

Fault Injection-based Assessment of Partial Fault Tolerance in Stream Processing Applications

Gabriela Jacques-Silva[†]♣, Buğra Gedik♣, Henrique Andrade♣*,
Kun-Lung Wu♣, Ravishankar K. Iyer[†]

[†]Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL
rkiyer@illinois.edu

♣Thomas J. Watson Research Center
IBM Research
19 Skyline Dr., Hawthorne, NY
{g.jacques,bgedik,klwu}@us.ibm.com

ABSTRACT

This paper describes an experimental methodology used to evaluate the effectiveness of partial fault tolerance (PFT) techniques in data stream processing applications. Without a clear understanding of the impact of faults on the quality of the application output, applying PFT techniques in practice is not viable. We assess the impact of PFT by injecting faults into a synthetic financial engineering application running on top of IBM's stream processing middleware, System S. The application output quality degradation is evaluated via an application-specific output score function. In addition, we propose four metrics that are aimed at assessing the impact of faults in different stream operators of the application flow graph with respect to *predictability* and *availability*. These metrics help the developer to decide where in the application he should place redundant resources. We show that PFT is indeed viable, which opens the way for considerably reducing the resource consumption when compared to fully consistent replicas.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Error handling and recovery*; H.2.4 [Database Management]: Systems—*Distributed databases*

General Terms

Reliability, experimentation

1. INTRODUCTION

Stream processing applications *continuously analyze* the heterogeneous incoming data and perform a variety of computations over generally incomplete information [5]. High availability is a key requirement of stream processing systems [25], since they process continuous live data. Typically,

*Currently employed by Goldman Sachs.
Email: henrique.c.m.andrade@gmail.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS'11, July 11–15, 2011, New York, New York, USA.
Copyright 2011 ACM 978-1-4503-0423-8/11/07 ...\$10.00.

streaming applications are designed by assembling *stream operators* as data flow graphs, which can be distributed over a set of nodes to achieve high performance and scalability [12]. A fault in a computing node or in a stream operator can result in massive data loss due to the typical high data rates of incoming streams.

Although many fault tolerance techniques for stream computing [13, 16, 19] provide guarantees that no data is lost or any inconsistency exists (e.g., duplicate delivery of the same data item, which we refer to as data duplication), these methods usually cause significant degradation in performance. Aiming at reducing such performance overhead, partial fault tolerance (PFT) techniques [4, 17, 21, 23, 30] have been proposed. These techniques assume that data loss and data duplicates are acceptable under faulty conditions. The rationale is that many streaming applications tolerate data imprecision by design, and, as a result, can still achieve correctness without using stringent fault tolerance methods.

While all of the above techniques require a careful evaluation to assess the fault tolerance achieved and the resulting performance degradation, this is especially true for methods that provide PFT. Hence, the use of PFT is not viable without a clear understanding of the impact of faults on the application output.

This paper describes a fault injection-based [14] experimental methodology to evaluate the effectiveness of PFT techniques in streaming applications. Our first goal is *to assess the impact of PFT on the output of an application*. We use a fault injector to mimic the effect of a fault in the application when a specific PFT mechanism is in place. To the best of our knowledge, we are the first to describe a thorough fault injection experiment that evaluates PFT techniques in a streaming application. Our second goal is *to characterize how each stream operator used by the application behaves under failures*, so the developer can decide if the PFT technique in place is adequate for the target operator. We characterize each operator by calculating *four evaluation metrics* over the application output generated during the injection trials. In addition, the metrics can be used to identify which operators are most *critical* to the application output quality. Prioritizing most critical operators when protecting an application with PFT methods leads to lower resource utilization while maintaining output quality.

Analyzing the impact of faults in streaming applications has many challenges that are not addressed by traditional fault injection methodologies [8, 14, 20, 29]. Streaming

applications can tolerate approximate results and be non-deterministic. In addition, these applications produce results continually, requiring a careful analysis of the output to estimate the fault impact. We address these issues by (i) defining an *output score function* (OSF) to measure the application output quality and compare it with the output under faults, and (ii) using the OSF over limited sections of the output to compute our proposed evaluation metrics.

Our fault injection experiment target is a simplified financial engineering application running on top of IBM’s stream processing middleware, System S [1, 12]. We considered a bursty data loss fault model, which emulates the crash of a stream operator. Our results show the following: (i) the impact on the application output quality varies widely for faults in different stream operators, demonstrating that operator sensitivity to faults is an important differentiator in deploying PFT; (ii) the tested application provided some surprising results; specifically, one stream operator with high selectivity turned out to be the least critical in terms of quality degradation when subjected to data loss. This indicates that PFT can be a powerful technique to maintain the accuracy of the results and preserve computing resources by replicating only parts of the application.

2. PARTIAL FAULT TOLERANCE

Several researchers [4, 15, 17, 21, 22, 23, 30] have described PFT techniques that are applicable to stream processing applications. While many leverage the partial and often strategic employment of fault tolerance techniques to lower the performance loss, no technique can guarantee *perfect* application output under faulty conditions. Different PFT mechanisms have different *effects* on the *input/output stream* of the failed operator and, as a result, on the application output. Hence there is a need to conduct experiments to understand the behavior of streaming applications under faults.

One particular technique [17] is based on a stream operator checkpoint mechanism that leverages code generation to automatically provide specialized state serialization methods depending on the stream operator type. When an operator fails, its upstream operators do not buffer outgoing tuples¹ unless they are required to produce a semantically correct result after a fault. Checkpointing [17] results in *bursty tuple loss* on the operator *input stream*.

Another example of PFT is to employ *free running replicas*, as proposed by Murty and Welsh [21]. This technique does not enforce determinism among the stream operator replicas, resulting in different effects on the operator *output stream*, such as *tuple reordering*, *duplication*, *loss*, and *value divergence*.

S4 [22], the open-source distributed stream processing middleware from Yahoo!, provides fault tolerance by restarting an operator from scratch once a failure is detected. The state of the operator is rebuilt by processing new incoming stream data items. This approach is similar to the strategy defined in [15] as gap recovery. Both techniques lead to *bursty tuple loss* on the occurrence of failures.

To evaluate how applications behave under PFT techniques, it is critical to understand the *effect* of faults on input/output stream of the operator. Note that we are not concerned with the specific mechanism used by the stream

processing middleware to detect a fault in a stream operator (e.g., heartbeats) and to restart the operator. Nevertheless, these detection and recovery times are important to determine the duration of the fault effect.

3. EVALUATION METHODOLOGY

The experimental methodology uses fault injection to characterize the output quality of a streaming application in the presence of faults while a specific PFT mechanism is in use. We assume that a fault detector and a fault tolerance mechanism are already in place and have been validated. We also assume that a stream operator fails by crashing in a fail-stop manner (i.e., without producing corrupted results) [6]. The selected fault model is broad because operator failures could be due to several distinct and indirect causes: a node failure (e.g., operating system kernel crash), a transient software fault (e.g., race condition), or a transient hardware fault (e.g., a memory bit flip that causes a process crash).

Characterizing the error behavior of streaming applications presents many challenges. Traditional fault injection methodologies evaluate the behavior of an application by checking its output after the injection of a fault [20, 29]. A deterministic application can be checked for correct behavior under faults by comparing the output of the faulty run to the output of the fault-free run (also called the *golden run*) [8]. In streaming applications, checking the *correct behavior* cannot be done by a simple bit-by-bit comparison of the faulty and fault-free outputs. Such applications are often non-deterministic², and are typically able to tolerate imprecise results.

To understand how a fault affects the application output, we compute an application-specific *output score function* (OSF). The OSF is calculated over a set of tuples of the output of the application, and is applied for the faulty and the golden output. Figure 1 shows two samples of streaming application outputs [1, 28]. Figure 1(a) shows a sample output of our target financial application described in Section 6.1. The output contains the *ticker* symbol of a company in the stock market and the projected *financial gain* obtained by buying a stock at a specific time. In this case, the OSF is a summation of the *financial gain* (i.e., 101.10). Figure 1(b) shows a sample output of a chip manufacturing application [28]. The output contains the *predicted wafer yield* of a manufacturing process, and a *prediction error*, indicating the extent to which the prediction deviates from the ground truth. In this case, the OSF is the *prediction error* average (i.e., 0.06). The result of the defined OSF over the application output data must have little variation (i.e., within error thresholds) when considering different non-faulty execution trials of a non-deterministic application.

To assess the *quality* of the output under faults, we define a *quality score* (QS), which is the *ratio* of the average OSF calculated over the faulty output (i.e., the application output produced in the presence of an operator fault) by the average OSF calculated over the golden output. This ratio estimates the fractional deviation of the faulty result and the correct result. The OSF average is obtained by executing the application multiple times with the same configuration (e.g., same injected fault). The average accounts

²A source of non-determinism is the multiplexing of multiple streams, where the relative order of tuple arrival to an operator is arbitrary.

¹A tuple is the basic unit of data in a stream.

Ticker	Financial gain	Predicted wafer yield	Prediction error
IBM	41.66	95	0.06
BAC	30.56	98	0.02
TWX	28.88	92	0.10

(a)

(b)

Figure 1: Examples of output for a financial (a) and a chip manufacturing (b) application. The OSF is applied over the application result (e.g., financial gain or prediction error values).

for the stochastic deviations on output caused by the application non-determinism. The average OSF and the QS are the basis for the proposed evaluation metrics. In addition, our metrics consider that stream operators can fail at different execution times, and that the time to detect and repair from such failure can vary. Figure 2 illustrates a possible failure scenario. Different execution times are represented in our methodology by the injection of faults into operators at different *stream offsets* from the beginning of the input stream. Different detection and repair times are represented by the injection of faults with different *outage durations*. We do not make assumptions about the operator failure probability distribution.

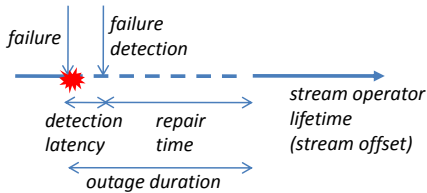


Figure 2: Timeline of a stream operator failure. The failure is detected after an average detection latency, and the detector triggers a repair procedure. The outage duration comprises the time to detect the failure and the repair time.

Our evaluation metrics characterize each operator in the following terms. (i) The *outage duration impact*, defined as the correlation coefficient between the outage duration and the QS computed over the part of the output stream affected by the fault. If the correlation coefficient is high, there is a direct quality improvement by applying techniques with lower recovery time for the target operator. (ii) The *data dependency*, defined as the standard deviation and analysis of variance (ANOVA) test of the QS obtained under faults injected into different stream offsets. A high standard deviation and rejected ANOVA test indicates that the impact of the fault on the quality is dependent on the specific data affected by the fault. (iii) The *recovery time*, defined as the P percentile of the QS observations over time that fall outside an error *threshold*. A high value for this metric indicates that the application takes a long time to stabilize and to start producing correct results again. (iv) The *quality impact*, defined as the sum of squares of the difference between the faulty QS and the golden QS evaluated over time. A high sum value indicates that answers produced by the application under faults are far away from the correct result. More details on the computation of each metric can be found in Section 5.2.

Our experimental methodology comprises the following steps:

1. *Choose a fault model according to the in place PFT technique* – the fault model is selected in correspondence to a recovery technique used to restore the operator upon a fault. In our experiments, we consider a *bursty tuple loss* model.
2. *Optimize fault injection target operators* – conditioned on the chosen fault model, only certain operators must be selected and subjected to fault injection. For example, operators that are not sensitive to the selected fault model do not need to be selected as an experimentation target.
3. *Use the actual expected data rate from stream sources to realistically model the effect of a fault* – for example, the knowledge of the real data input rate allows quantifying how much data is dropped when a fault occurs.
4. *Inject faults at different stream offsets and with distinct outage durations* – injecting faults at different stream offsets mimics random fault arrival times during the operator execution. Different outage durations mimics variations on the detection and recovery times;
5. *Evaluate the experimentation results* – based on the OSF function, each operator is characterized using the proposed metrics. These metrics can quantify the relative sensitivity of the operators to faults, and use it as a basis to compare different fault tolerance techniques.

4. FAULT INJECTION FRAMEWORK

To assess the impact of PFT, we built a fault injection framework to emulate the effect of these techniques on the input/output streams of a target stream operator. Currently, the framework supports a bursty tuple loss fault model, but it can be extended to include other fault models (e.g., tuple duplication). Bursty tuple loss can emulate the following situations: (i) a stateless operator crashes and restarts, and no in-flight tuples are saved; (ii) a stateful operator crashes and restores its state from a checkpoint upon restart, and no in-flight tuples are saved. The checkpoint preserves the operator state immediately before the occurrence of the fault; (iii) a stateful/stateless operator crashes and performs a failover to a replica. The operator has only one input stream and the backup replica is operating at approximately the same pace as the primary replica. In addition, injecting bursty tuple loss in the source operator can emulate faults affecting the real stream source (e.g., sensors) and data drop due to bursty data arrival and limited input buffer size. Emulating operator crashes by manipulating its input/output streams allows us to better control the fault injection experiment. Dropping approximately the same set of tuples in each experiment trial helps us to reduce the variation on the OSF result of different experiment trials.

4.1 Emulating Faulty Behavior

Our fault injection framework is designed to work seamlessly with SPADE (Stream Processing Application Declarative Engine) [12], the declarative stream processing language that is part of System S. SPADE offers language extensibility by allowing the implementation of *User-defined Built-in Operators* (UBOPs). When a developer identifies a general-purpose operation, he can describe it as a new type of stream operator, effectively extending the SPADE language.

Our framework uses UBOPs to extend the language with stream operators that mimic the faulty behavior of an operator when using a specific PFT technique. Figure 3 shows

how the framework operates. First, it receives as input a SPADE application, a target operator, a fault model, and its injection parameters. Based on the target operator and the fault model, the framework modifies the SPADE program to embed a *fault injection operator* (FIOP) in specific positions in the data flow graph. For example, to emulate tuple loss at the *input ports* of an operator, all the operators connected to these input ports are re-routed to send their output streams to the FIOP. The FIOP is connected to the target operator. Based on the new flow graph, the framework generates multiple SPADE programs, each of them with a FIOP configured with a different fault injection parameter. After the compilation, the application is ready for fault injection runs.

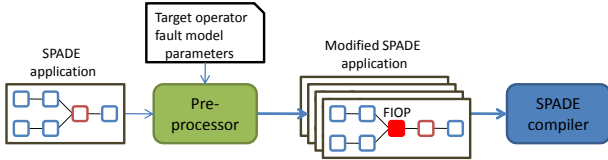


Figure 3: Fault injection framework changes the original application dataflow graph to include a fault injection operator (FIOP). The modified application is re-compiled and then executed with the fault injection parameters.

Figure 4 depicts how the injection occurs at runtime. The figure shows the injection of the *bursty tuple loss* fault model into the input port of operator OP2. In this example, OP1 sends tuples containing a stock symbol and a price to OP2. After the graph pre-processing, OP1 connects to the FIOP, which connects to OP2. The FIOP is placed right before the target operator and receives the following parameters: (i) the *outage duration*, specified in terms of the number of tuples to be dropped, and (ii) the *stream offset*, specified in terms of the number of tuples processed by the operator up until the fault. In Figure 4, the FIOP triggers a fault after it processes the stock symbol IBM at price USD 123.24. The duration of the fault is 2 tuples, leading the FIOP to drop the tuples with stock symbol YHOO and GOOG. After the FIOP drops the specified number of tuples, its operation goes back to normal, i.e., forwarding tuples received by OP1 to OP2.

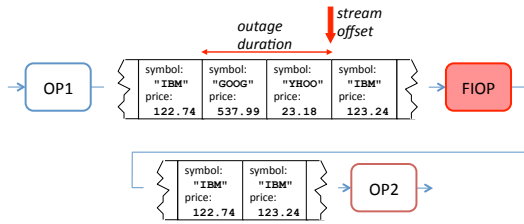


Figure 4: FIOP emulates a bursty tuple loss model by dropping tuples received after a specified *stream offset* and during a configured *outage duration*.

As described earlier in this Section, our framework does not actually crash and restart an operator during the injection. Even though the operator continues to run, it does not send or receive any new tuples for the time corresponding to the fault detection and recovery. As a result, its internal state (if any) remains unchanged.

4.2 Placing Injection Operators

To understand how the application behaves under faults,

in the worst case we may need to inject faults in all operators. However, streaming applications can have a substantially large number of operators. To reduce the number of required fault injection targets when evaluating a target application, the framework pre-analyzes the data flow graph. It selects as injection targets only those operators in which the injected fault results in a behavior that is different than a fault injected into another operator in the graph. An operator not chosen as fault injection target is assumed to have the same behavior as its upstream operator.

For the bursty tuple loss fault model, the inspection starts by selecting all *source operators* as injection targets. Injecting faults into the source mimics a fault affecting the stream feed originated outside the stream processing middleware (e.g., the raw sensor data feed), or the source operator itself.

From each source operator, the analysis continues to all downstream operators by doing a breadth-first traversal, until the *sink* is reached. A bursty tuple loss operator is placed in the data flow graph immediately before a chosen target operator. The framework selects an operator as a target if its position in the data flow graph meets one of the following properties:

1. *An upstream operator produces more than one output stream* – a common pattern in streaming applications is for one operator to have its outputs consumed by more than one operator downstream. As illustrated by Figure 5(a), both OP2 and OP3 consume the stream produced by OP1. If OP1 fails, part of its input stream is lost, affecting both OP2 and OP3. If OP2 fails, OP1 can continue to send data to OP3, but all data sent to OP2, while it is offline, is lost. These two different scenarios can impact the application output in different ways. Therefore, both scenarios must be emulated when evaluating the application behavior under faults.

2. *The operator consumes more than one input stream* – stream operators can consume outputs produced by more than one upstream operator. One such example is the *join* operator, which correlates events coming from two different streams. This is shown in Figure 5(b), where OP1 and OP2 send data to OP3. If OP1 fails, OP3 stops receiving data from one of its input ports, but it continues to process data coming from OP2. If OP3 fails, data sent by both OP1 and OP2 are lost. Since these two scenarios represent two different error modes, both of them should be emulated during the fault injection experiments.

3. *The upstream operator is stateful* – a stream operator can either be stateful or stateless. For example, an operator that filters a stream based on the attributes of the current tuple does not keep any state related to previously processed tuples. Figure 5(c) shows a data flow graph where a stateless operator OP1 sends data to a stateful operator OP2, which sends data to OP3. If OP1 fails, it loses input data from its upstream operator while it is offline. As a result, OP2 also does not receive input data during the time OP1 is offline and does not update its internal state. If OP2 fails, the behavior is analogous to a fault in OP1. OP2 loses its input data and does not update its internal state while it is recovering. However, the error behavior changes when OP3 fails. OP3 loses the input data, but OP2 still updates its internal state. Once OP3 is back up, OP2 is ready to send up-to-date information and does not spend any time rebuilding its internal state. These scenarios have different impact on the application output, and both must be evaluated.

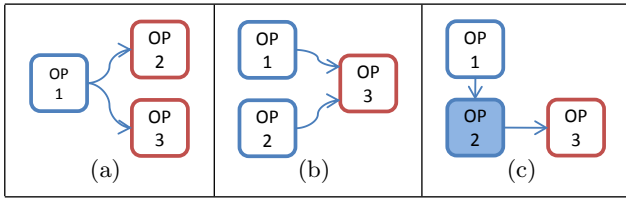


Figure 5: Placement rules for bursty tuple loss FIOPs consider the number of inputs and outputs of each operator, and if the operator is either stateless or stateful.

5. EVALUATING THE FAULT INJECTION OUTCOME

As mentioned in Section 3, we evaluate a fault injection outcome based on an application-specific *output score function* (OSF). The OSF characterizes how the application performs, and it can be computed independently of which stream operator failed. Next sections describe how we handle the continuous output of streaming applications and how we use the OSF to compute each of our metrics.

5.1 Handling Continuous Output

Stream processing applications typically produce output results continuously. If the output is not carefully analyzed, variations in the output due to the application non-determinism can be confused with the effects of a fault. This can lead to an overestimation of the faulty effect. We minimize this problem by limiting which segments of the continuous output stream are analyzed for estimating the faulty outcome. For example, results produced before the fault occurrence are ignored when computing the metric. In addition, results produced after the application started to produce output within error thresholds when compared to the golden run are also ignored.

Continuous output can also mask the effects of faults. For example, an application can manifest a fault by missing x alerts and misdetecting y alerts. When applying globally an OSF that considers the total number of detected alerts, the misdetected alerts compensate for the missed ones. This can erroneously lead the developer to think that the fault had low impact on the application output. We minimize this problem by computing the OSF over *local sections* of the output stream rather than once over the complete output set. The last two metrics described in Section 5.2 use a local OSF computation.

5.2 Evaluation Metrics

Table 1 summarizes our evaluation metrics. The first two metrics indicate the *predictability* of the stream operator behavior under faults. An operator that does not have predictable behavior under faults is not a good target for applying the PFT mechanism under test, because if such operator fails in the field, the application outcome is unknown. The last two metrics are related to system *availability*, and allow the assessment of which operators are more critical for the application to preserve output quality under faults.

Outage duration impact

By computing the *correlation coefficient between outage duration and quality score* (C^{oq}) we can assess the impact of the outage duration (Figure 2). If the QS and the outage

Metric	Operator characteristic	Definition
C^{oq}	Outage duration impact	Correlation coefficient between outage duration and quality score
D^{oq}	Data dependency	Standard deviation and analysis of variance test of the quality score for different stream offsets
R^{lq}	Recovery time	P percentile of local quality scores outside a threshold value
I^{lq}	Quality impact	Sum of squared errors of local quality score

Table 1: Summary of evaluation metrics

duration are highly correlated (i.e., the correlation coefficient is close to 1 or -1), the developer can use off-the-shelf curve fitting methods to find a function that describes the quality loss in relation to a certain outage. We can then feed this function with outage parameters extracted from real failures in the field and evaluate the risk (in terms of quality degradation) of using the evaluated PFT technique. If such behavior poses unacceptable risk to the application, this operator should be protected against faults.

The C^{oq} metric can be computed in the following way. A fault injection experiment for a single operator injects faults at m different stream offsets using n different outage durations. Each stream offset is referred to as SO_i , where $i \in [1..m]$, and each outage duration is referred to as OD_j , where $j \in [1..n]$. For each SO_i and OD_j , there are p repetitions, where each one generates an output stream with only a single section affected by the injected fault. Such section is estimated based on the SO_i and the maximum OD_j value. The OSF for the affected section of the stream is referred to as $FO_{i,j,k}$, where $k \in [1..p]$.

The average output score function $\overline{OSF_{i,j}}$ for each OD_j and a particular SO_i is computed as

$$\overline{OSF_{i,j}} = \frac{\sum_{k=1}^p FO_{i,j,k}}{p} \quad (1)$$

The OSF for the golden run is calculated over the section of the output stream affected by the fault with maximum OD_j value³. The golden run is executed q times, where each execution generates one $GO_{i,l}$, where $l \in [1..q]$. The *quality score* is referred to as $QS_{i,j}$ and is computed as

$$\overline{QS_{i,j}} = \frac{\overline{OSF_{i,j}}}{(\sum_{l=1}^q GO_{i,l})/q} \quad (2)$$

After this step, a particular SO_i has n OD_j values associated with it and their corresponding $\overline{QS_{i,j}}$ results. With these two sets of data, we compute the Spearman's rank correlation coefficient, which assesses if two sets of values have a monotonic relationship [24]. This step results in associating a correlation coefficient CC_i with each SO_i . Correlation coefficients have bounds $[-1..1]$.

The operator C^{oq} is then calculated as

$$C^{oq} = \frac{\sum_{i=1}^m CC_i}{m} \quad (3)$$

Data dependency

The D^{oq} metric is the *standard deviation* (σ^q) and *analysis of variance test* (A^q) of the quality score for different stream

³The index j is omitted in all formulas using a single fixed value of OD_j .

offsets. This metric evaluates how the *same fault* (i.e., the same fault model and outage duration) affects the output quality when injected at different stream offsets (Figure 2). A high variability in the application output quality under the same fault indicates *data dependency*, i.e., the impact on the output depends on the data being affected by the fault. An operator with a high σ^q and a rejected ANOVA test [27] is not a good candidate for PFT, since the result of a fault in the field is highly unpredictable. An operator with a low σ^q and an accepted ANOVA test indicates that the fault has a similar impact in output quality, independent of where the fault was injected.

To compute D^{oq} , we first calculate σ^q , similarly to the C^{oq} metric. The difference is that we choose the same fixed OD_j value for each SO_i , instead of considering all OD_j values. As before, we compute the \overline{QS}_i for each SO_i .

The σ^q is calculated with the standard deviation formula, using the \overline{QS}_i of each stream offset SO_i as data samples.

The analysis of variance A^q is a one-way ANOVA hypothesis test. This test assesses if there is a statistically significant difference between the observed means of different groups, where each group is obtained under a distinct condition [27]. For the D^{oq} metric, the test decides whether the changes in the stream offset of the injected fault affect the fault's impact on the application QS. If H_0 (null hypothesis) is *accepted*, it means that the target operator is not data dependent. A *rejected* H_0 indicates the opposite. Equation 4 shows the parameters for invoking the ANOVA test that returns an *accept* or *reject* value.

$$A^q = ANOVA((QS_{1,1}, \dots, QS_{1,k}); \dots; (QS_{i,1}, \dots, QS_{i,k})), \quad (4)$$

where $QS_{i,k} = FO_{i,k} / ((\sum_{l=1}^q GO_{i,l}) / q)$. D^{oq} is the tuple (σ^q, A^q) .

Recovery time

The R^{lq} metric is the P percentile of quality scores outside a threshold value. This metric estimates how long it takes for the application to recover and to start producing *normal* outputs after the occurrence of a fault. The larger the value of this metric, the larger is the impact of an operator failure on the application availability. This metric assesses the deviation of the application output quality *locally*, i.e., by computing the OSF over different intervals of the output stream (e.g., all tuples produced during a 1 second interval). For this metric, an OSF data point is considered *normal* when the difference between the faulty OSF and the golden OSF is less than a certain *threshold* (e.g., faulty OSF is less than 3% away from the golden OSF). Any difference greater than the threshold is considered to be an *erroneous* output. Our metric considers the coverage of $P\%$ of the erroneous output as it can provide enough accuracy in evaluating the recovery time of the application. In our experiments, we considered a P of 90%.

To compute R^{lq} , we choose the same single outage duration OD_j for all stream offsets SO_i . Each experiment trial k generates one output stream, which is divided in s sections. For each section, we compute the local OSF referred to as $LO_{i,k,t}$, where $t \in [1..s]$. The average of $LO_{i,k,t}$ over each experiment trial is referred to as $\overline{LO}_{i,t}$, and is computed similarly to Equation 1. A similar procedure is followed for each of the q trials of the golden run. The OSF for each section of the golden output stream is referred to as $\overline{GLO}_{i,t}$.

$\overline{GLO}_{i,t}$ refers to the average of $GLO_{i,l,t}$ over each trial, and is calculated similarly to Equation 1.

In the next step, we build an *error array* based on $\overline{LO}_{i,t}$ and $\overline{GLO}_{i,t}$, with t starting at S_{begin} , where S_{begin} is the first section of the output stream produced by the golden run after the fault injection point. Each position of the array is referred to as $EQ_{i,u}$, where $u \in [1..s - S_{begin}]$, and is computed as

$$EQ_{i,u} = \frac{|\overline{LO}_{i,t} - \overline{GLO}_{i,t}|}{\overline{GLO}_{i,t}}. \quad (5)$$

For each position u in the *error array*, we compute the number of error values that are greater than the established *threshold* up until and including u^{th} error value $EQ_{i,u}$. This is denoted by $NE_{i,u}$ and is represented formally as

$$NE_{i,u} = \sum_{v=1}^u \mathbf{1}[EQ_{i,v} > threshold]. \quad (6)$$

Then, we compute the *index* R_i^{lq} where $P\%$ of the erroneous QS observations fall. That is:

$$R_i^{lq} = \min u \text{ s.t. } NE_{i,u} \geq p * NE_{i,s-S_{begin}} \quad (7)$$

The final step is to obtain the maximum index for all stream offsets SO_i , that is $R^{lq} = \max_i R_i^{lq}$. Picking the maximum allows the assessment of the risk by considering the worst case manifested during experimentation.

Figure 6 shows an example of the R^{lq} metric. The line with a circle marker shows the QS values of the faulty run in relation to the golden run (square marker) for each section of the output stream. The dashed line shows the allowed error *threshold*. The R^{lq} metric covers 90% (P) of the data points that lie outside the threshold values (S13) after the fault is injected, showing an approximation of how long the application output takes to stabilize after a fault.

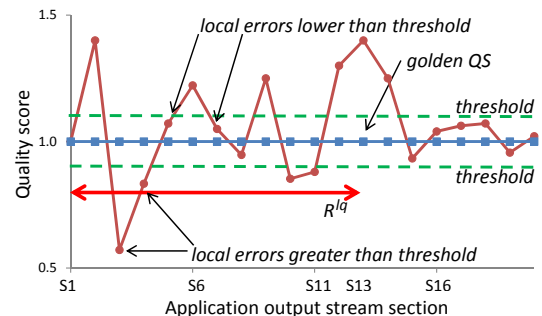


Figure 6: R^{lq} metric considers the number of local QS observations that fall outside a specified error threshold when compared to the golden run.

Quality impact

The I^{lq} metric is the *sum of squared errors (SSE)* of local quality score, which allows us to compare the fault impact of different operators on the application output quality. Similar to the R^{lq} , we consider local OSF values that are outside of a *threshold* tolerance. The *magnitude of the fault impact* is obtained by summing the squares of all local errors throughout the application execution after the injection up until the chosen P percentile of the R^{lq} metric.

The computation of this metric is similar to the R^{lq} computation. Instead of applying Equation 6, we calculate the

SSE of a single SO_i (referred to as I_i^{lq}) as

$$I_i^{lq} = \sum_{v=0}^{R_i^{lq}} (EQ_{i,v})^2 [EQ_{i,v} > \text{threshold}]. \quad (8)$$

I_i^{lq} is then $\max_i I_i^{lq}$.

6. EXPERIMENTAL EVALUATION

Our experiment target is a simplified application from the finance engineering domain called *Bargain Discovery* [1]. In our experiments, we assume the following: (i) a stream operator crash is *always* detected, (ii) all stateful operators of the application are being checkpointed [17], and (iii) an operator restores its state to the state immediately before the fault occurrence.

6.1 Target Application

The target application processes stock *trades* and *quotes*, and outputs information on all stocks in which there is a *potential money gain* on buying the stock at a given time. Figure 7 shows the Bargain Discovery flow graph.

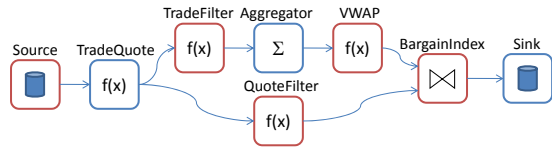


Figure 7: Stream operator graph of the Bargain Discovery application. One path processes only stock trades, while the other path processes only stock quotes. The streams are later correlated to decide if a given stock quote has a low price compared to its price history.

The application has one *source* operator, which reads trades and quotes events from a file⁴. Each entry in the file corresponds to a real event from the stock market. Each entry (tuple) contains a *ticker*, which is the symbol that identifies a company on an exchange. The *type* indicates the action taken by an investor. A *trade* action means an investor bought a certain number (*volume*) of shares at a specific *price*. A *quote* indicates an investor wants to sell a number (*ask size*) of stocks at a certain price (*ask price*).

The processing logic starts with the *TradeQuote* operator, which reduces the size of the tuple. Two different operators consume the output stream of *TradeQuote*, generating two branches in the application flow graph.

The first branch of the flow graph starts with *TradeFilter*, which filters all tuples of type *trade*. The *Aggregator* consumes all trades and sums up the *total volume* and *total price* for the 5 *most recent* trades of a given stock symbol. The operator generates a new sum every time a new trade of the corresponding symbol is processed on the input stream. The *VWAP* operator processes the *Aggregator* output stream and generates a tuple with the moving average price of a given stock symbol. The second branch of the flow graph has only the *QuoteFilter* operator, which outputs only tuples with type *quote*.

The processing logic finishes with the *BargainIndex*, which correlates the output streams of *VWAP* and *QuoteFilter*.

⁴In a real application deployment, data arrives as a continuous stream.

For every incoming *quote* tuple, it checks what is the most recent moving average stock price for the given ticker symbol. The operator estimates the potential money gain by multiplying the *ask size* by the difference between the moving average and the *ask price*. All outputs produced by the *BargainIndex* are stored into a file by the *Sink* operator.

For the purpose of fault injection, we modified the application input file by adding a *primary key* for each entry of the file. This key follows a strict ascending order. The application propagates this key until the *Sink*. With such key, we can precisely identify segments of the faulty output stream and match them with the equivalent segment of the *golden run's* output stream. This allows an accurate comparison between the application OSF with and without faults.

6.2 Experiment Parameters

The input stream used in our experiments consists of real market trades and quotes transactions from December 2005. We limited the number of processed trades and quotes to 5 million events. This dataset has the following characteristics: (i) the average event rate is 500 tuples/second, with peak rate of 2200 tuples/second; and (ii) *quote* transactions account for 80% of input stream events.

For this experiment, we chose 6 different outage durations. They are specified in seconds and have the following values: 0.5, 1, 2, 4, 8 and 16. The value 0.5 second was estimated by measuring how long it takes the System S runtime to detect a crashed process and restore it to its normal operation. The value 16 seconds is the time that the System S runtime takes to detect that a node has failed and to migrate a stream operator to a different machine. As described in Section 4, the FIOP that emulates the bursty tuple loss fault model expects as parameter the outage duration specified in terms of the number of tuples. Each of the outage duration values was converted to the number of tuples that would be lost due to the fault using both the average and peak input rate we observed in our dataset. The average and peak input rates are converted according to the processing rates of each operator. We used System S built-in instrumentation features [11] to obtain the processing rates of each operator.

The chosen stream offset trigger values are the following: 0.5, 1.5, 2.5, 3.5, and 4.5 million. Similar to the outage duration, we approximate the offset trigger based on the number of tuples processed by each operator.

Due to the application non-determinism, the golden run was executed 300 times. Each outage duration and stream offset combination was executed 5 times, totaling 300 fault injections per operator. The target operators for this application are *Source*, *TradeFilter*, *QuoteFilter*, *VWAP*, and *BargainIndex*. They are highlighted in Figure 7 and were chosen based on the optimization criteria described in Section 4.2. All experiments ran on a single node with Linux operating system, 4 Intel Xeon 3GHz processors, and 8GB of RAM. The experiments ran on a single node to reduce the effects of non-determinism on the application output.

Output Score Function. We defined the OSF of the application as the total sum of the financial gain. This application can misbehave in two ways: (i) underestimating the OSF (QS below 1), i.e., the application fails to indicate opportunities for buying profitable stocks; and (ii) overestimating the OSF (QS above 1), i.e., the application is estimating that certain stocks are more profitable than they are in reality. This can lead a trader to make wrong decisions.

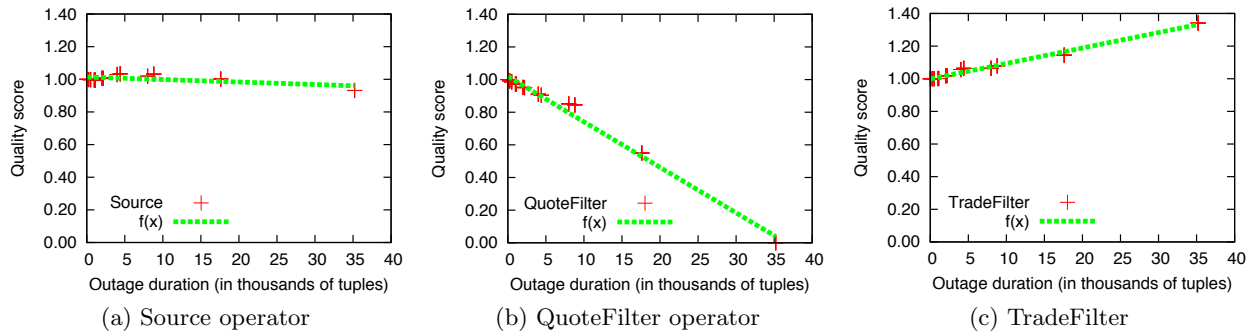


Figure 8: Quality score for fault injection trials with stream offset 1.5 million tuples and different outage durations for operators Source, QuoteFilter, and TradeFilter. Figure 8(a) shows that the QS underestimations overcomes the overestimations when the Source operator is offline for a long period of time. Figure 8(b) shows that the longer the outage duration, the more the QS decreases. Figure 8(c) indicates that the longer the outage of the TradeFilter operator, the more the application overestimates the trading financial gain.

6.3 Results

Outage duration impact.

The C^{oq} metric assesses if the outage duration caused by a fault in an operator directly impacts the application output quality. As described in Section 5.2, this metric considers only the section of the output stream that is directly affected by the fault. In the Bargain Discovery, we estimate the affected output stream by identifying all trades that were lost due to the injected fault and all trades that were correlated with a moving average that was miscalculated due to the injected fault. For example, when the fault injection target is the *Source* operator, the downstream operators lose both *trade* and *quote* tuples. In this case, the affected segment of the output stream considers both the dropped quotes and the quotes correlated with miscalculated moving averages. However, when the fault injection target is the *QuoteFilter*, the affected segment of the output stream consists only of the dropped quotes. Note that this analysis is application-specific, and should be customized for each application, if a precise estimate of the affected output stream is desired.

Table 2 shows the C^{oq} result for all target operators. Note that the QS result is highly correlated with the outage duration when faults are injected into *TradeFilter*, *QuoteFilter*, and *VWAP*.

Operator	C^{oq}	D^{oq}	R^{lq}	I^{lq}
Source	-0.38	(0.21,R)	340	21.23
TradeFilter	0.97	(0.76,R)	340	48.70
QuoteFilter	-1.00	(0.00,A)	6	6.00
VWAP	0.99	(0.14,R)	73	30.87
BargainIndex	-0.69	(0.08,A)	43	7.89

Table 2: Operator metrics results for Bargain Discovery application

Figure 8 illustrates how the *quality score* (QS) varies under different outages for three operators. The x axis is the outage duration and the y axis is the QS result. The function $f(x)$ is the result of least-squares fitting of a linear function.

For the *Source* (Figure 8(a)) there is an OSF overestimation when the outage duration is small (less than 5000 tuples), and an OSF underestimation when the outage duration increases (greater than 15000 tuples). When a fault affects the *Source*, both trades and quotes are lost. When

data loss is small, not as many quotes are dropped, resulting in many correlations with a miscalculated moving average. When the data loss is big, the financial loss due to not correlating quotes for a long time is greater than the overestimation due to miscalculated averages.

Figure 8(b) shows the high correlation between QS and outage duration for the *QuoteFilter* operator. Note that the QS is 0.00 at the maximum injected outage duration. The QS is 0.00 because the C_{oq} metric considers only the affected output stream under the maximum outage duration for its computation. When we inject a fault into *QuoteFilter* with the maximum outage duration, the *BargainIndex* does not perform correlations with any quote tuple, and, as a result, the application does not produce any output during the outage period.

Figure 8(c) shows the QS values for the *TradeFilter* operator. When this operator fails, the operator that maintains the most recent trades stops adjusting the moving averages based on new trade values. This can lead to the evaluation of a non-profitable stock as profitable (case 1) and vice versa (case 2). In our dataset, the magnitude of case 1 was always greater than case 2. This indicates that the financial loss for buying non-profitable stocks is greater than the financial loss incurred because purchase opportunities were missed.

Data dependency. Table 2 shows the D^{oq} metric results. A stands for an accepted A^q , and R stands for a rejected A^q . The operator with the greatest σ^q is *TradeFilter*, and the one with the lowest σ^q is *QuoteFilter*. The only operators with an accepted ANOVA test (i.e., low QS variability under faults) with $\alpha = 0.05$ are *QuoteFilter* and *BargainIndex*. Figure 9 shows the QS result (y axis) for all injected stream offsets (x axis) under the maximum outage duration. The QS for *QuoteFilter* is 0.00 independently of which stream offset the fault is injected into. This is because *BargainIndex* cannot do any stream correlation when *QuoteFilter* fails. For *TradeFilter*, the lowest QS is 1.34 and the greatest is 3.35. This represents a considerable variation, which indicates that effect of a fault on the QS depends to a great degree on what data the outage affects.

Recovery time. The R^{lq} estimates how long the application takes to produce output below an error threshold once an operator fails. For the target application, we consider a threshold of 3% away from the OSF of the golden run. We computed the local OSF values considering the primary key

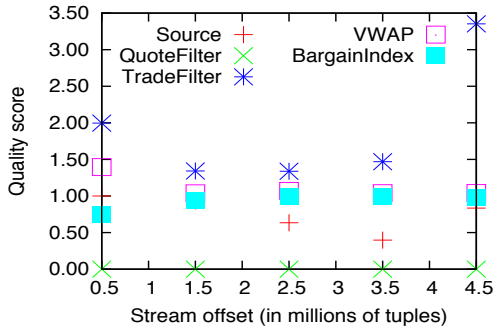


Figure 9: Quality score for different stream offsets under maximum outage duration. The impact of a failure in the TradeFilter operator on the application QS is highly data dependent, while the impact of a failure in QuoteFilter is independent of the data affected by the fault.

of the resulting tuple. All tuples with key values falling into an interval of 5000 units are grouped into one stream section (e.g., tuples with keys between 5000 and 10000). This approximates to one local observation at every 2.2 seconds when the input stream is producing tuples at its peak rate. Table 2 shows the R^{I^q} for all target operators for an outage duration of 16 seconds under peak rate. Figures 10 and 11 show the R^{I^q} value as vectors. The displayed value is the R^{I^q} for the injected stream offset, and not the maximum value among all injected stream offsets.

The operators with highest recovery time are *Source* and *TradeFilter*. These two operators have the same R^{I^q} value because during the injection trials the same exact set of tuples were dropped with respect to the *Aggregator*. Additionally, they have the highest R^{I^q} because they both affect the state of the *Aggregator*, which maintains the history of the recent trades. Once new tuples are processed, the *Aggregator* updates its internal state, producing moving average estimations with fresh data. As seen in Figure 10 and 11, the QS result stabilizes as more tuples are processed.

The *VWAP* and *BargainIndex* operators (Table 2) have smaller R^{I^q} values. When they fail, the state they affect downstream is quicker to rebuild in comparison to the *Source* and the *TradeFilter*. Once *BargainIndex* recovers from its checkpoint, its internal state contains outdated moving averages. However, it immediately starts receiving correct moving average values, allowing correct correlations with new incoming quotes. *QuoteFilter* has a small recovery time because it does not affect the state of operators downstream.

Quality Impact. The I^{I^q} evaluates the magnitude of the impact on the application output when an operator fails. The outage duration, error threshold, and interval of sections of the output stream are the same ones used by the R^{I^q} metric. Table 2 shows the I^{I^q} values for an outage duration of 16 seconds under peak rate. Our results reveal that a fault in *TradeFilter* affects the application output the most, while a fault in *QuoteFilter* has the lowest impact.

Figure 10 shows the QS result (y axis) for every section of the output stream (x axis) for 3 target operators. The x axis starts at section 300, which corresponds to the injection stream offset after 1.5 million tuples have been processed. Figure 10(a) shows the QS for the *Source* operator. After the fault injection there are no tuples present in the output stream, leading to 100% underestimation of the OSF when

compared to the golden run. Once the operator resumes sending tuples, the application overestimates its results up to 59% percent. Figure 10(b) shows the QS observations for the *QuoteFilter* operator. The *QuoteFilter* has a low I^{I^q} because it only affects the output during the outage period. When faults are injected into the *VWAP* operator (Figure 10(c)), the application produces high overestimates (up to 113% greater than the golden run OSF). This is because a fault in *VWAP* affects the *BargainIndex* state. As a result, the application continues to correlate new quotes with outdated moving average values. Note that when *VWAP* fails, the history of recent trades maintained by *Aggregator* is kept up-to-date. As a result, once *VWAP* recovers, it can immediately send up-to-date values downstream.

Figure 11 shows the QS for the *TradeFilter* when subjected to faults at stream offset after 3.5 million tuples have been processed and with different outage durations. Our experiments show that as the outage duration increases, the peak OSF overestimate for a certain stream offset and the I^{I^q} increase. When the outage lasts 2 seconds (Figure 11(a)), the maximum overestimate is 7% and the I^{I^q} is 0.25. For an outage of 8 seconds (Figure 11(b)), the maximum overestimate is 72% and the I^{I^q} is 12.85. The peak overestimate for an outage duration of 16 seconds (Figure 11(c)) is 144% and the I^{I^q} is 48.70. Our results show that even though the *TradeFilter* and *Source* are losing the same set of tuples under a injected fault with the same outage duration, the QS deviation is higher when the fault is injected into the *TradeFilter*. When *TradeFilter* fails, the *Source* continues to send new data, and the *BargainIndex* continues to correlate quotes with an obsolete moving average value. When the *Source* fails, the application stops the correlation of any data during the outage duration, resulting in errors with lower magnitude.

6.4 Discussion

Our results show the following regarding the Bargain Discovery application:

1. *Influence of operator output stream on the state of the downstream flow graph determines the criticality of the operator* – the total state size (in bytes) of operators downstream of a failed operator determines how long the application takes to fully rebuild its state and for how long the application produces erroneous results. As a result, operators with greater influence on the downstream state are more critical to maintain the application output quality. For example, the *TradeFilter* is the most critical operator with respect to bursty tuple loss, both in terms of the quality impact and recovery time, making it a top priority for protection against bursty tuple loss. Even though the *TradeFilter* operator is a stateless filter, it directly affects the stateful *Aggregator* and *BargainIndex* downstream. Another example is *QuoteFilter*, which is the least critical operator with respect to bursty tuple loss. This operator has low impact on application output quality, short recovery time, and very predictable behavior under faults ($A^q = \text{accept}$ and $\sigma = 0.00$). Although *BargainIndex* is stateful and consumes data from *QuoteFilter*, the *BargainIndex* does not keep internal state related to *QuoteFilter*'s output stream.

2. *Checkpoint is not adequate to protect operators when faults have a long outage duration* – our results show that checkpointing [17] provides good protection against faults with short outage duration (e.g., *TradeFilter* and *Aggrega-*

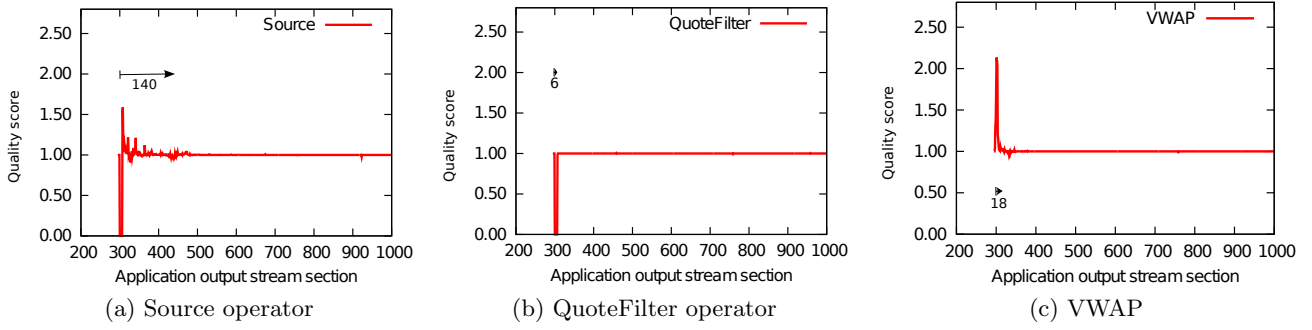


Figure 10: Quality score for each section of the output stream with stream offset 1.5 million tuples and outage duration of 16 seconds under peak rate. Vector shows the value of the R^{lq} metric for each operator. Figure 10(a) shows that *Source* operator has a high R^{lq} value, while Figure 10(b) shows that the application quickly recovers from a failure in the *QuoteFilter* operator. Figure 10(c) shows that a *VWAP* failure has a short R^{lq} when compared to *Source*, but it generates QS observations with higher values.

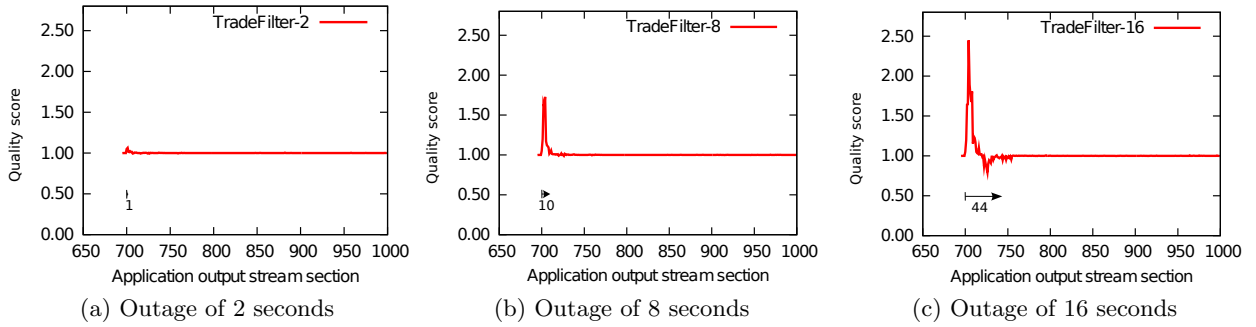


Figure 11: Local QS observations of the output stream with stream offset of 3.5 million tuples and different outage durations under peak rate for *TradeFilter* operator. I^{lq} and R^{lq} values increase as the outage duration increases.

tor in Figure 11(b)), but is not enough for faults with long recovery time.

3. *Position in the flow graph is not an adequate heuristic for deciding operator criticality* - although other researches [4, 21] suggest that the position in the flow graph can be used to deploy PFT, our study indicates that the position on the flow graph, and the type of operator alone are not adequate heuristics to characterize operator criticality. Although *QuoteFilter* and *TradeFilter* have both similar position in the flow graph and the same operator type, they have very distinct behavior under faults.

4. *The proposed metrics can be used to reconfigure the application fault tolerance and to observe a measurable improvement in application output quality* - based on our experimental results, we can improve the fault tolerance of the application by applying, for example, a technique with lower recovery time, such as *high availability groups* [17]. This technique maintains active replicas of operator groups of the application flow graph. Once an operator in the *active* group fails, the *backup* group becomes active. The failover time from one replica to the other is at most 2 seconds in System S. Our experiments show that by replicating a group of critical operators, such as the *TradeFilter*, *Aggregator* and *VWAP* (shown in Figure 12), we can see an improvement on the output quality of the application under faults. When either *TradeFilter*, *Aggregator*, or *VWAP* fails, the R^{lq} is 2 and the I^{lq} is 2. This is a significant improvement when compared to the previous values (I^{lq} of 48.70 and R^{lq} of

340). Under faults in the *Source* and the *TradeQuote*, the new I^{lq} is 1.05 and the R^{lq} is 39, in contrast to an I^{lq} of 21.23 and R^{lq} of 340 obtained in our previous tests. For other operators, such as the *QuoteFilter*, the metrics show that a simple restart is enough to maintain good application output quality. This has a big positive impact on resource utilization for our target application, given that *QuoteFilter* has an input selectivity of 80%, as described in Section 6.2.

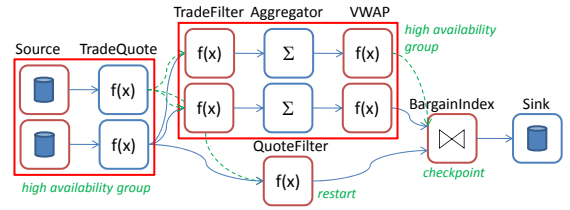


Figure 12: Fault tolerance reconfiguration of Bargain Discovery considers that operators with high metric values should be replicated, resulting in improved output quality on the occurrence of failures.

An important factor in deriving our conclusions was the OSF definition, which closely follows the semantics of our application. Our experimental methodology, together with a well-defined OSF, enabled us to make informed decisions with respect to fault tolerance, since we can evaluate the cost associated with the applied fault tolerance technique and the benefits such protection yields. Our results with the Bargain

Discovery application have also demonstrated robustness to different choices of OSF. We tested both *average financial gain* and *number of produced tuples* as OSFs, and the relative criticality of operators in terms of I^{lq} and R^{lq} were similar to the one obtained by the *total sum of financial gain*.

7. EXPERIMENTAL METHODOLOGY LIMITATIONS

Injecting faults into a streaming application and evaluating its results has many challenges, mainly imposed by the possible non-determinism of the application and its tolerance to approximate results. In this paper, we described an experimental methodology we used to evaluate the impact of PFT techniques on the output of a financial engineering application. We envision that our experimental methodology can be further generalized to be applicable to other streaming applications. In this section, we discuss some of the limitations of our current approach which should be studied in order to generalize the proposed methodology.

1. *Application-specific quality metric* - our basis to compare a faulty and a golden run is via an application-specific OSF. The OSF approximates the behavior of the application output and it copes with the non-determinism of the application output. For some applications, it may be hard to define an OSF. As a result, generic OSFs (e.g., number of tuples) can be investigated as an alternative for such cases.

2. *Concurrent failures* - our experiments assumed that operators fail independently. However, depending on the physical deployment of the application, multiple operators may fail together (e.g., if they reside in the same process [12] or run on the same node). We plan to investigate the emulation of such failures by considering the co-located operators as a single operator with a more complex internal logic.

3. *Number of required injection trials for applications with large number of operators* - our methodology injects faults in *each* operator considering different *outage durations* and *stream offsets*. In addition, if the application is non-deterministic, we execute the same injection configuration multiple times. This can result in a high number of experimental trials if the application has a large number of operators. In Section 4.2, we describe an optimization step based on the properties of the operator graph to help reducing the number of target operators. Considering co-located operators as a single operator can also reduce the number of required experiments.

8. RELATED WORK

Many fault tolerance techniques for streaming applications consider that no data can be dropped or duplicated [13, 16, 19], which depends upon the implementation of expensive buffer management and consistency protocols. These techniques do not evaluate the application output quality when faults occur, since they assume that the application produces the same output despite the occurrence of faults. Balazinska et al. [3] propose to produce *tentative* (lower precision) results during the occurrence of faults. The evaluation of this technique was based on the number of tentative tuples produced during faults, but there was no evaluation with respect to their impact on the output quality.

Previous literature on PFT techniques [4, 17, 21, 23, 30] do not describe how to systematically evaluate the impact of faults on the application output quality. Bansal et al. [4]

assume that the importance of each component in a streaming application can be described as a linear combination of importance of the inputs it is consuming. Our experiments show that this is not the case for our target application. Our earlier work [17] relies on the application designer to specify which components need to be fault-tolerant, while in this technique we inject faults to understand the application behavior under faults. Murty and Welsh [21] and Pietzuch [23] do not evaluate their proposed fault tolerance methods with respect to effects on the quality of an application’s results. Zhu et al. [30] assess the output quality of their proposed fault tolerance method in terms of a Sum of Squared Errors. We propose three other evaluation metrics.

There is vast research on evaluating fault tolerant systems with fault injection methods [8, 14, 29]. Streaming applications have unique characteristics, such as non-determinism, and continuous data processing despite the occurrence of faults. This brings up additional challenges to the evaluation methodology, which cannot be attacked with the techniques described in the literature.

Model-based dependability evaluation techniques [18] are complementary to fault injection experimentation. One use of model-based approaches is to understand the impact of faults on the application availability under a given failure distribution. Fault injection assumes that a fault that can happen on a real installation has occurred in the system, independent of its frequency.

Our work is related to research in load shedding of streaming applications, where the stream processing middleware can drop data once it detects that the system is operating over its capacity. Tatbul et al. [26] study the problem of using application semantics to drop tuples via a loss-tolerance graph. This graph can only have stream operators of specific types. Babcock et al. [2] proposes an accuracy metric similar to our QS. We propose four different metrics which are based on the QS. Previous work [9] considers the operator type to establish a specific quality metric. Our methodology uses a quality metric that is application-specific, but that is independent of the types of operators used by the application.

Condie et al. [7] evaluate the accuracy of applications running on top of a MapReduce implementation that supports continuous queries. The characteristics of their target applications are distinct from ours. Although their target application is implemented as a continuous query, it has a final *exact* and *deterministic* answer (e.g., number of page views). Our scenario is more complex, since our application is non-deterministic and *every* tuple produced is a true independent outcome of the application, and not a value that gets adjusted over time.

Fiscato et al. [10] propose a model for streaming applications with quality metrics based on the importance of a tuple. The importance of tuples produced by an operator depends on the importance of its input tuples. Tuples coming from the same base stream are assumed to be equally important. In our experiments, this does not hold true. Specifically, although *QuoteFilter* and *TradeFilter* output streams are derived from the same input stream, they show very different behavior under faults.

9. CONCLUSION

Partial fault tolerance techniques aim at decreasing their impact on performance of streaming applications by allowing

the application to lose and duplicate data when faults occur. In this paper, we described a fault injection methodology used to evaluate the impact of using PFT techniques on the output of a streaming application. Our results show that PFT can lead to better resource consumption, and that fault injection can be used to learn how to selectively deploy fault tolerance techniques in the application processing graph. In addition, our experiments indicate that some heuristics proposed by previous research [4, 21] on partial fault tolerance may not be the best option to maintain application output quality under faults.

As future work, we plan to evaluate other streaming applications to further generalize our experimental methodology and evaluation metrics. We believe there is a need for more fault injection experiments on stream processing applications, so that our insights and the body of knowledge with respect to the behavior of streaming applications under faults can be broadened.

10. REFERENCES

- [1] H. Andrade, B. Gedik, K.-L. Wu, and P. Yu. Scale-up strategies for processing high-rate data streams in System S. In *ICDE*, 2009.
- [2] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, 2004.
- [3] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1):1–44, 2008.
- [4] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. Turaga, and C. Venkatramani. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM*, 2008.
- [5] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB*, 2002.
- [6] S. Chandra and P. Chen. How fail-stop are faulty programs? In *FTCS*, June 1998.
- [7] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI*, 2010.
- [8] D. Costa, T. Rilho, and H. Madeira. Joint evaluation of performance and robustness of a COTS DBMS through fault-injection. In *DSN*, 2000.
- [9] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.
- [10] M. Fiscato, Q. H. Vu, and P. Pietzuch. A quality-centric data model for distributed stream management systems. In *QDB*, 2009.
- [11] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *CIKM*, 2009.
- [12] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD*, 2008.
- [13] 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*, 2009.
- [14] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
- [15] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *ICDE 2005*, 2005.
- [16] J.-H. Hwang, U. Çetintemel, and S. B. Zdonik. Fast and highly-available stream processing over wide area networks. In *ICDE*, 2008.
- [17] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu. Language level checkpointing support for stream processing applications. In *DSN*, 2009.
- [18] G. Jacques-Silva, Z. Kalbarczyk, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer. Modeling stream processing applications for dependability evaluation. In *DSN*, 2011.
- [19] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant Stream Processing Using a Distributed, Replicated File System. In *VLDB*, 2008.
- [20] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *DSN*, 2000.
- [21] R. N. Murty and M. Welsh. Towards a dependable architecture for Internet-scale sensing. In *HOTDEP*, 2006.
- [22] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *KDCLOUD*, 2010.
- [23] P. Pietzuch. Challenges in dependable Internet-scale stream processing. In *WDDDM*, 2008.
- [24] C. E. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 14:72–101, 1904.
- [25] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.
- [26] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.
- [27] K. S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., Chichester, UK, 2002.
- [28] D. S. Turaga, O. Verscheure, J. Wong, L. Amini, G. Yocum, E. Begle, and B. Pfeifer. Online FDC control limit tuning with yield prediction using incremental decision tree learning. In *Sematech AEC/APC*, 2007.
- [29] K. Whisnant, R. K. Iyer, Z. Kalbarczyk, P. H. J. III, D. A. Rennels, and R. R. Some. The effects of an armor-based sift environment on the performance and dependability of user applications. *IEEE Transactions on Software Engineering*, 30(4):257–277, 2004.
- [30] Q. Zhu, L. Chen, and G. Agrawal. Supporting fault-tolerance in streaming Grid applications. In *IPDPS*, 2008.