PARTIAL FAULT TOLERANCE IN STREAM PROCESSING APPLICATIONS - METHODS AND EVALUATION TECHNIQUES

BY

GABRIELA JACQUES DA SILVA

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Associate Professor Steven S. Lumetta
Professor Klara Nahrstedt
Professor William H. Sanders
Assistant Professor Shobha Vasudevan
ABSTRACT

Stream processing emerged as a paradigm to continuously process incoming live data streams, such as audio, video, and business feeds. These applications are assembled as dataflow graphs, where each vertex of the graph is a stream operator and each edge is a stream connection. In this environment, a fault in a stream operator can result in massive data loss or in the generation of inaccurate results. Most of the fault tolerance solutions proposed for streaming applications aim at guaranteeing that no data is lost or that no data item is delivered to the application more than once. These techniques result in high performance overhead, given the need to coordinate the state stored in checkpoints of distributed components or maintain consistency between replicas. In this dissertation, we investigate partial fault tolerance methods, which protect only the most critical stream operators of a streaming application. These methods take advantage of the fact that stream processing algorithms are approximate by nature and, as a result, can still achieve acceptable results under data loss and duplicate data delivery. The methods proposed in this dissertation include a checkpoint-based mechanism and a partial graph replication technique. Both techniques were implemented in System S, IBM Research’s stream processing middleware. In addition, this dissertation describes two different fault tolerance evaluation techniques. The first technique is based on fault injection and is used to emulate the effects of partial fault tolerance on a streaming application. With the fault injection results, the developers can understand the impact of faults on the application output and identify the most critical operators on their streaming application. The second evaluation technique is a model-based framework which provides generic abstractions for representing streaming applications with the stochastic activity network formalism. The framework allows the comparison of different fault tolerance techniques under varying fault models. Based on the results, the developers can evaluate the trade-offs that a certain technique provides when applied to their target application.
ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Prof. Ravi Iyer, for giving me the opportunity to work in his research group and letting me pursue the area of fault tolerance in stream processing applications. In addition, I’m grateful to Dr. Zbigniew Kalbarczk for always being available to discuss my research problems. I have learned a lot while in Illinois and in the DEPEND group, and I am deeply thankful to them for that. I also would like to thank my doctoral committee for their valuable feedback during my preliminary examination and in the past year. Super special thanks go to Dr. Jim Giles, Dr. Kun-Lung Wu, Dr. Buğra Gedik, and Dr. Henrique Andrade, from IBM Research, who introduced me to the topic of stream processing systems and had extensive discussions with me about how to provide fault tolerance for these systems. I am also grateful to Heidi Leerkamp, my other colleagues from the DEPEND group, and friends from UIUC and Brazil, for giving me support throughout my doctoral years. I also thank CAPES/Brazil (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), the Fulbright Association, and IBM for sponsoring my Ph.D. program. Finally, I am especially grateful to my family, namely (in alphabetical order!), Caroline, Celso, Eliseu, Joana, and Marlise. They have provided invaluable help to me during these tough, tough years in Illinois. Thanks for making me stick to it.
CONTENTS

LIST OF ABBREVIATIONS ........................................ vi

Chapter 1  INTRODUCTION ..................................... 1
  1.1 Motivation ................................................. 1
  1.2 Goals ....................................................... 2
  1.3 Contributions ............................................. 2
  1.4 Dissertation Organization ................................. 4

Chapter 2  RELATED WORK ...................................... 6
  2.1 Nomenclature ............................................... 6
  2.2 Stream Processing Systems ................................. 8
  2.3 Stream Processing Languages ............................. 11
  2.4 Fault Tolerance for Stream Processing Applications ... 14
  2.5 Summary ................................................... 16

Chapter 3  CHECKPOINTING FOR STREAM PROCESSING
  APPLICATIONS ................................................. 18
  3.1 System S under Failures ................................... 19
  3.2 Checkpoint Design ....................................... 21
  3.3 Experimental Evaluation .................................. 31
  3.4 Related Work ............................................. 39
  3.5 Summary ................................................... 40

Chapter 4  FAULT INJECTION INTO STREAM PROCESSING
  APPLICATIONS ................................................ 41
  4.1 Partial Fault Tolerance .................................... 43
  4.2 Evaluation Methodology ................................... 43
  4.3 Fault Injection Framework ................................. 47
  4.4 Evaluating the Fault Injection Outcome ................. 51
  4.5 Experimental Evaluation .................................. 59
  4.6 Related Work ............................................. 70
  4.7 Summary ................................................... 71
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>5.1 Stochastic Activity Networks</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>5.2 Application Model</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>5.3 Fault and Failure Model</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>5.4 Error Propagation Model</td>
<td>83</td>
</tr>
<tr>
<td>5.5</td>
<td>5.5 Modeling Fault Tolerance Techniques</td>
<td>88</td>
</tr>
<tr>
<td>5.6</td>
<td>5.6 Evaluation</td>
<td>94</td>
</tr>
<tr>
<td>5.7</td>
<td>5.7 Model Validation</td>
<td>103</td>
</tr>
<tr>
<td>5.8</td>
<td>5.8 Model Limitations</td>
<td>105</td>
</tr>
<tr>
<td>5.9</td>
<td>5.9 Related Work</td>
<td>106</td>
</tr>
<tr>
<td>5.10</td>
<td>5.10 Summary</td>
<td>107</td>
</tr>
<tr>
<td>6.1</td>
<td>6.1 Lessons Learned</td>
<td>110</td>
</tr>
<tr>
<td>6.2</td>
<td>6.2 Future Work</td>
<td>111</td>
</tr>
<tr>
<td>A.1</td>
<td>A.1 Source Operator</td>
<td>114</td>
</tr>
<tr>
<td>A.2</td>
<td>A.2 Sink Operator</td>
<td>115</td>
</tr>
<tr>
<td>A.3</td>
<td>A.3 Functon Operator</td>
<td>116</td>
</tr>
<tr>
<td>A.4</td>
<td>A.4 Aggregate Operator</td>
<td>117</td>
</tr>
<tr>
<td>A.5</td>
<td>A.5 Join Operator</td>
<td>119</td>
</tr>
<tr>
<td>A.6</td>
<td>A.6 Sort Operator</td>
<td>120</td>
</tr>
<tr>
<td>A.7</td>
<td>A.7 Barrier Operator</td>
<td>121</td>
</tr>
<tr>
<td>A.8</td>
<td>A.8 Punctor Operator</td>
<td>121</td>
</tr>
<tr>
<td>A.9</td>
<td>A.9 Split Operator</td>
<td>122</td>
</tr>
<tr>
<td>A.10</td>
<td>A.10 Delay Operator</td>
<td>123</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
<td>125</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ANOVA</td>
<td>Analysis of Variance</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CDF</td>
<td>Cumulative Distribution Function</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DGM</td>
<td>Dataflow Graph Manager</td>
</tr>
<tr>
<td>FIOP</td>
<td>Fault Injection Operator</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HA</td>
<td>High-Availability</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JMN</td>
<td>Job Manager</td>
</tr>
<tr>
<td>MNC</td>
<td>Master Node Controller</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>OSF</td>
<td>Output Score Function</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PFT</td>
<td>Partial Fault Tolerance</td>
</tr>
<tr>
<td>QM</td>
<td>Quality Metric</td>
</tr>
<tr>
<td>QS</td>
<td>Quality Score</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>RMN</td>
<td>Resource Manager</td>
</tr>
<tr>
<td>RMSE</td>
<td>Root Mean Square Error</td>
</tr>
<tr>
<td>SAN</td>
<td>Stochastic Activity Network</td>
</tr>
<tr>
<td>SDC</td>
<td>Silent Data Corruption</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SPADE</td>
<td>Stream Processing Application Declarative Engine</td>
</tr>
<tr>
<td>SPC</td>
<td>Stream Processing Core</td>
</tr>
<tr>
<td>SSE</td>
<td>Sum of Squared Errors</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UBOP</td>
<td>User-Defined Built-in Operator</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UDOP</td>
<td>User-Defined Operator</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over IP</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1
INTRODUCTION

1.1 Motivation

Stream processing is a paradigm to analyze live streaming data, such as audio, sensor readings, and news feeds. Traditional solutions for data analysis, such as database management systems (DBMSs), are usually not suitable for the extremely high rates of data streams. DBMSs execute queries that have exact answers over the stored data. Stream-oriented applications continuously query data that arrives asynchronously and compute answers over incomplete information [1, 2]. Examples of streaming applications include fraud detection [3], financial applications [4], network monitoring [5, 6], system anomaly prediction [7, 8], road traffic monitoring [9, 10, 11], analysis of geophysical events [12, 13], and sensor-based patient monitoring [14].

Streaming applications process continuous live data, making high-availability a key requirement [15]. Developers build streaming applications by assembling stream operators as dataflow graphs, which can be distributed over a set of nodes to achieve high performance and scalability. A failure in a computing node or in the stream operator itself can result in massive loss of data or in a delayed application output. On one hand, large amounts of data loss can lead to imprecise results. On the other hand, in a number of streaming applications, it may be better to produce partial results within a time bound than to produce complete results too late [16].

Most fault tolerance techniques for stream computing [17, 18, 19, 20, 21] consider that no stream data can be lost or duplicated, and the state of replicas of the same component should not diverge. This assumption results in extra overhead in the communication substrate, decreasing the overall system throughput [22]. In systems with high throughput and real-time requirements, resources should be allocated with parsimony. Fault tolerance
techniques that enforce no data loss and duplication under failures have a high impact on the performance of a streaming application, which can become unaffordable if we consider the data rates projected for the coming years [23]. Moreover, in this scenario, it is not necessary to ensure such semantics in order to achieve correctness under failure events. The rationale is that many streaming applications tolerate data imprecision by design, and, as a result, can still achieve correctness with data loss and duplication. Techniques for handling streaming systems that are operating over their load capacity also take advantage of this application characteristic. In overload situations, the system sheds its load by randomly dropping stream data items [24, 25, 26].

1.2 Goals

The first goal of this dissertation is to protect a streaming application against faults with methods that have low performance and resource overhead but still achieve correctness under the occurrence of faults. By providing configurable fault tolerance approaches, the developers can tune the fault tolerance solution specifically for their application and pay performance and resource overheads only for the parts of the stream processing graph that are necessary. The second goal of this work is to provide experimental and model-based methods to evaluate the impact of faults on the application output when a specific fault tolerance technique is applied. With such methods, the developers can understand how faults can affect their applications, compare the trade-offs between different techniques, and decide how to better deploy fault tolerance in their applications.

1.3 Contributions

The main contributions of this research can be summarized as follows.

1. Partial fault tolerance techniques for streaming applications, which allow the application developer to selectively apply fault tolerance mechanisms to the stream processing graph [27]. The techniques are deployed via language level annotations. As a result, only applications that need a given protection pay for the incurred performance overhead. Applying techniques directly on
the stream processing system middleware decreases the system throughput for all applications, even for the ones that do not require such protection.

We developed specialized checkpointing and partial graph replication techniques that do not strictly enforce that a stream data item should not be lost or duplicated (i.e., duplicate delivery of the same data item), avoiding high cost protocols in the communication substrate used between the stream operators of a streaming application. These techniques favor performance and application output timeliness over precise computation of every output data item. Both techniques were implemented and evaluated over SPADE (Stream Processing Application Declarative Engine) [28]. SPADE is the declarative stream processing language that is part of System S, IBM Research’s stream processing middleware [3, 29, 30, 31, 32] and basis for IBM’s commercial stream processing solution InfoSphere Streams [33, 34]. Parts of the developed checkpointing techniques are included in the InfoSphere Streams Version 1.2 distribution [35]. This feature has been used in real streaming applications, such as the checkpointing of bloom filter data structures used to detect duplicate incoming call detail records1.

2. An experimental evaluation methodology tailored to streaming applications, which is based on injecting faults into different stream operators and observing the fault’s effect on the application output quality. This methodology helps the developers to assess if the chosen fault tolerance technique is adequate for their application and to identify which stream operators are the most critical for the application to maintain quality of service.

In short, the proposed methodology emulates the effect of a fault in a streaming application when a specific fault tolerance mechanism is in place. The method uses a user-defined output score function to measure the application output quality under faults and then uses this function to compute four metrics that characterize the behavior of individual stream operators under faults. The proposed evaluation metrics are the outage duration impact, the data dependency, the recovery time, and the quality impact. We demonstrate this methodology by injecting a bursty data loss fault model in a financial engineering streaming application [4]. Our results confirm that faults affecting different stream operators quite differently impact the application output quality, demonstrating that operator sensitivity to faults can

---

1A call detail record contains data related to a telephone exchange.
be used to deploy partial fault tolerance techniques.

3. A framework for model-based evaluation of fault tolerance techniques applied to streaming applications, which allows the evaluation of trade-offs of different techniques and under varying fault models. Our framework is based on three key abstractions: stream operators, stream connections, and tuples. By composing these abstractions within a stochastic activity network (SAN) [36], we allow the modeling and evaluation of complete streaming applications.

The evaluation framework considers faults that lead to data loss and to silent data corruption (SDC). Importantly, the model captures the error behavior of these two failure modes by retrofitting the behavior observed in real fault injection experiments into the model. The model also captures how faults originating in one operator propagate to other operators down the stream processing graph. Finally, one of the unique aspects of the proposed framework is the evaluation of the application through the combination of the effects of faults on application data and its state.

We demonstrate the extensibility of our framework by evaluating the impact of faults when three different fault tolerance techniques are applied: checkpointing, partial graph replication, and full graph replication [22]. Our study shows 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 silent data corruption can break the no data duplication guarantees of a full graph replication-based fault tolerance technique.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 describes related work on the area of stream processing systems. Chapter 3 describes the design of the checkpointing technique, language extensions to SPADE and application output accuracy and performance experiments. Chapter 4 describes the experimental evaluation methodology, the proposed evaluation metrics, and an experiment conducted with a financial engineering application. Chapter 5 describes our modeling framework, detailing the application model and its mapping to the SAN formalism. This chapter also includes
a description of the partial graph replication technique. Finally, Chapter 6 concludes this dissertation and proposes future work.
Chapter 2

RELATED WORK

This chapter describes research related to this dissertation. First, we introduce the nomenclature related to stream processing applications. Sections 2.2 and 2.3 show an overview of current middleware and languages for deploying streaming applications. Finally, Section 2.4 reports on fault tolerance techniques designed specifically for streaming applications.

2.1 Nomenclature

As described in Chapter 1, streaming applications aim at extracting knowledge from live data sources online. To achieve this objective, these applications are organized as dataflow graphs, where a node processes the incoming data, and sends it to the next node in the graph. Figure 2.1 shows the nomenclature associated with these applications, which is used throughout this dissertation.

Figure 2.1: Nomenclature associated with a stream processing application.

- Stream source - a stream data source is a device that generates data that is of interest to a given application. The application connects to the stream source, which periodically forwards the available data to be processed by the flow graph. Examples of a stream source include
sensors [37] and servers hosting user-generated content (e.g., Twitter [38, 39] and Facebook [40] feeds).

- **Sink** - a sink is a component that stores, displays, or forwards the data produced by the application. Examples of a sink are a database or a user interface that shows the processing results in real-time.

- **Stream operator** - receives data from its input streams and executes a function over the incoming data item. Depending on the semantics of the function (i.e., operator type), it may send out a data item to the next operator. Examples of stream operators are filter, window-based aggregate [41], window-based join [41], barrier, and union.

- **Stream connection** - interconnects and transmits data items between different stream operators. If stream operators reside in the same process of an operating system [28, 42], stream connections can be implemented as a producer-consumer data structure. If operators reside in different nodes of a distributed system, stream connections can be implemented via inter-node communication protocols (e.g., TCP, UDP).

- **Tuples** - a data item being processed by an operator or flowing through a stream connection. A tuple has a finite number of attributes, and each attribute has a value associated to it.

- **Upstream set** - the upstream set of a given operator is the set with all other operators connected directly and indirectly (via another operator) to its input streams (i.e., operators towards the stream source). In the example depicted in Figure 2.1, the operators in the upstream set of operator $op_5$ are $op_3$, $op_4$, $op_1$, and $op_2$.

- **Downstream set** - the downstream set of a given operator is the set with all other operators connected directly and indirectly to its output streams (i.e., operators towards the sink). In Figure 2.1, the operators in the downstream set of operator $op_4$ are $op_5$ and $op_6$.

A formal model of a streaming application is given in Chapter 5.
2.2 Stream Processing Systems

Many stream processing systems have been developed in past years, both in academia and in industry. One of the first attempts to build a stream processing engine was the Aurora project [43], developed at Brandeis University, Brown University, and MIT. Aurora allows the developer to build streaming applications by composing a flow graph by interconnecting different operators. The flow graphs can consume data from both live data streams and historical data (e.g., DBMS). The original Aurora prototype focused on executing the dataflows in a single node. In contrast with Aurora, Medusa [44] aimed at composing applications where the resources (e.g., stream sources, sinks) can be geographically distributed. Both systems later evolved into the Borealis stream processing engine [45]. Borealis uses the same application model as Aurora, but improves on it by allowing the operator graph to be distributed in several nodes. In addition, it allows dynamic query modification, so that the operator logic can be adjusted during application runtime.

TelegraphCQ [46] is another academic project related to stream processing. It was developed at the University of California, Berkeley. TelegraphCQ was developed on top of the PostgreSQL relational database [47]. It adds new constructs to SQL in order to allow development of continuous queries. The first prototype of TelegraphCQ focused on a multi-threaded implementation instead of a distributed version.

More recent academic approaches for processing live data streams include StreamMine [48] and MapReduce Online [49]. StreamMine, developed at the Technische Universität Dresden, aims at supporting distributed stream processing applications. It uses a software transactional memory infrastructure to allow speculation when operators process parallel events. MapReduce Online was developed at the University of California, Berkeley, and is an adaptation of the original MapReduce paradigm [50] to support continuous queries. The main difference between this implementation and other data stream processing systems is that it uses buffers to send the output of a map task to a reduce task. Data from the map task to the reduce task are only transmitted after the buffer is full, increasing the processing latency.

Industry initiatives include Microsoft StreamInsight [51, 52], Streambase [53], Sybase Alery Streaming Platform [54], and IBM Research’s System S [3, 29, 30, 31, 32]. The next section discusses the architecture of System S.
S, which was the middleware used to test the fault tolerance methods and evaluation techniques proposed in this dissertation.

2.2.1 IBM System S

IBM System S is a distributed stream processing system that aims at running large-scale streaming applications. One of its characteristics is to be non-transactional, since it does not have atomicity or durability guarantees. This is typical in stream processing systems where applications are continuously running and quickly producing results.

In System S, independent executions of an application with the same input may generate different outputs. There are two main reasons for this non-determinism. First, stream operators often consume data from multiple inputs (e.g., \(op_5\) in Figure 2.1). If the data transport subsystem does not enforce message ordering across data coming from different sources, there is no guarantee in terms of which message the operator will consume first. Second, operators can use time-based windows. Some stream operators (e.g., aggregate and join) produce output based on the data within specified window boundaries. For example, if a programmer declares a window which accumulates data over 20 seconds, there is no guarantee that two different runs receive the same amount of data in the defined time interval.

System S deploys each application as a job. A job is composed of multiple processing elements (PEs), which are containers for the operators that make up an application dataflow graph. A PE is mapped to a process running on top of an operating system. PEs can be placed in any node available to the System S runtime infrastructure. Each PE hosts one or more stream operators. Stream operators are fused into the same PE when the cost of processing a tuple locally is smaller than sending it via the network [42, 28, 55]. Figure 2.2 shows a possible deployment of the sample application in Figure 2.1 when running on top of System S. In this example, operators \(op_1\) and \(op_3\) are fused into the same PE and are running in a different node from the other operators of the processing graph.

Figure 2.3 shows the simplified architecture of the System S infrastructure. To run a given job, the user contacts the job manager (JMN) [31, 56], which is responsible for dispatching the PEs to remote nodes. The JMN contacts a
Operators may be placed in the same process and in the same node.

Figure 2.2: Deployment of a streaming application running on System S.

resource manager (RMN) to check for available nodes in the system. Then, JMN contacts the master node controller (MNC) on the remote nodes, which instantiates the PEs locally. Once the PEs are running, the stream processing core (SPC) [29, 30] is responsible for carrying out the stream connections and transporting tuples between PEs. The dataflow graph manager (DGM) guides the SPC by informing which PEs should be connected to each other. The SPC also transports data between PEs. The storage subsystem is a shared file system accessible to all nodes.

Figure 2.3: System S architecture.

This dissertation focuses on the fault tolerance of streaming applications. The proposed fault tolerance methods are implemented directly into stream operator code and into their support libraries (Chapter 3). The problem of
fault tolerance in infrastructure components of a stream processing middleware and its validation is tackled in our earlier work [31, 56].

2.3 Stream Processing Languages

Many high-level languages have been developed in the past few years that target stream processing applications. They aim at providing the user direct abstractions that fit the stream paradigm and facilitate the development of streaming applications. Examples of such abstractions are the representation of a stream and the ability to express time-based operations.

CQL [57], StreamSQL [58], GSQL [59], and ESL [60] are SQL-like languages that allow the development of continuous queries by defining streams and allowing a variety of windowing semantics. The motivation for a SQL variation comes from its widespread use for querying relational databases. Once the users submit their continuous queries, the stream processing system generates a representation of the queries as dataflow graphs, which express how the operators send data to each other. Once the operators start consuming data from stream sources, the application starts producing results.

Another set of languages allows composing a streaming application directly as a graph. One example is the Aurora system, which provides a way for the developer to assemble the application by connecting boxes and arrows [43]. StreamIt [61] is a programming language developed at MIT which allows the composition of graphs by using the following three types of operator interconnection patterns: pipeline, split-join, and feedback loop. StreamIt gives freedom to the programmers to specify their own logic for each stream operator. SPADE [28] is the language and compiler for creating distributed data stream processing applications to be deployed on IBM’s System S. In Chapter 3, we show how we extended SPADE to support the fault tolerance methods proposed in this dissertation. The next section describes the structure of SPADE and its main features.

2.3.1 SPADE

SPADE [28] is a language and a compiler for creating distributed data stream processing applications to be deployed on System S. SPADE offers (i) a lan-
guage for flexible composition of parallel and distributed dataflow graphs; (ii) a toolkit of type-generic built-in stream processing operators; (iii) an extensible operator framework, which supports the addition of new type-generic and configurable operators (user-defined built-in operators [UBOPs]) to the language, as well as new user-defined non-generic operators (user-defined operators [UDOPs]) used to wrap existing, possibly legacy analytics; and (iv) a broad range of edge adapters used to ingest data from outside sources and publish data to outside destinations, such as network sockets, databases, and file systems.

The SPADE language provides a stream-centric, operator-based programming model. The stream-centric design implies a programming language where an application writer can quickly translate the flows of data from a block diagram prototype into the application skeleton by simply listing the stream dataflows. The second aspect, i.e., operator-based programming, is focused on designing the application by considering the smallest possible building blocks that are necessary to deliver the computation an application has to perform. In summary, SPADE programs are dataflow graphs, where operators are connected via streams and serve as basic computational units performing stream transformations. Each stream follows a defined schema, which can be built from SPADE basic types (e.g., integer, string). All tuples going through a stream follow the schema defined for that stream.

Figure 2.4 shows a sample application written in SPADE. This application uses as a stream source the error log produced by host targetHost.crhc.illinois.edu. The data from the stream source is fetched to the application via a source (Source) operator, which then produces a Log stream. The Log stream is consumed by two other functor (Functor) operators. These two operators filter the incoming events by the value of the eventPriority attribute. The first functor produces the InfoEvents stream, which contains events with priority INFO. The second functor filters events with priority ALERT and produces an AlertEvents stream. This stream is consumed by a UDOP (Udop), which sends an email to the host administrator notifying the alert. After the UDOP processes the event, it forwards the event to the Sink operator, which stores a copy of the event in a local file called AlertEvents.log. Events related to the InfoEvents stream are stored into the InfoEvents.log file. The equivalent flow graph of this application is shown in Figure 2.5, where each SPADE op-
operator maps into a box. In this figure, the boxes are labeled according to
the output stream it produces or the operator type (Sink operator). Note,
however, that SPADE associate names to streams and not to operators.

```plaintext
stream Log(dateTime: String, eventPriority: String,
    eventMessage: String)
    := Source()
    ["ctcp://targetHost.crhc.illinois.edu:9876/"{}

stream InfoEvents(dateTime: String, eventMessage: String)
    := Functor(Log)[eventPriority = "INFO"]{}

stream AlertEvents(dateTime: String, eventMessage: String)
    := Functor(Log)[eventPriority = "ALERT"]{}

stream SendEmailAdmin(dateTime: String, eventMessage: String)
    := Udop(AlertEvents)[]{}

Nil := Sink(SendEmailAdmin)["file:///AlertEvents.log"]{}
Nil := Sink(InfoEvents)["file:///InfoEvents.log"]{}
```

Figure 2.4: Sample SPADE code for a log processing application.

![Diagram](image.png)

Figure 2.5: Corresponding dataflow graph for a log processing application.

A key distinction between SPADE and other stream processing middleware is its emphasis on code generation. Given an application specification in the SPADE language, the SPADE compiler generates specialized application code based on the computation and communication capabilities of the runtime environment. This specialization includes many aspects, such as rate-adaptiveness [62] and code fusion [42, 55]. In this dissertation, we take advantage of SPADE’s code generation capability to generate specialized fault tolerance algorithms.
2.4 Fault Tolerance for Stream Processing Applications

A number of fault tolerance techniques target streaming applications. In this dissertation, we classify them into the following two categories: (i) **stringent**, where the fault tolerance method aims to guarantee that no tuple is lost and that no tuple is duplicated; and (ii) **partial**, where the fault tolerance technique allows tuple loss and tuple duplication under faults. These techniques favor performance over a no-data-loss and no-data-duplication guarantee. The techniques presented in this dissertation fall into the partial category.

2.4.1 Stringent Fault Tolerance Techniques

Shah et al. [20] present Flux, a variant of the process-pair technique [63]. To avoid tuple loss and duplication, the technique proposes the addition of special *ingress* and *egress* operators, which help to ensure message delivery, message ordering, and failover between replicated operators. This approach requires stream operators to be deterministic and replicated operators to receive all tuples in the same order.

Balazinska et al. [17, 64] propose a protocol called DPC (delay, process and correct). DPC aims to achieve both availability and result consistency. To reduce the processing latency upon the occurrence of a failure, DPC allows stream operators to produce tentative tuples. These tentative tuples are corrected once the failed component recovers. Similarly to Flux, DPC requires operators to be deterministic.

Hwang et al. [65] describe three techniques for high-availability in streaming applications: (i) passive standby, (ii) active standby, and (iii) upstream backup. In the passive standby approach, operators checkpoint their internal state and communication queues to a backup replica. In active standby, the backup replica actively processes tuples. The output *tuples* produced by the replica are logged, so they can be replayed in case of a failure. Upstream backup uses the upstream nodes as backups for the downstream neighbors by preserving tuples in their output queues until their downstream neighbors have processed them. In these schemes, the communication substrate is modified to coordinate which tuples should be present in checkpoints (if any) and in input/output queues of other operators in the system.

Cai et al. [66] propose a hybrid approach for maintaining backup repli-
cas. In this technique, replicas are brought up-to-date periodically. The frequency of the checkpoint increases when a system monitor predicts a failure. Similarly to the approaches proposed in [65], this technique checkpoints the internal operator state and logs the outgoing tuples.

SGuard [19] provide fault tolerance for streaming applications via checkpointing. The application checkpoints are saved to disk using a distributed and replicated file system. SGuard provides a memory management middleware, which uses application-level copy-on-write to perform asynchronous checkpointing. The application developer uses specialized classes that allow the memory management middleware to transparently checkpoint the application state. Like previous approaches, SGuard checkpoints the internal data structures of each operator and logs input/output tuples in case they need to be replayed after a crash.

Hwang et al. [22] describe a full replication technique that allows each operator to receive inputs from replicated dataflows. In this technique, both operator replicas are active and de-duplicate data items coming from redundant streams. For de-duplication of tuples, the operator implementation logic is modified so that replicas can work in a deterministic fashion. In Chapter 5, we use the modeling framework proposed in this dissertation to provide the evaluation of this technique when faults leading to silent data corruption occur.

StreamMine provides two methods for fault tolerance. The first is based on checkpointing [67]. In this approach, the operator checkpoints internal states and logs non-deterministic events to allow precise replay upon a recovery. The second approach is based on active replication [68]. Replicated operators can be single or multi-threaded. In this implementation, the communication substrate uses an atomic broadcast protocol to deliver messages to the replicas. The time spent on achieving consensus is overlapped with processing time by using speculative processing of tuples.

In [69], Gu et al. propose the sweeping checkpoint method. This technique checkpoints the internal state and the output queues of operators. It triggers the checkpoint after receiving acknowledge messages from downstream operators. Zhang et al. [70] use the same checkpoint method to deploy a hybrid fault tolerance technique, which combines passive and active operator replicas.
2.4.2 Partial Fault Tolerance Techniques

Besides stringent fault tolerance techniques, Hwang et al. [65] also mention the use of an *amnesia* technique. This technique does not save any data related to operator state and input/output communication queues. Their work does not evaluate the impact of such techniques on the application output, which is the focus of Chapters 4 and 5 of this dissertation.

Murty and Welsh [71] enumerate fault tolerance techniques that can be used for applications that process live data. One such technique is *structured operator replication*, which replicates operators that are considered important to the query. One example of replication criteria is the *depth* of the operator in the flow graph. Another technique is *free-running operators*, which does not require operators to be in strict consistency. The authors do not discuss the evaluation of the proposed techniques or the fault tolerance deployment criteria.

LSS [21] (lightweight summary structure) is a checkpointing technique for streaming applications. The authors describe stream operators as having the following behavior: read data from streams, process data, and accumulate intermediate results. The authors propose to save the state of the stream operator at the end of each loop iteration. In this approach, tuples can be lost during operator failure and recovery.

Zen [16] is a resource allocation framework for partial fault tolerant stream processing applications. It considers the importance of operators and the failure characteristic of a given cluster to decide how to assign stream operators to computing nodes. In Zen, it is assumed that the importance of a component can be defined as a linear combination of the importance of its inputs. Our fault injection experiments with a streaming application presented in Chapter 4 show that this assumption does not necessarily hold.

2.5 Summary

This chapter summarizes research on the area of stream processing systems. Both academic and industry research initiatives in this area are introduced. In addition, the chapter details the architecture of IBM’s stream processing middleware System S and its programming language SPADE. Finally, fault tolerance approaches for streaming applications are described. Additional
comparisons between these techniques and those proposed in this dissertation are included at the end of each chapter of this dissertation.
Chapter 3

CHECKPOINTING FOR STREAM PROCESSING APPLICATIONS

For streaming applications to generate semantically correct results, even in the presence of failures, it is essential to employ fault tolerance techniques. For instance, sensor-based patient monitoring applications require rigorous fault tolerance, since data loss or computation errors may lead to catastrophic results. On the other hand, there are applications that do not have such strict requirements. One such example is an application that discovers pairs of caller/callee by data-mining a set of VoIP streams [72]. In case of failures, VoIP packets may be lost or a user can get disconnected from the VoIP system. The application can still infer the caller/callee pairs, although with less confidence. Such a class of applications is called partial fault tolerant. Moreover, in some streaming applications it may be better to produce partial results within a time bound than to produce complete results too late [16]. In systems that aim to provide maximum data throughput, resources must be spent with parsimony. Therefore, one-size-fits-all fault tolerance is not the best approach for streaming applications.

To achieve massive parallelism and scalability, stream processing systems distribute the application over nodes in a cluster. Hence, they are subject to the failure model of distributed systems [73], where messages can be omitted, duplicated, nodes can crash, and the network can become partitioned. A successful technique to provide fault tolerance to large-scale distributed systems is synchronous checkpointing [74]. While it can be applicable to streaming applications, more specialized approaches can be more appropriate. A streaming application is one particular type of parallel application. While the latter, in general, has clear synchronization points (barriers), the former is constantly changing its state based on the incoming data. Additionally, a typical streaming application has real-time constraints. Stopping the application to coordinate a checkpoint operation directly affects the timing requirements. Many fault tolerance techniques developed specifically for
stream computing [19, 21, 65] do not consider the semantics of the application and apply fault tolerance throughout the system. This results in unnecessary performance overhead and an overall degradation of performance when contrasted with a more targeted approach.

In this chapter we describe the design of checkpointing techniques for partial fault tolerant streaming applications. We deployed such techniques in SPADE and System S. To provide flexibility to the user, we do not enforce a single checkpointing policy for the whole application. More specifically, we allow the application developers to annotate their SPADE application source code, so that they can choose which parts of their application should be fault tolerant. The use of language-level annotations is a natural approach to specify such policies, since the developers know their application semantics and failure behavior. To carry out the behavior chosen by the user, we take advantage of SPADE’s code generation framework to automatically produce the extra code required by the fault tolerance policies.

The main contributions of the work described in this chapter are (i) a framework for applying fault tolerance policies for streaming applications via language annotations; (ii) an incremental checkpointing algorithm for sliding window-based stream operators; and (iii) a code generator that outputs specialized checkpointing code based on the stream operator type and instance.

The rest of this chapter is organized as follows. Section 3.1 discusses the behavior of System S under failures. Section 3.2 describes the checkpointing techniques for user-defined operators and window-based operators. Section 3.3 demonstrates the applicability and benefits of the technique and includes several performance studies using a real-world manufacturing application as well as synthetic applications. Section 3.4 compares the proposed approach with previous research. Finally, Section 3.5 provides a summary of the chapter.

3.1 System S under Failures

The System S middleware has many self-healing features. As a central component, JMN plays a fundamental role in this. Besides dispatching PEs, JMN also monitors their life-cycle. Each MNC monitors which PEs are alive...
in its local node and sends this information to JMN. If a PE fails, JMN detects it and re-dispatches the PE in the same node. If the PE has crashed due to a node failure, JMN may restart the PE in a different node. During the recovery time, the behavior of the PEs connected to the crashed PE differs. The behavior depends on the specific position of the PE in the dataflow graph, as it is shown by Figures 3.1 and 3.2.

Figure 3.1 shows an example of a graph with six PEs. PE 1 sends the same data to PE 3 and PE 4. PE 4 also consumes data from PE 2. PE 5 and 6 consume data from PE 3 and 4, respectively. Figure 3.2 shows the consequence of a failure in PE 4. As expected, PE 6 does not have input streams to process; therefore, it does not produce any data. The behavior differs for PE 1 and 2, since they are data producers (also referred to as source PEs). SPC discards all the new data PE 2 sends, given that there is no PE to consume it. PE 1 still has one live connection, so it continues to send new data to PE 3, but it stops sending data to PE 4. Once PE 4 is reintegrated, the connections are re-established. At this point, PE 2 stops discarding data and PE 1 resumes sending new data to both links. More details on SPC and System S failure behavior can be found in earlier work [29, 31, 56].

The operator fusion feature of SPADE also has an important implication for how applications can fail in System S. As described in Section 2.2.1, fusing operators results in placing multiple operators inside a single PE. Instead of using the regular stream transport, the streams are converted into function calls [4]. Because all operators in a PE reside in a single process and share the same address space, if an operator in the fused group crashes, the whole set of operators hosted by a PE also crashes.
3.2 Checkpoint Design

As described in Section 2.3.1, Spade has an operator-based programming model. To checkpoint an operator, it is important to define the behavior of the operator under failure and the state it should have once it recovers. To minimize the performance overhead, we have to develop techniques that are specific to individual operator types and that can be customizable by different instances a developer might employ in the context of one application. Here is where we can take advantage of Spade’s compiler and code generation framework. With the knowledge of the application, Spade can generate code that uses specialized checkpoint techniques for each operator instance based on their parameters. For each operator we have to take into account the following three aspects: (i) the minimal operator state required in order to recover after a failure, (ii) whether the operator is able to produce semantically correct results after a recovery from a failure, and (iii) whether the restored state contains stale data. We assume a fail-stop model for an operator failure.

Regarding operator state, Spade has both stateless and stateful operators. An example of an operator that can be either stateless or stateful is a functor. Functors perform tuple-level manipulations such as filtering, projection, and mapping. For every incoming tuple, the functor generates an output, unless the input tuple does not satisfy the filter predicate. If the filter predicate does not involve any variables other than the attributes of the current tuple (e.g., a stream attribute in a tuple is greater than a constant value), there is no need to save state. If the operator crashes and restarts, it can still filter tuples with the same predicate. The developer can also customize the functor operator to update state variables when it receives a tuple (e.g., to compute the running average for an attribute). In this case, a functor is stateful and
the state variables should be checkpointed. Note that the variable value can be affected by tuple loss (e.g., maximum value of an attribute). This should be considered when deploying the checkpoint technique.

Depending on the operator type, checkpointing the internal operator state and restarting it may not be sufficient to provide correct operator semantics. One such case is the **barrier** operator. Barriers synchronize logically related streams. The operator emits an output tuple every time it has at least one tuple from all of its inputs. As described in Section 3.1, when a PE fails, tuples may be lost during the recovery time. For a barrier operator to provide correct results after recovery, we need to save in-flight tuples. If tuples are lost, there is no guarantee that the logical pairing the operator produces is correct. For this type of operator, it is mandatory to apply other techniques that save in-flight tuples, such as upstream backup [65]. The advantage of our technique is that through code generation, we can enforce in-flight tuple buffering only for the operators that require such semantics.

In streaming applications, it is possible that the operator state is valid only during a certain time frame. One example is the **aggregate** operator. The aggregate operator groups and summarizes the incoming tuples according to an operation (e.g., sum, average, among others). It performs the operation over all the tuples that are inside a window boundary. One option available to the developer is to parameterize the window behavior based on size ($x$) and an output interval ($y$). The size of the window can be defined as all the tuples accumulated over the last $x$ seconds. As new tuples arrive, the operator discards tuples older than $x$ seconds. At every $y$ seconds, the operator computes the aggregate function based on the current contents of the window. On the event of a failure, the restored state of an aggregate contains all the tuples that are inside the window at the time of the checkpoint. This means that on recovery the middleware must handle stale data. In normal operation, some of these tuples would have been discarded due to the arrival of new tuples. Therefore, the recovery routine has to eliminate the expired tuples.

In the next sections we show how we modified the SPADE infrastructure to support checkpointing and how we added fault tolerance to *user-defined* operators as well as the built-in *join* operator, which is an example of a windowed operator. Our checkpointing technique applies to most other windowed operators; however, this chapter focuses only on the join.
3.2.1 SPADE Support

To selectively provide fault tolerance, we allow the users to define in their source code what parts of their application should be fault tolerant by employing language constructs. After the users annotate their application, the SPADE compiler generates code that saves the state of the selected operators with a pre-established frequency. For built-in SPADE operators, the compiler automatically generates checkpoint methods. The state of each operator is assumed to be independent from the others. When operators are fused in the same PE, each of their states is saved at their specified frequency. We do not save them all at the same time to allow maximum possible throughput. Due to performance overheads, the user may not want to checkpoint an operator with a large state at the same frequency of an operator with small state. The state independence also applies to PEs. During a PE recovery, the states of other interconnected PEs are not rolled back.

Figure 3.3 is an example of how to specify that an operator should be checkpointed. The example has a source (Source) and a UDOP (Udop) operator. The source stream (CountStream) produces tuples with an integer (count) and a string (str), extracted from the packets coming from a TCP client located at src.somewhere.com:789. The UDOP creates an AverageStream where each tuple contains an integer (avg) and a string (str). Note that the UDOP contains the checkpoint keyword and the associated checkpoint frequency in seconds (10).

```plaintext
stream CountStream(count: Int, str: String)
:= Source()
["ctcp://src.somewhere.com:789"]{}

stream AverageStream(avg: Int, str: String)
:= Udop(CountStream)["Avg"]{}
-> checkpoint=10 # 10 seconds
```

Figure 3.3: Checkpoint annotation in SPADE.

The SPADE compiler generates code both for the operators and the PEs (with or without fusion). For the operators, it generates extra code so it can implement the checkpointing policy. The extra code is dependent on the operator type and configuration. Further details on the generated code are given in Sections 3.2.3 and 3.2.4. The compiler modifies the configuration of
the PEs to selectively enable checkpoint services for its operators.

3.2.2 Runtime Support

The PE execution flow changes if any of the operators in a PE have the `checkpoint` keyword. Figure 3.4 shows the PE operation steps with checkpointing. A PE is comprised of a `PEWrapper`, which manages all the operators it contains (OP1, OP2, and OP3). When the PE starts, it initiates a thread (`PECheckpoint`) that is responsible for carrying out the checkpoint policy (step 1). `PECheckpoint` verifies which operators should checkpoint and builds a priority queue with their next checkpoint timeout. In step 2, the thread removes the next operator to checkpoint from the queue (`getExpiringOperator()`) and sleeps until it is time to save the next state snapshot. When the thread wakes up, it invokes the `getCheckpoint()` method of the operator (step 3). This method contains the logic to serialize the operator state. While the `getCheckpoint()` method executes, the operator cannot process any new incoming tuples. Once the method call returns, the thread saves the serialized state to the storage subsystem via the `OPState` class (step 4). `OPState` saves the new state in a temporary file first, which is later renamed to a permanent file through an atomic `rename()` operation.

![Figure 3.4: PE checkpoint operation.](image)

The PE recovery is similar. Its operation is shown in Figure 3.5. When the PE starts up, it invokes a method from the `PECheckpoint` class to start the recovery operation (step 1). This method searches for the checkpoint files based on the PE and the operator IDs (step 2). These identifiers are constant throughout the lifetime of the PE. At first, the restore procedure checks the
integrity of the checkpoint file via a hash value computed and stored by the \texttt{OPState} class. If the file is corrupted, the procedure discards the state and the operator restarts with a fresh state. One option is to maintain different versions of checkpoint files and restore an older checkpoint in case of corruption of the latest one. File corruption is handled differently when the operator rebuilds its state from multiple files. Such a case is described in Section 3.2.4 in the context of the join operator. When the checkpoint file is valid, the restore routine invokes the method \texttt{restoreCheckpoint()} in the operator class (step 3). Similarly to the serialization function, the de-serialization implementation is specialized for each operator type.

![Figure 3.5: PE recovery operation.](image)

3.2.3 User-Defined Operator

In \texttt{Spade}, the user has the capability to extend the basic set of built-in operators via user-defined operators (UDOPs). With UDOPs, the developers can use external libraries and implement customized operations for their particular application. \texttt{Spade} generates skeleton code so the user can easily handle tuples from System S streams, process them with the specialized code, and send them over as a System S stream to other operators.

To checkpoint UDOPs, the \texttt{Spade} compiler adds checkpoint method interfaces to the generated skeleton. The user has to fill in the methods with the appropriate serialization logic. This approach is similar to the technique employed in the fault tolerant CORBA standard for application-level state checkpointing [75]. The \texttt{PECheckpoint} thread automatically invokes the serialization methods at the specified frequency.
Figure 3.6 gives the generated interfaces and an example of how to add the serialization code. This is part of the Spade output for the code given in Figure 3.3. In this example, the state of the UDOP has two member variables, namely \texttt{avgCount} and \texttt{numCount}. The user receives a reference to a serialization buffer object (\texttt{SBuffer}), used both for the state saving and restoring methods. The user has to serialize/de-serialize the data to/from the buffer in the same order. Because other methods can modify the member variables during checkpoint, they must be protected by a mutual exclusion construct.

```cpp
void UDOP_Avg::getCheckpoint(SBuffer &checkpoint) {
    AutoMutex am(mutex);
    checkpoint << avgCount << numCount;
}

void UDOP_Avg::restoreCheckpoint(SBuffer &checkpoint) {
    AutoMutex am(mutex);
    checkpoint >> avgCount >> numCount;
}
```

Figure 3.6: UDOP checkpoint interface.

### 3.2.4 Join Operator

The \textit{join} operator correlates two streams. The streams are paired up based on the join predicate and the window configuration. Two different windows (one per incoming stream) group the tuples from each incoming stream. Each stream can have a different window configuration. The window keeps the input tuples in the order of arrival. Once the operator receives an input tuple from \textit{stream 1}, it evaluates the predicate condition against all tuples in the window of \textit{stream 2}. If the predicate evaluates to true, the operator pairs the matching tuples and sends them downstream. After the pairing stage, the operator inserts the input tuple into its corresponding window. If the window is full, the oldest tuple is discarded, i.e., the window \textit{slides}.

A join operator can have an arbitrarily large window. For example, the application described in Section 3.3.1 has three join operators. The sizes of two of them are 512,000 and 128,000 tuples. Tuples in join operators may accumulate over a long period of time, depending on the stream input rate.
If the operator crashes and there is no checkpoint, the operator produces few outputs for a long time, since it has to fill up its windows in order to produce matches at full rate. With checkpointing, we can recover most of the window content. Therefore, the operator is able to produce matches right after the restore operation.

If for the join operator we use the same checkpoint technique employed for UDOPs, all the tuples inside the window should be serialized. This results in the serialization of large chunks of data, which introduces a prohibitive performance overhead. To overcome this problem, we devised an incremental checkpointing technique for sliding window-based operators, such as joins. Incremental checkpointing algorithms for stream computing require performing a checkpoint maintenance operation per tuple that arrives to an operator [21]. For low-cost checkpointing, both tuple insertion and serialization should be lightweight operations.

Incremental Checkpoint Algorithm

In a sliding window configuration, as new tuples arrive, the older ones are evicted from the window. This behavior can be implemented with a double-ended queue data structure. New tuples are inserted at the tail of the queue and old tuples are removed from the head of the queue. Between two checkpoints, the state of the operator can be described by two possible configurations. In the first configuration the window only has new tuples, meaning that the total number of new tuples since the last checkpoint is greater than the size of the window. In this case there is no common state between the last checkpoint and the current one. If the number of new tuples is less than the size of the window, the serialization time can be decreased by not repeating this operation for tuples that are part of both the last and the current checkpoint.

We devised a circular buffer data structure that divides the sliding window into fixed groups of tuples. As a result, we can control the number of checkpoint files and avoid the garbage collection problem. At every checkpoint interval, we verify which groups have new tuples and save their contents to disk, serializing both new and old tuples in a group. We limit re-serialization by dividing the window into smaller groups. Groups that did not change between two checkpoint intervals do not need to be re-saved. Since it is a
sliding window, only groups with more recent tuples change.

Each position in the circular buffer contains the following data: (i) a checkpointing file name; (ii) a dirty bit, which indicates if group should be serialized to disk; (iii) the current number of tuples in the group; and (iv) the window index of the most recent tuple in the group, so we can correctly index the double-ended queue data structure. To decrease the performance overhead, our algorithm updates the circular buffer only at every checkpoint interval.

The number of positions in the circular buffer data structure is based on the number of tuples we want to save per checkpoint operation. We divide the window size by the number of tuples per checkpoint file and add one extra position. The extra position accounts for the window slide.

Figure 3.7 gives an example of a count-based 15-tuple sliding window. In this example, we divide the window in groups of five tuples. This results in four checkpoint groups. At checkpoint time $t_1$, the window contains 13 new tuples (A-M). G1 and G2 contain five tuples each, while G3 contains three. The checkpoint routine evaluates the circular buffer structure and finds that G1, G2, and G3 are dirty. At this point, it serializes the tuples based on the indexes maintained by the circular buffer. As in UDOPs, no tuple processing is allowed during serialization. After the serialization, the checkpoint routine cleans the dirty bits in the circular buffer and saves the tuple contents to disk. At time $t_2$, the window has five new tuples. G3 has tuples N-O and G4 has tuples P-R. G1 lost three tuples (A-C) due to the window slide. The checkpoint method checks that G3 and G4 are dirty and serializes all their contents to disk (K-R). Even though G1 lost tuples, its corresponding file is not updated. This file is invalid only after G1 loses all its tuples. The checkpoint thread reuses this file after the window slides by a whole group size.

Restore Algorithm

To recover the operator state, we read all files related to a window. Since we use a circular buffer, the first valid tuple of the window can be in any group. We start to rebuild the window from the file that has the oldest write timestamp. The oldest file corresponds to the window segment that was not updated for the longest time; hence, it is the beginning of the sliding window.
Figure 3.7: Join checkpointing operation.

Restoring all the tuples from the oldest file may result in a window bigger than its maximum size. Thus, we discard all tuples that exceed the total size of the window.

As mentioned in Section 3.2, another factor we have to consider during state restore operations is stale data. If the operator had not crashed, some of the tuples from the beginning of the window would have been discarded. We eliminate the stale data by estimating how many tuples would have been discarded during normal operation. The estimate is used to remove the top $N$ tuples from the window. We compute the number of stale tuples by the following formula:

$$N_{tuples} = (T_{recovery} - T_{serialization}) \times N_{tuples/sec}$$

$T_{recovery}$ is the time after the de-serialization routine completes. The time of operator state serialization ($T_{serialization}$) and the number of tuples per second ($N_{tuples/sec}$) are retrieved from the checkpoint file. Both data are obtained at runtime and are serialized with the tuples to the checkpoint file. Note that we calculate an approximation of the number of stale tuples, since there may be a variance on the input tuple rate.

Our recovery routine also handles corrupted checkpoint files. As the operator state is divided in different files, even if one of the files is corrupted by a disk failure, we can still recover the operator state. The implication of a corrupted file is the loss of, at most, the same number of tuples contained in a checkpoint group. If the application cannot tolerate tuple loss, the underlying file system has to apply replication techniques. Kwon et al. [19] showed that using replicated file systems to store checkpoint files has acceptable
performance overhead.

Code Generation

To enable checkpointing, the Spade compiler generated modified join code. The join operator code has two main methods – one for each input port. These methods are changed to include a mutual exclusion variable and a counter of new tuples per window. This is the only code added to the operator critical path. For per-group\(^1\) join operators [4] – where the operator allocates a new sliding window depending on the tuple key attribute content – we add code to dynamically allocate our circular buffer data structure and the new tuple counter.

The new counter helps to estimate the operator input rate and to update the indices contained in the circular buffer. Once the checkpoint method runs, it slides the circular buffer data structure by the number of new tuples. The circular buffer code is generic and does not need to be specialized for each join configuration. The only parameters it needs are the window size and the number of tuples each checkpoint group has. This number can be estimated based on the tuple size and operator input rate and can be learned during the operator profiling phase [42].

For incremental checkpointing, the checkpoint method interface changes. Since we need independent serializable buffers, we add extra checkpoint methods at the operator interface. The checkpoint thread invokes the correct checkpoint/restore method according to the operator type.

For per-group join operators we automatically generate a specialized class that associates serialization buffers and per-group keys. Since the key type is dependent on the tuple type, which is defined at the language level, the checkpoint thread does not know the key type. This specialized class abstracts the serialization buffer key-based access to the checkpoint thread. In Section 3.3.2 we show the overhead imposed by our technique both for join operators with one sliding window and for per-group joins.

\(^1\)A per-group operator emulates the existence of disjoint instances of the operator processing tuples associated with one logical group; for example, joining stock market transactions related to IBM independently from transactions related to Google. Note that, in this example, the per-group key attribute is the company attribute.
3.3 Experimental Evaluation

To evaluate our technique, we conducted two different experiments. The first employs checkpointing in a UDOP that is part of a real-world manufacturing application. The application builds statistical models over streaming data. The experimental objective is to evaluate how the application behaves under a UDOP failure by quantifying the crash impact on the accuracy of the application output. The second experiment evaluates the performance overhead of our checkpoint technique for join operators. We devised synthetic applications with different join configurations and evaluate how different checkpoint parameters impact the operator performance.

3.3.1 Application Output Deviation

To quantify the impact of our checkpoint technique on the application output, we used an application called FAB [76]. FAB generates two statistical models from several sensors embedded in semiconductor manufacturing tools used in IBM’s chip manufacturing facilities. The models are built to predict the wafer yield from the input sensors. One of FAB’s outputs is a quality metric (QM), which compares the value predicted by the statistical model and the ground truth, i.e., the actual wafer yield metric. For our runs, the actual wafer yield was collected along with sensor reading from the real manufacturing environment during the production of 9000 wafers.

For this experiment, we use FAB’s QM to quantify how our checkpoint technique performs when facing crashes in the operator that creates and maintains the incrementally built classification model. This operator is implemented as a UDOP, which generates model parameters based on information accumulated during runtime. If the UDOP crashes and no checkpointing is implemented, it loses all its collected information. Therefore, after a UDOP restore, it rebuilds its model parameters from scratch, losing valuable historic data previously used to fine-tune the classification model. When checkpointing is in place, all the UDOP state variables are maintained. After recovery, the UDOP produces model parameters from the same state it had before the crash. For this application, we do not buffer tuples while the operator is recovering. If any input tuple is sent to the UDOP while it is offline, the tuple is discarded. Note that while there is the potential for data loss, it does
not critically affect the accuracy of the classification model.

Figure 3.8 shows FAB’s dataflow graph, which has 79 operators. For the purpose of this experiment, we run one operator per PE. If we inject a crash fault in the statistical model operator (PE 74 in Figure 3.8), all the other 78 operators will continue to run. The PEs are distributed across 10 nodes, each with 4 Intel Xeon 3 GHz processors. The average runtime of the application is 30 minutes. However, we consider that one experimental run is complete only when all the sensor inputs available have been processed. To quantify the output deviation of the application, we ran FAB in the following scenarios: (i) FAB without UDOP checkpoint and no PE crash (S1); (ii) FAB with UDOP checkpoint at every 1 second and PE crash (S2); and (iii) FAB without UDOP checkpoint and PE crash (S3).

Figure 3.8: FAB dataflow graph.

We ran each scenario 40 times. For S1, all runs produced the same predicted wafer yield. Even though FAB contains non-deterministic operators (e.g., time-based operators), they did not affect the QM output during our runs. We used this output as the baseline to compute output deviation under the PE crash scenarios.

Since the time of the failure can impact the output deviation due to the amount of accumulated state in the classification model, each run has a different crash time. For S2, we randomly pick a time between the beginning of the application and the average application runtime (30 minutes). At that time, we inject a fault in the statistical model PE via a kill command. We
restore the PE after 2 to 5 seconds, which is the estimated failure detection time in System S. Once all input is processed, we finish the application and collect all its results (e.g., predicted wafer yield) and injection data (e.g., crash time). S3 employs the same fault injection parameters as S2, so we can compare outputs produced with equivalent failure times. For example, if in a run the PE crashed after 10 minutes and its recovery took place after 3 seconds, we try to repeat the same failed run for the new configuration. To minimize timing variations and more accurately approximate the crash time, we collect the number of processed tuples upon a crash in S2 and inject a fault once this number is reached in S3. Despite not being a fine-grained fault trigger, such as a low-level instruction address, the approach has shown to be effective in practice. The average of total processed tuples for the two scenarios is very close (5370.85 for S2 and 5370.875 for S3, where 5381 is the total number of tuples for S1).

S1 produces 107 wafer yield predictions and outputs the aforementioned QM for each one of them. We compare each QM produced by the run without PE crash and the run with PE crash. The difference between the two QMs is called prediction error. To evaluate the overall output deviation between failed runs and the golden run we computed the root mean squared error (RMSE) of the prediction errors. The RMSE shows how far from the correct answer the failed runs are on average. Note that we compare wafer yield prediction errors starting from the failure point onward. Every prediction produced before the failure is discarded from the RMSE computation.

The RMSE for S2 is 4.80, while that for S3 is 7.79. Our experiments show that the prediction error distribution for the runs without checkpointing have longer tails than the runs with checkpointing. This means that runs without checkpoint produced QM samples very distant from the correct value. This can be seen in Figure 3.9, which shows the cumulative distribution function (CDF) for the prediction errors in one of our runs. The prediction errors are in log scale. For this run, the checkpoint scenario generated results where the predicted value was at most 6 points away from the correct value. On the other hand, the checkpoint-free scenario produced predictions that reached up to 52 points away from the correct value.

The performance overhead for checkpointing FAB’s statistical model operator is negligible, since it did not change the tuple processing rate. It also did not affect the results produced by the non-deterministic operators down-
stream. We observed this by checking that all the measured QMs before operator crash were identical to the ones produced by the checkpoint-free run.

3.3.2 Checkpoint Performance Overhead

To quantify the performance overhead of our checkpointing technique for join operators, we devised two synthetic SPADE applications. Each application contains four operators, as shown by Figure 3.10. Two of them are source operators, which send data to a join operator. The join operator, as shown by Figure 3.10 correlates the data and sends the output to a sink operator. The sink writes the join output to a file. The objective is to stress the join operator and evaluate the checkpoint technique under high data loads.

The two applications differ by their join window configuration. They both correlate stream 1 (S1) with stream 2 (S2); however, the second application uses the per-group window modifier for S1. The attributes in S1 are two integers and one 70-byte string. S2 contains one integer and one 10-byte string. The join matches the streams when the integer of S1 is equal to the...
one in S2. The join output stream contains all attributes from S1 plus the string attribute from S2.

The operators are hosted by four PEs, which are placed in four different nodes interconnected by Gigabit Ethernet. Each node runs Linux on four Intel Xeon 3 GHz processors. Both source and sink operators read/write data to their local hard disk. The join operator saves its checkpoint files in a shared file system, in this case NFS. The NFS server is shared with approximately another 200 nodes.

Single-Window Join

For the first application, we evaluate the checkpoint performance varying the window size and the group size. The window of S1 is parameterized with the following sizes: 8192, 16,384, 32,768, 65,536 and 131,072. The window of S2 is fixed with size 0. Tuples from S2 are compared against S1 to generate an output tuple, but the operator does not maintain S2 tuples in its internal window buffers. Therefore, the checkpoint routine serializes only tuples from the first window. The statistical distribution for both data sources is uniform. We evaluated the checkpoint overhead for this scenario under two checkpoint frequencies: 1 and 10 seconds. To assess how the window group size impacts the performance, we divided the window of S1 in four different group sizes. Each window size was divided in the following number of tuples/group: 512, 1024, 2048 and 4096. We ran each configuration 30 times, where each execution lasted 200 seconds.

The checkpoint overhead is compared to the operator performance when running without checkpoint (I/O rate with checkpoint / I/O rate without checkpoint). Each window configuration has a different input/output (I/O) rate, which is shown in Table 3.1. The input rate is the sum of the inputs from S1 and S2. In all configurations, the input rate is evenly distributed (approximately 50% for each input stream). The rate is in tuples/second. Note that the input rate decreases as the size of the window increases. Because the operator has a greater number of tuple comparisons to make, it cannot process as many tuples as it could in the case of smaller windows. This creates back pressure on the output ports of the source operators, leading to lower input rates.

Figure 3.11 shows how much the checkpoint operation at every 1 second
Table 3.1: I/O rates for single-window join in tuples/second.

<table>
<thead>
<tr>
<th>Window Size</th>
<th>Input Rate</th>
<th>Output Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>9350.94</td>
<td>25,451.04</td>
</tr>
<tr>
<td>16,384</td>
<td>5054.39</td>
<td>27,180.69</td>
</tr>
<tr>
<td>32,768</td>
<td>2503.25</td>
<td>25,643.36</td>
</tr>
<tr>
<td>65,536</td>
<td>970.19</td>
<td>14,099.27</td>
</tr>
<tr>
<td>131,072</td>
<td>889.64</td>
<td>13,062.59</td>
</tr>
</tbody>
</table>

affects the I/O rate of the join operator. When the window size is 8192, the checkpoint overhead is greater than 5% in all configurations. As Table 3.1 shows, the input rate for this operator is 9350.94 tuples/second. This means that the number of new tuples is more than half of the window size, implying that, most of the time, we have to save the full window. Since there are few common states between checkpoint intervals, our technique does not have good performance. As the window size increases, we can see the benefit of our algorithm. For a window of size 32,768, the configuration with 512 tuples/group shows less than 1% performance overhead. A reason this scenario has better performance than configurations with a bigger group size is that a group of 512 tuples results in less data re-serialization. Note that as the number of groups increases, the overhead of managing more files increases as well. However, this overhead did not affect the performance here, since it had the least performance penalty.

Figure 3.11: I/O relative performance for single-window join with checkpoint interval of 1 second.

Figure 3.12 shows the I/O rates for a checkpoint interval of 10 seconds. The
I/O rates are higher compared to the 1 second scenario, since the checkpointing routine spends less time in serialization. When the checkpoint interval is larger, the performance is better for bigger group sizes. As the checkpoint interval increases, it is expected that a configuration with higher numbers of tuples/group performs better, since it needs to manage a smaller number of files.

![Figure 3.12: I/O relative performance for single-window join with checkpoint interval of 10 seconds.](image)

**Per-Group Join**

The second application uses the join operator with the *per-group* modifier for S1. The window has size 1024 tuples for S1 and 0 for S2. The *per-group* modifier generates one window of size 1024 based on a key attribute of the stream. In this case, the attribute is one of the integers of S1. For this experiment, the stream data has 50 different integers for S1 and 600 different integers for S2 uniformly distributed. This generates 50 different windows of size 1024. Therefore, at every checkpoint interval, we have to serialize new tuples in 50 different windows. Here, each window is divided in four different groups sizes. The numbers of tuples/group are 32, 64, 128 and 256. The checkpoint intervals are 1 and 10 seconds. We ran each configuration 30 times, where each run lasted 200 seconds.

Figure 3.13 shows the I/O relative performance for the per-group join. The averages for input and output for the configuration without checkpoint are 2026.52 tuples/second and 75,856.67 tuples/second, respectively. When the
checkpoint interval is 1 second, the group size of 64 is the configuration with lower performance impact. The number of input tuples/second decreases by 1.37%, while the output rate decreases by 1.83%. When the checkpoint interval is 10 seconds, the configuration with better I/O performance is associated with a group size of 32. However, our data shows that the checkpoint thread is able to start a serialization, on average, once every 10.5 seconds. Not meeting the checkpoint interval deadline results in a higher loss of tuples during recovery, since there will be more stale data in the files. When the group size is 256, the performance impact is higher, but it gets the operator state at every 10.09 seconds. In general, the performance impact is higher when the join operator has the per-group modifier. A per-group join has more checkpointing data structures to maintain and more files to save; hence, the serialization phase takes longer.

Figure 3.13: I/O relative performance for per-group join.

We conducted experiments to evaluate the per-group join checkpointing when the key attribute of the source data follows a Zipf distribution. Word frequency is an example of a distribution that follows Zipf’s law [77]. In our scenario, this means that some windows have a greater number of new tuples than others. Our data shows that the I/O performance impact is lower than the per-group join under uniform distribution (2%-3% overhead). However, the checkpoint thread is not able to meet the checkpoint deadlines. Instead of serializing the operator state at every 1 second, the checkpointing routine serializes the operator state at every 6 seconds on average. This happens because a lot of data must be rewritten to disk for windows with a low tuple insertion rate. For example, we have to re-save a whole chunk of data even
when there is only one new tuple in the group. Our results suggest that, when the stream data does not follow a uniform distribution, we should divide the window corresponding to different keys with different group sizes.

3.4 Related Work

Many techniques tailored for streaming applications add fault tolerance by changing the communication substrate. We argue that this slows down the maximum throughput of the system and should only be applied in selected parts of the application. Balazinska et al. [64] propose a protocol called delay, process and correct (DPC). When a failure occurs, stream operators may produce tentative tuples that need to be corrected later on. The developer has to know how to correct results that were produced with tentative tuples. Our aim is to abstract out the fault tolerance from the user by providing a language-level abstraction. Additionally, DPC may require operator checkpointing to recover from a failure. Our techniques can complement DPC.

Another set of fault tolerance techniques is based on operator replication. Passive standby, active standby [65], and process-pairs [20] were adapted to the streaming context. Hwang et al. [65] describe upstream backup, which enforces tuple backup in upstream operators. Tuples are replayed in case of failure. This technique also changes the communication substrate. Cai et al. [66] propose a hybrid replication-based technique. Replicas are brought up-to-date via checkpoints of the active replica.

LSS [21] (lightweight summary structure) is a checkpoint technique more similar to our approach. Zhu et al. assume that data loss is acceptable during the failure and recovery process of a streaming application. LSS provides an API that should be embedded in the operator code right after a tuple is processed. We automate this process by using the SPADE’S code generation framework to output specific checkpoint methods based on the operator type. Hwang et al. [18] propose delta checkpoints for both aggregate and join operators. Our work differs by proposing different checkpoint techniques depending on the operator failure semantic. SGuard [19] employs memory management middleware to track application-level memory pages. It uses copy-on-write to perform asynchronous checkpoints. Our technique lets the user choose which operators should apply a checkpointing scheme, decreasing the overall performance overhead. Our incremental checkpoint technique also
handles corrupted checkpoint files without requiring a replicated file system like SGuard does.

Some research aims to provide application fault tolerance at the language level [78, 79]. Szentiványi et al. [80] use aspect-oriented programming features for building fault tolerant applications. CATCH (Compiler-Assisted Techniques for Checkpointing) [81] is a compiler-based approach to transparently checkpoint applications. It does not use language extensions; however, it uses compile-time information to establish checkpoint interval and size. CATCH is a process-level technique and does not apply to our concept of operator-based checkpointing. Bronevetsky et al. [82] introduce a pre-compiler that instruments MPI programs for automatic checkpointing. The users annotate their application with potential checkpointing sites. This approach aims at parallel applications, whose behavior is different from that of streaming applications. Even though streaming applications are distributed, we propose techniques to save stream operators based on their semantic and failure behavior.

3.5 Summary

Large-scale stream processing is becoming a paradigm for developing long-running applications that will monitor, control, and extract knowledge from the critical infrastructure in a wide array of operational areas: monitoring and acting on sophisticated manufacturing tools and fabrication processes [76], business processes such as algorithmic trading [4], surveillance and fraud detection systems, personal healthcare, and public health systems. All demonstrate how critical it is to develop mechanisms that will ensure these applications and the components that make up the middleware supporting them have the means to stay up and operating continuously, even in the presence of software and hardware failures.

This chapter has shown how language primitives coupled with code generation can provide a flexible mechanism for specifying well-targeted state checkpointing to large-scale applications. We have also shown how incremental checkpointing can be carried out for a stream join operator, which is a fundamental building block in applications that carry out data correlation on live streams.
Although stringent fault tolerance techniques for stream computing [19, 22, 69] provide guarantees that no data is lost or that 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 [16, 21, 27, 71, 83] have been proposed. These techniques assume that data loss and data duplicates\(^1\) 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 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 chapter describes a methodology used for evaluating the effectiveness of PFT techniques in streaming applications. Our first goal is *to provide the developers a method to assess the impact of PFT on the output of their application*. We propose the use of fault injection [84] 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 methodology to experimentally evaluate PFT techniques in a streaming application. Our second goal is *to characterize how each stream operator 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

\(^{1}\)TCP provides reliable communication as long as the communicating processes do not crash.
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 involves many challenges that are not addressed by traditional fault injection methodologies [84, 85]. Streaming applications can have multiple independent outputs, tolerate approximate results, and be non-deterministic. In addition, these applications produce results continuously, 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.

We illustrate our methodology by assessing an application running on top of System S. Our experiments with a bursty data loss fault model in a financial engineering streaming application show the following results: (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. These results indicate 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.

This chapter is organized as follows. Section 4.1 discusses the effect of partial fault tolerance techniques on stream processing applications. Section 4.2 describes the evaluation methodology. Section 4.3 describes the fault injection framework. Section 4.4 discusses the set of proposed metrics and how to evaluate the injection outcome. Section 4.5 presents our results applied in the financial engineering domain. Section 4.6 contrasts related work with our methodology, and Section 4.7 summarizes the chapter.
4.1 Partial Fault Tolerance

Several researchers [16, 21, 27, 71, 83] have described PFT techniques that are applicable to stream processing applications. While all leverage the partial and often strategic employment of fault tolerance techniques to lower 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, assessment methodology and appropriate metrics are needed.

The technique described in Chapter 3 and in [27] 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. The state serialization methods are based on the stream operator type and failure semantic. When an operator fails, its upstream operators do not buffer outgoing tuples unless they are required to produce a semantically correct result after a fault. Checkpointing [27] 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 [71]. 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.

To evaluate how applications behave under PFT techniques, it is critical to understand the effect of faults on the input and 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.

4.2 Evaluation Methodology

Our 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-fast manner (i.e., clean crash). 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 that causes a crash (e.g., memory bit flip causes a process crash).

Characterizing the error behavior of streaming applications presents many challenges. 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*) [85]. 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-deterministic\(^2\), and are typically able to tolerate imprecise results.

To understand how a fault affects the application output, we compute an *output score function* (OSF). The OSF is calculated over a set of tuples of the output of the application and is applied for both the faulty and the golden output. Figure 4.1 shows two samples of streaming application outputs [4, 76]. Figure 4.1(a) shows a sample output of the financial engineering application described in Section 4.5.1. 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 4.1(b) shows a sample output of a chip manufacturing application [76]. 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 OSF is application-specific and can be complex depending on the application. In addition, the OSF should be sensitive to faults, i.e., change its value statistically significantly under faults. In our experiments, we evaluate the OSF sensitivity by conducting a two-sample Kolmogorov-Smirnov hypothesis test [86].

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

\(^2\)A source of non-determinism is the multiplexing of multiple streams, where the relative order of tuple arrival to an operator is arbitrary.
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 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 4.2 gives 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 4.2: Failure of a stream operator.

Our evaluation metrics characterize each operator in the following terms. (i) The outage duration impact is 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 is 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** is 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** is defined as the sum of the 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 distant from the correct result. More details on the computation of each metric can be found in Section 4.4.2. Note that our metrics are related to the effects that a fault has on the application output. These metrics can be used in addition to other usual performance evaluation metrics, such as application throughput and the end-to-end latency of a tuple.

Streaming applications can also have multiple independent outputs (called **sinks**). This situation arises when parts of the dataflow graph are re-used for a different computation (e.g., by using different statistical models [76]). Figure 4.3 shows an application with 10 stream operators. In this example, three different **sources** are processed to generate the results output by two different **sinks**. Each sink stores the results of a different computation over the incoming data.

![Figure 4.3: Sample stream processing application.](image)

To optimize the fault injections, we consider only the stream operators on the path of a specific sink, i.e., the **sink dependency graph**. The operators on the sink dependency graph are shown with filled boxes in Figure 4.3. A different OSF should be defined for each sink of the application.

The methodology to evaluate each **sink** 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 mimic variations of the detection and recovery times.

5. **Evaluate the experimental results.** Based on the OSF function, each operator is characterized using the proposed metrics. Using these metrics, the developer can quantify the relative sensitivity of the operators to faults, and use them as a basis to compare different fault tolerance techniques.

### 4.3 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; and (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. These faults are not protected by any of the fault tolerance techniques that guarantee no tuple loss [19, 22, 64, 69].

4.3.1 Emulating Faulty Behavior

Our fault injection framework is designed to work seamlessly with SPADE which 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 4.4 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 dataflow 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 4.4: Fault injection framework.](image)

Figure 4.4 depicts how the injection occurs at runtime. The figure shows the injection of the bursty tuple loss fault model into the input port of oper-
ator 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 4.5, the FIOP triggers a fault after it processes the stock symbol IBM at price USD 123.24. The duration of the fault is two 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. Note that the figure depicts the FIOP for an operator that receives a single stream and has one input port. For stream operators with two or more ports, there is a different version of the FIOP with the equivalent number of ports.

![Figure 4.5: Emulation of bursty tuple loss.](image)

Note that 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. This is equivalent to an operator crash and restart from a checkpoint with its most up-to-date state preserved before the fault.

4.3.2 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 an application, the framework pre-analyzes the dataflow graph. It selects as injection targets only those operators which are sensitive to the chosen fault model (e.g., operators which process a single input stream are not sensitive to tuple reordering), or the injected fault results in a behavior that is different from a fault injected into another operator in the graph.

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 that 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 dataflow graph immediately before a chosen target operator. The framework selects an operator as a target if its position in the dataflow 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 shown by Figure 4.6(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 4.6(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 (e.g., a stream attribute $x$ is less than a value) does not keep any state related to previously processed tuples. Figure 4.6(c) shows a dataflow 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 impacts on the application output, and both must be evaluated.

Figure 4.6: Placement of bursty tuple loss FIOPs.

Note that if the framework does not select an operator as a fault injection target, it assumes that the operator error behavior with respect to the application output quality is the same as the behavior of its upstream operator.

### 4.4 Evaluating the Fault Injection Outcome

As mentioned in Section 4.2, 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. The next sections describe how we handle the continuous output of streaming applications and how we use the OSF to compute each of our metrics.
4.4.1 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.

Figure 4.7 shows an example of a focused segment of the application output in which part of its tuples is affected by the injected fault. The first top bar represents the sequence of input tuples of the target operator. The fault is injected at a given stream offset and with some specified outage duration. The third bar represents the application output under a faulty run, where only parts of the tuples are affected by the fault (boxes in red). The QS is then computed over the tuples identified as affected by the fault on the faulty output and their pair (if any) on the output generated by the golden run (boxes in yellow). The first two metrics described in Section 4.4.2 consider such focused segments of the stream for their computation.

Continuous output can also mask the effects of faults. For example, an application can manifest a fault by missing \( x \) alerts (false negative) and misdetecting \( y \) alerts (false positive). When applying 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 4.4.2 use a local OSF computation.

Figure 4.8 depicts an example of the QS computation over local observations of the application output. Similarly to Figure 4.7, the first top bar represents the operator input stream into which the fault is injected. The third bar shows how the tuples are grouped for the local QS computation. Note that the local segments of the output contain both tuples that are affected by the fault and tuples that are correct. The QS is computed by comparing the OSF of the local group of tuples in the faulty output to the OSF of the matching group in the golden run output. The graph in the figure shows the QS behavior of the output as the stream progresses (i.e., over time). Once the fault is injected, it can cause a perturbation on the output. However, as more tuples are processed by the application, the operators recover from the fault and the output stabilizes. The last two metrics described in Section 4.4.2 use local OSF computations.

![Figure 4.8: Example of local observations of the QS on the application output.](image)

53
4.4.2 Evaluation Metrics

Table 4.1 summarizes our evaluation metrics. The first two metrics ($C^{oq}$ and $D^{oq}$) 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 an operator fails in the field, the application outcome is unknown. The last two metrics ($R^{lq}$ and $I^{lq}$) are related to system availability and allow the assessment of which operators are more critical for the application to preserve output quality under faults.

Table 4.1: Summary of evaluation metrics.

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

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, which is one of the properties of an operator failure (Figure 4.2). If the QS and the outage 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. The developer can 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 an 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 a 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

\[
\overline{OSF}_{i,j} = \frac{\sum_{k=1}^{p} FO_{i,j,k}}{p}
\]

(4.1)

The OSF for the golden run is calculated over the section of the output stream affected by the fault with maximum \(OD_j\) value\(^3\). 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

\[
QS_{i,j} = \frac{\overline{OSF}_{i,j}}{(\sum_{i=1}^{q} GO_{i,1})/q}
\]

(4.2)

After this step, a particular \(SO_i\) has \(n\) \(OD_j\) values associated with it and their corresponding \(QS_{i,j}\) results. With these two sets of data, we compute the Spearman rank correlation coefficient, which assesses whether two sets of values have a monotonic relationship [87]. This step results in associating a correlation coefficient \(CC_i\) with each \(SO_i\). Correlation coefficients have bounds \([-1..1]\).

The operator \(C^{eq}\) is then calculated as

\[
C^{eq} = \frac{\sum_{i=1}^{m} CC_i}{m}
\]

(4.3)

Data Dependency

The \(D^{eq}\) metric is the standard deviation \((\sigma^q)\) and analysis of variance test \((A^q)\) of the quality score for different stream 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, which is the

\(^3\)The index \(j\) is omitted in all formulas using a single fixed value of \(OD_j\).
other property of an operator failure (Figure 4.2). 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 \(\sigma^q\) and a rejected ANOVA test \([86]\) is not a good candidate for PFT, since the result of a fault in the field is highly unpredictable. An operator with a low \(\sigma^q\) 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 \(\sigma^q\), 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 \(QS_i\) for each \(SO_i\).

The \(\sigma^q\) is then calculated with the standard deviation formula, using the \(QS_i\) 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 \([86]\). 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 (4.4) shows the parameters for invoking the ANOVA test that returns an accept or reject value.

\[
A^q = ANOVA((QS_{1,1}, ..., QS_{1,k}); ..., (QS_{i,1}, ..., QS_{i,k})),
\]

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

Recovery Time

The \(R^{oq}\) 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 the impact of an operator failure on the application availability. As described in Section 4.4.1, 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 2% 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.

To compute $R_l^q$, we choose the same single outage duration $OD_j$ for all stream offsets $SO_i$. Each experimental 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 experimental trial is referred to as $\overline{LO}_{i,t}$, and is computed similarly to Equation (4.1). 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 $GLO_{i,t}$. $GLO_{i,t}$ refers to the average of $GLO_{i,l,t}$ over each trial, and is calculated similarly to Equation (4.1).

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

$$EQ_{i,u} = \frac{|LO_{i,t} - GLO_{i,t}|}{GLO_{i,t}}$$ (4.5)

For simplicity, our definition of $EQ_{i,u}$ in this dissertation considers the absolute value of the error. However, it can also consider the signal of the error (e.g., output is erroneous only when $LO_{i,t} - GLO_{i,t} \geq 0$, and $EQ_{i,u}$ is 0 otherwise). Such a condition depends on the semantic of the application, and should be defined by the developers when evaluating their program.

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

$$NE_{i,u} = \sum_{v=1}^{u} 1[EQ_{i,v} > threshold]$$ (4.6)

Then we compute the index $R_l^q$ where $P\%$ of the erroneous QS observa-
tions fall. That is

\[ R^l_i = \min u \text{ s.t. } NE_{i,u} \geq p \times NE_{i,s-S_{\text{begin}}} \]  \hspace{1cm} (4.7)

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

Figure 4.9 shows an example of the \( R^l_i \) 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. In this example, the \( R^l_i \) 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 4.9: Example of the \( R^l_i \) metric.](image)

Quality Impact

The \( I^l_i \) 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. Similarly to the \( R^l_i \), 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 \( R^l_i \) metric.

The computation of this metric is similar to the \( R^l_i \) computation. Instead of applying Equation (4.6), we calculate the SSE of a single \( SO_i \) (referred to as \( I^l_i \)) as
\[ I_{lq}^i = \sum_{v=0}^{R_i} (EQ_{i,v})^2 [EQ_{i,v} > \text{threshold}] \quad (4.8) \]

\[ I_{lq} \text{ is then } \max_i I_{lq}^i. \]

4.5 Experimental Evaluation

Our experimental target is a prototype application from the financial engineering domain called bargain discovery [4]. 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 [27], and (iii) an operator restores its state to the state immediately before the fault occurrence.

4.5.1 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 4.10 shows the flow graph of the bargain discovery application.

![Stream operator graph of the bargain discovery application.](image)

The application has one Source operator, which reads trades and quotes events from a file\(^4\). 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).

\(^4\)In a real-application deployment, data arrives as a continuous stream.
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 five 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 in 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. For every incoming quote tuple, it checks for 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 flow graph propagates this key until the Sink. With such a 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.

4.5.2 Experimental Parameters

The input stream used in our experiments consists of real market trades and quotes transactions during 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 six 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 4.3, the FIOP that emulates the bursty tuple loss fault model expects as a 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 because of the fault. In this conversion, we used 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 [42] 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. Similarly to the outage duration, we approximate the offset trigger based on the number of tuples processed by each operator.

Because of the application non-determinism, the golden run was executed 300 times. Each outage duration and stream offset combination was executed five 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 4.10 and were chosen based on the optimization criteria described in Section 4.3.2. All experiments ran on a single node with the Linux operating system, four Intel Xeon 3 GHz processors, and 8 GB of RAM.

**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 trading decisions.

We evaluated the OSF sensitivity-to-faults by using a two-sample Kolmogorov-Smirnov hypothesis test (KS-test) [86]. The $H_0$ of a KS-test is that both datasets come from the same distribution. In our case, one dataset comes from the OSF samples of the golden run, while the other dataset comes
from the OSF samples of the faulty run. A rejected $H_0$ means that the OSF is sensitive to the injected faults. For the KS-test, we considered the results of injected faults with maximum outage duration and all target operators. In addition, we considered the samples obtained from local OSF observations, i.e., one KS-test for each local OSF. Because at least one KS-test rejected $H_0$ with a level of significance ($\alpha$) of 0.05, we concluded that the chosen OSF is sensitive to faults.

4.5.3 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 4.4.2, 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 because of the injected fault and all trades that were correlated with a moving average that was miscalculated because of 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 4.2 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.

Figure 4.11 shows 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. The figure shows the QS data points with faults injected at stream offset 1.5 million.

For the Source (Figure 4.11(a)) there is an OSF overestimation when the outage duration is small (less than 5000 tuples), and an OSF underestima-
Table 4.2: Operator metrics for bargain discovery application.

<table>
<thead>
<tr>
<th>Operator</th>
<th>$C^{oq}$</th>
<th>$D^{oq}$</th>
<th>$R^l_q$</th>
<th>$I^l_q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>-0.38</td>
<td>(0.21,R)</td>
<td>340</td>
<td>21.23</td>
</tr>
<tr>
<td>TradeFilter</td>
<td>0.97</td>
<td>(0.76,R)</td>
<td>340</td>
<td>48.70</td>
</tr>
<tr>
<td>QuoteFilter</td>
<td>-1.00</td>
<td>(0.00,A)</td>
<td>6</td>
<td>6.00</td>
</tr>
<tr>
<td>VWAP</td>
<td>0.99</td>
<td>(0.14,R)</td>
<td>73</td>
<td>30.87</td>
</tr>
<tr>
<td>BargainIndex</td>
<td>-0.69</td>
<td>(0.08,A)</td>
<td>43</td>
<td>7.89</td>
</tr>
</tbody>
</table>

Figure 4.11: Quality score for fault injection trials with stream offset 1.5 million tuples and different outage durations for operators Source, QuoteFilter, and TradeFilter.

When the outage duration increases (greater than 15,000 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 large, the financial loss due to not correlating quotes for a long time is greater than the overestimation due to miscalculated averages.
Figure 4.11(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 4.11(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 4.2 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 $\sigma^q$ is TradeFilter, and the one with the lowest $\sigma^q$ 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 4.12 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 the stream offset into which the fault is injected. 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 the effect of a fault on the QS depends to a great degree on what data the outage affects.

**Recovery time.** The $R^{bg}$ 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 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 4.2
Figure 4.12: Quality score for different stream offsets under maximum outage duration for all target operators.

shows the $R_{lq}$ for all target operators for an outage duration of 16 seconds under peak rate. Figures 4.13 and 4.14 show the $R_{lq}$ value as vectors. The displayed value is the $R_{lq}$ for the injected stream offset, and not the maximum value among all injected stream offsets.

Figure 4.13: 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.
The operators with highest recovery time are Source and TradeFilter. These two operators have the same $R_l^q$ value because during the injection trials the same exact set of tuples was dropped with respect to the Aggregator. Additionally, they have the highest $R_l^q$ 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 Figures 4.13(a) and 4.14, the QS result stabilizes as more tuples are processed.

The VWAP and BargainIndex operators (Table 4.2) have smaller $R_l^q$ 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 down-
stream.

**Quality impact.** The \( I^q \) 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 \( R^q \) metric. Table 4.2 shows the \( I^q \) 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 4.13 shows the QS result (y axis) for every section of the output stream (x axis) for three 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 4.13(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 4.13(b) shows the QS observations for the QuoteFilter operator. The QuoteFilter has a low \( I^q \) because it only affects the output during the outage period. When faults are injected into the VWAP operator (Figure 4.13(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 4.14 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 \( I^q \) increases. When the outage lasts 2 seconds (Figure 4.14(a)), the maximum overestimate is 7% and the \( I^q \) is 0.25. For an outage of 8 seconds (Figure 4.14(b)), the maximum overestimate is 72% and the \( I^q \) is 12.85. The peak overestimate for an outage duration of 16 seconds (Figure 4.14(c)) is 144% and the \( I^q \) 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 error is higher when the fault is injected into
the TradeFilter. When TradeFilter fails, new quotes continue to correlate with an obsolete moving average value. This results in errors with greater magnitudes.

4.5.4 Discussion

Our results show the following regarding the bargain discovery application:

1. *The influence of the 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 ($A^o = accept$ 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 provides good protection against faults with short outage duration (e.g., TradeFilter and Aggregator in Figure 4.14(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 researchers [16, 71] suggest that

---

5 An operator not chosen as fault injection target is assumed to have the same behavior as its upstream operator (Section 4.3.2).
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 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 (Chapter 5). 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 4.15), we can see an improvement on the output quality of the application under faults. When either TradeFilter, Aggregator, or VWAP fails, the $R_{Iq}$ is 2 and the $I_{Rq}$ is 2. This is a significant improvement when compared to the previous values ($I_{Rq}$ of 48.70 and $R_{Iq}$ of 340). Under faults in the Source and the TradeQuote, the new $I_{Rq}$ is 1.05 and the $R_{Iq}$ is 39, in contrast to an $I_{Rq}$ of 21.23 and $R_{Iq}$ 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 great positive impact on resource utilization for our target application, given that QuoteFilter has an input selectivity of 80%, as described in Section 4.5.2.

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.

4.6 Related Work

Many fault tolerance techniques for streaming applications consider that no data can be dropped or duplicated [19, 22, 69], 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. [64] 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 [16, 21, 27, 71, 83] does not describe how to systematically evaluate the impact of faults on the application output quality. Bansal et al. [16] assume that the importance of each component in a streaming application can be described as a linear combination of the importance of the inputs it is consuming. Our experiments show that this is not the case for our target application. Zhu et al. [21] 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 [84, 85]. Streaming applications have unique characteristics, such as non-determinism, and continuous data processing despite the oc-
currence of faults. This brings up additional challenges to the evaluation methodology, which cannot be attacked with the techniques described in the literature.

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. [25] 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. Our approach is independent of operator types. Babcock et al. [24] propose an accuracy metric similar to our QS. We propose four different metrics which are based on the QS. Previous work [88] considers the operator type to establish a specific quality metric. Our methodology uses a quality metric that is independent of the operator type.

Fiscato et al. [89] 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.

4.7 Summary

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 chapter, we described a methodology and fault injection study to evaluate the impact of using PFT techniques on the output of a streaming application. The evaluation uses four different metrics to characterize how each stream operator in the application flow graph behaves under faults.

The results show that in the tested application the operator that processes approximately 80% of the source stream impacts the application output quality the least. This shows that partial fault tolerance can lead to considerable decrease in resource consumption and, as a result, better overall application performance. In addition, the chapter describes how the proposed methodol-
ogy can be used to learn how to selectively deploy fault tolerance techniques in the application processing graph.
Due to the low latency and high throughput nature of streaming applications, a variety of low-cost fault tolerance techniques have been proposed in past years [19, 21, 22, 27, 65]. To understand the benefits of applying a fault tolerance technique to a given target application, it is critical to evaluate its effect on the application output, especially considering the differing resource consumption of alternative techniques and the varying failure rates of distinct computer systems [90].

Previous research on the evaluation of fault tolerance techniques for streaming applications has focused mostly on their performance impact [21, 22]. In Chapter 4, we propose 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 the many ways that an application flow graph can fail (e.g., concurrent failures of stream operators).

In this chapter, we describe a modeling framework to evaluate the dependability provided by different fault tolerance techniques under varying fault models. The framework considers faults that lead to data loss and data corruption. To the best of our knowledge, this is the first work to consider the problem of data corruption in streaming applications. The error behavior modeled for both data loss and data corruption follows the behavior observed in real fault injection experiments. In addition, the proposed 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 has not been addressed by the research community either.

The framework is based on generic models specified with the stochastic activity network (SAN) formalism [36], which is ideal for expressing the prob-
abilistic behavior of faults and parallelism of streaming applications. The framework provides an abstraction for the key components of a streaming application: stream operators, stream connections, and tuples. These components are then assembled to represent a complete application flow graph as an SAN. We also define how the fault models under evaluation and their resulting error propagation are captured in our SAN models. The models for error propagation are independent of stream operator types and only differentiate them in terms of being stateful or stateless. The fault propagation mechanism forms the basis of our evaluation of fault tolerance techniques under different fault models.

Our modeling framework captures the best of the experimental evaluation world and the best of the model-based evaluation world. By injecting faults into the application we can learn its behavior as soon as a fault occurs. By retrofitting such behavior into a model, we can observe how faults can affect the application in terms of metrics that require a long observation time (e.g., availability) and better understand the risk of a fault when given realistic fault distributions.

We use our framework to evaluate the effectiveness of three different fault tolerance techniques, namely checkpointing [27], high-availability groups, and full replication [22]. The experiments with faults that cause data loss show that high-availability groups have a great advantage in maintaining the accuracy of the application output when compared to checkpointing. In addition, the results show that faults that lead to data corruption can break the no data duplication guarantee provided by the modeled full replication technique.

The current implementation of the framework assumes that different stream operators fail and recover independently. It also considers a generic representation for all operators, without any specialization for different operator types. We consider that extending the model both to force the parallel failure of more than one operator and to further specialize the model for different operator types is trivial. More details on developing more precise models for specific operator types can be found in Appendix A.

The main contributions of this chapter are (i) a framework with generic models to compose streaming applications and to evaluate the dependability and resource consumption trade-offs provided by different fault tolerance techniques and under different fault models; (ii) a characterization of how errors affecting a stream operator can propagate to other stream operators.
on the processing graph, which considers the probabilities that the stream connections of an application are used and 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.

This chapter is organized as follows. Section 5.1 provides a brief description of the SAN formalism. Section 5.2 describes the application model and its mapping to the SAN formalism. The fault and error propagation models are discussed in Sections 5.3 and 5.4. Sections 5.5 and 5.6 show how we added different fault tolerance techniques to the framework and their evaluation. Sections 5.7 and 5.8 discuss model validation experiments and some limitations of our framework. Section 5.9 describes related work, and Section 5.10 concludes the chapter.

5.1 Stochastic Activity Networks

Our modeling framework uses the SAN formalism [36] to model stream processing applications and the occurrence of failure events. SANs are a variation of stochastic Petri nets [91] and have been used to evaluate the performance and dependability of many complex systems [92, 93, 94].

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. An activity can be timed or instantaneous. A timed activity expresses how long a transition takes to complete and can be described as a random variable. An instantaneous activity expresses a transition that takes a non-significant amount of time to complete with respect to the modeled system. Activities can have a set of cases, which are used to model the possible outcomes of a transition. Timed activities are graphically represented by ovals, while instantaneous activities are represented by bars. Cases are depicted as small circles attached to the activity representation.

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 (◁).

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 (⊿).

For a formal definition of SAN, refer to Sanders and Meyer’s earlier work [36].

### 5.2 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 \( (k^\text{th} \text{ output port of operator } o \in O, \text{ where } k \in [1..p_{\langle o \rangle}^\triangleright]) \) to an operator input port \( (k'^\text{th} \text{ input port of operator } o' \in O, \text{ where } k' \in [1..p_{\langle o' \rangle}^\triangleleft]) \). 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 \( \tau_{\langle c \rangle} \).

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,

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

In other words, \( C_{\langle o \rangle,i}^\triangleleft \subset C \) defines the set of stream connections attached to the \( i^\text{th} \) 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 $f_{(o)i}$ defined as

$$f_{(o)i} : \cup_{c \in C_{(o)i}^\prec} \tau_c \rightarrow \mathbb{R}^+.$$ 

For each port, we also define an average processing cost per tuple $\overline{d}_{(o)i}$ as

$$\overline{d}_{(o)i} = \frac{\sum_{c \in C_{(o)i}^\prec} \sum_{m=1}^{\mid\tau_c\mid} f_{(o)i}(\tau_c,m)}{\sum_{c \in C_{(o)i}^\prec} \mid\tau_c\mid}.$$ 

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

An operator $o$ with no output ports ($p_{(o)}^\succ = 0$) 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 $j \in [1..p_{(o)}^\succ]$ as $C_{(o)i}^\succ$. Formally,

$$C_{(o)i}^\succ = \{(\langle o',k' \rangle, \langle o'',k'' \rangle) : o' = o \land k' = j\}.$$ 

In other words, $C_{(o)i}^\succ \subset C$ 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 $O_{(o)i}^\prec$. Formally,

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

The set of operators connected to all input ports of an operator is denoted as $O_{(o)}^\prec = \bigcup_{i=1}^{p_{(o)}^\prec} O_{(o)i}^\prec$. We also define the set of all operators in the upstream of an operator $o$ as

$$U_{(o)}^\prec = \begin{cases} 
\emptyset & \text{if } p_{(o)}^\prec = 0 \\
O_{(o)}^\prec \cup \bigcup_{o' \in O_{(o)}^\prec} U_{(o')}^\prec & \text{otherwise.}
\end{cases}$$

Finally, we define the set of operators on the upstream of a specific input
port \( i \) of operator \( o \) as

\[
U_{(o),i}^\triangleleft = \begin{cases} 
\emptyset & \text{if } p_{(o)}^\triangleleft = 0 \\
O_{(o),i}^\triangleleft \cup \bigcup_{o' \in O_{(o),i}^\triangleleft} U_{(o'),i}^\triangleleft & \text{otherwise}
\end{cases}
\]

### 5.2.1 Stream Operators

To represent an operator \( o \) using the SAN formalism, we first consider its input stream connections, \( C_{(o)}^\triangleleft = \bigcup_{i=1}^{p_{(o)}} C_{(o),i}^\triangleleft \). For each input stream connection \( c \in C_{(o)}^\triangleleft \), we generate a corresponding place in the SAN. These places are labeled as input stream connection. Figure 5.1 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 5.1. 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 \( p_{(o)}^\triangleleft \) 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 5.1. The processing stage considers selectivity\(^1\) 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 go back to the waiting stage. The processing stage also considers average processing cost \( d_{(o),i} \) for each processing tuple place. This cost is used as the activity parameter and can be obtained by profiling the modeled application [42]. If there is an output to be generated, the operator moves to the sending stage. For the sending stage, \( p_{(o)}^\triangledown \) places are created (one for each output port). These places are labeled as sending output, as shown in Figure 5.1. The operator

---

\(^1\) Selectivity is generally used for operators with predicates, such as filters and join. In this work, 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 produced by any operator type.
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 5.1. 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. Only a single output gate is used per output port. Each output stream connection $c \in C_{(o)}$ is also mapped to a *place*. These places are labeled as *output stream connections*. The *output stream connection* places associated with a given output port (i.e., $C_{(o),j}$, where $j \in [1..p_{(o)}]$) are connected to the output port’s *output buffer* place. Figure 5.1 shows an example for a single output connection.

**Figure 5.1:** SAN for a stream operator with two input connections on a single input port and one output stream connection on a single output port. Model captures three tuple processing stages (waiting, processing, and sending), operator selectivity, and tuple transmission to output stream connection.

5.2.2 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 [95]. This formalism allows places in different models to be shared, effectively allowing communication between the composed models. Stream connections are modeled as the state sharing between the output stream connection place of one operator and the input stream connection place of another operator. In the composed model, we create one shared pair input and 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 5.2. This example shows the equivalent composed model for the application segment given in Figure 5.3. This application segment, written in SPADE has three operators, mapping to a composed model with three base stream operator models and two stream interconnections.

![Figure 5.2: Example of composed model equivalent to the streaming application depicted in Figure 5.3. The sharing of output stream places of \( op_1 \) and \( op_2 \), and input stream places of \( op_3 \) models the interconnection between operators.](image)

5.2.3 Tuples

We represent each tuple \( \tau_{(c,m)} \) 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 or the stream operator itself. Tuple processing within an operator is modeled by removing a token from the input stream connection place and optionally adding a token (after the
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"]{}

Figure 5.3: 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.

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 5.2.1). Discarded tuples are seen as tokens that are ef-
effectively removed from the model execution. Note that the tokens placed in
stream connection and output buffer places are the only ones that represent
application data. All other tokens in the model represent the state in which
each operators is (e.g., waiting, processing).

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.
5.3 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., device driver error leads to kernel crash), a Heisenbug (e.g., race condition), or a hardware transient error (e.g., 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 an operator failure, it did not result in any erroneous output value being produced and the operator state stored in the checkpoint file [19, 27] (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 [19, 22, 65]. On the other hand, partial fault tolerance techniques [21, 27] 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 [96]. Dixit et al. [97] 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 [19, 21, 22, 27, 65] assume a fail-stop fault model, which does not always hold true [98]. In stream operators, hardware errors can affect the internal state of the operator and the attribute values of a tuple.

Similarly to stream operators, we consider both clean crashes and SDC for the stream sources. Crashes affecting stream sources can lead to data loss even when the fault tolerance technique applied to the application guarantees
no data loss. Although the streaming middleware can control the execution of stream operators and maintain enough redundant execution context and state for recovering the application, it cannot control the availability of external sources. With our model, the developer can assess the effects of such faults on the application.

5.4 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 5.2) 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, or (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 5.2.1 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 semantic is ignored. Figure 5.4 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.

![Figure 5.4: 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 streams connections, allowing the propagation of tainted tuples.](image)

### 5.4.1 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 described in Chapter 4. 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 distribution\(^2\) with a constant rate of \(\lambda_c\) failures per second. After the failure activity fires, all data (i.e., tokens) in the input and output

\(^2\)Analysis of field failure data has shown that exponential distribution is a good enough approximation of real failure rates [99].
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 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 \( p_{(o)}^{\text{in}} \) 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 \( p_{(o)}^{\text{in}} \) input ports maintains any data based on the previously processed tuples and uses such data 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 5.4, 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 [65] or partial loss [27] of its internal state. Once the operator restarts, its state is different from the state it would have in a failure-free run. Figure 5.5 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). Section 5.5 provides more details on how we established the stabilization time for the modeled techniques.

Figure 5.5: Timeline for a stateful stream operator under a data loss fault model. Once the operator crashes and restores, the operator runs under an unstable mode and produces tainted tuples. After processing tuples and stabilizing its internal state, the 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., $o \in U_{(o,i)}^<$) associated to a $F_{(o,i)}$ that maintains state crashes. As shown in Figure 5.6, a fault in an operator that is in the upstream set (i.e., filter with predicate $\text{int} < 9$) of a port with a stateful function (i.e., aggregation) can affect the operator’s 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 5.6(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 5.6(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 crash.

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
Figure 5.6: 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.

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.

5.4.2 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 [100], 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 $\lambda_s$ faults per second. The $\lambda_s$ value considers only faults that lead to an 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 5.7 shows an example of an 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.

![Diagram](image)

Figure 5.7: 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.

5.5 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 [27], partial replication via high-availability groups, and a full replication-based approach [22]. We provide a summary of each technique and how it is
mapped to the SAN formalism.

5.5.1 Checkpointing of Operator State

Checkpointing of operator state (Chapter 3 and [27]) is a lightweight fault tolerance technique developed on top of SPADE. 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. Similarly to streams, the heartbeat is modeled by adding shared variables via the replication/join formalism. The shared variable lets an operator $o$ monitor the alive state (Section 5.4.1) 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 [29, 30]. This behavior should be changed if the intercommunication protocol uses buffers to hold tuples and avoid tuple loss [65]. In our model, this can be done by changing the function associated to an output gate.

For all stateless operators of an application, our model assumes that there is no checkpoint from which to recover. 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 because of an upstream failure, as described in Section 5.4.1.

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

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

The number of tuples to be processed until stabilization, denoted as $Z_{o,i}$, depends on two components in our model. First is the state size, denoted as $S_{o,i}$, and second is the total number of tuples that were lost since the last checkpoint, denoted as $L_{o,i}$. The value for $L_{o,i}$ 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 $Z_{o,i}$. We use the weights $\alpha$ and $\beta$ 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 $Z_{o,i}$ can take using a constant factor on the state size, called state cap factor, denoted as $\gamma$. 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 (Chapter 4). If this does not hold for a given target application, the developer can change this parameter in our model accordingly.

The occurrence of hardware transient errors that cause silent data corruption can also be added to the checkpointing model. Gu et al. [101] have shown that a significant number of transient errors can take more than 1 million CPU cycles to be detected. Long latency failures can lead to checkpoint corruption. This means that even in the case of an operator crash and restore from a checkpoint, the checkpoint can contain a corrupted state. In our
modeling framework, this could be added by marking a checkpointed state as tainted if the checkpointing activity is triggered and the internal operator state is tainted. We do not consider this in our model because we already consider that checkpointed stateful operators generate tainted tuples upon restore because of data loss. However, this can be considered when modeling fault techniques that use checkpointing but still guarantee no data loss [19].

5.5.2 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 subgraphs (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 all these base models 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 implemen-
tation in SPADE, only one replica is active at a time, while all others are in backup mode.

Similarly to the checkpoint model, we add a heartbeat monitor for each stream connection \( c \in C \) of the application. The operator model discards output tuples if the heartbeat indicates that the downstream operator has failed. In addition, we add extra heartbeats (via shared variables) for groups of operators that were selected to be in an HA group. For all the group edges of an HA group (one group edge for each replica), we append a model similar to the one described in Figure 5.8. Each group edge monitors all operators participating in that group, both from the active and backup replicas. The failover and deactivate activities are triggered based on the liveness condition of the operators participating in the HA group. The failover activity is parameterized according to the average failover time between replicas. As in our implementation in SPADE, only one group edge starts in the active mode, while all other copies of the same group are set to be in the backup mode.

![Figure 5.8: SAN for an HA group of two operators and two replicas. Places associated to operators (\( op_{1,A} \), \( op_{2,A} \), \( op_{1,B} \), and \( op_{2,B} \)) indicate whether each operator is alive. Active replica deactivates upon detection of a failed operator. Backup replica triggers a failover upon the crash of an operator of the active replica. Tokens displayed in the figure indicate that the replica is active and all monitored operators are alive.](image)

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 5.4.1, 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.

5.5.3 Full Replication

Hwang et al. [22] 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 marks\(^3\) 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, as shown in Figure 5.9 by the dashed lines. During the model simulation, the operator writes tokens to both the original and replicated streams.

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 if the number of tuples received from this stream is less 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

\(^3\)A punctuation is special mark that can be included in a stream.
Figure 5.9: Operator graph for the full replication technique. Dashed arrows represent the additional stream interconnections required by this mechanism. Operator replicas send data to both operator copies downstream. Operators deduplicate tuples when consuming data from redundant streams.

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 5.1) 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 pseudorandom 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 a 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 5.4.2. 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 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.

5.6 Evaluation

To illustrate the applicability of the proposed framework, this section reports on evaluating the dependability 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 and the
following techniques are applied: checkpointing of stateful operators (Table 5.1 - configuration 1) and HA groups with checkpointing (Table 5.1 - 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 5.2). All our models and experiments were implemented with the Möbius tool and its simulator [102].

Table 5.1: First experiment compares configuration 1 to configuration 2 under the same fault model. Configuration 1 deploys the bargain discovery application with two stateful operators configured to checkpoint. Configuration 2 deploys the same application, but applies the HA groups technique to two different operator groups, and checkpointing of one operator.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Configuration 1</th>
<th>Configuration 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application graph</td>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>checkpoint [27]</td>
<td>HA groups and checkpoint [27]</td>
</tr>
<tr>
<td>Fault model</td>
<td>crash leading to data loss</td>
<td>crash leading to data loss</td>
</tr>
</tbody>
</table>

Table 5.2: Second experiment evaluates configuration 3 under SDC errors. Configuration 3 applies the full replication mechanism to the bargain discovery application.

<table>
<thead>
<tr>
<th>Configuration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>Fault tolerance</td>
</tr>
<tr>
<td>Fault model</td>
</tr>
</tbody>
</table>

5.6.1 Target Application

Our target application is bargain discovery [4], 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 5.1 - configuration 1. 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).

### 5.6.2 Experiment Parameters

In our experiments, we used a constant input stream data rate with the following values: 250, 350, 450, and 550 tuples/second. Our values were based on real market data from December 2005, which has an average rate of 500 tuples/second. The average tuple processing time and selectivity for each operator in the graph are shown in Table 5.3.

Table 5.3: Parameters used in the bargain discovery model. Processing time (in seconds) and selectivity obtained via Spade profiler. State size (in tuples) obtained from the application code and trace data.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Processing time (seconds)</th>
<th>Selectivity</th>
<th>State size (tuples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>1.26E–06</td>
<td>1.0</td>
<td>0</td>
</tr>
<tr>
<td>Filter 1</td>
<td>1.26E–06</td>
<td>1.0</td>
<td>0</td>
</tr>
<tr>
<td>Filter 2</td>
<td>5.29E–07</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>Aggregate</td>
<td>1.32E–05</td>
<td>1.0</td>
<td>15,495</td>
</tr>
<tr>
<td>Filter 3</td>
<td>5.47E–07</td>
<td>1.0</td>
<td>0</td>
</tr>
<tr>
<td>Filter 4</td>
<td>2.25E–07</td>
<td>0.8</td>
<td>0</td>
</tr>
<tr>
<td>Join (port 1)</td>
<td>1.27E–06</td>
<td>0.0</td>
<td>3099</td>
</tr>
<tr>
<td>Join (port 2)</td>
<td>1.27E–06</td>
<td>0.182</td>
<td>0</td>
</tr>
<tr>
<td>Sink</td>
<td>7.43E–07</td>
<td>1.0</td>
<td>0</td>
</tr>
</tbody>
</table>

As described in Section 5.4, 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, where we compared the number of tuples produced during a fault-free application run and a run under faults.
5.6.3 Evaluation Metrics

Our evaluation metrics are the following:

1. **Availability.** The application is 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 [103].

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 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 total application output.

5.6.4 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 (configuration 1) and when adding two HA groups (configuration 2). 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: 5.56E–04 (30 minutes), 3.33E–04 (50 minutes), 2.38E–04 (70 minutes), and 1.85E–04 (90 minutes). Stream operators fail at the same rate and independently. Although these rates are high considering supercomputer failure rates [90], 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 (Section 5.5) 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. The state penalty factor after a crash and the staleness factor were obtained by using a sample input stream of the
Aggregate operator and analyzing the tuples falling inside a window. The window is obtained by selecting a random point of the input stream and by assuming a given input rate and different recovery times. For each random point of the input stream, we evaluate how many count-based windows (Section 5.6.1) would be affected if data were to be dropped at that point. By doing this operation 1000 times for each recovery time and input rate combination, we obtain an average of how many windows are affected during a fault. This indicates approximately how many tuples should be reprocessed to fully recover the affected windows. We did this operation for the four tested source input rate values (5.6.2) and with recovery times ranging from 1 second to 17 seconds. This values were used to find the parameters that best fit the function introduced in Section 5.5 to the real data. Figure 5.10 shows the values obtained from the real data and the values obtained by fitting function $Z_{(o),i}$ under different input rates.

![Figure 5.10: Total number of tuples required to be processed after a crash of the Aggregate operator: (a) input rates of 250 and 350 tuples/second; and (b) input rates of 450 and 550 tuples/second. Graphs show the average obtained from real data (Average) and the number obtained by the fitted $Z_{(o),i}$ function (Model).](image)

A similar procedure was followed for the Join operator and its stateful port (port 1). The value obtained for the state penalty factor after a crash is 0.03, and the value obtained for the staleness factor is 0.41. Similarly to the Aggregate, a state penalty factor of 0.005 was considered when receiving tainted tuples in its inputs. Figure 5.11 shows the values obtained from experiments with the real data and the fitted function $Z_{(o),i}$ under different input rates. Note that when compared to the Aggregate operator, the parameters for the
Join operator are lower. This is due to the smaller internal state size of the Join operator.

Figure 5.11: Total number of tuples required to be processed after a crash of the Join operator: (a) input rates of 250 and 350 tuples/second; and (b) input rates of 450 and 550 tuples/second. Graphs show the average obtained from real data (Average) and the number obtained by the fitted $Z_{(o),i}$ function (Model).

We assumed that all checkpointed operators had their latest checkpoint 1 second before the crash occurred. We also parameterized the state cap factor as 1, which means a full window refresh. We evaluated both techniques considering a crash recovery time of 3.5 seconds and 15 seconds. 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 second, which is the average failover time of our implementation in SPADE.

Figure 5.12 shows how the evaluation metrics behave when the source input rate increases. Figure 5.12(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 5.12(b) shows that the number of tainted tuples also increases. This behavior occurs 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 5.12(a) and 5.12(b) is that the deploy-
Figure 5.12: Results for configuration 1 and configuration 2 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 of Figure 5.12(a) applies for Figures 5.12(b) and 5.12(c). Figure 5.12(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 5.12(b)), but its fraction decreases when compared to the total number of tuples produced by the application.

ment with HA groups is better than checkpointing even for a recovery time of 15 seconds. Although the availability provided by this technique is slightly lower when compared to configuration 1 with 3.5 seconds 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. Note that when the input rate is higher (550 tuples/second), the availability provided by the HA groups approach is higher than the checkpoint approach even if the operator recovery time is 15 seconds. We believe this occurs because the Aggregate is
the operator with highest the stabilization time. Even though the Aggregate
takes a longer to stabilize after a crash when under the HA group approach
(because it recovers from scratch), it stabilizes faster due to the higher input
rate. As a result, if there is a failover to an unstable replica, the unstable
time is smaller than the recovery time of 3.5 seconds and the stabilization
time after recovering from a checkpoint. In Figure 5.12(c), we can see that
the number of tainted tuples in relation to the total number of tuples pro-
duced 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 5.13 shows the impact of different fault rates on the application
availability. The tested scenario includes configurations 1 and 2. As ex-
pected, we can see that as the fault rate increases, the availability of the
application decreases. When comparing configurations 1 and 2, we can see
that the difference in availability is low (at most 0.009) if the extra com-
puting resources required to deploy the HA groups technique are considered.
On the other hand, this technique can pay off if we consider the number of
erroneous tuples produced because of the faults. In the scenario with fault
rate 5.56E–04, configuration 2 generates approximately half the number of
tainted tuples generated by configuration 1. Figure 5.13 also shows the avail-
ability of the application when our availability metric does not consider fault
propagation. We can see that the application availability is overestimated,
providing an unrealistic number to the developer.

Figure 5.13: Results for configuration 1 and configuration 2 under crash
faults occurring at different fault rates. Displayed availability is for a source
input rate of 450 tuples/second and crash recovery time of 3.5 seconds.
5.6.5 Impact of Silent Data Corruption

In this experiment, we tested the impact of SDCs when bargain discovery uses a full replication technique (configuration 3 in Table 5.2). For this scenario, the SDC rates follow an exponential distribution with \( \lambda \) values of 1.52E–04 (110 minutes) and 1.39E–04 (120 minutes). When the fault triggers in stateful operators, 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 seconds. Once a punctuation arrives, the model clears the counters for both clean and tainted tuples.

Figure 5.14(a) shows the availability under different input rates and different fault rates (1.52E–04 and 1.39E–04). Similarly 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 5.14(b) shows the percentage of tainted tuples in the application output. We can see that the number of tainted tuples is very low if 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 \((\text{tainted tuples} + \text{non-tainted tuples})\) is greater than the total number of tuples (and its confidence interval) produced during a fault-free run. This indicates that \textit{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 should be guaranteed. We believe this is a problem that should be studied given the high processing rates and low latency requirements of stream processing systems.
Figure 5.14: (a) Availability for a full replication deployment under SDC (configuration 3). The availability increases as input rate increases. (b) Percentage of tainted tuples under the same fault tolerance technique. The relative impact of error decreases as tuple rate increases.

5.7 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 top of System S and running the application model in our framework. Figure 5.15 shows the result of our experiment, where we plot the number of output tuples produced by the application sink over time. The output rate in the real execution is more unstable when compared to the one produced by the model. This occurs because the operator selectivities vary within a range over time according to the data distribution, while our model captures the average selectivity. Note, however, that once the real application fills up its internal windows, the average output rates seen by the sink are similar to the ones produced by a single execution of our model (60.03 and 65.78 tuples/second, respectively). In future work, we plan to increment our base model to better capture the possible dynamics of the stream data.

To validate our fault and error propagation model, we compared the results of the bargain discovery model with the results 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 Chapter 4. Table 5.4 shows our measurements when faults are injected into different operators. The crash recovery time and source input rate considered for both real and simulated execu-
Figure 5.15: Comparison of bargain discovery executing on top of System S and its simulation in our framework. After the application stream operators fill up its internal windows, the average output rate observed in the real sink is similar to the one produced by our model.

The table shows the measurements of the availability and the recovery time ($R_{tq}$) metrics. In short, the $R_{tq}$ metric assesses how long, after the injection of a fault, the application takes to produce results that are approximate to a fault-free run. Although the two metrics do not measure the same thing, we can say that $R_{tq}$ is an indirect metric of the application availability (i.e., the greater the $R_{tq}$, the lower the availability). We can see that the relative order of the operators’ availability is the same, excepting the Aggregate operator. This occurs because in the model we consider that the checkpoint of the Aggregate operator is taken 1 second before the crash, and in the real injections we consider that its checkpoint is taken immediately before the crash. The table also shows the measurements for the number of tainted tuples and the quality impact ($I_{tq}$) metrics. Briefly, the $I_{tq}$ metric computes the sum of squares of the differences between the tuple attribute values of a faulty run and a fault-free run. We can say that the greater the number of tainted tuples, the greater the value of $I_{tq}$. However, because the number of tainted tuples does not capture the error magnitude that $I_{tq}$ captures, there might be some discrepancy. This is the case for the Filter 3 operator, which has a higher $I_{tq}$ 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.
Table 5.4: Comparison between results obtained from real fault injections into bargain discovery (Section 4.5.3) 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Availability</th>
<th>$R^{lq}$</th>
<th>Tainted tuples</th>
<th>$I^{lq}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0.84533</td>
<td>340</td>
<td>15,506.32</td>
<td>21.23</td>
</tr>
<tr>
<td>Filter 1</td>
<td>0.84536</td>
<td>340</td>
<td>15,500.01</td>
<td>21.23</td>
</tr>
<tr>
<td>Filter 2</td>
<td>0.84533</td>
<td>340</td>
<td>21,752.34</td>
<td>48.7</td>
</tr>
<tr>
<td>Aggregate</td>
<td>0.83987</td>
<td>340</td>
<td>22,520.63</td>
<td>48.7</td>
</tr>
<tr>
<td>Filter 3</td>
<td>0.93676</td>
<td>73</td>
<td>8898.29</td>
<td>30.87</td>
</tr>
<tr>
<td>Filter 4</td>
<td>0.95555</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Join</td>
<td>0.93599</td>
<td>43</td>
<td>2753.54</td>
<td>7.89</td>
</tr>
</tbody>
</table>

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.

5.8 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 param-
eters 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 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 overtainting of tuples for some operator types. One such example is a 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 simulating the model, all these tokens have to be processed and go through the processing graph, increasing the execution time of the simulation. We believe that this problem can be tackled by reducing the simulation time and the input rates, making them proportional to the original stream source rate. By using such techniques, the developer can still assess fault tolerance mechanisms and derive insights with our framework. Another possible optimization is to only produce tuples once a failed is triggered in the model or any of the operators in the model is in unstable mode. With such optimization, the time consumed by the simulator when the application is behaving normally is much shorter.

5.9 Related Work

The evaluation of the impact of fault tolerance techniques in streaming applications has focused mostly on performance [65, 22, 19]. Other works have focused on evaluating the impact of data loss in the application results via ex-
perimentation [27, 21]. The work described in Chapter 4 focuses on proposing a fault injection methodology to learn how to deploy partial fault tolerance techniques. 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. [104] develop a modeling framework for fork/join tasks to target the problem of optimal resource allocation. This work focuses on the occurrence of faults and their effects on the application output.

Our work can also be compared to other uses of probabilistic models to evaluate fault tolerant systems. Singh et al. [93] model publish/subscribe systems to evaluate their survivability. Cinque et al. [92] model the dependability of wireless sensor networks. Wang et al. [94] 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. Moreover, one of the fundamental differences of our framework is that it combines the modeling of application data and application state. By modeling the effects of faults into the application data and its propagation throughout the application, we can evaluate a target application in terms of the impact of the fault on its output.

Previous research has analyzed the effects of error propagation in software [105, 106]. These works consider the effect of data corruption errors. We also consider how data loss can affect other components of the processing graph.

5.10 Summary

This chapter described a modeling framework tailored to evaluate 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 can be used to evaluate different fault tolerance techniques and test new techniques even before implementing them in a streaming middleware.

We show the generality and extensibility of our framework by modeling
three different fault tolerance mechanisms. The framework showed that high-availability groups has a great advantage over checkpointing, mainly when the fault detection and recovery time is long. In addition, the results show that SDC can break the no data duplication guarantee provided by a full graph replication-based technique [22]. This result motivates the development of low-cost error detectors tailored for streaming applications.
Chapter 6

CONCLUSIONS

This dissertation describes methods for providing partial fault tolerance for stream processing applications. Contrasting with current designs, these methods do not enforce a \textit{one-size-fits-all} approach. We propose a checking pointing and partial graph replication scheme that can be selectively applied to the stream processing graph. As a result, the application developer can configure a solution that is ideal for a given application, achieving better performance and lower resource consumption. Our solutions were implemented in IBM Research's middleware for stream computing System S and partly integrated into IBM's InfoSphere Streams Version 1.2 [33, 35]. The fault tolerance feature has been used by real applications, such as applications that process live call data records.

In addition, this dissertation focuses on evaluation methods for understanding the impact of faults in a given streaming application. Our first approach is experimental and is based on the emulation of the effect of faults in each stream operator of the processing graph. By analyzing the impact of these faults on the application output, we characterize each operator with a set of evaluation metrics. These metrics can be then used to estimate the sensitivity of each stream operator to faults. The most sensitive operators should be given priority when partially protecting the stream processing graph. Our experiments with a financial engineering application show that the influence of an operator output stream on the state of the downstream flow graph determines the criticality of the operator.

Our second fault tolerance evaluation approach is based on modeling. This approach allows the comparison of the dependability provided by different fault tolerance techniques and under varying fault models. The framework offers abstractions tailored for the stream processing domain, namely stream operators, stream connections, and tuples. These abstractions are assembled to compose a complete streaming application. The proposed framework con-
siders the effects of error propagation, which allows a more precise estimate of the availability provided by a given fault tolerance technique and the mechanism by which faults can affect the application output. Our results show that partial graph replication has a great advantage in maintaining the accuracy of the application output when compared with checkpointing, mainly when we consider that the system can take a long time to recover failed stream operators. In addition, our framework revealed that silent data corruption can break the no duplication guarantee of a full replication technique [22].

6.1 Lessons Learned

Some of the lessons learned throughout the development of this dissertation include the following.

1. **Fault tolerance techniques should cope with application non-determinism to give flexibility to the application developer and maximize application throughput.** The fault tolerance mechanisms proposed in this dissertation account for non-determinism, and do not enforce replicas of stream operators to process incoming data in the same order or generate the same output every time. As a result, our techniques allow the application developer to implement a stream operator with a tuple processing function with *any* logic. **SPADE** is an example of stream processing language that allows developers to define their own operator processing logic. Developers can devise multi-threaded operators and use any operating system call and legacy libraries. Enforcing determinism limits which kinds of functions the developer can use (e.g., functions that return wall clock time). In addition, enforcing determinism among replicas can have a very high cost [107] and may lead to the violation of the application throughput requirement.

2. **Partial fault tolerance may be essential when considering streaming applications with real-time constraints.** In real-time systems, the application is considered to have failed if it produces an output after a given deadline. Fault tolerance methods with high performance overhead may cause messages being processed by the application to be delayed, even without any stream operator crash. The evaluation techniques
described in this dissertation can help a developer to find parts of the application where fault tolerance is not needed. This results in reduced performance overhead and, as a result, a lower number of messages with missed deadlines.

3. **Evaluation techniques need to consider large-scale applications.** The experimental evaluation technique described in Chapter 4 characterizes the impact of stream operators on the application output by injecting faults into each operator individually. However, the total number of stream operators in an application can be large (i.e., hundreds), which can affect the scalability of our technique. In Section 4.3.2, we propose placement rules to reduce the number of required fault injection experiments when considering a tuple loss fault model. Similar placement rules can be developed for other fault models. When considering the model-based evaluation approach, we use SAN simulation to compare different fault tolerance techniques. In this case, the greater number of stream operators in an application, the longer the simulation time is. However, the simulation of our model can be trivially parallelized. This is because different stream operator models can be placed in different nodes or threads, and only input and output stream connection places and operator liveness state values need to be shared between distinct nodes.

6.2 Future Work

This research opens many opportunities for future work, both in terms of developing new partial fault tolerance techniques and exploring new approaches to evaluate its efficacy. A few of opportunities are itemized below.

- Our current fault tolerance techniques can cause a *bursty tuple loss* in the stream being processed by a given operator. One opportunity to improve this technique is to consider the partial buffering of tuples when operators crash. Instead of backing up outgoing tuples in all upstream operators [65], we can randomly discard some tuples in order to decrease the number of data items replayed after a crash. This can
improve accuracy if compared to a bursty tuple loss, but still reduce the recovery time if compared to an upstream backup approach.

- Another approach to improve accuracy in our partial graph replication approach is by developing smart replica failover algorithms. Because the state of the replicas can diverge as a result of the possible non-determinism of operators, the failover algorithm has to be smart in order to reduce the number of duplicated or omitted tuples.

- The current version of our fault injection tool (Chapter 4) supports the emulation of a bursty tuple loss fault model. Other models that can be implemented include tuple duplication, tuple reordering, replicated tuples with divergent attribute values, and corruption of attribute values. Additional fault models allow the observation of the application behavior under different situations. Such observations are critical to the development of new protection techniques, since they indicate what behavior should be enforced upon the occurrence of a fault.

- The outcome of our experimental methodology approach is in terms of four evaluation metrics that characterize each stream operator in the processing graph. The current approach to improve the application fault tolerance is to greedily protect the operators with higher measurements of the quality impact and recovery time metrics. However, protecting all operators with high metric values can still break the performance requirements of the application. A venue for future work is to formulate an optimization problem, whose objective is to find the best fault tolerance deployment for a given application, so that it still meets the performance constraints but minimizes the output quality degradation.

- In the current version of the modeling framework, the application composition and the generation of base operator models are done manually according to the algorithms described in Chapter 5. This process can be automated by developing a tool that interprets the streaming application source code and derives the required information (e.g., number of operators, input and output ports, stream connections). In addition, this tool can provide a standard interface that allows new fault tolerance techniques to be plugged into the model.
The results presented in this dissertation assume that operators fail independently. However, operators can be co-located in the same process of the operating system or run in different processes of the same computing node to achieve lower processing latency [28, 42, 55]. The current version of the framework can be modified to consider the physical deployment of the application and analyze the application behavior in such situations.
This appendix shows the equivalent SAN model for different built-in stream operator types contained in the SPADE language [28, 35]. The models described here follow the algorithm described in Section 5.2.1. In addition, the chapter reports how the framework models the propagation of punctuation marks. For the sake of simplicity, the figures shown in this appendix do not include the propagation of tainted tokens.

A.1 Source Operator

The source operator brings data to the application from an external data source. Figure A.1 gives an example of a source operator coded in SPADE. This operator processes incoming data from host some.node.com and sends the data to its output. The stream operator has a single output port, which is represented by the keyword stream and is named as SomeSourceStream.

```plaintext
stream SomeSourceStream(id: String, attr1: String, attr2: String) := Source()
  ["ctcp://some.node.com:2210/"]
//
```

Figure A.1: Source operator code.

Figure A.2 shows the equivalent SAN model for the source operator shown in Figure A.1. Note that the source operator includes an input gate (IG3) and an activity directly connected to the input stream connection place. The input gate enables the activity whenever there is space in the input buffers of the operator, which is represented by the input stream connection place. A token is added to the input stream connection place once the activity fires.

The semantic of a source operator indicates that the the operator should forward all messages it receives, i.e., the operator selectivity is 1.0. This
means that the probability assigned to the transition from the processing stage to the waiting stage is 0.0.

### A.2 Sink Operator

The sink operator externalizes the result stream of a streaming application. The output of this operator is sent to an external system component (e.g., a file, a database, or a GUI). Figure A.3 gives an example of a sink operator coded in SPADE. This sink operator receives as input the *SomeSourceStream* and stores its tuples into a file named (*appResult.dat*).

```plaintext
Nil
:= Sink(SomeSourceStream)
   ["file:///appResult.dat", csvFormat]
{}   
```

Figure A.3: Sink operator code.

Figure A.4 shows the SAN model generated by our modeling framework for a sink operator. The model is similar to the source operator model. Note that the sink operator does not have an activity connected to its input stream connection place. This is because the sink operator receives data from another operator in the processing graph, which, in this case, is an operator generating the *SomeSourceStream*. In addition, a sink operator does not have an output stream connection place. The place named *application output* stores all tokens generated by the model.
A.3 Functor Operator

A functor operator can filter, project, and transform attribute values of an incoming tuple. Figure A.5 gives an example of a functor that filters the tuples from the incoming streams (`SomeSourceStream`, `AnotherSourceStream`, and `YetAnotherStream`) based on the `id` attribute value. All tuples that contain the attribute `id` with value "AString" are sent forward. All others get discarded. Figure A.6 gives a functor with a different functionality. The operator consumes data from the same three streams. Instead of filtering tuples, it adds a `timestamp` attribute to the stream.

```plaintext
stream AStringStream(attr1: String, attr2: String) := Functor(SomeSourceStream,AnotherSourceStream, YetAnotherStream)
    [id="AString"]
    {}
```

Figure A.5: Functor operator with filtering function.

```plaintext
stream TimestampedStream(timestamp:Long, schemaFor(SomeSourceStream))
    := Functor(SomeSourceStream,AnotherSourceStream, YetAnotherStream)
    []
    {timestamp := timeMicroseconds()}
```

Figure A.6: Functor operator with transformation function.

Figure A.7 shows the SAN model generated by the framework for both functors (Figures A.5 and A.6). Each `input stream connection` place maps to one input stream represented in the source code. Note that although the two functors have different functionality, their corresponding SAN model is
the same. This is because their structure (i.e., number of I/O ports and number of I/O stream connections) is the same. The only change between the two functors is the case probability values assigned to the activity that represents the operator selectivity. While the functor in Figure A.6 has a selectivity of 1.0 (i.e., no tuple is discarded), the functor in Figure A.5 can have a selectivity lower than 1.0 (i.e., some tuples may be discarded due to the operator predicate).

![SAN model for functor operator.](image)

Figure A.7: SAN model for functor operator.

### A.4 Aggregate Operator

An aggregate applies functions over a window (i.e., a collection) of tuples. The developer can configure the how this window accumulates tuples. The window can contain a specific number of tuples (i.e., count-based) or collect all tuples received over a period of time (i.e., time-based). Figure A.8 shows an example of a count-based aggregate. In this example, the operator maintains the 20 most recent incoming tuples (\texttt{count(20)}). Figure A.9 shows an example of a time-based aggregate. Only tuples received in the last 15 seconds (\texttt{time(15)}) are maintained.

The aggregate operator also has a triggering policy, which defines when the operator should compute the aggregation function and emit a tuple in its output port. One possible triggering policy is to generate an output tuple after a specific number of tuples are received. In the code displayed in Figure A.8, the operator generates an output every time a new tuple is received (\texttt{count(1)}). Another triggering policy is to generate an output tuple based on a time interval. Figure A.9 shows an example where a tuple is emitted at every 2 seconds (\texttt{time(2)}).
**stream MaxIntStream(max:Integer)**

```java
:= Aggregate(IntegerStream <count(20),count(1)>)
  []
  {max := Max(intValue)}
```

Figure A.8: Aggregate with count-based triggering policy.

**stream CountAStringStream(count:Integer)**

```java
:= Aggregate(AStringStream <time(15),time(2)>)
  []
  {count := Cnt()}
```

Figure A.9: Aggregate with time-based triggering policy.

Figure A.10 shows the equivalent SAN model for the aggregate with count-based triggering policy. Because the triggering policy for this aggregate is to emit a tuple at every new incoming tuple, the model should have its *selectivity* parameterized to 1.0. Note that this selectivity parameter does not need to be derived from the code itself. The parameter can be obtained by profiling the application, as discussed in Section 5.2.

Figure A.10: SAN model for aggregate with count-based triggering policy.

Figure A.11 shows the SAN model for the aggregate coded in Figure A.9. To emulate the triggering of an output based on time, the model includes a phony input port. This input port is represented by the *input stream connection* place labeled *time trigger*. Based on the time interval specified in the application code, this input stream connection receives a new token that is processed by the operator. When this token is processed, the operator produces an output (selectivity of 1.0). All other incoming tokens from the original input stream connection are discarded (selectivity of 0.0). Note that the behavior of this operator could also be approximated by observing the profile of the application. Adding a time trigger allows a more precise modeling of the time-based behavior, since it copes with the possibly different
incoming data rates of upstream operators.

Figure A.11: SAN model for aggregate with time-based triggering policy.

A.5 Join Operator

The join operator correlates streams associated to two different ports according to a predicate condition. Similarly to an aggregate, the join operator looks for correlations based on the contents of a window. This operator only triggers correlations upon the arrival of new tuples. More details on the join operator logic can be found in Section 3.2.4.

Figure A.12 gives an example of a join operator where tuples coming from Stream1 and Stream2 are correlated with tuples from Stream3 according to an equality predicate. Stream1 and Stream2 are associated to the first input port, while Stream3 is associated to the second input port. The operator maintains a window of 30 tuples (count(30)) for both ports.

```
stream CorrelatedStream(id:Integer, attr1:String, attr2:String) := Join(Stream1, Stream2 <count(30)>;
    Stream3 <count(30)>);

[{$1.id = $2.id}]
{id := $1.id, attr1 := $1.attr, attr2 := $2.attr}
```

Figure A.12: Join operator code.

Figure A.13 shows the equivalent SAN model for the join operator coded in Figure A.12. The three input streams are represented by the three input places tagged as input stream connections. The two processing tuple places represent the two input ports of the join operator. Although the join has two
input ports, it has only one output port (CorrelatedStream). In the model, this is represented by a single sending output place.

![SAN model for join operator.](image)

Figure A.13: SAN model for join operator.

A.6 Sort Operator

The sort operator arranges the tuples contained in a window according to the specified ordering configuration (ascending or descending). Figure A.14 gives an example of a sort operator with a time-based window of 20 seconds (time(20)). Once the operator sorts all tuples accumulated over a 20 second interval, it sends all of them to the operator output. After that, it sends a punctuation to mark the end of a window.

```plaintext
stream SortedStream(max:Integer) := (IntegerStream <time(20)>) [intValue(asc)]{}
```

Figure A.14: Sort operator with time-based window.

Figure A.15 shows the model for the sort operator in Figure A.14. Similarly to the time-based aggregate, the model includes a phony input port to trigger the punctuation mark once the window ends. The emulation of a punctuation mark can be done through a phony output port. All tokens sent through this port are considered punctuation marks and should be processed by an equivalent phony input port on downstream operators. Note that this model does not buffer all tuples before sending them to operators downstream. A model that considers tuple buffering can be achieved with minor modifications to the processing stage.
A.7 Barrier Operator

The barrier operator synchronizes multiple streams that are logically related. Once the operator receives at least one tuple in each input port, it emits a new tuple. Figure A.16 gives an example of code for a barrier operator. This example shows the synchronization of tuples from Stream1 and Stream2 with tuples from Stream3.

```
stream SyncStream(id1:Integer, attr1:String, attr2:String) := Barrier(Stream1,Stream2; Stream3) []
{id1 := $1.id, attr1 := $1.attr, attr2 := $2.attr}
```

Figure A.16: Barrier operator code.

The equivalent SAN model for this operator is identical to the one shown in Figure A.13. This model is an approximation of a barrier. A more precise barrier model can be made with minor changes to the base model.

A.8 Puncator Operator

A puncator operator inserts a punctuation mark in a stream once a given predicate evaluates to true. Figure A.17 gives an example of an operator that inserts a punctuation mark after it receives a tuple where the value of attribute id is greater than 100.
A.17 PunctuatedStream(id:Integer, attr:String) := Punctor(Stream1) [id > 100, after] {} Figure A.17: Punctor operator code.

Figure A.18 shows the SAN model for the punctor operator coded in Figure A.17. The processing stage activity of a punctor has an additional case probability associated with it. While the first two cases represent the probability of discarding a tuple and the probability of sending a tuple to the operator output port, the third case represents the probability of emitting a punctuation mark. If this case is taken, the operator sends a regular tuple through its output stream connection and an extra tuple through the punctuation output stream connection.

Figure A.18: SAN model for punctor operator.

A.9 Split Operator

The split operator allows an input stream to be divided into multiple streams according to a specified mapping. Figure A.19 gives an example of a split operator where Stream1 is split into three other streams (Attr1Stream, Attr2Stream and Attr3Stream) according the mapping defined in the file map.txt. Each output stream maps to one output port.

Figure A.20 shows the SAN model for the split operator given in Figure A.19. This model includes three different output stream connections,
where each of them is associated to one output port. The probability of writing a tuple to each output port is represented by the cases associated to the processing activity.

A.10 Delay Operator

The delay operator forces a given delay for each incoming tuple. Figure A.21 gives an example of a delay operator where each tuple is delayed for 2 seconds.

The model of this operator is the same as the aggregate shown in Figure A.10. The activities of the model can be parameterized with the average...
time to emit a tuple, as in the other operators. As described in Section 5.2, such parameters can be obtained by profiling the application.
REFERENCES


