

Modeling Stream Processing Applications for Dependability Evaluation

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

[†]Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{kalbarcz,rkiyer}@illinois.edu

[♣]Thomas J. Watson Research Center
IBM Research
{g.jacques,bgedik,klwu}@us.ibm.com

Abstract—This paper describes a modeling framework for evaluating the impact of faults on the output of streaming applications. Our model is based on three abstractions: stream operators, stream connections, and tuples. By composing these abstractions within a Stochastic Activity Network, we allow the modeling of complete applications. We consider faults that lead to data loss and to silent data corruption (SDC). Our framework captures how faults originating in one operator propagate to other operators down the stream processing graph. We demonstrate the extensibility of our framework by evaluating three different fault tolerance techniques: checkpointing, partial graph replication, and full graph replication. We show that under crashes that lead to data loss, partial graph replication has a great advantage in maintaining the accuracy of the application output when compared to checkpointing. We also show that SDC can break the *no data duplication* guarantees of a full graph replication-based fault tolerance technique.

I. INTRODUCTION

Stream processing applications continuously process multiple sources of live data (e.g., audio and business feeds), analyze them on-the-fly, and generate results. Examples of these applications include algorithmic trading, fraud detection, and health monitoring systems. Streaming applications are assembled as dataflow graphs, where each vertex of the graph is a *stream operator*, and each edge is a *stream connection*. To achieve high performance, stream operators can run across different nodes of a distributed system. In this environment, a fault in a stream operator can result in massive data loss or in the generation of inaccurate results.

Fault tolerance techniques must be used to achieve application resiliency to errors. To understand the benefits of applying a certain fault tolerance technique to a streaming application, it is critical to evaluate its effect on the application output, especially considering the differing resource consumption and performance impact of alternative techniques [1], [2], [3], [4].

Previous research on the evaluation of fault tolerance techniques for streaming applications has mostly focused on their performance overhead [2], [4]. Our earlier work [5] proposes the evaluation of the impact of faults on the application output via fault injection. While fault injection can be applied directly

to the real system and get accurate results, it can be very time-consuming and expensive to deploy, especially if we consider that operators can fail concurrently.

In this paper, we describe a modeling framework to evaluate the dependability provided by different fault tolerance techniques under varying fault models. The framework allows us to compare the relative merits of different techniques, so that the user can determine which technique performs best for an application at hand. The framework considers faults that lead to *data loss* and *data corruption*. To the best of our knowledge, we are the first to consider the problem of data corruption in streaming applications. In addition, our framework considers the consequences of *error propagation*, i.e., the impact that a fault at one stream operator can have on the downstream operators and on the application output. This is an important problem that also has not been addressed by the research community.

The developed framework is based on generic models specified with the Stochastic Activity Network (SAN) formalism [6]. One of the main innovations of our approach is to provide SAN-based abstractions for the key components of a streaming application: stream operators, stream connections, and tuples. By assembling these components, we represent the complete dataflow graph of the target application as a SAN. Furthermore, we devise techniques to capture the error propagation behavior of various fault models in the SAN representation of a streaming application, making it possible to evaluate the dependability achieved by various fault tolerance techniques under different fault models.

The framework is used to evaluate the effectiveness of three different fault tolerance techniques, namely checkpointing [3], high-availability groups [5], and full replication [2]. Our experiments with faults that cause data loss show that high availability groups have an advantage in maintaining the accuracy of the application output when compared to checkpointing. Our results also indicate that faults that lead to data corruption can break the *no data duplication* guarantee provided by the modeled full replication technique. We evaluate the accuracy of our approach by comparing the results obtained by running a target application in the proposed framework and in System S - a stream processing middleware developed at IBM Research.

The main contributions of this paper are (i) a framework with generic models to compose streaming applications and

This work was supported by an IBM PhD Fellowship (awarded to Gabriela Jacques-Silva) and IBM-UIUC Open Collaborative Research Project.

[‡]Currently employed by Goldman Sachs.
Email: henrique.c.m.andrade@gmail.com

to evaluate the dependability and resource consumption trade-offs provided by different fault tolerance techniques and under different fault models; (ii) a formalization of how errors affecting a stream operator propagate to other stream operators on the processing graph. The propagation model considers the probabilities that the stream connections of an application are used as well as the state size of stateful operators in the application; and (iii) an extensible framework to test new fault tolerance techniques before deployment and compare their behavior with existing approaches.

II. STOCHASTIC ACTIVITY NETWORKS

Our modeling framework uses the SAN formalism [6] to model stream processing applications and the occurrence of failure events. In addition to facilitating a unified modeling of performance and dependability aspects of a system, SANs can express the non-deterministic behavior and the parallelism of streaming applications.¹

Informally, the basic constructs of SANs are the following.

1. *Place* - a place contains a natural number of *tokens* and can represent, for example, a possible state of the modeled system. Places are represented graphically as a circle.

2. *Activity* - an activity indicates transitions between *places*. It expresses how long a transition takes to complete, and it can be described as a random variable. This construct can have a set of *cases*, which are used to model the possible outcomes of a transition. Activities are graphically represented by ovals, and cases are depicted as small circles attached to the ovals.

3. *Token* - a token is an item residing in a *place* and is depicted as a dot.

4. *Input gates* - input gates enforce a condition for an activity to be enabled and are illustrated as a left triangle (\triangleleft).

5. *Output gates* - output gates allow the execution of a function after the completion of an activity. Output gates can be used to update the state of the model. These gates are depicted as a right triangle (\triangleright).

For a formal definition of SAN, refer to Sanders and Meyer [6].

III. APPLICATION MODEL

This section describes the streaming application model considered by the framework. The later subsections describe the abstractions provided for stream operators, stream connections, and tuples. In addition, we detail how each of these components is mapped into the SAN formalism.

A *streaming application* is a directed dataflow graph $G = \langle O, C \rangle$. The vertices O represent a set of *stream operators*. Each stream operator $o \in O$ has an associated number of input ports $p_{\langle o \rangle}^{\triangleleft} \in \mathbb{Z}^*$ and a number of output ports $p_{\langle o \rangle}^{\triangleright} \in \mathbb{Z}^*$.

The directed edges C represent a set of *stream connections*. Each connection $c = (\langle o, k \rangle, \langle o', k' \rangle) \in C$ connects an operator output port (k^{th} output port of operator $o \in O$, where $k \in [1..p_{\langle o \rangle}^{\triangleright}]$) to an operator input port (k^{th} input port of

operator $o' \in O$, where $k' \in [1..p_{\langle o' \rangle}^{\triangleleft}]$). An item flowing through a stream connection c is called a *tuple* and is denoted by $\tau_{\langle c \rangle, m}$, where m represents the index of the tuple in the stream connection $c \in C$. The set of all tuples for a given connection c is denoted by $\tau_{\langle c \rangle}$. In a streaming application, $|\tau_{\langle c \rangle}|$ is infinite. In this paper, we assume that $|\tau_{\langle c \rangle}|$ is finite to estimate the rate of a given stream connection.

An operator o with no input ports ($p_{\langle o \rangle}^{\triangleleft} = 0$) is called a *source operator*. A source operator channels data directly from a raw data source (e.g., video camera) maintained by a component outside the stream processing graph.

For a non-source operator o , we denote the set of incoming connections on its input port $i \in [1..p_{\langle o \rangle}^{\triangleleft}]$ as $C_{\langle o \rangle, i}^{\triangleleft}$. Formally,

$$C_{\langle o \rangle, i}^{\triangleleft} = \{(\langle o', k' \rangle, \langle o'', k'' \rangle) \in C : o'' = o \wedge k'' = i\}.$$

In other words, $C_{\langle o \rangle, i}^{\triangleleft} \subset C$ defines the set of stream connections attached to the i^{th} input port of operator o . Each input port is associated with a processing logic function $F_{\langle o \rangle, i}$. This function is characterized with a tuple processing cost function $f_{\langle o \rangle, i}$ defined as

$$f_{\langle o \rangle, i} : \cup_{c \in C_{\langle o \rangle, i}^{\triangleleft}} \tau_{\langle c \rangle} \rightarrow \mathbb{R}^+.$$

For each port, we also define an average processing cost per tuple $\bar{d}_{\langle o \rangle, i}$ as

$$\bar{d}_{\langle o \rangle, i} = \frac{\sum_{c \in C_{\langle o \rangle, i}^{\triangleleft}} \sum_{m=1}^{|\tau_{\langle c \rangle}|} f_{\langle o \rangle, i}(\tau_{\langle c \rangle, m})}{\sum_{c \in C_{\langle o \rangle, i}^{\triangleleft}} |\tau_{\langle c \rangle}|}.$$

The operator invokes $F_{\langle o \rangle, i}$ every time there is a tuple available in any of the stream connections associated with $C_{\langle o \rangle, i}^{\triangleleft}$. As a result, the average processing cost $\bar{d}_{\langle o \rangle, i}$ considers tuples from all connections in $C_{\langle o \rangle, i}^{\triangleleft}$.

An operator o with no output ports ($p_{\langle o \rangle}^{\triangleright} = 0$) is called a *sink operator*. A sink operator stores its results into a component outside the stream processing graph (e.g., database).

For a non-sink operator o , we denote the set of outgoing connections of its output port $j \in [1..p_{\langle o \rangle}^{\triangleright}]$ as $C_{\langle o \rangle, j}^{\triangleright}$. Formally,

$$C_{\langle o \rangle, j}^{\triangleright} = \{(\langle o', k' \rangle, \langle o'', k'' \rangle) \in C : o' = o \wedge k' = j\}.$$

In other words, $C_{\langle o \rangle, j}^{\triangleright} \subset C$ defines the set of stream connections attached to the j^{th} output port of operator o .

We define the set of operators connected to an input port i of operator o as $O_{\langle o \rangle, i}^{\triangleleft}$. Formally,

$$O_{\langle o \rangle, i}^{\triangleleft} = \{o' \in O : (\langle o', k' \rangle, \langle o'', k'' \rangle) \in C \wedge o'' = o \wedge k'' = i\}.$$

The set of operators connected to all input ports of another operator is denoted as $O_{\langle o \rangle}^{\triangleleft} = \cup_{i=1}^{p_{\langle o \rangle}^{\triangleleft}} O_{\langle o \rangle, i}^{\triangleleft}$. We also define the set of all operators in the upstream of an operator o as

$$U_{\langle o \rangle}^{\triangleleft} = \begin{cases} \{\} & \text{if } p_{\langle o \rangle}^{\triangleleft} = 0 \\ O_{\langle o \rangle}^{\triangleleft} \cup \cup_{o' \in O_{\langle o \rangle}^{\triangleleft}} U_{\langle o' \rangle}^{\triangleleft} & \text{otherwise.} \end{cases}$$

Finally, we define the set of operators on the upstream of a

¹Other modeling formalism could also be used (e.g., stochastic Petri nets). However, SAN offers simplicity while still allowing to express complex system behavior.

specific input port i of operator o as

$$U_{\langle o \rangle, i}^{\triangleleft} = \begin{cases} \{\} & \text{if } p_{\langle o \rangle}^{\triangleleft} = 0 \\ O_{\langle o \rangle, i}^{\triangleleft} \cup \bigcup_{o' \in O_{\langle o \rangle, i}^{\triangleleft}} U_{\langle o' \rangle}^{\triangleleft} & \text{otherwise.} \end{cases}$$

A. Stream Operators

To represent an operator o using the SAN formalism, we first consider its input stream connections, $C_{\langle o \rangle}^{\triangleleft} = \bigcup_{i=1}^{p_{\langle o \rangle}^{\triangleleft}} C_{\langle o \rangle, i}^{\triangleleft}$. For each input stream connection $c \in C_{\langle o \rangle}^{\triangleleft}$, we generate a corresponding *place* in the SAN. These places are labeled as *input stream connection*. Figure 1 shows an example where we have two input stream connections. The tuple handling within an operator is modeled in three stages, namely *waiting*, *processing*, and *sending*. In the *waiting* stage, the operator waits for input from any of the input stream connections. This is achieved by connecting all the *input stream connection* places to a single input gate (IG1). The input gate is also connected to a place labeled *waiting for input*, as shown in Figure 1. Once there is data available, the input gate (IG1) enables the transition out of the *waiting for input* place and the operator moves to the *processing* stage. For this stage, we create $p_{\langle o \rangle}^{\triangleleft}$ places, one for each input port. These places are labeled as *processing tuple* and are connected to the input gate IG1 via an *activity* as shown in Figure 1. The *processing* stage considers *selectivity*² to determine if a new tuple should be sent out or not. This behavior is modeled by an *activity with two cases* on the tuple processing transition. If there is no output to be generated, the operator waits for new input data by transitioning back to the *waiting for input* place, in which case we return to the *waiting* stage. The *processing* stage also considers average processing cost $\bar{d}_{\langle o \rangle, i}$ for each *processing tuple* place. This cost is used as the *activity* parameter and can be obtained by profiling the modeled application [7]. If there is an output to be generated, the operator moves to the *sending* stage. For the *sending* stage, $p_{\langle o \rangle}^{\triangleright}$ places are created (one for each output port). These places are labeled as *sending output*, as shown in Figure 1. The operator transitions from a *processing tuple* place to a *sending output* place through the *cases* defined on the activity connecting the two. The operator stays in the *sending output* place until there is available space in the operating system's protocol stack buffer for data transmission (enforced by an input gate (IG2)). This emulates the possible back pressure caused by downstream operators. Furthermore, an additional place for each output port is added to represent the output buffer for the port. These places are labeled as *output buffer* as shown in Figure 1. The input gate IG2 is connected to the output buffer places through an activity and an output gate OG1. The output gate is also connected to the *waiting for input* place. The output gate is used to submit the tuple to the output buffer and immediately go back to the *waiting* stage by transitioning to the *waiting for input* place. There is only a single output gate used per

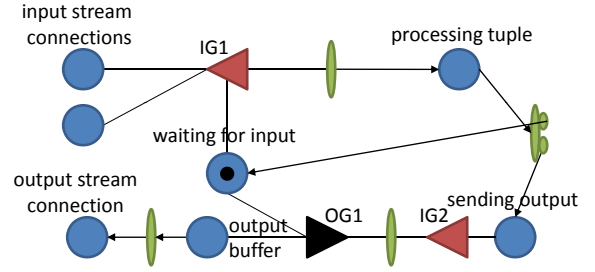


Fig. 1: SAN for a stream operator with two input connections on a single input port and one output stream connection on a single output port. The model captures three tuple processing stages (*waiting*, *processing*, and *sending*), operator selectivity, and tuple transmission to output stream connection.

output port. Each output stream connection $c \in C_{\langle o \rangle}^{\triangleright}$ is also mapped to a *place*. These places are labeled as *output stream connection*. The *output stream connection* places associated with a given output port (i.e., $C_{\langle o \rangle, j}^{\triangleright}$ where $j \in [1..p_{\langle o \rangle}^{\triangleright}]$) are connected to the output port's *output buffer* place. Figure 1 shows an example for a single output connection. For other examples, refer to our earlier work [5].

B. Stream Connections

Stream connections represent the intercommunication channels between operators executing in different processes or nodes of a distributed system. We represent stream connections by composing stream operator models via the *Replicate/Join* formalism [8]. This formalism allows places in different models to be *shared*, effectively allowing the communication between the composed models. Stream connections are modeled as the *state sharing* between the *output stream connection* place of one operator with the *input stream connection* place of another operator. In the composed model, we create one shared pair *input-output stream connection* for each $c \in C$. Because places are shared, once a token is written into an *output stream connection* place of one operator, it is immediately visible in the *input stream connection* place of the next operator.

A single composed model with all stream interconnections represents the application dataflow graph, as shown by the example in Figure 2. This example shows the equivalent composed model for the application segment given in Figure 3. This application segment, written in SPADE [9], has three operators, mapping to a composed model with three base stream operator models and two stream interconnections. Note that each operator model has a token indicating its current state. This makes the operators execute concurrently, effectively mimicking the operation of a parallel and distributed streaming application.

C. Tuples

We represent each tuple $\tau_{\langle c \rangle, m}$ as a *token* that transitions through the operators via the stream connections. Tokens placed in *input stream connection*, *output stream connection*, or *output buffer* place model tuples residing in communication queues of the operating system (OS) or the stream operator itself. Tuple processing within an operator is modeled by removing a token from the *input stream connection* place

²Selectivity is generally used for operators with predicates, such as filters and joins. In this paper, we use selectivity to represent the ratio *average number of output tuples per second in an output port/average number of input tuples per second in an input port* for any operator type.

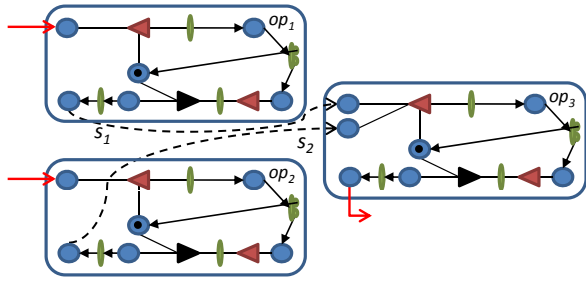


Fig. 2: Example of composed model equivalent to the streaming application depicted in Figure 3. The sharing of output stream places of op_1 and op_2 , and input stream places of op_3 models the interconnection between operators.

```

stream OP1(tag: String, company: String,
           article: String)
:= Functor(CNNSource)
[tag = "technology"]{}

stream OP2(tag: String, company: String
           article: String)
:= Functor(FoxNewsSource)
[tag = "technology"]{}

stream OP3(company: String, article: String)
:= Functor(OP1, OP2)
[company = "IBM"]{}

```

Fig. 3: Synthetic streaming application coded in SPADE containing three operators. OP1 and OP2 consume data from external sources (CNNSource and FoxNewsSource) and filter for articles related to technology. OP3 consumes the output of both OP1 and OP2, filtering for articles related only to IBM.

and optionally adding a token (after the processing time) to the *output stream connection* place. The token is written to the *output stream connection* place according to the specified operator selectivity (Section III-A). Discarded tuples are seen as tokens that are effectively removed from the model.

In our framework, source operators are not connected to any upstream operators and, as a result, have no data to consume. We represent external sources by adding an activity that fires according to a random variable with a configurable statistical distribution. The distribution describes the inter-arrival time of tuples into one of the source operators feeding the application. Once this activity fires, we insert a token into the *input stream connection* places of all source operators consuming data from the same external source. This token is then transmitted to all downstream operators, until it reaches one of the application sinks. For this purpose, all source operator models are augmented with an *input stream connection* place, even though they do not have input ports.

IV. FAULT AND FAILURE MODEL

For a streaming application to achieve maximum data throughput and minimum end-to-end latency, the stream processing graph can be distributed over a set of computing nodes. Each operator or set of operators can be mapped to a process that can run on different nodes of a distributed system.

Our fault model assumes that different *stream operators* can

fail independently or concurrently. The cause of failure can be a node crash (e.g., a device driver error leads to OS kernel crash), a Heisenbug (e.g., a race condition), or a hardware transient error (e.g., an error affects ALU output).

One possible outcome of a fault is a *clean crash* of the operator. A clean crash means that even though the fault has led to operator failure, it did not result in any erroneous output value being produced and the operator state stored in the checkpoint file [3] (if any) was not corrupted. The consequences of a clean crash can vary depending on the fault tolerance technique being used. Many techniques aim to guarantee *no data loss* under operator crashes [1], [2]. On the other hand, partial fault tolerance techniques [3], [4] favor improved runtime-performance over a no data loss guarantee. When failures occur under these schemes, both the data present in the communication channels and the data being processed by the operator at the time of failure may not be recoverable.

Another possible outcome of a fault in an operator is a *silent data corruption* (SDC). SDC is the result of a fault that goes undetected, but corrupts parts of the application data. Previous works have shown that a non-negligible percentage of transient hardware faults lead to the corruption of the application output [10]. Dixit et al. [11] indicate that transient error rates may increase as the microprocessor feature size decreases. As a result, data corruption becomes a significant problem to be handled by the application layer. To the best of our knowledge, all fault tolerance techniques proposed for streaming applications [1], [2], [3], [4] assume a fail-stop fault model, which does not always hold true [12]. In stream operators, hardware errors can affect the *internal state of the operator* and the *attribute values of a tuple*.

V. ERROR PROPAGATION MODEL

To accurately evaluate the fault impact on the application output, our framework models *error propagation*. This allows us to capture the effects that a fault in one operator can have in other operators. For example, if an operator corrupts a tuple because of a fault, this tuple can spread to all connected operators and eventually reach the application output. We model the probability that a corrupted tuple reaches the output based on operators' selectivities and the state size of stateful operators. In this section we detail how we augment our base model (Section III) to include operator failures and propagation of errors. We describe the model extensions for both clean crashes and SDCs.

In our framework, we refer to tuples affected by a fault as *tainted tuples*. A tainted tuple (i) may have attribute values different than it would have during a fault-free execution, and (ii) may be generated under a predicate evaluation that uses corrupted data and/or a corrupted internal state. Tainted tuples can have attribute values that are approximate to its fault-free values and thus be tolerable by a given application. The proposed model operates over tokens that have no value and, as a result, does not capture the degree to which a tuple is tainted.

To differentiate between correct and tainted tuples, we augment the stream operator model described in Section III-A to include the *generation* and *propagation* of tainted tuples under faulty conditions. To distinguish correct tuples from tainted ones, we create an extra stream interconnection for each connection $c \in C$ in the application. These extra connections, called *tainted stream connections*, only carry tainted tuples. Once the operator consumes a tuple from a tainted connection, it goes through state transitions that may lead to the production of one or more new tainted tuples. To model error propagation, we distinguish whether an operator is stateless or stateful. Any other specific operator semantics are ignored. Figure 4 shows an example SAN model for a stateless operator that includes additional places for tainted stream connections. The example includes an extra processing state, which depicts the processing of *tainted tuples*. The next sections describe the conditions under which operators might generate tainted tuples based on the fault model under evaluation.

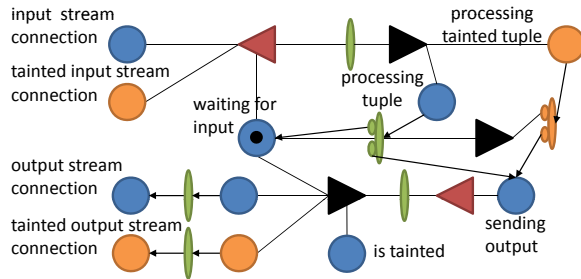


Fig. 4: Extended SAN for a stateless stream operator with a single input and output stream connection. Model includes extra places to represent operator interconnection via tainted stream connections, allowing the propagation of tainted tuples.

A. Operator Crash Failure

A stream operator crash can result in data loss. Data loss can affect the internal state of stream operators, leading to inaccurate application output. To evaluate the extent to which the data loss affects the application outcome, we analyze the *conditions under which operators can generate corrupted data*. Our analysis is based on the fault injection experiments with a streaming application [5] running on top of System S. These experiments emulate operator crashes via a bursty data loss fault model and evaluate the deviation of the application output under faulty conditions.

In our framework, an operator crash is modeled via a *failure activity* that transitions from an *alive* state to a *crashed* state. This transition follows an exponential distribution³ with a constant rate of λ_c failures per second. After the *failure activity* fires, all data (i.e., tokens) in the input and output buffers of the operator are discarded. While in crashed mode, the operator does not receive or send any tuple. After the failure, the operator transitions again to an *alive* state via a *recovery activity*. This activity is parameterized with the average recovery time of an operator. Once the operator is

³Analysis of field failure data has shown that exponential distribution is a good enough approximation of real failure rates.

alive, it restarts to process input data. Depending on the operator behavior under crashes, the data it sends out can be either a correct or a tainted tuple.

To model the behavior of a stream operator o under data loss, we classify the operator into the following categories: (i) *stateless*, where all functions $F_{(o),i}$ associated to $p_{(o)}^{\triangleleft}$ input ports decide their outcome solely based on the current tuple being processed (e.g., filter); and (ii) *stateful*, where at least one function $F_{(o),i}$ associated to $p_{(o)}^{\triangleleft}$ input ports maintains any accumulated state (e.g., data from previously processed tuples or local OS data) and uses such state to compute attribute values of the output tuple (e.g., window-based join).

Stateless operators. When a stateless operator crashes, it can still generate correct tuples from its input streams after a restart. As a result, stateless operators *do not generate tainted tuples* after a crash and restart.

If stateless operators *receive tainted tuples* from their input streams, the operator may send out a tainted tuple (i.e., *propagate the error*). As shown in Figure 4, we add an extra processing state to process tainted tuples. The corrupted value of tainted tuples can also affect the selectivity of the operator. For example, tuples that normally would not pass a filter predicate can pass the filter because of a corrupted attribute value. This behavior is modeled by adding an activity with a variation on the original *case* probabilities values defined by the operator selectivity.

Stateful operators. If a stateful operator crashes, there may be both data loss in its input ports and total [1] or partial loss [3] of its internal state. Once the operator restarts, its state is different from the state it would have in a failure-free run. Figure 5 shows how we model the operator timeline in this situation. We consider that the operator *produces tainted tuples until the internal state of the operator stabilizes* and can output tuples with approximate or perfect results. We model this behavior by changing the state of the operator to an *unstable mode* once it crashes and restores. While the operator is in an unstable mode, all the tuples that it produces are tainted. Once the operator stabilizes its internal state, it transitions to a *stable mode* and can again produce correct tuples. We assume that the duration of the unstable period lasts until a certain number of correct tuples is processed (e.g., full window size) [5]. Section VI provides more details on how we established the stabilization time for the modeled techniques.

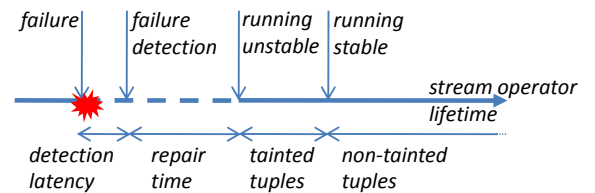


Fig. 5: Timeline for a stateful stream operator under a data loss fault model. Once the operator crashes and restores, it runs in an unstable mode producing tainted tuples. After processing tuples and stabilizing its internal state, operator produces non-tainted tuples.

Another situation in which a stateful operator o *produces tainted tuples* is when an operator in the *upstream set of input*

port i (i.e., $o \in U_{(o),i}^{\triangleleft}$) associated to a $F_{(o),i}$ that maintains state crashes. As shown in Figure 6, a fault in an operator that is in the upstream set (i.e., filter with predicate `int < 9`) of a port with a stateful function (i.e., aggregation) can affect the operator internal state. The example shows a filter operator at time t_1 and an operator with a sliding aggregation window of size 4 at time t_2 . In Figure 6(a), the operator slides its window after consuming the incoming tuples (5 and 2). After that, the operator sums up the current values of the window and generates an output tuple (16). Figure 6(b) shows the execution when the filter crashes at time t_1 . Because of the crash, the tuples with value 5 and 2 are dropped. As a result, the aggregator does not slide its internal window. The window slides only after the filter recovers at time t_2 and sends tuples 1, 8, and 5. As the figure shows, the aggregator ends up generating tuples with different values (12 and 18) than the fault-free scenario. Similarly to the stateful operator crash scenario, we consider that the operator also has to process a certain number of incoming tuples to stabilize its internal state after an upstream operator crashes.

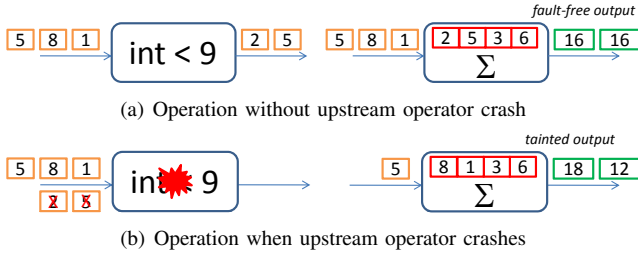


Fig. 6: Example of an upstream operator crash that impacts the internal state of a stateful operator downstream. Even though the stateful operator does not crash, its internal state ends up with a different content when compared to a fault-free run.

The last situation that a stateful operator may generate tainted data is when it *receives a tainted tuple from an upstream operator*. In contrast with a stateless operator, when a stateful operator receives a tainted tuple on a port i with $F_{(o),i}$ that maintains state, the operator internal state gets tainted, potentially leading to the generation of multiple tainted tuples until the stabilization of its state. We expect that the stabilization time due to a tainted tuple is shorter than the stabilization time due to tuple loss. The intuition is that a bursty loss of data affects more tuples than a single tainted tuple. Note that this stabilization time is also a parameter in our model. If this assumption does not hold for a certain application, the stabilization time can be adjusted accordingly.

B. Operator Silent Data Corruption Failure

To understand and properly model SDC in stream operators, we conducted fault injection experiments on a real application to gather data on how the SDC manifests in terms of tainted tuples produced by a faulty operator. We used SymPLFIED [13], a symbolic fault injector, to carry on the measurements. Two stream operators were used as fault injection targets: (i) a *stateless filter* that consumes data from a single input stream and sends data to a single output stream. We observed that

many injection runs resulted in the corruption of a *single tuple* on the operator output; and (ii) a *stateful aggregator* that computes a sum over a sliding window of size 5. SymPLFIED found injection runs where both *single* and *multiple tuples* were corrupted on the operator output.

SDC is modeled by transitioning an operator from an *error-free* state to an *error* state. The activity follows an exponential distribution with a fault rate of λ_s faults per second. The λ_s value considers only faults that lead to a SDC.

Stateless operator. The outcome of a fault in a stateless operator is to *generate one tainted output tuple*. The processing stage of a corrupted tuple can have a different selectivity than the processing of a correct tuple. An extra activity in the model with case probabilities different than the original processing activity models different selectivity values under faults. Stateless operators also *propagate tainted tuples* coming from its tainted input stream connections. Error propagation occurs according to the selectivity of tainted tuples.

Stateful operator. A fault in a stateful operator can cause the *generation of one or more tainted tuples*. If the error leads to the corruption of a single tuple, the operator sends a tainted tuple to its tainted output streams connections. If the error leads to the corruption of the internal state, the operator sends tainted tuples until its internal state stabilizes.

If the operator *receives tainted tuples* in a port associated with a function $F_{(o),i}$ that maintains state, it also considers the generation of tainted tuples until its internal state stabilizes. Figure 7 shows an example of a SDC of a tuple processed by a stateless filter at time t_1 . The figure also shows the aggregator at time t_2 , after consuming tuple 3 and the tainted tuple. The erroneous tuple taints the internal state of the operator, resulting in multiple erroneous tuples.

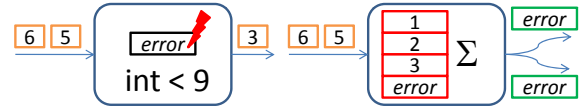


Fig. 7: Hardware transient fault affects the filter operator, generating a corrupted output tuple. The error propagates to the Aggregator, which consumes an erroneous tuple and stores it in its internal window. Corrupted tuples spread to multiple output stream connections.

VI. MODELING FAULT TOLERANCE TECHNIQUES

In this section, we show how our framework can be extended to model the following three fault tolerance mechanisms: checkpointing of operator state [3], partial replication via high-availability groups [5], and a full replication-based approach [2]. We provide a summary of each technique and how it is mapped to the SAN formalism.

A. Checkpointing of Operator State

Checkpointing of operator state [3] is a lightweight fault tolerance technique developed in SPADE [9], the programming language for IBM's System S. This technique allows the developer to configure the fault tolerance applied to the application by selectively annotating the stream operator processing graph. For every annotated operator, the SPADE

compiler automatically generates specialized checkpointing code based on the operator type and instance. The System S runtime asynchronously invokes this checkpointing code at the frequency specified together with the annotation. Because the operator checkpoint is not updated at every processed tuple, the checkpointed state is slightly *outdated* with respect to the operator state right before the crash. In addition, every time a stream operator crashes, its upstream operators discard all outgoing tuples, causing a *bursty tuple loss*. This solution favors performance and application output timeliness over precise computation of every output tuple.

Our checkpointing model considers crash failures by adding a heartbeat monitor for each stream connection c of the application. Similar to a stream, the heartbeat is modeled by adding shared variables via the *Replication/Join* formalism. The shared variable lets an operator o monitor the liveness state (Section V-A) of all operators connected to its output ports. Once the operator detects a crash, it immediately starts discarding all outgoing tuples. If the operator has more than one output stream connection, it stops sending data only to the failed one. This follows the implementation of one of System S' operator intercommunication protocols [14].

For all stateless operators of an application, our model assumes that there is no checkpoint to recover from. For stateful operators, we consider checkpointing only for operators annotated in the target application. If the operator is not annotated, it assumes it restarts from scratch. This means that after a restart, the operator does not send output tuples at its full output rate (i.e., sends tuples with a lower selectivity) until it is stable. After the stabilization time, the operator resumes its normal behavior. In this study, the stabilization time is equal to the number of tuples corresponding to a full operator state (e.g., window size). During the stabilization time, the operator does not send tuples as tainted. Because the operator is starting from scratch, we consider that whatever it produces is correct, similarly to when the real streaming application is starting from scratch. Note, however, that this situation can still result in tainted tuples in the application output. If there are stateful operators downstream, they may start to produce tainted tuples due to an upstream failure, as described in Section V-A.

For stateful operators, we model the number of tuples to be processed until stabilization with the following formula:

$$Z_{\langle o \rangle, i} = \min(\alpha * S_{\langle o \rangle, i} + \beta * L_{\langle o \rangle, i}, \gamma * S_{\langle o \rangle, i}).$$

The number of tuples to be processed until stabilization, denoted as $Z_{\langle o \rangle, i}$, depends on two components in our model. First is the state size, denoted as $S_{\langle o \rangle, i}$, and the second is the total number of tuples that were lost since the last checkpoint, denoted as $L_{\langle o \rangle, i}$. The value for $L_{\langle o \rangle, i}$ can be estimated based on the *age* of the last checkpoint and the tuple rate for input port i . We use a linear combination of these two components to estimate $Z_{\langle o \rangle, i}$. We use the weights α and β to scale the two components. The former is called the *state penalty factor*, and the latter is called the *state staleness factor*. We also limit the maximum value $Z_{\langle o \rangle, i}$ can take using a constant factor on the

state size, called *state cap factor*, denoted as γ . The impact of *propagated errors* is estimated only by considering a *state penalty factor* over the state size. Note that this formula is an approximation of the behavior observed in our fault injection experiments with data loss in a streaming application [5]. If this does not hold for a given target application, the developer can change this parameter in our model accordingly.

B. High-Availability Groups

High-availability (HA) groups is a fault tolerance technique implemented in the SPADE compiler that allows an annotated set of interconnected operators of a stream processing graph to be replicated during runtime. Each set of interconnected operators is referred to as a *group*. There is no determinism enforced between the replicas of a group.

When SPADE replicates a group of operators, one of the replicas is configured as *active* and the others are configured as *backups*. While both active and backup replicas consume and process their input streams, only the tuples produced by the active replica are consumed by the operators downstream. Tuples produced by the farthest downstream operator of the backup sub-graphs (also called the *group edge*) are discarded. When *any* operator in the active replica fails, the backup replica detects the fault (via the absence of heartbeats) and starts sending its tuples to the operators downstream. This technique can cause tuple loss during the failover time.

To model the HA groups technique, we first replicate the base model of each operator annotated to be in an HA group and then add them to the composed model of the application. The replicated operators are interconnected following the same flow of the original stream processing graph.

To trigger the failover between active and backup replicas, we add to the model a heartbeat (via shared variables) for each operator selected to be in an HA group. These heartbeats are monitored by each group edge of an HA group (one edge for each replica). The group edge is responsible to trigger *failover* and *replica deactivation activities* depending on the liveness condition of the replicated operators. The *failover activity* is parameterized according to the average failover time between replicas. As in our implementation in SPADE, only one replica is *active* at a time, while all others are in *backup* mode.

For all non-annotated operators, our model assumes that the operators have no replica and restart from scratch after a crash. Stateful operators in the HA group are also assumed to restart from scratch, unless annotated with a checkpoint capability. Although an operator in an HA group may produce tainted tuples upon restart, these tainted tuples do not propagate down the graph if the replica is not active. Generation of tainted tuples by the active replica can only occur if there are simultaneous failures in both replicas, and the active replica ends up with an operator in an *unstable* state.

Following the model described for tuple loss in Section V-A, a stateful operator in a graph with HA group annotations can still produce tainted tuples upon the crash of an upstream operator. However, the difference here is that some operators are replicated. As a result, operators in the HA group only

start producing tainted tuples when a non-replicated operator in its upstream crashes *or* when there is no active replica for an HA group contained in the upstream set (e.g., during failover time). In such cases, the operator produces tainted tuples until its stabilization time. We model the number of tuples to be processed until stabilization using the same formula introduced for the checkpoint technique (Section VI-A).

C. Full Replication

Hwang et al. [2] propose a technique based on full replication of the stream processing graph. Each operator replica sends its output to all operator replicas downstream. This technique *guarantees no tuple loss and no tuple duplication upon crashes* by making all operators deterministic and by deduplicating tuples coming from redundant streams. If an operator sees a tuple for the first time, it immediately sends the tuple downstream and stores its value in an internal table. Operators discard all tuples detected as duplicates. The operator internal table is periodically cleaned by punctuation marks⁴ sent by source operators at a specified interval.

To add this technique to our base model, we first replicate all operators except the sources and sinks. For each replicated operator in the application, we add one extra operator to the composed model. We then add extra regular and tainted streams to fully interconnect the replicated graph.

To model tuple deduplication, we add one extra place per incoming stream. This extra place counts how many tuples are received in each stream. When a tuple arrives to an operator input stream, it checks whether the number of tuples received from this stream is *fewer or equal* to the number of tuples received from the redundant stream. If this predicate evaluates to true, the operator considers the tuple a duplicate and discards it. The operator cleans up all of its counters when a *punctuation activity* fires. The deduplication logic works for both tainted and non-tainted stream connections. Note that deduplicating tuples exclusively by counters adds a little imprecision to our model. The imprecision also comes from the fact that SANs are non-deterministic. This means that although the operators are semantically replicas, the outcome of the activities that represents their selectivity (Figure 1) can be different even if they are processing a tuple with the same count value. We envision that operator replicas can be made deterministic by starting the pseudo-random number generator associated to each *processing activity* with the same seed. In addition, we can extend the model so that a token can be handled as a data structure containing a sequence number. The sequence numbers can then be used for precise deduplication.

Since this technique guarantees no data loss and no data duplication under crashes, we consider its evaluation only under SDC errors. The error propagation model follows the rules described in Section V-B. For the stabilization time of a stateful operator after a fault or error propagation, we follow a similar formula to that introduced for the checkpointing

⁴A punctuation is special mark that can be included out-of-band in a stream to delimit groups of tuples that should be operated on together.

technique. If a stateful operator receives a tainted tuple, or a fault affects its internal state, we assume its stabilization time is proportional to the state size.

VII. EVALUATION

To illustrate the applicability of the proposed framework, this section reports on dependability evaluation of a real streaming application while employing different fault tolerance techniques. Our first experiment compares the fault tolerance provided to the same application when crashes occur. The following techniques are applied: checkpointing of stateful operators (Table I - Configuration 1) and HA groups with checkpointing (Table I - Configuration 2). The second experiment evaluates the impact of SDC in the output of the application when full replication is deployed (Configuration 3 in Table I). All our models and experiments were implemented with the Möbius tool and its simulator [15].

A. Target Application

Our target application is Bargain Discovery [16], which aims at processing financial market trades and quotes to discover stocks that have a low price at a certain point in time. This application has eight stream operators and its original processing graph is shown in Table I (C1). The stream operators include four filters, one aggregate, and a join. Both the aggregate and join operators process data over count-based windows. These operators instantiate new windows according to the stock symbols contained in the incoming data stream (SPADE's *per-group* modifier).

B. Experiment Parameters

In our experiments, we used a constant input stream data rate with the following values: 250, 350, 450, and 550 tuples/sec. Our values were based on real market data from December 2005, which has an average rate of 500 tuples/sec. The average tuple processing time and selectivity for each operator in the graph are shown in Table II.

As described in Section V, the selectivity of an operator can change (if lower than 1) when processing tainted tuples. We considered a variation of selectivity of 10% for stateful operators and 5% for stateless operators. These variations are based on data from our fault injection experiments [5], where we compared the number of tuples produced during a fault-free application run and a run subjected to failures.

C. Evaluation Metrics

1. *Availability* - the application is deemed available when all of its operators (or one of its replicas) are *alive* and *stable*, no operator is processing tainted tuples, and no operator has a tainted state. This metric considers the fraction of time that the application is providing correct service [17].

2. *Total number of tainted tuples* - this metric considers the total number of tainted tuples stored by the sink operator. It gives an estimate of how many tuples can have its attribute values affected by a fault.

3. *Percentage of tainted tuples with respect to fault-free run* - this metric is the fraction of tainted tuples stored by the sink

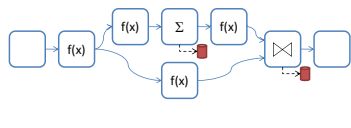
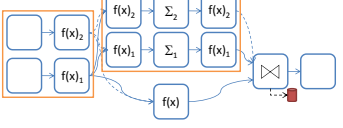
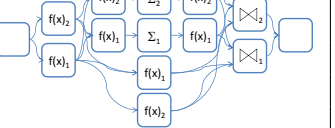
	Configuration 1 (C1)	Configuration 2 (C2)	Configuration 3 (C3)
Application graph			
Fault tolerance	checkpoint [3]	HA groups and checkpoint [3]	full replication [2]
Fault model	crash leading to data loss	crash leading to data loss	silent data corruption

TABLE I: Summary of experiments. First experiment compares Configuration 1 (C1) to Configuration 2 (C2) under the same fault model. C1 deploys the Bargain Discovery application with two stateful operators configured to checkpoint. C2 deploys the same application, but applies the HA groups technique to two different operator groups, and checkpoint of one operator. Second experiment evaluates Configuration 3 (C3) under a SDC errors. C3 applies the full replication mechanism to the same target application.

during a run with faults over the total number of tuples stored by the sink during a fault-free run. It indicates the fault impact on the final application output.

Operator	Processing time	Selectivity	State size
Source	1.26	1.000	0
Filter 1	1.26	1.000	0
Filter 2	0.52	0.200	0
Aggregate	13.2	1.000	15,495
Filter 3	0.54	1.000	0
Filter 4	0.22	0.800	0
Join (port 1)	1.27	0.000	3099
Join (port 2)	1.27	0.182	0
Sink	0.74	1.000	0

TABLE II: Parameters used in the Bargain Discovery model. Processing time (in μsec) and selectivity obtained via SPADE profiler. State size (in tuples) obtained from the application code and trace data.

D. Checkpoint versus High-Availability Groups

In this first experiment, we used our modeling framework to compare the impact of crash faults on the application when applying only checkpointing of stateful operators (C1) and when adding two HA groups (C2). We used a simulation time of 2 hours, and each experiment configuration had at least 1000 repetitions. The simulation stops after all evaluation metrics reach an average result with a confidence level of 95%. We simulated the models with the following exponential fault rates: $5.56\text{E-}04$ (30 min), $3.33\text{E-}04$ (50 min), $2.38\text{E-}04$ (70 min) and $1.85\text{E-}04$ (90 min). Stream operators fail at the same rate and independently. Although these rates are high considering supercomputer failure rates [18], we used such rates to project the impact of faults in the application in scenarios where failures can be more frequent.

For the aggregate operator, we considered a state penalty factor of 0.035 after a crash or an upstream failure and a state penalty factor of 0.005 when receiving tainted tuples in its inputs. In addition, we considered an operator staleness factor of 1.98. For the join operator and its port associated to a stateful function (port 1), we considered a state penalty factor after a crash of 0.03 and a staleness factor of 0.41. These numbers were obtained by fitting the function $Z_{(o),i}$ to the numbers obtained by analyzing real stream data and observing how many windows are affected by a fault. More details on this experiment can be found in our earlier work [5].

We also parameterized the state cap factor as 1 (i.e., a full window refresh). We evaluated both techniques considering crash recovery times of 3.5 sec and 15 sec. These numbers are based on the average and worst-case times for restarting an operator in System S. For the HA groups technique, we considered a failover time between replicas of 1 sec, which is the average failover time of our implementation in SPADE.

Figure 8 shows how the evaluation metrics behave when the source input rate increases. Figure 8(a) shows that as the input rate increases, the availability also increases. When the input rate is higher, the operator windows can be *refilled* more quickly, resulting in a lower stabilization time. Even though the availability is higher for increased input rates, Figure 8(b) shows that the number of tainted tuples also increases. This is because of the *join* operator. Although the operator state stabilizes more quickly, the join operator also correlates more data from one of its input streams when its state is tainted, leading to a greater number of tainted tuples on the output.

Another observation from Figures 8(a) and 8(b) is that the deployment with HA groups is better than checkpoint even for a recovery time of 15 sec. Although the availability provided by this technique is slightly lower when compared to C1 with a 3.5 sec of recovery, the number of tainted tuples is significantly lower. This occurs because when the non-replicated operators fail, no output is produced by the application, reducing the number of tainted tuples. When other operators of the graph fail, there is a spare copy, leading to a quicker recovery. In Figure 8(c), the number of tainted tuples in relation to the total number of tuples produced by the application decreases. A fault tolerance technique that affects only a minor fraction of the output might be sufficient for a given target application.

Figure 9(a) shows the impact of different fault rates on the application availability. The tested scenario includes C1 and C2. As expected, the availability of the application decreases as the fault rate increases. Comparing C1 and C2, the difference in availability is low (at most 0.009) if we look the extra computing resources required to deploy the HA groups technique. On the other hand, this technique can pay off if we consider the number of erroneous tuples produced due to the faults. In the scenario with a fault rate of $5.56\text{E-}04$, C2 generates approximately half the number of tainted tuples generated by C1. Figure 9(a) also shows the availability of the application when our availability metric does not consider

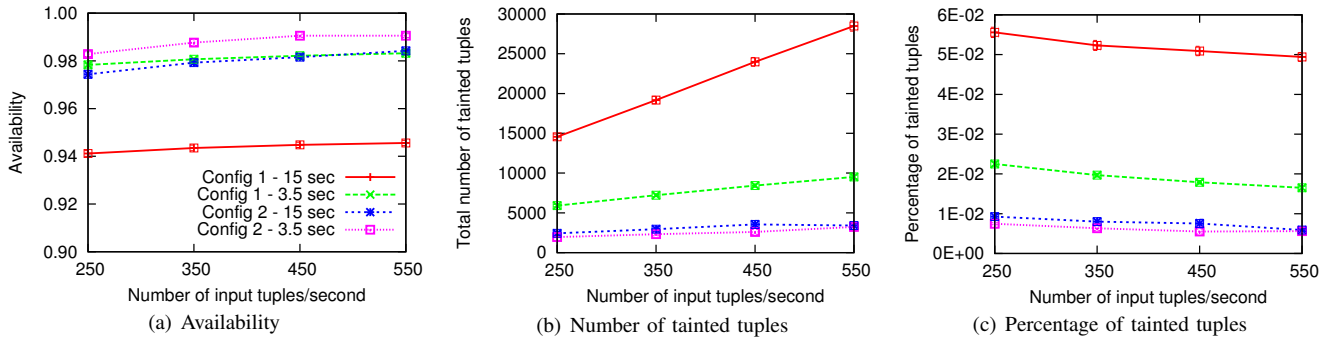


Fig. 8: Results for C1 and C2 for crash faults with an exponential rate of $1.85E-04$. Graphs include numbers for crash recovery times of 3.5 and 15 seconds. The legend in Figure 8(a) applies to Figures 8(b) and 8(c). Error bars denote the confidence interval. Figure 8(a) shows that availability increases as the input rate increases for both techniques. The number of tainted tuples produced increases when source input rates are higher (Figure 8(b)), but its fraction decreases when compared to the total number of tuples produced by the application.

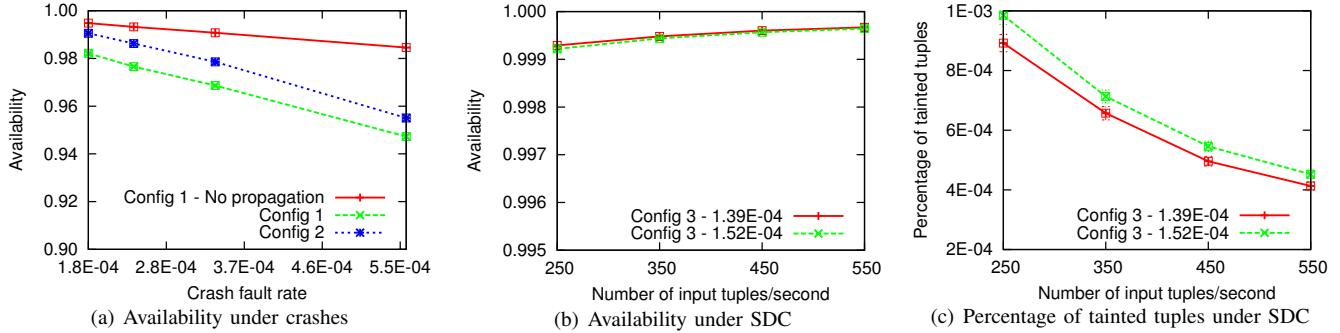


Fig. 9: Figure 9(a) shows the results for C1 and C2 under crash faults occurring at different fault rates. Error bars denote the confidence interval. Displayed availability is for a source input rate of 450 tuples/second and crash recovery time of 3.5 seconds. Figure 9(b) shows the availability for a full replication deployment under SDC (C3). Availability increases as input rate increases. Figure 9(c) shows the percentage of tainted tuples under the same fault tolerance technique. The relative impact of error decreases as tuple rate increases.

fault propagation. The application availability is overestimated, providing an unrealistic number to the developer.

E. Impact of Silent Data Corruption

In this experiment, we tested the impact of SDCs when Bargain Discovery uses a full replication technique (C3). For this scenario, the SDC rates follow an exponential distribution with λ_s values of $1.52E-04$ (110 min) and $1.39E-04$ (120 min). When the fault triggers in a stateful operator, it corrupts a single tuple or its internal state with 50% chance each (i.e., equal chance of corruption before or after processing a tuple). The stabilization time assumed for this scenario is the same as for propagated errors in the previous experiment. We used a punctuation interval of 15 sec. Once a punctuation arrives, the model clears the counters for both clean and tainted tuples.

Figure 9(b) shows the availability under different input rates and different fault rates ($1.52E-04$ and $1.39E-04$). Similar to experiments with crash faults, as the source input rate increases, the availability increases. Intuitively, the availability also decreases as the fault rate increases. Note that the impact of the availability of SDC is much lower than crash faults. The reasons are (i) the assumed fault rate is lower, (ii) operators never go offline due to a fault, and (iii) the application produces tainted tuples only via error propagation, which has a lower stabilization time. Figure 9(c) shows the percentage of tainted tuples on the application output. The number of tainted

tuples is very low, if it is compared to the total number of tuples produced by the application in a fault-free run (less than 300 tainted tuples in our experiments). However, the total number of tuples produced by the application (*tainted tuples + non-tainted tuples*) is greater than the total number of tuples (and its confidence interval) produced during a fault-free run. This indicates that *SDC breaks the no data duplication guarantee provided by the full replication technique*. This means that stream operator developers should consider high coverage but low cost fault detectors if no data duplication must be guaranteed. We believe this is a problem that should be studied given the high processing rates and low latency requirements of stream processing applications.

VIII. MODEL VALIDATION

Model validation is a critical step for obtaining trustworthy results. Our first validation test is to assess the similarity between the performance of an application running on System S and running the application model in our framework. Figure 10 shows the result of our experiment, where we plot the number of output tuples produced by the application sink over time when the source input rate is 450 tuples/second. The output rate in the real execution is more unstable when compared with the one produced by the model. This occurs because the operator selectivity varies within a range over time according to the data distribution, while our model captures

the average selectivity. To compare the average output rate of the real execution and the simulated application, we executed the application with 2000 different input data sets. The average output rates seen by the sink are similar to the ones produced by a single execution of our model (59.29 and 65.78 tuples/sec, respectively). In addition, 71% of the output rates observed in sink of the application are within the min-max output rate boundaries observed on the model execution. As future work, we plan to improve our base model to better capture the possible dynamics of the stream data.

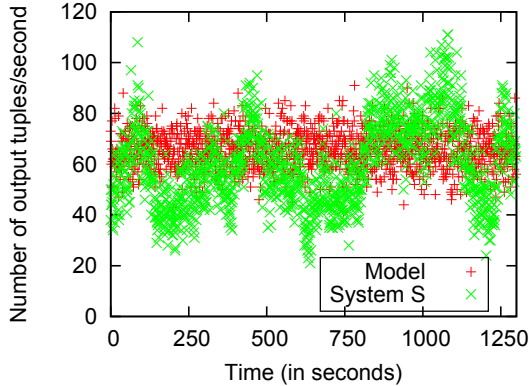


Fig. 10: Comparison of Bargain Discovery executing on System S and its simulation in our framework. After the application stream operators fills up its internal windows, the average output rate observed in the real sink is similar to the one produced by our model.

To validate our fault and error propagation model, we compared the results of the Bargain Discovery model with those obtained by injecting faults into the same application in System S. We exercised the model by forcing the occurrence of a single crash fault on one operator, imitating the injection methodology proposed in our earlier work [5]. Table III shows our measurements when faults are injected into different operators. The crash recovery time and source input rate considered for both real and simulated executions are the same (16 sec and 2200 tuples/sec). The table shows the measurements of the *availability* and the *recovery time* (R^{lq}) [5] metrics. In short, the R^{lq} metric assesses how long, after the injection of a fault, the application takes to produce results that are similar to a fault-free run. Although the two metrics do not measure the same thing, R^{lq} can be used as an indirect metric of the application *availability* (i.e., the greater the R^{lq} , the lower the *availability*). The relative order of the operators' availability is the same, except the *Aggregate* operator. This occurs because in the model the checkpoint of the *Aggregate* operator is taken 1 sec before the crash, but in the real injections the checkpoint is taken immediately before the crash. The table also shows the measurements for the *number of tainted tuples* and the *quality impact* (I^{lq}) [5] metrics. Briefly, the I^{lq} metric computes the sum of squares of the differences between the tuple attribute values of a faulty run and a fault-free run. The greater the number of tainted tuples, the greater the value of I^{lq} . However, because the number of tainted tuples does not capture the *error magnitude* that I^{lq} captures, there might be some discrepancy. This is the case for the *Filter 3* operator, which has a higher I^{lq}

value than *Source*, but produces a smaller number of tainted tuples on the application output. The similarity between our measurements confirms the accuracy of our model.

Operator	Availability	R^{lq}	Tainted tuples	I^{lq}
Source	0.84533	340	15,506.32	21.23
Filter 1	0.84536	340	15,500.01	21.23
Filter 2	0.84533	340	21,752.34	48.70
Aggregate	0.83987	340	22,520.63	48.70
Filter 3	0.93676	73	8898.29	30.87
Filter 4	0.95555	6	0.00	6.00
Join	0.93599	43	2753.54	7.89

TABLE III: Comparison between results obtained from real fault injections into Bargain Discovery [5] and results obtained from our model. *Availability* maps to the *recovery time* (R^{lq}) metric. *Number of tainted tuples* maps to the *quality impact* (I^{lq}) metric.

For validating the propagation model for SDC errors, we modified our target application to include an extra attribute in the tuple schema. The extra attribute indicates whether the tuple is corrupted. We then forced the propagation of this attribute through all operators of the application (e.g., an aggregate propagates the maximum value of the attribute value for all tuples in its window). We observed that even when the source operator contains only one corrupted tuple, it may result in the corruption of multiple tuples in the output.

The validation of the fault tolerance mechanisms followed an approach similar to the objectives of *fault injection*. By forcing the failure of operators in the model, we can observe if the detection and recovery transitions are taking place accordingly. Finally, we added multiple monitoring functions that check for *invariants* of our model during its execution. By checking the value of these assertions each time the model simulates, we confirm that the model is operating as expected. One example of this assertion is that two replicas of an HA group should never be active at the same time.

IX. MODEL LIMITATIONS

As in any modeling approach, our modeling framework can provide only an approximation of the system behavior. In this section, we discuss some of the limitations of our approach.

1. *Parametrization* - in our experiments, we derived some of the parameters based on previous fault injection observations and on the application source code. However, for other applications that contain operators with processing logic defined by the user (e.g., user-defined operators in SPADE), we may need to conduct fault injection experiments to derive parameters such as the stabilization time. Although fault injection is a time-consuming task, its results can be used for multiple applications that use the same operator and for testing different physical deployments (e.g., concurrent operator failures caused by operator co-location in the same node).

2. *Overtainting of tuples* - our assumption for stateful operators is that during the stabilization time, all tuples produced are tainted. This can cause an overestimation on the number of tainted tuples produced by some operator types. One such example is the join operator, where only some of the correlated tuples may be tainted. As future work, we plan to create additional model templates based on the operator semantics.

3. *Simulation with high data rates* - one of the key abstractions of our model is the tuple representation, where each tuple is mapped to a token in the SAN. When we simulate the model, all these tokens have to be processed and go through the processing graph, increasing the execution time of the simulation. This problem can be tackled by parallelizing the execution of the SAN. In our framework, different operators share few states with one another, which facilitates the efficient execution in a distributed setting.

X. RELATED WORK

The evaluation of the impact of fault tolerance techniques in streaming application has focused mostly on performance [1], [2]. Other works have focused on evaluating the impact of data loss in the application results via experimentation [3], [4]. Our previous work focused on proposing a fault injection methodology to learn how to deploy partial fault tolerance techniques [5]. This work proposes the evaluation of the impact of faults on a streaming application by providing generic and extensible models that consider error propagation.

Zhao et al. [19] developed a modeling framework for fork/join tasks to target the problem of optimal resource allocation. This paper focuses on the occurrence of faults and their effects on the application output. Viglas and Naughton [20] model streaming applications by considering rate estimates of each operator on the processing graph. Our model includes the representation of internal state of operators, which is critical to evaluate dependability aspects of the application.

Our work can also be compared to other uses of probabilistic models to evaluate fault tolerant systems. Singh et al. [21] model publish-subscribe systems to evaluate their survivability. Cinque et al. [22] model the dependability of wireless sensor networks. Wang et al. [23] develop a model for evaluation of synchronous checkpointing of distributed applications considering error propagation. Our modeling framework differs from these approaches by providing abstractions tailored for streaming applications. This allows us to evaluate the application in terms of the impact of faults on its output.

Previous research has analyzed the effects of error propagation in software [24], [25]. These works consider the effect of data corruption errors. We also consider how data loss can affect other components of the processing graph.

XI. CONCLUSION

This paper describes a modeling framework tailored to evaluating the dependability of streaming applications under faults. Our framework provides abstractions for three key components of an application: stream operators, stream connections, and tuples. By assembling these components, we can compose a given application and evaluate how it behaves under faulty conditions. Our framework allows the developer to compare different fault tolerance technique and decide which technique has the best cost-benefit for his application.

We show the generality and extensibility of our framework by modeling three different fault tolerance mechanisms. Our results show that SDC can break the no data duplication

guarantee provided by a full graph replication-based technique [2]. This result motivates the development of low-cost error detectors tailored for streaming applications.

As future work, we plan to add performance metrics to our framework and to automate the model composition based on the application source code and its physical deployment.

REFERENCES

- [1] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, "High-Availability Algorithms for Distributed Stream Processing," in *ICDE*, 2005.
- [2] J.-H. Hwang, U. Çetintemel, and S. B. Zdonik, "Fast and highly-available stream processing over wide area networks," in *ICDE*, 2008.
- [3] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu, "Language Level Checkpointing Support for Stream Processing Applications," in *DSN*, 2009.
- [4] Q. Zhu, L. Chen, and G. Agrawal, "Supporting fault-tolerance in streaming Grid applications," in *IPDPS*, 2008.
- [5] G. Jacques-Silva, "Partial fault tolerance in stream processing applications - methods and evaluation techniques," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2010.
- [6] W. H. Sanders and J. F. Meyer, "Stochastic Activity Networks: formal definitions and concepts," in *LNCSS 2090*, 2001.
- [7] B. Gedik, H. Andrade, and K.-L. Wu, "A code generation approach to optimizing high-performance distributed data stream processing," in *CIKM*, 2009.
- [8] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, jan 1991.
- [9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: the System S declarative stream processing engine," in *SIGMOD*, 2008.
- [10] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *DSN*, 2004.
- [11] A. Dixit, R. Heald, and A. Wood, "The impact of new technology on soft error rates," in *SELSE 6*, 2010.
- [12] S. Chandra and P. Chen, "How fail-stop are faulty programs?" in *FTCS*, 1998.
- [13] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLIFIED: Symbolic program-level fault injection and error detection framework," in *DSN*, 2008.
- [14] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: a distributed, scalable platform for data mining," in *DMSSP*, 2006.
- [15] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, 2002.
- [16] 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.
- [17] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, 2004.
- [18] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *DSN*, 2006.
- [19] H. C. Zhao, C. H. Xia, Z. Liu, and D. Towsley, "A unified modeling framework for distributed resource allocation of general fork and join processing networks," in *SIGMETRICS 2010*, 2010.
- [20] S. D. Viglas and J. F. Naughton, "Rate-based query optimization for streaming information sources," in *SIGMOD*, 2002.
- [21] S. Singh, A. Agbaria, F. Stevens, T. Courtney, J. F. Meyer, W. H. Sanders, and P. Pal, "Validation of a survivable publish-subscribe system," *International Scientific Journal of Computing*, vol. 4, no. 2, 2005.
- [22] M. Cinque, D. Cotroneo, C. Di Martino, and S. Russo, "Modeling and assessing the dependability of wireless sensor networks," in *SRDS*, 2007.
- [23] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, L. Votta, C. Vick, and A. Wood, "Modeling coordinated checkpointing for large-scale supercomputers," in *DSN*, 2005.
- [24] M. Hiller, A. Jhumka, and N. Suri, "Epic: Profiling the propagation and effect of data errors in software," *IEEE Transactions on Computers*, vol. 53, pp. 512–530, 2004.
- [25] A. Jhumka, M. Hiller, and N. Suri, "Assessing inter-modular error propagation in distributed software," in *SRDS*, 2001.