A Hybrid Approach to High Availability in Stream Processing Systems

Zhe Zhang*
National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN USA
zhezhang@ornl.gov

Yu Gu*
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN USA
yugu@cs.umn.edu

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

Hao Yang
Nokia Research Center
White Plains, NY USA
hao.2.yang@nokia.com

Minkyong Kim, Hui Lei
IBM T.J. Watson Research Center
Hawthorne, NY USA
{minkyong,hlei}@us.ibm.com

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

***Abstract*—Stream processing is widely used by today's applications such as financial data analysis and disaster response. In distributed stream processing systems, machine fail-stop events are handled by either active standby or passive standby. However, existing high availability (HA) schemes have not sufficiently addressed the situation when a machine becomes *temporarily unavailable* due to data rate spikes, intensive analysis or job sharing, which happens frequently but lasts for short time. It is not clear how well active and passive standby fare against such *transient unavailability*. In this paper, we first critically examine the suitability of active and passive standby against transient unavailability in a real testbed environment. We find that both approaches have advantages and drawbacks, but neither is ideal to provide fast recovery at low overhead as required to handle transient unavailability. Based on the insights gained, we propose a novel hybrid HA method that switches between active and passive standby modes depending on the occurrence of failure events. It presents a desirable tradeoff that is different from existing HA approaches: low overhead during normal conditions and fast recovery upon transient or permanent failure events. We have implemented our hybrid method and compared it with existing HA designs with comprehensive evaluation. The results show that our hybrid method can reduce two-thirds of the recovery time compared to passive standby and 80% message overhead compared to active standby, allowing applications to enjoy uninterrupted processing without paying a high premium.**

I. Introduction

The world is entering a "digital era", where digital devices are widely deployed and massive amounts of data are generated continuously. It is reported that a total of 500 TB per year of imaging data is created in the UPMC medical system, and 200 of London's traffic cameras generate 8TB of data each day [9]. Stream processing systems that can analyze massive and continuous data streams in real-time are critical to process such data in many important application scenarios, from financial analysis, traffic management to network intrusion detection.

In these systems, multiple stream processing jobs share a cluster of machines connected with high speed networks. One job may contain many processing elements and run on multiple machines. A processing element (PE) is a software module that takes certain data input, processes it and generates data output for other PEs to consume. To maximize the system utilization, a machine is often shared among different jobs and may host partitions of different jobs concurrently [2], [7], [5], [21].

This sharing creates interdependencies among jobs: a job that experiences a sudden increase of incoming data rate, or encounters some data that requires intensive processing, may consume significantly more resources than before. It can "squeeze" the resources available to other jobs sharing the same machine. Those other jobs can suffer significantly increased processing delay, sometimes as if they have stopped processing data. We call such phenomena *transient unavailability* or *transient failures*.[1]

Our measurements on a 100+ node environment (details in Section II-B) show that such transient unavailability lasts for short periods of time (e.g., around 10 s) but can happen frequently (e.g., once every minute or tens of seconds). This is very different from *fail-stop* events (e.g., machine crashes, power outages) that happen much less frequently and do not recover without explicit interventions. Nevertheless, such transient unavailability can cause over 100% increase in the average end-to-end processing delay, and 8 fold increase during periods of unavailability (in Section V-B). For many delay sensitive applications, such performance degradation warrants a solution.

Techniques such as load shedding and traffic shaping [3], [21] may alleviate load spikes by dropping some incoming data or reducing the burstiness in data rate. However, they do not completely solve the problem when applications are loss-sensitive, or when it is the nature of data, not the volume, that

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

[1] In the following discussion we use transient failures and transient unavailability interchangeably.

demands more resources for intensive processing. Scheduling and load balancing techniques [22] can migrate jobs to less loaded machines. However, they usually operate for resource variations occurring at larger time scales, and are not agile enough for short yet frequent transient unavailability.

Existing high availability (HA) mechanisms in stream processing systems [4], [6], [16], [20] have largely focused on *fail-stop* events and neglected transient failures. There are generally two basic HA approaches, namely *active standby* (AS) and *passive standby* (PS). In active standby, two or more copies of a 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.

While AS and PS have been widely used in many stream processing systems, little is known about their effectiveness and performance under transient failures caused by the aforementioned interdependencies among jobs. AS should be able to handle them as long as the primary and secondary machines each has a diversified mixture of jobs such that load spikes do not occur concurrently. However, AS doubles the processing resources and can quadruple the traffic amount. While requiring less processing resources and traffic amount, PS has much higher detection delay and may not be fast enough to react to transient failures; higher detection delay increases the overall end-to-end application delay significantly.

In this paper, we propose a hybrid HA method that combines the advantages of both active and passive standby approaches: The system behaves like passive standby during normal conditions and uses smaller amounts of computing and bandwidth resources.[2] When a transient failure is detected, it quickly switches to the active standby mode by activating a predeployed secondary copy that was in "suspension". Once the primary copy becomes responsive again (e.g., after a load spike is over), the system rolls back to the passive standby mode. Thus the system incurs small overhead most of the time, while providing fast recovery during failures and unavailability.

We have implemented our hybrid HA method, and have evaluated its performance through extensive experiments. Results show that the hybrid method can reduce recovery time to about 1/3 of that for passive standby, while incurring at least 80% less message overhead than active standby. As such, our hybrid method presents a favorable performance / cost tradeoff for transient failures, suitable for agile reaction without paying the high premium of active standby.

The hybrid approach utilizes a *sweeping* checkpointing mechanism that we proposed in our earlier work [11]. The focus of that work was to empirically analyze the performance and cost tradeoffs of AS and PS under fail-stop events, and to prove the correctness of sweeping checkpointing against multiple, consecutive and concurrent PE failures. It did not address the transient unavailability problem that we target in this work.

We have made three contributions in this work. First, to the best of our knowledge, we are the first to identify transient failures, study their impact on application performance and the suitability of traditional AS and PS solutions in the context of stream processing. Second, we have designed a novel hybrid HA method that addresses both fail-stop and transient failures. It provides fast failure recovery and small system overhead that is particularly suitable for handling transient failures. Third, we have evaluated the performance and cost of the hybrid method using a real stream processing prototype, and compared with those of traditional AS and PS.

The rest of the paper is organized as follows: Section II presents the stream processing model, our measurements on transient failures, and the goals of our investigation. In Section III, we briefly summarize the sweeping checkpointing mechanism and highlight its differences to other checkpointing variants. Section IV presents the design of our hybrid HA method. We provide a comprehensive evaluation of AS, PS and Hybrid under transient failures, and compare different detection methods for transient failures in Section V. We compare with related work in Section VI and discuss future work and lessons learned in Section VII. Section VIII concludes the paper.

II. Models, Motivations and Goals

A. Stream Processing Model

We briefly describe the system model we assume for this work. The stream processing is done in a shared environment where multiple users can submit jobs. A job consists of multiple processing elements (PEs) which can be placed on different machines. The subset of a job's PEs running on the same machine constitutes a subjob of that job. A PE takes input data from possibly multiple input queues, processes them and produces output data into possibly multiple output queues. In addition to the data in input/output queues, a PE can maintain internal processing states. For these stateful PEs, it is critical to correctly restore the processing states during failure recovery.

The system typically has a scheduling component [22] in the system that determines the placement of PEs on machines based on their respective resource requirements and availability. When the resource available on a machine or the resource requirement of a running subjob changes significantly and remains stable for an extended period of time, the scheduling component may migrate subjobs across machines to maintain the resource matching. However, the scheduler is not the right place to handle short yet frequent transient failures. The cost of frequent migration can be prohibitively high, and the durations of transient failures may be much shorter than the time to migrate subjobs.

B. Impact and Characteristics of Transient Failures

We have a stream processing environment consisting of over 150 machines. More than 70 developers and researchers routinely develop and test their applications in this shared

[2]For stream processing systems that we study, CPU cycles and network bandwidth are more precious and affect the system performance more than storage, which is relatively abundant. Our previous work [11] has compared the message overhead of AS and PS.

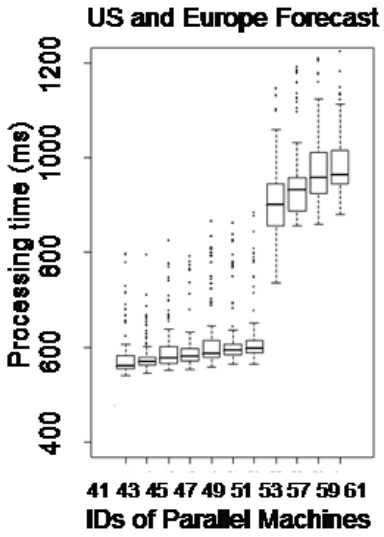

Fig. 1: Impact of transient failures on processing time. About 50% increase in average processing time for machines affected by transient failures.

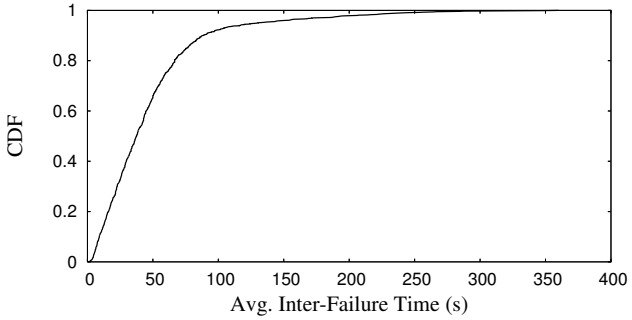

Fig. 2: CDF for transient failure frequency. About 75% machines have transient failures happen more frequently than once every 60 seconds.

environment. Unless reserved, there is no constraint regarding which machines one application can use.

Transient failures can greatly increase the processing time in such a shared environment. We show the processing time of a weather forecast application in Figure 1. It was measured in an uncontrolled experiment where other applications may use the same machines. The application is parallel and runs on multiple machines concurrently. On machines 41-53, it takes around 0.58 second to finish processing; on machines 55-61, it takes about 0.9 second, a 50% increase. Later investigations show that some other applications were using machine 55-61 during the experiment, causing higher overall load, and thus increasing the processing time. Although this 50% increase may seem moderate, our evaluation with controlled background load shows that the average processing time can increase 100% for all data, and 8 fold for those data arriving during transient unavailability (Section V-B). Such degradation is a problem for many delay-sensitive applications.

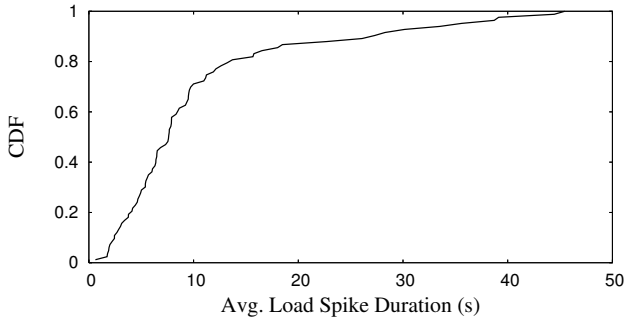

Fig. 3: CDF for transient failure duration. About 80% of them last for less than 15 seconds.

We measured the frequency and length of transient unavailability on 83 machines (excluding infrastructure and special hardware machines). A sample of CPU load was taken every 0.25 s and the measurement continued for 24 hours. All 83 machines exhibited transient unavailability. Using a threshold of 95% CPU utilization to delineate the start and end of transient unavailability, Figure 2 shows the CDF of average inter-failure time. It is clear that transient unavailability indeed occurs frequently. For example, over 75% of machines have transient failures that are more frequently than once every 60 s. Figure 3 shows the CDF of average duration of transient failures on those 83 machines. We can see that they usually last a few seconds (e.g., above 70% last shorter than 10 s); some longer ones (about 20%) can last more than 20 s.

C. Goals and Scope

We have two goals in our investigation. The first one is to understand the suitability of the two basic high availability approaches (AS and PS) to handle transient failures. Using experiments on real testbeds, we seek to gain an in-depth understanding of whether they provide appropriate performance/cost tradeoff against transient failures. Secondly, based on the insights learned, we want to design a new HA method specifically tailored for transient failures, such that applications can enjoy short recovery time without paying excessive overhead. The method should work for both stateful and stateless PEs, under transient failures and fail-stop events. It should produce the same results for deterministic PEs, and guarantee no loss of data (but possibly different results) for non-deterministic PEs, under single or multiple consecutive failures. We do not address disastrous scenarios such as the power outage of the whole cluster.

III. A Low-Cost Passive Standby Method

Earlier analytical comparisons [13] between active standby and passive standby have shown that AS is always superior to PS: PS has almost the same overhead as AS, while taking at least 50% more time in recovery. Contrary to this earlier result, we proposed a *sweeping checkpointing* mechanism [11] that gives PS a magnitude lower overhead than that of AS. In our hybrid design we adopt this low-cost passive standby method as a building component. In this section, we first define the

operations of active standby and passive standby, then briefly explain the main issues in checkpointing and summarize the sweeping checkpointing. Details on its performance and formal proof of correctness can be found in [11].

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. 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 the states (such as input / output queue data and internal states) of a subjob and sends the checkpoint message to the secondary machine. When the primary machine fails, a secondary copy of the subjob is started using the last known checkpoint. It will reconnect with the upstream and downstream subjobs, and resumes the processing.

A. Main Issues in Checkpointing

The performance of passive standby depends heavily on the checkpointing mechanism. There are three main issues in checkpointing:

What data to include in checkpoint messages. A PE has data in input/output queues, and internal states. To correctly recover a PE, its internal states and/or input/output queues may need to be persisted. In the earlier work [13] they are all checkpointed. Note that the internal states are not the PE's memory image (which can be huge), but information that a PE maintains for processing data. A simple example is a counter value for a PE counting the number of received data elements. These states affect the output and are usually much smaller than the complete memory image.

When to trim (i.e., remove) data from output queues. A PE produces data in output queues and sends them to the input queues of downstream PEs. If an data element has been received and successfully processed by those downstream PEs, it is no longer needed and can be removed from the output queue. However, one cannot remove data immediately after they are received at downstream input queues. This is because the downstream PEs may crash before finish processing all data received in their input queues. Such data must be preserved and reprocessed after the downstream PEs recover from failures.

When to checkpoint for multi-PE subjobs. Given multiple PEs in the same subjob, the timing of checkpointing each PE significantly affects 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. Because checkpointing happens after all PEs are suspended, this method is usually relatively slow. Another option is *individual checkpointing*, where each PE has its own timer to drive its own checkpointing procedure.

B. A Brief Summary of Sweeping Checkpointing

In sweeping checkpointing, a checkpoint message includes *the internal states and output queues, but not input queues*, of a PE. We trim data from an output queue after an *accumulative acknowledgment*, indicating those data have been successfully received and processed, and the resulting states have been checkpointed, at downstream PEs. For each PE, checkpoints happen *immediately after its output queue is trimmed*.

Specifically, an output queue assigns an incremental sequence number to each newly produced data element. When a downstream PE successfully receives certain data, processes them and checkpoints the resulting states, it sends an acknowledgment to each of the upstream output queues. The acknowledgment includes the highest sequence number of those data elements, indicating they are no longer needed. If an output queue sends data to multiple downstream input queues, it removes a data element only when all downstream input queues indicate that data element is no longer needed.

Such a queue trimming method ensures that a data element is either processed (and resulting states checkpointed), or can be retrieved from upstream output queues if recovery is needed. The immediate checkpointing after an output queue trimming ensures that a checkpointing message contains as few data elements in output queues as possible. Excluding input queues from checkpoint messages also significantly reduces the message overhead, because many PEs produce much less analyzed, derived data than the raw data they consume.

In [11], we have formally proved that the sweeping checkpointing method guarantees correct recovery for stateful PEs, even when multiple, consecutive PEs fail concurrently in the system. We have also shown through test-bed evaluation, that it is 4X faster and incurs 10% message overhead than alternative synchronous checkpoint and individual checkpoint methods.

IV. A Hybrid Method for Transient Failures

An effective solution to transient failures requires fast failure detection. Otherwise the application will experience long end-to-end delay because new data cannot be produced before failures are detected and recovery actions are taken. It also demands efficiency, and in particular, small message overhead. We first investigate which failure detection methods are effective, then we propose a hybrid method that combines the advantages of both active and passive standby mechanisms.

A. Fast and Reliable Detection

Since transient failures are usually short, they must be detected fast enough. Otherwise the failures could be over even before any action is taken. The detection should be reliable in the sense that all transient failures should be detected, with minimum numbers of false alarms.

We explored different detection methods, including a conventional heartbeat based method and several other more sophisticated ones, e.g., a "benchmarking" method. Interestingly, we find that the convention wisdom stands out.

The conventional method detects failures through heartbeat misses. The rationale is that when unavailability happens, a

machine will be too busy to respond to heartbeat messages. Usually one monitoring machine sends periodic ping messages to another (e.g., the primary) machine. The latter sends back a reply for each ping. When a threshold (usually 3) number of consecutive replies are missed, a failure is declared.

Another intuitive idea one may have is to measure the processing time or throughput of a PE for detecting transient unavailability. The reason is that an overloaded PE tends to process data more slowly, leading to longer processing time, or lower throughput.

The more sophisticated methods follow the above reasoning. In the "benchmarking" method, we measure the processing delay to detect transient failures. The time for a PE to process a standard set (e.g., 20 or so) of data elements are first measured on an idle machine with the same hardware resources as the deployment machines. That measurement is the benchmark.

At runtime, the PE is configured with the benchmark and embedded with the standard set of data elements. On each machine, a thread monitors the CPU load at fine granularities (e.g., 50 ms) through system calls. When the CPU load exceeds a threshold L_{th}, the thread triggers the PE to process the standard set, and compares the result against the benchmark. If the result exceeds the benchmark by a threshold P_{th}, a detection is declared.

However, when we compared these different ways of failure detection, we had the interesting observation that the simple heartbeating method can be as fast as, and more reliable than, the sophisticated benchmarking one (details in Section V-C). The detection through lower throughput or longer processing is far from straightforward. First, the throughput has natural ups and downs, because the data input rate depends on external data feeds and may not be constant. Second, the throughput depends on the nature of input data and the PE's processing logic. Even a constant input rate does not always lead to a constant output rate. Similarly, the processing time for each data element depends on what information is contained in that data element. It is not constant either. Thus a change in throughput or processing time is not necessarily caused by a transient failure.

The benchmarking method ignores these factors. Given the same resource availability, ideally the embedded data set should have the same processing time. However, we find it to be too sensitive to bursty traffic, which is common in stream processing. This makes it prone to false alarms even under medium CPU load.

With comparable detection delay but much less false alarms, in our system we pair the conventional heartbeating approach with our hybrid method. Nevertheless, the key of our hybrid design is the *speculative switching between AS and PS in face of suspicious failures*; it is compatible with different detection mechanisms, such as the failure prediction mechanisms proposed by Gu et al. [10] which monitor multiple resources. As long as one can detect such transient unavailability quickly and reliably, our hybrid HA method can readily take advantage of it.

B. The Hybrid Approach

Our hybrid method can behave like either passive or active standby depending on failure events. Under normal conditions it operates like PS and incurs low overhead. When it detects a failure, it immediately switches over to the active standby mode. After a transient failure is over, it rolls back to the PS mode. If permanent failures happen, a new secondary machine will be activated. [3] It provides a middle ground between the two approaches, recovering in 30% delay of that of PS while costing about 10% of the overhead of AS.

Different from passive standby where a copy is deployed on demand after a failure, we *pre-deploy* a secondary copy on another machine when the primary subjob is deployed. To avoid consuming CPU cycles, we suspend this job immediately after its deployment. The PE's processing loop is stopped when a flag is set to indicate suspension. When we switch over to active standby, we only need to reset the flag to resume the processing loop. Experiments (Section V-B) show that this takes only 1/4 of the time compared with the option of deploying a subjob on demand.

Instead of storing the checkpoint states on disk, we keep them in memory. Whenever new states come we *refresh the PE memory* directly. Our PE implementation has an interface named *storeJobState(jobState)* to overwrite the old state with the new one. By doing this we avoid the expensive disk I/O operations for storing/retrieving states. Note again that the state is not the complete memory image of a PE, but output queue data and internal states included in the checkpointing message.

To further reduce the delay of activating the secondary copy, we apply *early connection*. We connect the output queues of upstream subjobs to the input queues of pre-deployed subjobs on the secondary machine, yet without actually sending data to them. Such connections have an *isActive* field set to *false*. An output queue does not send data to such inactive downstream input queues, which avoids duplicate processing that consumes CPU cycles. When switching happens, we just need to set that field to *true*. Again, experiments (Section V-B) show a reduction of about 50% in latency compared to the option of establishing connections on-demand.

After a transient failure passes, the primary copy resumes and the system will rollback. The secondary copy is suspended again, with upstream connection field set back to *false*. For fail-stop failures, after a threshold time of unresponsiveness from the primary machine, the secondary copy becomes a new primary copy, and another secondary copy is instantiated.

During transient unavailability, the primary copy might be processing data much more slowly than the secondary copy. Under high data rates, the secondary copy may have already processed a significant amount of data when the primary copy comes back. It may take long time for the primary to finish processing all backlogged data. To speed up the resumption of processing, we apply *Read State on Rollback:* the job on

[3] In this paper we do not address the problem of selecting secondary machines, which has been studied in [19].

the primary machine will read states back from the suspended secondary copy, including output queues and internal states. Thus it can "jump" to the latest state directly.

Due to the lack of *rollback* capability, conventional passive standby suffers from the high latency of disconnecting from the primary copy and completely switching to the secondary copy when there are frequent false alarms. It does not trigger any action until it is very confident that failures have happened (e.g., three heartbeat misses). We find this to be a main cause for slow reaction under transient unavailability. In contrast, our hybrid method can afford false alarms to certain extent, because it can quickly roll back to the normal mode. To minimize the recovery delay, we decide to trigger action after the *first* heartbeat miss. The downside is unnecessary switching and rollback upon false alarms, which we find acceptable with the heartbeat based detection.

Experiments show that heartbeating is a reasonably reliable method. For example, we find that with a heartbeat interval of 110 ms, and the CPU usage around 60%, a false alarm occurs once every 11 minutes on average. A detailed study is presented in Section V-C. This ensures that false alarms happen rarely, thus the cost of unnecessary switching and rollback is small.

V. Evaluation

A. Experimental Environment and Methodology

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 [5] work. It can support four different HA modes for each subjob: **No HA (NONE)**, where a single copy is deployed and no action is taken when failures happen; **Active Standby (AS)**, where two copies are deployed and both process data and send output downstream; **Passive Standby (PS)**, where another machine runs the secondary copy upon failures; and **Hybrid**, where another machine has a pre-deployed but suspended secondary copy. Each subjob in the same job can use a different HA mode.

Each PE has three interfaces that collectively implement the checkpointing operations. A Checkpoint Manager (CM) is responsible for checkpointing the states of PEs. Driven by a timer (synchronous or individual checkpointing) or output queue trimming events (sweeping checkpointing), it calls a PE's $pause(controller)$ method to suspend it, giving its own interface $controller$ as the callback parameter. When the PE has suspended, it calls the $ackPePause()$ method of the CM. The controller will call the $checkpoint()$ method of the PE to obtain its internal state, variables that affect the output, but not the complete memory image of the PE. After storing the state on the secondary machine, the controller calls the $resume()$ method to resume the PE.

The experiments are performed on a cluster of Redhat Enterprise Linux 4.4 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 logics, so the difference in results can be exclusively attributed to different HA approaches.

To generate transient failure load on a machine, we run a computation-intensive program that can be parameterized to take approximately a required share of CPU. By starting and stopping the program at different times, we can impose both regular and Poisson arrivals of such failures. The average inter-arrival time and failure length are tunable. We generate transient failures on all primary machines except the first one in the chain, since it is also where stream input is generated and transient failures can cause unstable input data rates. Unless specified, each data point in the figures is the average of 5 independent runs and each run lasts for 100 seconds.

B. Performance against Transient Failures

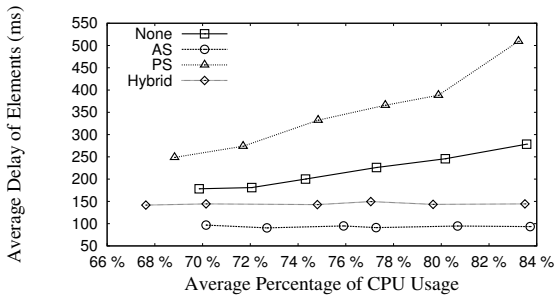

Fig. 4: Average element delay under transient failures

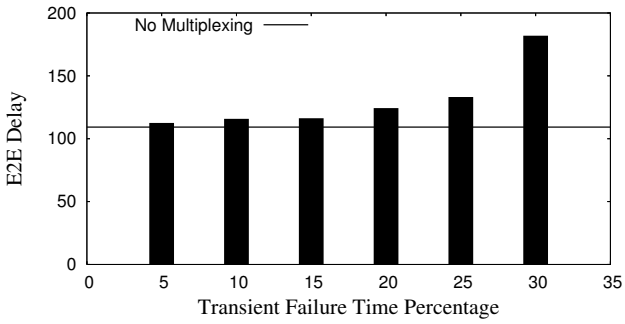

Fig. 5: E2E Delay vs. percentage of transient failure time

Average End to End Delay To evaluate the benefits of the Hybrid method against transient failures, we apply NONE, PS, AS and Hybrid on one subjob, and generate independent transient failure loads of the same severity and distribution on the subjob's primary and secondary machines. We measure the average end-to-end delay of data elements. During each transient failure, the overall CPU usage is increased from

60% (used by the application) to $95\% \sim 100\%$. We set the checkpoint interval at 500 ms, heartbeat interval at 100 ms, and incoming data rate at 10K elements/s. To simulate different degrees of severity of transient failures, we vary the fraction of time when transient failures are present. A large fraction leads to more performance degradation. We vary the fraction between $30\% \sim 80\%$.

Figure 4 shows the average end-to-end delay of data elements as the average CPU utilization varies from 65% to 85%. AS has the lowest delay (about $90ms$) and remains relatively stable; this is because the downstream subjobs receive data from both copies and will use whichever arriving first. Since the transient failures on the primary and secondary machines are uncorrelated, usually one of them can process the same data at normal speeds.

For both NONE and PS, the delay increases about linearly as transient failures become more severe, with PS having higher delays. For NONE, the increase is due to the increased severity of transient failures on the primary machine. For PS, because it does not roll back, the subjob experiences the same degree of transient failures before (on the primary) and after (on the secondary) the switching. The higher delay was caused by the extra overhead in the slow detection and migration of subjobs to the secondary machine.

Because the transient failures on the primary and secondary machines are independent, Hybrid can switch to the secondary machine when transient failures occur on the primary machine, or switch back to the primary machine when they are gone. Thus it can almost always use the machine without transient failures. The fast switching also ensures the extra overhead is minimum. Thus its delay remains flat and is smaller than that of NONE and PS. It is higher than that of AS because the switching still takes some time and happens more than once, thus increasing the delay for some data elements.

As a final note for this experiment, we observe transient failures indeed severely slow down the processing and increase delay. For example, at CPU utilization of 85%, the E2E delay during transient failure periods increase more than 8-fold on average.

Multiplexing Gains One major advantage of our hybrid method is to allow multiple primary machines to share one secondary machine. As long as transient failures on these primary machines are not correlated (the scheduler can help avoid deploying correlated subjobs on these primary machines), most of the time the secondary needs to run one subjob.

To evaluate the benefits of such multiplexing, we let three primary machines share one secondary machine, generate failure load on the primary machines only and vary its severity. Figure 5 shows how the E2E delay changes as the fraction of time with transient failures increases from 5% to 30%. The maximum fraction is 30% because that is when on average, at least one of the three primary machines will experience transient failures and have an active subjob running on the shared secondary machine. The horizontal line is the E2E delay when each machine has a dedicated secondary machine.

The hybrid method has very small increase in E2E delay as transient failures become more frequent. The delay increases less then 25% as transient failures become present for up to 20% of the time (on average for 60% of the time an active subjob is running on the secondary machine). The increase becomes significant (about 80%) only when transient failures are present for 30% of the time for each primary machine. This is because failures on different machines, even uncorrelated, are likely to have some overlap when they become more frequent. Thus the secondary may need to run more than one subjob at some time, increasing the E2E delay.

Traffic Amount We compare the amount of traffic in the number of elements transmitted under different HA policies. This helps us understand the cost they pay to achieve the previous performance. We vary the data rate from 1K to 25K elements/s. For PS and Hybrid, we set the checkpoint interval to 500 ms. [4] The PE's internal state is set to have a size of 20 data elements.

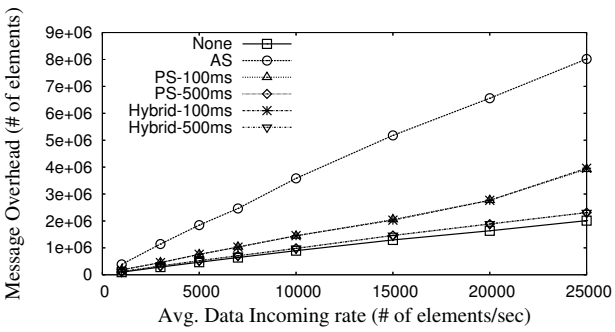

Fig. 6: Overhead vs. Data Rate

Figure 6 shows that the total traffic amount under AS is around four times of that under NONE. This is because with two copies for each subjob, both the primary and the secondary send two copies of each message to the two downstream subjobs, leading to a 4X increase of traffic. For both PS and Hybrid, the increased traffic amount is only around 10% due to the sweeping checkpointing. The results show that the Hybrid method only incurs marginal overhead in terms of redundant messages.

Recovery Time Comparison We then compare the failure recovery time, defined as the time from the inception of a transient failure to the producing of the first new output data after the switch, for PS and Hybrid. The recovery delay for PS consists of three parts: *failure detection*, *job redeployment*, and *data retransmission/reprocessing* (reprocess data sent to the primary subjob but whose results are not produced yet due to failures). Since secondary jobs are pre-deployed in Hybrid, it does not have the *job redeployment* stage, but instead a *job resume* stage.

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. The failure detection time of Hybrid is about 1/3 of that of PS, and both

[4]We tried larger intervals, but found they have similar results to that of 500 ms.

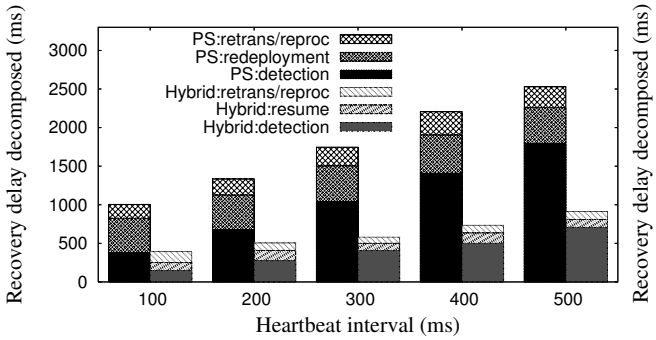

Fig. 7: Recovery time decomposition vs. Heartbeat interval

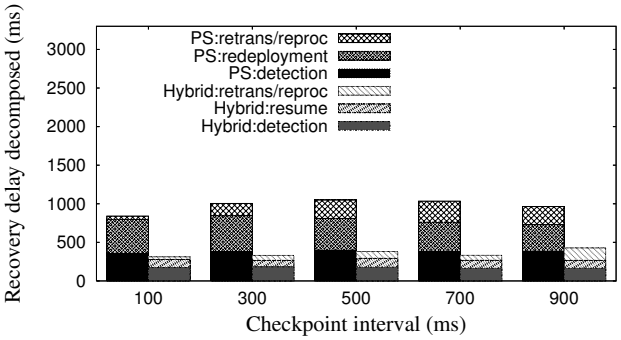

Fig. 8: Recovery time decomposition vs. Checkpoint interval

increases about linearly respect to the heartbeat interval. This is because the detection time dominates the total recovery, and it is a multiple of heartbeat intervals (1 and 3 for Hybrid and PS), thus increasing linearly. The subjob resume time for Hybrid and deployment time for PS remain about the same, simply because they do not depend on the heartbeat interval.

Figure 8 shows the impact of the checkpoint interval with a fixed heartbeat interval of 100 ms. With larger checkpoint intervals there are more data to retransmit and reprocess. Thus 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 and 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 affects the amount of data retransmission.

Gains of the Hybrid Optimization Techniques From Figure 7 and 8, we can see the gains of some hybrid optimization techniques in Section IV-B. Due to the pre-deployment, we reduce the average redeployment time from 464.61 ms in PS to 112.82 ms in hybrid for resuming the suspended secondary job, which is a 75% reduction. The early connection creation reduces the retransmission/reprocessing time from 239.54 ms in PS to 104.32 ms in hybrid. The speedup of state read back depends on the duration of transient failures. If the primary is almost stopped, the reduction is the time it takes to process all the data arriving during the failure, which can be the failure duration (e.g., $\sim$10s) when data rates are high.

Switching Overheads for Hybrid We study the time it takes Hybrid to switch over and rollback, and accompany message overhead, consisting of data sent to the primary machine during a failure and the the size of states the primary needs to read while rolling back. We vary the data rate, and overload the primary machine to make it unavailable for periods of 5 seconds and 10 seconds.

In switch-over, most of the time is spent to resume the pre-deployed copy and to activate the corresponding connection. Figure 9 shows that overall time increases only slightly. For example, for 5 sec unavailability duration, the time increases about 20%, from data rate of 1000 elements/s to 7000 elements/s. Of the two components, the switch-over time is relative stable across various data rates for both unavailability durations. The other component, rollback time, mainly includes the time for the read-state action and becomes larger with increasing data rate. Such increase is due to more elements in the input/output queues of secondary jobs under higher data rates, thus more time in reading back.

Figure 10 shows that the message overhead increases linearly with higher data rate for both unavailability durations; it is roughly equal to data rate times unavailability duration. This shows that the message overhead is dominated by the data elements sent to the unresponsive primary machine. The overhead for reading back state therefore is small. From these two figures, we can see that the Hybrid switches and rolls back very fast, and incurs small message overhead.

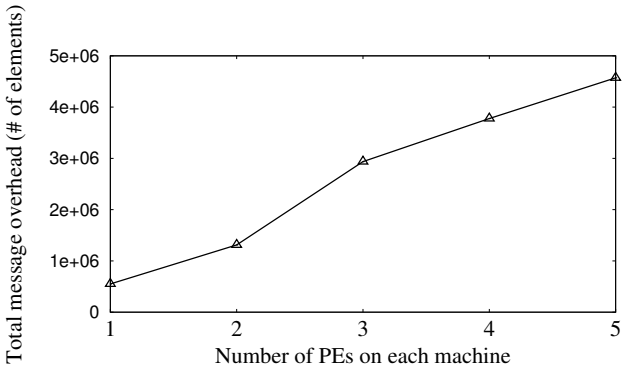

Fig. 11: Overhead vs. Number of PEs per Machine

We also study how the number of PEs on a machine affect the total amount of message overhead in Hybrid. Figure 11 shows that the overhead increases about linearly as the number of PEs on each machine increases. This is simply because each additional PE adds its own, and relatively stable overhead in checkpointing messages.

C. Transient Failure Detection

We compare the background load detection ratio and false alarm ratio (Figure 12 and Figure 13) of benchmarking and heartbeat based failure detection methods against transient failures. During the experiment we set the heartbeat interval to 110 ms and periodically generate over 200 transient load

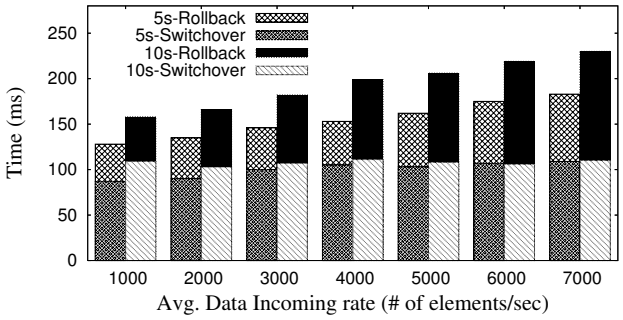

Fig. 9: Switchover and Rollback Time vs. Data Rate

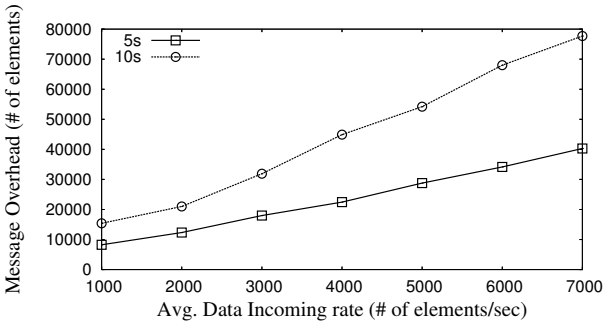

Fig. 10: Switchover and Rollback Message Overhead vs. Data Rate

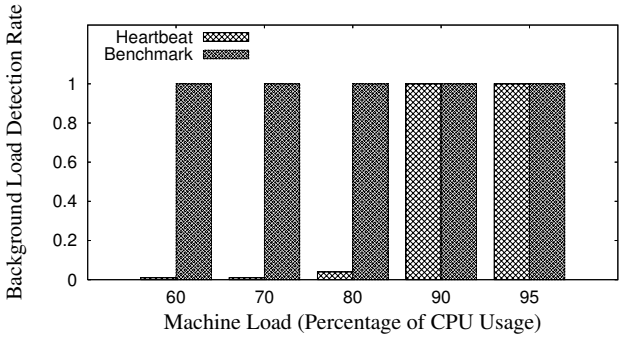

Fig. 12: Failure Detection Ratio vs. Machine Load

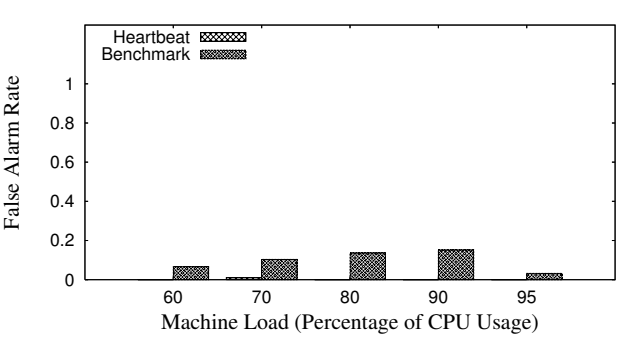

Fig. 13: False Alarm Ratio vs. Machine Load

increases that gives a certain background machine load.[5] The experiment is repeated from 60% to 95% machine load.

Background load detection ratio is defined as the ratio of the number of detected CPU load increases versus the total number of generated background machine loads. This metric indicates the sensitivity of a failure detection method. Generally, for an ideal detection method, background load detection ratio should be close to 0 when the machine load is small and the normal operation of the application is not affected. On the other hand, when the background load consumes a significant amount of CPU cycles and affect the normal operation of the application, this ratio should go toward 1. From Figure 12, we can see that benchmarking is overly sensitive to the background load. Even at relatively low machine load of 60% (normal processing is not affected), benchmarking still declares all generated machine loads. In contrast, heartbeat exhibits a close to 1 detection ratio at high background machine loads ($\geq$ 90%) and much lower ratios when background load is low.

The second metric we use to compare failure detection methods is false alarm ratio, which is defined as the fraction of incorrectly declared transient load spikes (no background machine load) among all those declared. This metric denotes the reliability of failure detection methods and this metric should be close to 0 for a reliable detection method. From Figure 13, we can see the false alarm ratio for benchmarking is also fairly high, which exceeds 15% even at 90% machine load. Heartbeat method, on the other hand, maintains very low false alarm ratio at all background loads. Such low false alarm ratios minimize unnecessary switch-over in Hybrid, allowing it to maintain almost identical system overhead as that of passive standby. At high background machine loads, the high detection ratios for heartbeat allows Hybrid to reliably react to transient failures.

Besides detection and false alarm ratios, we also studied the average detection delay, defined as the time from a load spike happening to a detection declared. Our measurement shows that heartbeat has an average detection delay of 143.58 ms, only slightly longer than that of benchmarking, 132.14 ms. With comparable detection delay and rate, but much less false alarms, we decide that heartbeat is a reasonable choice for transient failure detection.

VI. Related Work

High availability for stream processing systems has become an active research topic in recent years. Representative work includes those [4], [12], [16], [15] proposed in the context of Borealis [1], one of the first stream processing systems. Except [16], the other three are all based on active standby. [4] achieves flexible trade-off between availability and consistency by introducing tentative data concept; [12] has the highest availability by paying the cost of having multiple upstream copies sending data to multiple downstream copies;

[5]We tried heartbeat interval smaller than 100 ms but found they have higher false alarm ratio.

[15] allows replicas to execute without coordination but still produce consistent results; [16] 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; they do not study how to address the transient unavailability problem, or the suitability of the two approaches against such unavailability.

High availability has been studied in other stream processing and data flow systems. [17], [19] are in the context of System S [2], a stream processing system developed at IBM Research. [17] 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 have different requirements due to load spikes and tight coupling of subjobs across machines. [19] 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.* [20] 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.

Despite the prevalence of active and passive standby approaches in stream processing systems, there is not much work to systematically compare their performance tradeoff, especially different variants. As far as we are aware, Hwang et al.'s work [13], [14] is the only exception where their 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, they do not investigate their suitability under transient unavailability, which is the focus of this paper.

Checkpointing is a widely used technique for preserving critical state information. [8] provides a comprehensive summary and comparison of different checkpointing techniques. AQuA [18] achieves adaptive fault-tolerance through intelligent object replication according to user specified level of dependability. The sweeping checkpoint method we use in the system bears certain similarities to the Efficient Coordinated Operator Checkpointing (ECOC) proposed in [6]. However, our scheme is simpler than the two-phase protocol used by ECOC, and thus is more efficient and leads to the magnitude of lower overhead compared to conventional checkpointing variants.

The *upstream backup* method proposed in [13] 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. In contrast, we use sweeping checkpointing in the hybrid method such that the internal PE states can be checkpointed and recovered.

Our hybrid approach uses a simple form of failure detection: heartbeat messages. Other failure detection / prediction mechanisms have been proposed, mostly for fail-stop events. [23] summarizes a number of different failure detection techniques; Gu *et al.* [10] proposed online methods to predict failures for stream processing systems. Our hybrid approach can also work with these more advanced mechanisms, if they are shown to be suitable for fast and reliable detection of transient failures.

VII. Discussions

We discuss the limitation of our work and some lessons we learned during the investigation. Transient unavailability may happen due to squeeze of resources other than CPU, such as memory or disk I/O. Our study has focused on CPU because it is the most common phenomena we observe in our cluster environment. This does not preclude the possibility of other types of resource unavailability. Their characteristics and impact remain to be studied. However, we believe the basic idea of our approach, which is to adaptively switch between aggressive and conservative high availability methods in "suspicious" scenarios, is widely applicable.

The hybrid method refreshes the states of the secondary subjob copy directly in memory. Although this leads to faster checkpointing, the state can be lost when both the secondary and primary machines fail. If handling the failure of both is a goal, the state has to be persisted to a permanent storage, i.e., a disk. Some penalty in performance is expected.

We also require PEs to have certain interfaces to support a number of operations, such as suspension/resuming, reading/writing states. This is appropriate for new PEs that are developed with such support. For legacy PEs that do not have such interfaces, the hybrid method cannot be directly applied.

Due to the time constraint, our evaluation uses synthesized PE processing and transient failure load instead of real applications and failure load. Performance metrics such as transmission overhead and processing delay can be affected by application logic and real failure load. It would allow us to better understand the impact of actual transient failures on real applications. We would like to perform such evaluation in the future. We also want to study more complex PE topologies such as trees in addition to the chain structure studied in this work. There are interesting issues like how to trim multiple output queues around the same time to minimize checkpoint overhead.

The sweeping checkpointing reduces overhead mainly because it reduces the amount of output queue data to checkpoint. 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. The sweeping checkpointing may not achieve as good results in those cases.

The reliability and speed of heartbeat detection comes as a surprise. Initially it seems that by monitoring the processing

time or instantaneous throughput at fine time granularities, we can obtain better detection performance. However, experiments show that it is far from that straightforward. A simpler mechanism that relies on less assumptions usually applies to a wider range of circumstances. Thus that "conventional wisdom" indeed has a reason for its popularity. Nevertheless, we do not expect we have exhausted the potential space of detection mechanisms. We plan to study how suitable some other proposals are for transient unavailability and apply them for the hybrid method if appropriate.

VIII. Conclusion

High availability is essential for many stream processing applications. Transient unavailability due to suddenly increased data rates or processing intensity can cause significantly increased delay and reduced throughput on a shared infrastructure. Most of the stream processing high availability work deal with fail-stop. They do not provide fast reaction at low overhead, as required by the characteristics of transient unavailability.

We propose a hybrid method that achieves much faster recovery than passive standby, at a fraction of the overhead of active standby. It utilizes a sweeping checkpointing method that reduces the checkpointing overhead by one order of magnitude. Our hybrid method opens the door for many applications that desire fast recovery but cannot afford the premium of active standby.

References

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

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

[3] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 350, Washington, DC, USA, 2004. IEEE Computer Society.

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

[5] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, and F. Ye. 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.

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

[7] F. Douglis, M. Branson, K. Hildrum, B. Rong, and F. Ye. Multi-site cooperative data stream analysis. *SIGOPS Oper. Syst. Rev.*, 40(3):31–37, 2006.

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

[9] P. Gibbons. Data-rich computing: Where it's at. In *Hadoop Summit and Data-Intensive Computing Symposium*, 2009.

[10] X. Gu, S. Papadimitriou, P. Yu, and S.-P. Chang. Online failure forecast for fault-tolerant data stream processing. In *ICDCS '08: Proceedings of the 28th International Conference on Distributed Computing Systems*, 2008.

[11] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. An empirical study of high availability in stream processing systems. In *Middleware '09: the 10th ACM/IFIP/USENIX International Conference on Middleware (Industrial Track)*, 2009.

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

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

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

[15] J.-H. Hwang, S. Cha, U. Cetintemel, and S. Zdonik. 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*, pages 1303–1306, New York, NY, USA, 2008. ACM.

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

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

[18] Y. J. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. Aqua: An adaptive architecture that provides dependable distributed objects. *IEEE Trans. Comput.*, 52(1):31–50, 2003.

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

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

[21] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

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

[23] S. Q. Zhuang, D. Geels, I. Stoica, and R. H. Katz. On failure detection algorithms in overlay networks. In *INFOCOM 2005: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2112–2123, 2005.