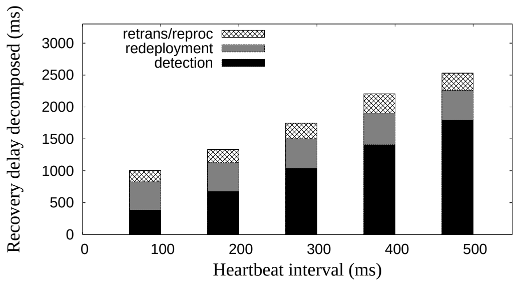

Figure 7: Recovery time decomposition vs. Heartbeat interval

The above two figures shed some light on improving the recovery delay for PS. First is to improve the detection, which is the largest component in the overall recovery delay. Next is to speed up redeployment, which is also a significant part (about 500 *ms*) in the total delay. Overall, the experiment results show that although AS has almost negligible recovery delay, PS only needs 10% of the overhead. Thus each approach has its suitable scenarios.

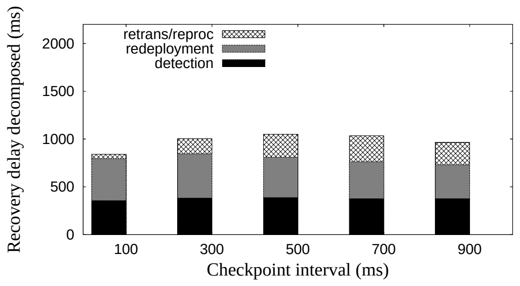

Figure 8: Recovery time decomposition vs. Checkpoint interval

6. RELATED WORK

High availability for stream processing systems has become an active research topic in recent years. Representative work includes those [3, 9, 12, 13] proposed in the context of Borealis [1], one of the first stream processing systems. Except [13], the other three are all based on active standby. [3] achieves flexible trade-off between availability and consistency by introducing tentative data concept; [9] has the highest availability by paying the cost of having multiple upstream copies sending data to multiple downstream copies; [12] allows replicas to execute without coordination but still produce consistent results; [13] studies the optimal checkpoint scheduling and backup machine assignment when multiple subjobs need checkpointing and many state storage machines are available. They choose one of the two basic approaches as basis and do not study the performance tradeoff between the two,.

High availability has been studied in other stream processing and data flow systems. [14, 15] are in the context of System S [2], a stream processing system being developed at IBM Research. [14] studies how to provide high availability for the system component of JMN by checkpointing related job state information. It does not study high availability for jobs, which is the focus on this paper. [15] studies how to pick the most suitable recovery machine when many are available to recover failed jobs, a related but different problem compared to ours. Shah *et al.* [16] is one early work that adopts active standby approach for parallel data flows. It coordinates multiple replicas and focuses on ensuring consistency and preventing deadlocks, important but different issues compared to us.

High availability in virtual machines has also been studied. Hypervisors [5] run a pair of virtual machines in a synchronized manner. Two virtual machines have identical system states at any point of time. Remus [7] reduces the overhead by relaxing the synchronization requirement; the states of two machines are synchronized only when the output of the primary is externally visible. They still adopt the basic active or passive standby approach. However, because of the significant differences in assumptions, design goals, and evaluation environments in VM, the detailed techniques may not be applied directly.

Despite the prevalence of active and passive standby approaches in stream processing systems, there is not much work to systematically compare their tradeoff, especially different variants. As far as we are aware, Hwang et al.'s work [10, 11] is the only exception where the results show that active standby is always superior. Although they classify basic high availability algorithms, their comparison is mostly analytical and simulation-based, thus cannot fully capture the complexities and delicacies in a real testbed. More importantly, the results were based on a specific variants of passive standby, thus the generality of the conclusion is limited. We propose a sweeping checkpoint method, use it to implement a new PS variant, and perform comprehensive empirical evaluations in a real prototype. We reach a very different conclusion that both active and passive standby have their suitable conditions and neither is always better than the other. This insight has inspired and motivated another piece of our work for designing a hybrid approach that combines the advantages of both approaches to provide fast recovery at affordable overhead. The details are beyond the scope of this work and we will publish in a separate paper.

Checkpointing is a widely used technique for preserving critical state information. [8] provides a comprehensive summary and comparison of different checkpointing techniques. Among existing checkpointing mechanisms for stream processing systems, the Efficient Coordinated Operator Checkpointing (ECOC) proposed in [6] is the closest to the sweeping checkpoint method used in our system. ECOC uses a two-phase protocol, where locally scheduled checkpoints are first stored in a "pending checkpoint list", and then executed after confirmations from all subsequent operators are received (indicating that the output queue can be trimmed). Compared with ECOC, our scheme does not require two phases and is much simpler and efficient, leading to the magnitude reduction in overhead compared to conventional checkpointing methods. Moreover, in ECOC it is assumed that output queues can be trimmed to empty, whereas under high data rates we find this is not the case. Some output elements are still produced between the trim and the checkpointing.

The *upstream backup* method proposed in [10] also keeps data elements in the output queue until confirmations from downstream machines are received, indicating that the data have been processed and resulting data received at further downstream machines. Because this method does not checkpoint PE's internal states, any data that affect a PE's internal states must be stored in the upstream PE's output queues almost infinitely; otherwise the correct internal state cannot be reproduced. This severely limits its applicability to stateful PEs where most data do affect internal states: the recovery delay under high data rates can be extremely long. Also, it cannot handle multiple concurrent failures, because data are preserved long enough only for the immediate downstream machine crash. Any downstream failure beyond that incurs inrecoverable data loss.

7. CONCLUSION

High availability is essential for many stream processing applications. Despite the popularity of AS and PS as the basis for many high availability solutions, their design and performance tradeoffs have received limited attention. We propose a novel sweeping checkpointing method that not only reduces the overhead by one order of magnitude compared to conventional checkpointing methods used in previous studies [10, 11], but also ensures no loss of data against multiple concurrent failures. Using the sweeping checkpointing, we also implement a new PS variant in a real prototype stream processing system. Our experimental evaluation shows that AS and PS each has its respective advantages and disadvantages. While AS provides fast recovery, PS needs much smaller overhead. There is no clear winner and each of them should be adopted under their most suitable scenarios.

The insights we gain from this study have motivated us to design a hybrid HA approach that combines the advantages of AS and PS, providing fast recovery and affordable overhead at once. The main idea is to allow individual PEs to switch between AS and PS modes dynamically depending on the occurrence of failure events. We will describe the design details and performance results in a separate paper.

8. REFERENCES

[1] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. The design of the borealis stream processing engine. In *Proceedings of CIDR'05: 2nd Biennial Conference on Innovative Data Systems Research* (2005).

[2] Amini, L., Jain, N., Sehgal, A., Silber, J., and Verscheure, O. Adaptive control of extreme-scale stream processing systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 2006), IEEE Computer Society, p. 71.

[3] Balazinska, M., Balakrishnan, H., Madden, S., and Stonebraker, M. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2005), ACM, pp. 13–24.

[4] Branson, M., Douglis, F., Fawcett, B., Liu, Z., Riabov, A., and Ye, F. Clasp: Collaborating, autonomous stream processing systems. In *Middleware '07: Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware* (New York, NY, USA, 2007), ACM.

[5] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM, pp. 1–11.

[6] Brettlecker, G., Schuldt, H., and Schek, H.-J. Efficient and coordinated checkpointing for reliable distributed data stream management. In *ADBIS* (2006), vol. 4152 of *Lecture Notes in Computer Science*, Springer, pp. 296–312.

[7] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: high availability via asynchronous virtual machine replication. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008),

USENIX Association, pp. 161–174.

[8] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (2002), 375–408.

[9] Hwang, J., Cetintemel, U., and Zdonik, S. Fast and reliable stream processing over wide area networks. In *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering* (2007), IEEE Computer Society, pp. 604–613.

[10] Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. High-availability algorithms for distributed stream processing. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 779–790.

[11] Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. High-availability algorithms for distributed stream processing. In *Technical Report CS-04-05* (2005), Department of Computer Science, Brown University.

[12] Hwang, J.-H., Cha, S., Cetintemel, U., and Zdonik, S. Borealis-r: a replication-transparent stream processing system for wide-area monitoring applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 1303–1306.

[13] Hwang, J.-H., Xing, Y., Cetintemel, U., and Zdonik, S. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering* (Los Alamitos, CA, USA, 2007), vol. 0, IEEE Computer Society, pp. 176–185.

[14] Jacques-Silva, G., Challenger, J., Degenaro, L., Giles, J., and Wagle, R. Towards autonomic fault recovery in system-s. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing* (Washington, DC, USA, 2007), IEEE Computer Society, p. 31.

[15] Rong, B., Douglis, F., and Xia, C. H. Failure recovery in cooperative data stream analysis. In *ARES '07: Proceedings of the The Second International Conference on Availability, Reliability and Security* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 77–84.

[16] Shah, M. A., Hellerstein, J. M., and Brewer, E. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2004), ACM, pp. 827–838.

[17] Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K.-L., and Fleischer, L. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *the 9th ACM/IFIP/USENIX International Conference on Middleware* (2008), pp. 306–325.