Failure Transparency in Stateful Dataflow Systems

Aleksey Veresov[1]
EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Jonas Spenger[1]
EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Paris Carbone
EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden
Digital Systems, RISE Research Institutes of Sweden, Stockholm, Sweden

Philipp Haller
EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Abstract

Failure transparency enables users to reason about distributed systems at a higher level of abstraction, where complex failure-handling logic is hidden. This is especially true for stateful dataflow systems, which are the backbone of many cloud applications. In particular, this paper focuses on proving failure transparency in Apache Flink, a popular stateful dataflow system. Even though failure transparency is a critical aspect of Apache Flink, to date it has not been formally proven. Showing that the failure transparency mechanism is correct, however, is challenging due to the complexity of the mechanism itself. Nevertheless, this complexity can be effectively hidden behind a failure transparent programming interface. To show that Apache Flink is failure transparent, we model it in small-step operational semantics. Next, we provide a novel definition of failure transparency based on observational explainability, a concept which relates executions according to their observations. Finally, we provide a formal proof of failure transparency for the implementation model; i.e., we prove that the failure-free model correctly abstracts from the failure-related details of the implementation model. We also show liveness of the implementation model under a fair execution assumption. These results are a first step towards a verified stack for stateful dataflow systems.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Checkpoint / restart

Keywords and phrases Failure transparency, stateful dataflow, operational semantics, checkpoint recovery

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.42

Related Version *Extended Version*: `https://arxiv.org/abs/2407.06738` [62]

Funding This work was partially funded by Digital Futures under a Research Pairs Consolidator grant (PORTALS).

1 Introduction

Stateful dataflow systems have seen wide adoption in the modern cloud infrastructure due to their ability to process large amounts of event-based data at ingestion time [24]. Apache Flink [13], for example, is used to power tens-of-thousands of streaming jobs with up to nine billion events per second at ByteDance [48], and several thousand streaming jobs at Uber [25]. An essential aspect of stateful dataflow systems is the recovery from failures, as failures are to be expected in any long-running streaming job [19]. However, failure recovery is non-trivial. For example, simply recovering from a failure by restarting a job from the

[1] Both authors contributed equally to this research.

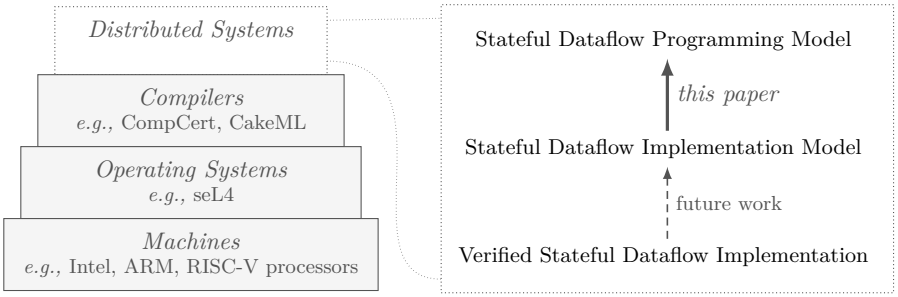


Figure 1 This work in the context of a fully verified stack for distributed programming.

very beginning would discard all progress up to that point, making the recovery prohibitively expensive. To balance the need for efficiency and reliability, stateful dataflow systems have to embrace complex failure recovery protocols. Because of their complexity, the correctness of these recovery protocols is a crucial problem for the reliability of stateful dataflow systems.

In previous work [21, 44, 26], a failure-masking recovery protocol is considered to be correct, if failures can be masked such that the user cannot observe the failures. This property is also known as *failure transparency*, *i.e.,* a user should be able to ignore failures as if they do not occur. Failure transparency has been shown for some distributed systems, such as Durable Functions [8], Reliable State Machines [51], reliable actors (KAR) [61], and serverless microservices (µ2sls) [30]. As for stateful dataflow, the core mechanism used in Apache Flink's [13] recovery protocol, namely Asynchronous Barrier Snapshotting (ABS) [11, 12], has been shown to be a correct snapshotting protocol [10]. However, the proof does not reason about failure transparency and its related aspects, such as modelling failures and the recovery from failures, as well as about the equivalence of observed executions. That is, there has been no formal proof that Apache Flink's entire failure recovery protocol provides failure transparency. Furthermore, the literature lacks a formal definition of failure transparency for systems described with distinct failure-related rules using small-step operational semantics, a widely-used method for defining program execution in programming languages theory.

An important approach for ensuring reliability and correctness is machine-checked formal verification, *i.e.,* proving that a system implements its specification. There is well-known prior work on verified compilers [41, 33, 53], operating systems [31], as well as processors [16, 29, 55] (Figure 1, left). However, there is an apparent lack of verified distributed systems, particularly, there is no verified stateful dataflow system. We believe that it is essential to address this gap in order to prevent disastrous outages of distributed infrastructure as known today.

This work is a first step towards the grand goal of providing a fully verified reliable stack for distributed programming, as shown in Figure 1. It addresses the highlighted gap by: (1) providing a definition of failure transparency, (2) formalizing a stateful dataflow system as a model in small-step operational semantics, under the assumptions of crash-recovery failures and FIFO-ordered channels, and (3) formally proving that the model permits abstracting from failures, *i.e.,* that it is failure transparent. Our definition of failure transparency is based on *observational explainability*, a property which, informally, says that the *explainable* implementation model generates the same observable output as is possible in the *explaining* abstract model. Using this property, a system is defined as failure transparent if the whole system is observationally explainable by its explicitly separated failure-free part. Finally, we prove that our formal model of a stateful dataflow system based on Asynchronous Barrier Snapshotting [13, 10] is failure transparent. This abstraction from failures is designed to serve the end users of the modelled system with less interest in its implementation details.

Contributions. In summary, this paper makes the following contributions.

- We provide the first small-step operational semantics of the *Asynchronous Barrier Snapshotting* protocol within a stateful dataflow system, as used in Apache Flink (Section 4).
- We provide a novel definition of *failure transparency* for programming models expressed in small-step operational semantics with explicit failure rules and the intuitions behind it (Section 5). It is the first attempt to define failure transparency in the context of stateful dataflow systems.
- We prove that the provided implementation model is failure transparent and guarantees liveness (Section 6).
- We provide a mechanization of the definitions, theorems, and models in Coq.[2]

Outline. Section 2 introduces background on failures, distributed systems, stateful dataflow, as well as some basic notation used throughout this paper. Section 3 informally introduces the stateful dataflow programming model and failure recovery via the Asynchronous Barrier Snapshotting (ABS) protocol. Section 4 provides a small-step operational semantics of a stateful dataflow system based on ABS. Section 5 defines failure transparency and observational explainability for programming models expressed in small-step operational semantics. Section 6 proves that the implementation model is failure transparent. Section 7 discusses related work, and Section 8 concludes this paper.

2 Background

2.1 Failures in Distributed Systems

A distributed system is a system of many processes communicating over a network [9]. The kind of distributed systems which are related to this work are event-based processing systems such as stateful dataflow systems [19, 67, 13, 52, 69, 60]. Failures within such systems are expected to happen, due to their typical large scale and longevity [19]. However, failures are notoriously hard to deal with within distributed systems. For this reason, *failure transparency* is a necessary abstraction, as it enables the user to abstract from failures. Failure transparency as a general concept, and failure recovery protocols are both well-studied topics in distributed systems [63, 65, 40, 44, 43, 26, 21]. Moreover, failure transparency has seen an increase in interest within the programming languages community in recent years [8, 30, 51, 61]. The goal of failure recovery is to provide automatic system means to recover from system failures, in ways which the system user may or may not notice. In contrast, the goal of failure transparency is to provide an abstraction of the system, such that the abstraction hides the internals of failures and failure recovery, masking the failures from the user [26]. For this reason, failure transparency greatly simplifies the programming model to the benefit of the end user.

2.2 Stateful Dataflow and Apache Flink

Stateful dataflow systems, sometimes also called stream processing or dataflow streaming systems, such as Apache Flink [13], have become ubiquitous for real-time processing of large amounts of data [48, 25]. Other well known dataflow systems include Google Dataflow [2], IBM Streams [18], Apache Spark [66] and Spark Streaming [68], Timely Dataflow [52],

[2] `https://github.com/aversey/abscoq`

NebulaStream [69], Portals [60], and more [7, 56]. The popularity and wide-spread use of dataflow systems [25, 48] is due to their ability to scale-out production workloads. In particular, they provide high throughput, low latency, and strong guarantees (such as failure transparency, sometimes referred to as exactly-once processing). The programming model of most stateful dataflow systems is based on acyclic dataflow graphs [24]. In these graphs, the nodes are stateful processing tasks, and the edges are streams of data. As failure transparency is an important aspect of the stateful dataflow programming model, it and its failure recovery protocol is the focus of this paper.

2.3 Asynchronous Barrier Snapshotting

The failure recovery protocol used in Apache Flink [13] is a checkpointing-based rollback recovery protocol [21], in which the system regularly takes checkpoints and, after a failure, recovers to the latest completed checkpoint. For batch execution systems, such as MapReduce [19], the general approach is to atomically execute one batch at a time, and if a failure occurs, the system restarts from the beginning of the current batch. In contrast, computation on stateful dataflow streaming systems is continuous [24], without predefined recovery points in its execution, complicating the failure recovery. The solution to recovery in continuous computations is the acquisition of causally consistent snapshots [14], which can be used for the recovery to a consistent system state after a failure [21]. The specific implementation of Apache Flink [13] and other stateful dataflow systems [60] use the Asynchronous Barrier Snapshotting (ABS) protocol [12], an extended and optimized variant for data processing graphs of the Chandy-Lamport snapshotting protocol [14], for taking causally-consistent snapshots. In contrast to the Chandy-Lamport snapshotting protocol, the ABS protocol is tailored to acyclic dataflow graphs and its snapshots do not contain any in-flight events. In contrast to batching protocols, the ABS protocol is fully asynchronous, and does not require blocking coordination. For these reasons, the ABS protocol greatly benefits the end-to-end latency and throughput of the system.

2.4 Basic Notation

Functions. We denote a function f similarly to set-builder notation as: $\big[k \mapsto t \;\big|\; k \in \mathrm{dom}(f)\big]$. The part after the bar defines the domain of the function. The part before the bar defines the value of the function at point k by the expression t. The expression t captures all variables defined on the right side of the bar, including k. A function with only one element in its domain is represented as $\big[x \mapsto x'\big]$, for example, $\big[3 \mapsto 7\big]$ is such a function that $\mathrm{dom}(\big[3 \mapsto 7\big]) = \{3\}$ and $\big[3 \mapsto 7\big](3) = 7$. We denote function update as $f\, g$, such that:

$$(f\, g)(x) = \begin{cases} g(x) & \text{if } x \in \mathrm{dom}(g) \\ f(x) & \text{if } x \notin \mathrm{dom}(g) \end{cases}$$

Sequences. We represent a sequence S as a function f with domain $\big\{i \;\big|\; i \in \mathbb{N} \wedge i < |S|\big\}$. The length of the sequence is represented by $|S|$ and may be infinite. The notation S_i stands for the i-th element of the sequence S and equals $f(i)$. To simplify our analysis of sequences, we use $\big[t\big]_i^n$ as a shorthand for $\big[i \mapsto t \;\big|\; i \in \mathbb{N} \wedge i < n\big]$, where t is an expression that captures i and represents the i-th element of the sequence. Therefore, for any sequence S, we have that $S = \big[S_i\big]_i^{|S|}$.

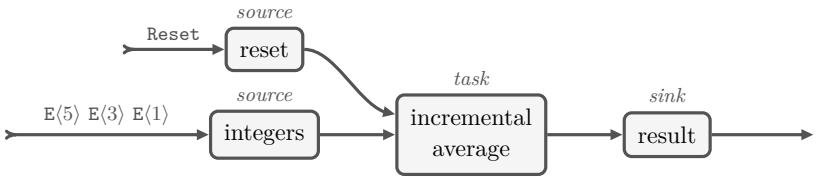


Figure 2 Example stateful dataflow program calculating the incremental average of a data stream of integers. Another stream is used to transfer control messages resetting the state of the program.

The usage of indices for variables standing for sequences may differ from other variables. If S stands for a sequence, then S_i corresponds to the i-th element of S. If, in contrast, x is not a sequence, then x_i is an independent variable and is not connected to x or any x_j. To avoid confusion, we name sets and sequences using uppercase and individual elements using lowercase.

Sequence concatenation can be used to extend or shrink existing sequences. We include a shorthand notation for sequence concatenation, concatenating S with S' as follows $S : S' \equiv \left[i \mapsto S_i \mid i \in \mathbb{N} \wedge i < |S|\right]\left[j + |S| \mapsto S'_j \mid j \in \mathbb{N} \wedge j < |S'|\right]$. To simplify extraction and addition of single elements, we denote single-element sequences $\left[x\right]_i^1$ as $\left[x\right]$, where x is the only value in the sequence. The empty sequence is represented as ε.

3 Stateful Dataflow

Stateful dataflow systems, sometimes also called *distributed dataflow*, *dataflow streaming*, or *stream processing* systems, are widely used for real-time processing of large amounts of streaming data. This section informally introduces the stateful dataflow programming model and its failure recovery mechanism, which we formalize and prove correct in later sections. It is mostly based on Apache Flink [13], a stateful dataflow system, however, the core concepts and techniques involved also apply to other similar systems [19, 66, 68, 2, 18, 52, 69, 60].

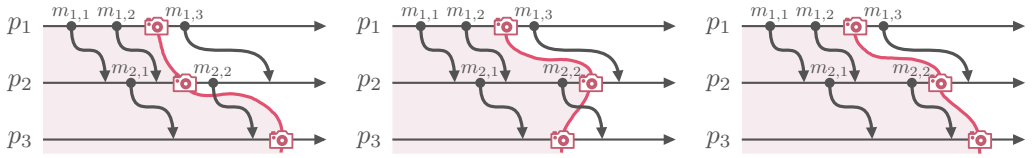
3.1 A Taste of Programming in Stateful Dataflow

Figure 2 shows a stateful dataflow example calculating the incremental average of a stream of integers. The example consists of two *sources* ingesting streams of events into the system. One source ingests a stream of integers $\texttt{E}\langle i \rangle$, and the other ingests a stream of `Reset` events. The term *stream* can be understood as an unbounded sequence of events, it may in general continue forever. The example also consists of a *task*, an internal processing unit, which calculates an incremental average of the integers. The incrementally computed averages are emitted to a *sink*, which is the output of the system.

A more detailed representation of the example is shown in Listing 1. Sources, tasks, and sinks are created using corresponding functions. The API enables users to: (1) create sources, tasks, and sinks; (2) specify the connections in the graph by providing input and output streams; and (3) to specify how the tasks process events by providing their processing functions. In this example, when the task receives an integer event $\texttt{E}\langle i \rangle$, it updates the average and emits the new average. When it receives a `Reset` event, it resets its local state, such that the average is reset to its initial state. To note is that the task is considered *stateful*, as it maintains local state for its computation of the incremental average, even though the processing function f is a pure function. Also to note is that it is possible to provide an easier-to-use API above this core API, for example an API based on higher-order functions (`map`, `flatMap`, etc.) [13, 60, 2, 52].

Listing 1 A stateful dataflow program calculating the incremental average of a data stream of integers (see Figure 2).

```
Source(input = "src_reset", output = "reset")
Source(input = "src_ints", output = "ints")
Sink(inputs = { "avgs" }, output = "sink_avgs")
Task(inputs = { "src_ints", "src_reset" }, output = "avgs",
  f = (event, state) => event match {
    case Reset =>
      val new_state = {sum = 0, count = 0}
      return (Nil, new_state)
    case E⟨value⟩ =>
      val new_state = {sum = state.sum + value, count = state.count + 1}
      val average = E⟨value = new_state.sum / new_state.count⟩
      return (average : Nil, new_state) })
```



(a) Inconsistent snapshot. **(b)** Chandy-Lamport snapshot. **(c)** ABS snapshot.

Figure 3 Examples of snapshots obtained in a distributed stateful dataflow system with three processes $p_1 \rightarrow p_2 \rightarrow p_3$.

3.2 Failure Recovery via Asynchronous Barrier Snapshotting

Failure recovery is a crucial aspect of stateful dataflow systems. In this section, we describe the failure recovery mechanism of the Asynchronous Barrier Snapshotting (ABS) protocol [12] as used in Apache Flink. More specifically, ABS is a *distributed snapshotting* protocol [14] which is used for the checkpointing-based rollback-recovery protocol [21] within Apache Flink [12]. After a failure, a checkpointing-based recovery will restart the system from the latest valid snapshot of the system [21].

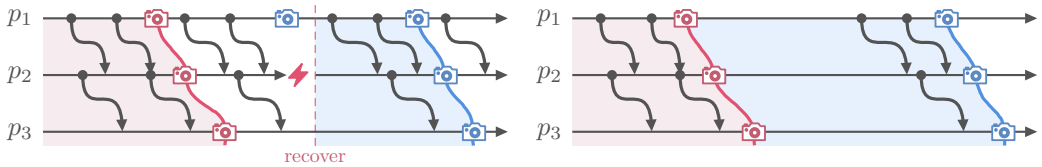
Distributed Snapshotting Protocols. A distributed snapshotting protocol is considered *causally consistent* if it captures snapshots that do not violate causality [14]. Causality, here, refers to the causal order relation [35], informally: two events are causally ordered if one event was part of a causal chain leading to the other event. Consequently, a causally consistent snapshot captures the state of a system such that all events causally preceding any other event in the snapshot are included. This definition is illustrated by three example executions of different snapshotting protocols for a dataflow graph consisting of three nodes, shown in Figure 3. An incorrect implementation (Figure 3a) would be to let the processes periodically capture a snapshot of their state without coordination. A snapshot captured with this method can be inconsistent, thus not suitable for recovery, as it may violate causality. In the example, the incorrect snapshot has captured that $m_{2,2}$ was received by p_3 but never sent by p_2, this is a violation of causality, and recovery from such a snapshot would be considered erroneous. In contrast, consistent snapshotting protocols do not violate causality. The Chandy-Lamport asynchronous snapshotting protocol [14] (Figure 3b) solves

Listing 2 Representation of an event handler within a stateful dataflow system implementing failure recovery using the ABS protocol [12, 10].

```
EventHandler Def TK⟨f, [S_i]_i^n, o⟩
  Vars state, snapshots
  On Event Receive < S_j, epoch, Event⟨w⟩ > If ∃v: state = < epoch, v > Do
    v', w' = f(v, w')
    state = < epoch, v' >
    emit(< o, epoch, Event⟨w'⟩ >)
  On Event Receive [< S_i, epoch, Border >]_i^n If ∃v: state = < epoch, v > Do
    snapshots.update(epoch ↦ v)
    state = < epoch + 1, v >
    emit(< o, epoch, Border >)
  On Event Fail Do
    state = Failed
  On Event Recover < recoverEpoch > Do
    state = < recoverEpoch, snapshots(recoverEpoch) >
```

this issue through distributed coordination by means of disseminating markers during its regular execution, separating pre-snapshot and post-snapshot messages. However, a snapshot captured with the Chandy-Lamport protocol may capture in-flight events: as shown in the example (Figure 3b), the message $m_{2,2}$ was sent (according to p_2's snapshot) but not yet received (according to p_3's snapshot). The Asynchronous Barrier Snapshotting (ABS) protocol [12, 10], in contrast, captures complete distributed computations without in-flight events by modification of the marker-based Chandy-Lamport protocol. As shown in Figure 3c, the snapshot does not include any in-flight events.

The ABS Protocol. A representation of the ABS protocol [12, 10] corresponding to our formalization in Section 4 is found in Listing 2. The handler has two mutable states: the processing task's volatile `state`, and the persistent `snapshots` state. The `state` is a tuple `< epoch, v >` consisting of the current `epoch`'s sequence number being processed, and the state `v` of the processing task. The first event handler consumes an event `Event`$\langle$`w`$\rangle$ from a stream with stream name S_j out of the sequence of stream names S for some `epoch` if it is not currently in a failed state. It processes the event w on its current state v, which produces an output event w' and new state v'. It then updates its mutable state, and emits the output on its outgoing stream with stream name o. The second handler processes the `Border` markers from the higher-level ABS protocol. It will consume all border events from all its incoming streams in a single step. In doing so, it will take a snapshot of the local state and update the epoch number, as well as disseminate the border marker further downstream. To note is that the first handler does not consume from a stream if that stream has a border marker as its next event, instead it will block such streams until the border step (*i.e.,* the second handler) has been taken. The first and second handlers implement the ABS protocol, whereas the third and fourth handlers implement the failure recovery. The third handler models the random failures of tasks, a task can randomly fail at any time, in which case it loses its volatile state. The fourth handler implements the failure recovery, and is triggered by some external coordinating instance once it has detected the failure. Once a failure has



(a) Execution with a failure. **(b)** Observed execution.

Figure 4 An execution with failures and its observed execution.

been detected, all tasks are recovered to the same epoch which corresponds to the latest snapshot of the system. When triggered, the fourth handler recovers the state back to the snapshot of the epoch found in the message.

Failure Recovery. The dataflow system can recover from failures using the ABS protocol. Figure 4a shows an execution using the ABS protocol in which p_2 fails. The coordinator (not displayed) will eventually discover the failure, and trigger a synchronous recovery step in which all processes recover to the latest completed snapshot and continue processing from there. Even though failures occur in the execution, the observer will be able to construct an idealized execution corresponding to our notion of failure transparency in which there are no failed events or incomplete epochs as shown in Figure 4b. This is, loosely speaking, achieved by ignoring the side effects from the failed epochs, and is explored in detail in Section 6.

4 Implementation Model

We now provide a formal model of the stateful dataflow system described above. The goal of this formalization is to capture and analyze key aspects of the implementation of the system, with focus on its failure recovery using the Asynchronous Barrier Snapshotting protocol [12, 10]. The formal model is presented in two parts: the first part presents an explicit evaluation rule for message passing, and the second part presents the evaluation rules for processing and failure recovery.

4.1 Streaming Model

The streaming model is based on processors (or tasks) that communicate via streams. A processor is a stateful entity that may consume an event from an incoming stream, process it, and produce events to its outgoing stream. Streams, in turn, transport the events between processors in a FIFO order. With this notion of processors and streams, we can execute computational graphs by means of steps. Note that, in this section, we discuss a general streaming model, leaving the implementation of processors abstract. Whereas, in the next section, we discuss concrete implementations of processors.

Syntax. Figure 5 shows the syntax of the streaming model. A configuration $c = \langle \Pi, \Sigma, N, M, D \rangle$ represents a point in an execution of a streaming program. The processors Π indexed by identifiers p represent processor definitions, for which Σ represents the states of the processors. The messages M are modeled as a sequence of all messages, for which a message m corresponds to a tuple of a sequence number n, a stream name s, and the message data d. The current sequence number from which a processor p reads from or writes to a stream s is represented by $N_p(s)$. The sequence numbers for all processors are represented by N. When

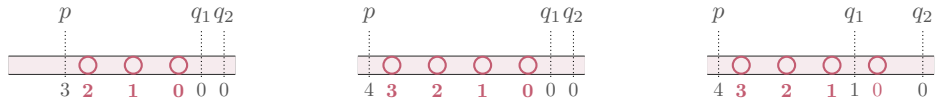
$p,\ q$ processor ID $\quad s,\ o$ stream name $\quad n \in \mathbb{N}$ sequence number

π processor $\quad \sigma$ state $\quad d$ message data $\quad D$ auxiliary data

$$\Pi ::= \left[\pi\right]_p^{|\Pi|} \quad \text{processors}$$
$$\Sigma ::= \left[\sigma\right]_p^{|\Pi|} \quad \text{states}$$
$$M ::= \left[m\right]_i^{|M|} \quad \text{messages}$$
$$N ::= \left[N_p\right]_p^{|\Pi|} \quad \text{sequence numbers}$$
$$N_p ::= \left[s \mapsto n \mid s\right] \quad \text{sequence numbers of } p$$

$$X ::= \left[x\right]_i^{|X|} \quad \text{actions}$$
$$x ::= \quad \text{action}$$
$$+\,s\,d \quad \textit{production}$$
$$\mid -\,s\,d \quad \textit{consumption}$$
$$m ::= n\,s\,d \quad \text{message}$$

Figure 5 Streaming syntax.



(a) Initial state. **(b)** Production by p. **(c)** Consumption by q_1.

Figure 6 Production and consumption to/from a stream with a producer p and consumers q_1 and q_2.

a processor processes a message, it may produce and consume messages. This production and consumption is represented by a sequence of actions X. A production action producing message d to stream s has the form $+\,s\,d$, similar to the consumption action $-\,s\,d$. The auxiliary data D is used to store global and additional execution information which is specific to the models; for example, it can be used to implicitly model the global coordinator. In the formalization here, the processor π, state σ, message data d and auxiliary global data D are seen as atomic values, that is, no information about their internal structure is provided. These limitations permit reusing the same syntax and rule for different instantiations of π, σ, d and D.

Figure 6 illustrates a stream as a sequence of messages with index numbers. When producing an event to a stream (Figure 6b), the event is appended to the stream with an incremented index number. This also increments the producer's index number for the stream from 3 to 4. Similarly, the consumer's index number points to the next event to be consumed. Figure 6c shows that the consumer q_1 has consumed the event 0, which in turn also increments its index number for the stream, pointing at the next event. Consumers and producers process the stream independently and asynchronously. The production of a message is a kind of broadcast, in the sense that all processors will have to consume it before consuming a newer message.

Step Rule. The streaming model essentially consists of a single rule (S-Step) which describes the processing of messages. Intuitively, a streaming step from configuration $\langle\, \Pi,\ \Sigma,\ N,\ M,\ D \,\rangle$ can be taken if there is a local step with actions X, such that the actions are applicable. A local step describes how the processor Π_p changes its current state Σ_p to its next state Σ'_p using actions X. The actions X are applicable to N_p and M if all messages consumed by X are available on the input streams of the processor. The application of the actions X results in N'_p and M', which are the incremented sequence numbers for the processor and the set of

messages M extended with the newly produced messages. In case of taking a streaming step, the configuration transitions to the new configuration $\langle \Pi, \Sigma\big[p \mapsto \Sigma'_p\big], N\big[p \mapsto N'_p\big], M', D \rangle$. In summary, the result of the streaming step is an update of the local state of the processor according to the local step, and an update of the sequence numbers and messages according to the actions X. To simplify the analysis of streaming steps, auxiliary information about the processor ID, its sequence numbers, and the actions of the step is placed on the arrow of the execution step. This information can be omitted when it is not needed by applying abstraction steps S-AbsX and S-AbsP.

$$\frac{\Pi_p \Vdash \Sigma_p \xrightarrow{X} \Sigma'_p \qquad X(N_p, M) = (N'_p, M')}{\langle \Pi, \Sigma, N, M, D \rangle \xrightarrow[p]{N_p, X} \langle \Pi, \Sigma\big[p \mapsto \Sigma'_p\big], N\big[p \mapsto N'_p\big], M', D \rangle} \text{ S-Step}$$

$$\frac{c \xrightarrow[p]{N_p, X} c'}{c \underset{p}{\Longrightarrow} c'} \text{ S-AbsX} \qquad \frac{c \underset{p}{\Longrightarrow} c'}{c \Rightarrow c'} \text{ S-AbsP}$$

The streaming rule can be applied if there exists a derivation of the form $\Pi_p \Vdash \Sigma_p \xrightarrow{X} \Sigma'_p$ for a processor Π_p. These are called local steps, since they have access only to the local data of a processor, *i.e.,* its definition, state and locally accessible messages. These rules describe the local step of a processor, in which the processor may produce and consume messages/actions X, and update its local state to Σ'_p. The produced actions X modify the sequence numbers of the processor N_p and the messages in the system after application. This is computed by the action application function $X(N_p, M)$ and results in the new sequence numbers N'_p and messages M' for the next configuration as defined below.

Action Application. The action application rule defines how actions modify the sequence numbers and messages. A production action $+\,s\,d$ increases the sequence number of the stream s for the producer, and adds the message to the sequence of messages. Each stream has at most one producer; thus, we do not need to specify the producer in the action or message. A consumption action $-\,s\,d$ increases the sequence number of the stream s for the consumer, but does not remove it from the sequence of messages, as there may be other consumers waiting to consume the message. To note is that the consumption action application is only defined if the message is present in the sequence of messages. Due to this, local steps may only be applied in the context of the S-Step rule if the consumed message is present in the sequence of messages. The remaining cases of the definition are for the recursive application of actions.

▶ **Definition 4.1** (Action Application).

$$\begin{aligned}
(+\,s\,d)(N_p, M) &= (N_p\big[s \mapsto N_p(s) + 1\big], M \cup \big\{N_p(s)\,s\,d\big\}) \\
(-\,s\,d)(N_p, M) &= (N_p\big[s \mapsto N_p(s) + 1\big], M) \text{ if } N_p(s)\,s\,d \in M, \text{ undefined otherwise} \\
(\big[x\big] : X)(N_p, M) &= X(x(N_p, M)) \\
\varepsilon(N_p, M) &= (N_p, M)
\end{aligned}$$

According to the definition, it is not always possible to apply an action. This may be the case if, for example, a message for some sequence number is not yet available on its stream. This enables indirectly "passing" messages to the local step rules. Whereas the local step rule is defined for all possible steps for all messages that it may consume, cases in which the message consumption is not applicable by the action application definition are ruled out by the streaming global step rule. This leaves only messages which are applicable to be applied to the steps, thus passing the message to the rule.

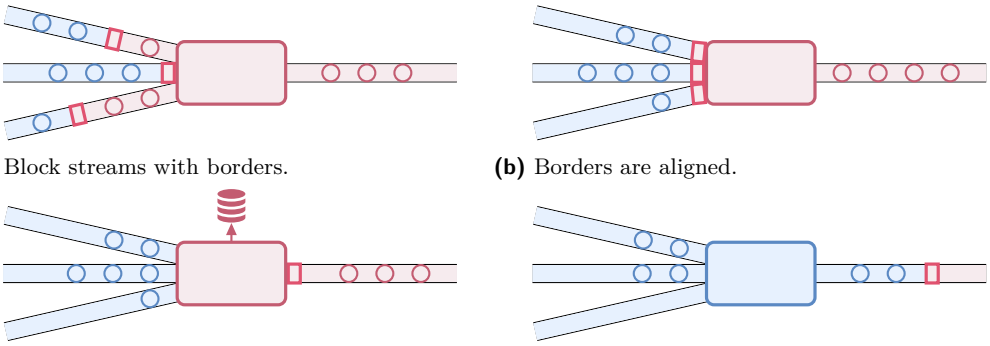
	$v,\ w$	value		$e \in \mathbb{N}$	epoch number
π	$::= \texttt{TK}\langle\, f,\ [S_i]_i^{\lvert S\rvert}\ ,\ o \,\rangle$	task	d	$::= \langle\, e,\ d_\mathrm{C} \,\rangle$	message
a	$::= [e \mapsto v \mid e]$	snapshot archive	d_C	$::=$	message cases
σ	$::= \langle\, a,\ \sigma_\mathrm{V} \,\rangle$	state		$\texttt{EV}\langle\, w \,\rangle$	*event*
σ_V	$::=$	volatile state		$\mid \texttt{BD}$	*epoch border*
	$\texttt{fl}$	*failed state*	D	$::= M_0$	initial input messages
	$\mid \langle\, e,\ v \,\rangle$	*normal state*			

Figure 7 Stateful dataflow syntax.

4.2 Stateful Dataflow Model

The presented stateful dataflow model consists of processing tasks, sources, and sinks. A processing task consumes messages from a set of input streams, and produces messages on its output stream. The task's behavior is defined by a function f which processes the messages. The function f takes the task's state and an input message, and produces a new state and a sequence of output messages: $f(v, w) = v', [W_i']_i^n$. The presented formal model does not provide a syntax and semantics for functions; they can be expressed using any suitable formalism. The sources of the model are emulated by streams which are initialized in the first configuration to contain all the messages which are to be consumed from the source. That is, each source is represented by its output stream, which in turn becomes an input to one of the tasks of the computational graph. Sinks are also emulated as streams, however, in contrast to sources, they are initially empty. The computation of the system, informally, takes inputs from the sources, processes them in the processing graph, and produces outputs to the sinks.

Syntax. The syntax of the implementation model (Figure 7) extends the shared streaming syntax and semantics (Figure 5) by providing concrete instances of processors/tasks, messages, and state definitions. A task $\texttt{TK}\langle\, f,\ S,\ o \,\rangle$ is a three-tuple of its processing function f, sequence of input streams S, and its output stream o. Tasks process messages which are tuples of an epoch number e and the message data d_C. There are two kinds of messages: normal events $\texttt{EV}\langle\, w \,\rangle$ and epoch borders $\texttt{BD}$. The epoch border messages are markers used for the snapshotting algorithm, whereas the events are the actual data processed by the tasks. When processing, the tasks manipulate state which consists of a persistent *snapshot archive* a, *i.e.,* a map from epoch numbers to the corresponding local snapshots, and some *volatile state* σ_V. The snapshot archive is a map from epoch numbers e to the state v of the processor at the end of the epoch. The volatile state is either a *failed state* $\texttt{fl}$ or a *normal state* $\langle\, e,\ v \,\rangle$, consisting of the current epoch number and the state data value v of the processor. As with the messages, normal states are tagged by epoch numbers. A processor is in a failed state if it has crashed and lost its volatile state. The auxiliary data D used for this model consists of the initial input messages for the system. As we may need to restore the messages which are yet to be consumed, we keep track of all the initial input messages as the global auxiliary data of the system.



(a) Block streams with borders. **(b)** Borders are aligned.

(c) Upload snapshot and propagate border. **(d)** Continue processing.

Figure 8 Epoch border alignment protocol (figure adapted from [10]).

4.2.1 Derivation Rules

The semantics of the model consists of seven rules. Three of the rules, I-Event, I-Border, and F-Fail, are local rules which enable deriving a local step of the form $\pi \Vdash \sigma \xrightarrow{X} \sigma'$. Whereas the I-Event and I-Border rules model the processing of the system, the F-Fail rule models nondeterministic crash-failures of a processing task within the system. These rules, together with the streaming rule S-Step and its abstraction rules S-AbsX and S-AbsP, are used for deriving global steps. The fourth rule, F-Recover, is a global rule used for recovering the state of all processors after a failure.

Event Rule. The first rule, I-Event, models tasks processing events:

$$\frac{f(v, w) = v', \left[W'_i\right]_i^n}{\texttt{TK}\langle\, f,\, S,\, o\,\rangle \Vdash \langle\, a,\, \langle\, e,\, v\,\rangle\,\rangle \xrightarrow{[-\, S_j\, \langle\, e, \texttt{EV}\langle\, w\,\rangle\,\rangle]:\left[+\, o\, \langle\, e, \texttt{EV}\langle\, W'_i\,\rangle\,\rangle\right]_i^n} \langle\, a,\, \langle\, e,\, v'\,\rangle\,\rangle}\ \text{I-Event}$$

The rule can perform a local step for a task $\texttt{TK}\langle\, f,\, [S_i]_i^{|S|},\, o\,\rangle$, if the current state of the task is a normal state $\langle\, e,\, v\,\rangle$, and the task can consume an event $\texttt{EV}\langle\, w\,\rangle$ from one of its inputs S_j. Applying a task's function f to its current state v and the consumed event w results in the task's next state v' and a sequence of output events $\left[W'_i\right]_i^n$. The rule updates the state of the task to the new state $\langle\, e,\, v'\,\rangle$ and produces the output events $[\texttt{EV}\langle\, W'_i\,\rangle]_i^n$ on the output stream o. The local step produces the actions which are the concatenation of the consumed and produced events. For example, $[-\, S_j\, \langle\, e, \texttt{EV}\langle\, w\,\rangle\,\rangle] : [+\, o\, \langle\, e, \texttt{EV}\langle\, w'\,\rangle\,\rangle]$ is the action of consuming the event $\texttt{EV}\langle\, w\,\rangle$ with epoch number e from the input stream S_j and producing the event $\texttt{EV}\langle\, w'\,\rangle$ with epoch number e on the output stream o.

Border Rule. Whereas the event rule consumes a single event from a stream, the border rule (I-Border) consumes one border event BD from *every incoming stream*:

$$\frac{}{\texttt{TK}\langle\, f,\, [S_i]_i^n,\, o\,\rangle \Vdash \langle\, a,\, \langle\, e,\, v\,\rangle\,\rangle \xrightarrow{[-\, S_i\, \langle\, e, \texttt{BD}\,\rangle]_i^n:[+\, o\, \langle\, e, \texttt{BD}\,\rangle]} \langle\, a\left[e \mapsto v\right],\, \langle\, e+1,\, v\,\rangle\,\rangle}\ \text{I-Border}$$

This consumption is enabled for a task if the next event to be consumed on every one of its incoming streams is a border event. In other words, the event rule consumes events up until all streams are aligned by the border events, at which point the border rule consumes

the border events from all its incoming streams. The rule is a local step which, in addition to consuming border events from all incoming streams and producing a border event on its outgoing stream, stores the current state v for epoch e to the snapshot storage a (by setting the new snapshot archive to $a[e \mapsto v]$), as well as incrementing the current epoch number.

Epochs are a key concept of Asynchronous Barrier Snapshotting. Each epoch is a sequence of data-bearing *events*, ending with an *epoch border*, and are used to define the boundaries of state snapshots. After regular processing for which some streams are blocked by border events (Figure 8a), the rule aligns the streams by the borders (Figure 8b), takes a copy of the current state of the processor storing it to the snapshot archive (Figure 8c), and propagates the epoch border message downstream and increments the epoch number, ready to process events from the next epoch (Figure 8d). The effect of this is that epochs of events are separated by the border events throughout the whole processing graph.

Failure Rule. Failures are introduced nondeterministically by the F-Fail rule:

$$\frac{}{\texttt{TK}\langle\, f, S, o \,\rangle \;\Vdash\; \langle\, a,\, \sigma_{\mathrm{V}} \,\rangle \to \langle\, a,\, \texttt{fl} \,\rangle} \text{ F-Fail}$$

The failure rule sets the task's state to failed $\langle a, \texttt{fl}\rangle$, thus losing the task's volatile state. Once a task is failed, it is no longer able to apply the steps I-Event and I-Border, and will remain idle until the F-Recover rule has been applied.

Failure Recovery Rule. The last rule, F-Recover, is a global rule which recovers the state of all failed tasks:

$$\frac{\langle\, a,\, \texttt{fl} \,\rangle \in \Sigma}{\langle\, \Pi,\, \Sigma,\, N,\, M,\, M_0 \,\rangle \Rightarrow \mathrm{lcs}(\langle\, \Pi,\, \Sigma,\, N,\, M,\, M_0 \,\rangle)} \text{ F-Recover}$$

The rule may be triggered nondeterministically if there exists a task in a failed state, and will reset the state of the system to the latest common snapshot. The full details of how the latest common snapshot (lcs) is computed is discussed further below, as it depends on additional definitions.

The latest common snapshot is constructed by: (1) calculating the greatest common epoch for which a snapshot has been taken by all processors in the system; (2) restoring the state of all processors to their local snapshots at the greatest common epoch; and (3) restoring sequence numbers and messages to undo any messages that were produced or consumed for epochs greater than the greatest common epoch. The greatest common epoch is calculated by finding the minimum (*common*) of the maximum (*greatest*) epoch numbers of the local snapshots of all the processors.

▶ **Definition 4.2** (Greatest Common Epoch Number). *The greatest common epoch number of a configuration* $c = \langle\, \Pi,\, \Sigma,\, N,\, M,\, D \,\rangle$ *is:*

$$\mathrm{gce}(c) = \min\big\{\max(\mathrm{dom}(a)) \;\big|\; \Sigma_p = \langle\, a,\, \sigma_{\mathrm{V}} \,\rangle\big\}$$

The persistent *output messages* of the system consist of all messages produced up to and including the greatest common epoch. These messages can be identified by comparing their epoch number e to the greatest common epoch number $e \leq \mathrm{gce}(c)$. The recovery purges any messages which are not part of this set, bar the initial input messages M_0, thereby making these output messages (identified by out) persistent.

(a) An execution with a failure. **(b)** Snapshot-view of the execution.

Figure 9 Executions viewed through the latest common snapshot.

▶ **Definition 4.3** (Output Messages). *For a configuration* $c = \langle \Pi, \Sigma, N, M, D \rangle$*, its output messages are:*

$$\mathrm{out}(c) = \left\{ n\, s\, \langle\, e,\, d \,\rangle \;\middle|\; (n\, s\, \langle\, e,\, d \,\rangle) \in M \wedge e \leq \mathrm{gce}(c) \right\}$$

▶ **Definition 4.4** (Messages on a Stream). *The subset* $M \downarrow s$ *of messages on a particular stream is defined as:*

$$M \downarrow s = \left\{ n'\, s'\, d' \;\middle|\; (n'\, s'\, d') \in M \wedge s' = s \right\}$$

The *lcs* function computes the latest common snapshot of a configuration for use as a recovery point in the F-Recover rule. Its computation makes use of the greatest common epoch number (gce), and the output messages (out). The states Σ' are restored by removing any stored snapshots with an epoch number larger than the gce, and the volatile states are restored to the states captured by the snapshot of the gce. The messages are updated to only keep the stable output messages $\mathrm{out}(c)$ and the messages which are yet to be consumed M_{in}. The sequence numbers N' are updated accordingly, setting the sequence number of a processor p for a stream s to the number of messages that the processor has either produced or consumed on the stream: $|\mathrm{out}(c) \downarrow s|$. Its complete definition is given below.

▶ **Definition 4.5** (Latest Common Snapshot). *The latest common snapshot of a configuration* $c = \langle \Pi, \Sigma, N, M, M_0 \rangle$ *is a configuration described by* $\mathrm{lcs}(c)$*:*

$$\mathrm{lcs}(c) = \langle\ \Pi,\ \Sigma',\ N',\ M_0 \cup \mathrm{out}(c),\ M_0\ \rangle, \textit{ where}$$

$$\begin{aligned} \Sigma' &= \left[p \mapsto \langle\, \mathrm{A}(a),\, \langle\, \mathrm{gce}(c) + 1,\, a(\mathrm{gce}(c)) \,\rangle \,\rangle \;\middle|\; \Sigma_p = \langle\, a,\, \sigma_{\mathrm{V}} \,\rangle \right] \\ \mathrm{A}(a) &= \left[e \mapsto a(e) \;\middle|\; e \in \mathrm{dom}(a) \wedge e \leq \mathrm{gce}(c) \right] \\ N' &= \left[p \mapsto \left[s \mapsto |\mathrm{out}(c) \downarrow s| \;\middle|\; s \in \mathrm{dom}(N_p) \right] \;\middle|\; p \in \mathrm{dom}(N) \right] \end{aligned}$$

Viewing computations through the lens of the latest common snapshot shows configurations which are caused by failure-free executions. Figure 9a shows an execution with a failed processor p_2 and an incompletely processed epoch (green). In contrast, the latest common snapshot view of the same execution (Figure 9b) shows only the two completed epochs (red, blue), masking the failed epoch. The snapshot is emulating an execution such that all the steps on epochs after the greatest common epoch are not taken, and all failed steps of incompletely processed epochs are ignored. This reasoning is further elaborated for the proof of failure transparency in the next section, where we show that the implementation model is failure transparent when viewed through the lens of the output messages function.

4.3 Assumptions

We make the following assumptions as a means to distill the essential mechanism of the failure recovery protocol. We assume that the message channels are FIFO ordered, a common assumption for snapshotting protocols [14]. With regard to failures, we make common

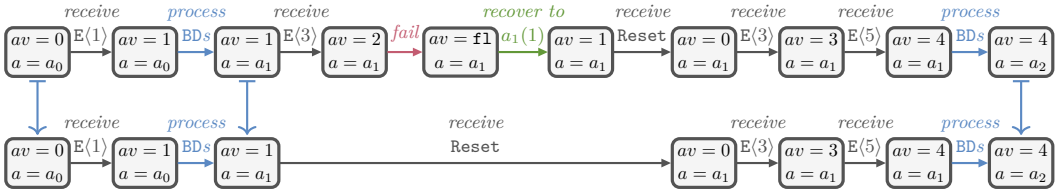


Figure 10 Execution of the incremental average task (Figure 2). Top: execution with a failure and subsequent recovery. Bottom: corresponding failure-free execution. Snapshot archives: $a_0 = \left[0 \mapsto 0\right]$, $a_1 = a_0\left[1 \mapsto 1\right]$, $a_2 = a_1\left[2 \mapsto 4\right]$.

assumptions to asynchronous distributed systems [9]. Failures are assumed to be crash-recovery failures, in which a node looses its volatile state from crashing. Further, we assume the existence of an eventually perfect failure detector, which is used for (eventually) triggering the recovery. With regard to system components, we assume the following components which can be found in production dataflow systems. The implicit coordinator instance is assumed to be failure free; in practice it is implemented using a distributed consensus protocol such as Paxos [37]. The snapshot storage is assumed to be persistent and durable; a system such as HDFS [57] would provide this. Further, the input to the dataflow graph is assumed to be logged such that it can be replayed upon failure. In practice, a durable log system such as Kafka [32] would be used for this. For our model, we make the following assumptions. The recovery is assumed to be an atomic, synchronous system-wide step. In practice, it may be implemented as an asynchronous atomic step, which allows tasks to start processing before all have been recovered. Further, the task's processing functions are assumed to be pure, *i.e.,* free from side effects. A function f may be re-executed multiple times due to failures; a common assumption in related work [8, 30].

5 Failure Transparency

In this section, we define failure transparency such that it can be applied to systems described in small-step operational semantics with distinct failure-related rules. We first provide a rationale behind failure transparency, followed by its formalization.

5.1 Rationale

The purpose of failure transparency is to provide an abstraction of a system which hides the internals of failures and failure recovery. In particular, we would like to be able to show that the implementation model presented in the previous section is failure transparent. In concrete terms, this entails showing that executions in the implementation model can be "explained" by failure-free executions, something which we explore in this section.

Consider the task of computing the incremental average from the previous example (Section 3, Figure 2). The task consumes regular events $\texttt{E}\langle i\rangle$, reset events, and border events `BD`. For this example, we consider a partial execution of the task in which it processes the events: $[\texttt{E}\langle 1\rangle, \texttt{BD}, \texttt{E}\langle 3\rangle, \text{fail}, \text{recover}, \texttt{Reset}, \texttt{E}\langle 3\rangle, \texttt{E}\langle 5\rangle, \texttt{BD}, \ldots]$. The task's configurations consist of the task's current average value av, and its snapshot archive, a. Figure 10 shows at the top an execution of the task with a failure and subsequent failure recovery as the fourth and fifth events. After the recovery step, in its sixth configuration, the task's state is reset to its state for the snapshot $a_1(1)$, at which point it had the average value 1.

The question we ask is whether we can rely on the behavior of the task? More specifically, can we use the average value $av = 2$ in the fourth configuration (after receiving the event $\mathtt{E}\langle 3\rangle$)? The problem is that the task will fail in its next step, and recover to a state in which the receiving of the event has been undone. Moreover, the task continues its execution after recovery by processing the reset event first, and does never reach a state again in which its average value is 2. For this reason, we cannot blindly rely on the observed behaviors of the task as we may observe things which are later undone. In more complex systems, failures may further result in duplications and reorderings of events, further complicating the reasoning about the system.

Dealing with these issues requires the observer of the system to reason about which events are effectful and which are to be discarded. In some sense, the observer should be able to reason about the observed execution as if it was an ideal, failure-free execution, *i.e.,* an execution in which all events are effectful. Put in another way, the solution is to find a corresponding failure-free execution, and reason about that one instead. Intuitively, the observer should find some failure-free execution which "explains" the execution. Considering the above example, a failure-free execution thereof would correspond to the bottom execution in Figure 10. Note that there are no failure or recovery steps in the failure-free execution, yet its state progresses in a similar way to the original execution.

Even though the failure-free execution on an intuitive level correspond to the original execution, we would like to have a formal notion for this. The idea is to lift the observed executions by means of "observability functions", to a level where failure-related events and states are hidden. For example, for the executions above, we could define an observability function which takes the configuration of the task and keeps only the snapshot storage. After this transformation, applying this function to every configuration in the executions, we will not be able to distinguish the two executions by observing the system at any point in time. That is, common to both executions, we will first observe a_0, then a_1, and finally a_2. On a technical level, for every configuration of the original execution, we can find a configuration in the failure-free execution which, after application of the observability functions, is equal to it (*e.g.,* the mapping from top to bottom configurations in Figure 10); this is what we mean by "observable explainability". Thus, we can explain the original execution by the failure-free execution using the provided observability function.

The essence of our definition of failure transparency is derived from the notion of explaining the original executions by failure-free executions using observability functions. Instead of reasoning about executions, we can reason about the observable output of executions at any given moment. Using observability functions effectively hides the internals of the model and enables the user to focus on the output of the system. That is, the user can reason about failure-free executions instead of faulty executions.

This informal introduction highlights three essential parts of failure transparency: the execution system, failures within the system, and the observability of the system. The goal of the rest of this section is to define these terms and to provide a formal definition of *failure transparency*.

5.2 Executions

The execution system for the failure transparency analysis is modelled as a transition system for which the transition relation is provided as a set of inference rules. In particular, we provide a formal definition for executions as a means to discuss the execution of systems. With this notion, distributed programs can be formally modelled in small-step operational

semantics, and consequently formally verified. Although it may seem unintuitive to model distributed systems as transition systems for which the transition relation is defined over the global state, this is in fact commonly done in other formal frameworks such as TLA$^+$ [38].

▶ **Definition 5.1** (Execution Step). *A statement $c \Rightarrow c'$ is called an execution step from c to c'. We denote the derivability of an execution step in the set of rules R by $R \vdash c \Rightarrow c'$.*

We reason about systems in terms of their executions. An execution is a sequence of configurations C, connected by execution steps derivable in a set of rules R, starting from some initial configuration C_0.

▶ **Definition 5.2** (Executions). *A sequence of configurations $\left[C_i\right]_i^n$ is called an* execution *in a set of rules R, if $\forall i < n.\ R \vdash C_{i-1} \Rightarrow C_i$. The set of all possible executions starting from C_0 in R is denoted as $\mathbb{E}^R_{C_0}$.*

The set of rules R of an execution specifies its reducibility relation by providing $c \Rightarrow c'$ as a conclusion of some of its rules. This approach is commonly known as *small-step operational semantics*. In our representation, the set of rules is explicit, whereas commonly it is implicit. This is due to our need to explicitly distinguish between separate execution systems. This allows us, for example, to separate an execution system into two parts: one with failures R *s.t.* the failure-related rules are a subset thereof $F \subseteq R$, and one without failures $(R \setminus F)$.

5.3 Observational Explainability

The observability function represents the observer's view of the system. It notably differs from the plain configurations in the following two ways: the observer may not observe all internal details of configurations, *i.e.,* some parts of the configuration are *hidden* from the observer (*e.g.,* hiding commit messages [8]); and the observer may observe some derived views of the configuration.

▶ **Definition 5.3** (Observability Function). *An* observability function *O of an execution system is a function which maps configurations to their observable outputs. It is required to be monotonic with respect to execution steps possible in the set of rules R for some partial order $\sqsubseteq_O$, that is: $\forall c, c'.\ (R \vdash c \Rightarrow c') \implies O(c) \sqsubseteq_O O(c')$.*

We say that an implementation's execution is observably explained by a specification's execution, if the observer cannot distinguish the two executions. This is the case when, for every configuration in the implementation's execution, there is a corresponding configuration in the specification's execution, such that their observed values are equal after application of the respective observability functions.

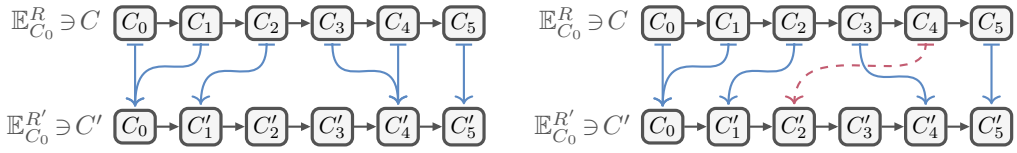
▶ **Definition 5.4** (Observational Explanation). *A sequence of configurations C of length n is explained by a sequence of configurations C' of length n' with respect to observability functions O and O', denoted as $C \ {}^{O}\!\rightleftharpoons^{O'} C'$, if:*

$$\forall m < n.\ \exists m' < n'.\ O(C_m) = O'(C'_{m'})$$

An implementation's system, in turn, is observably explainable by the specification's system, if for each execution of the implementation there exists an explaining execution in the specification. We call this property *observational explainability*.

▶ **Definition 5.5** (Observational Explainability). *The set of rules R is* observationally explainable *by R' with respect to their observability functions O and O' and the translation relation T, denoted as $R \ {}^{O}\!\overset{T}{\rightleftharpoons}^{O'} R'$, if:*

$$\forall\, c' \in \mathrm{dom}(T).\ \forall c.\ c' T c \implies \forall C \in \mathbb{E}^R_c.\ \exists\, C' \in \mathbb{E}^{R'}_{c'}.\ C \ {}^{O}\!\rightleftharpoons^{O'} C'$$



(a) Monotonic mapping. **(b)** Non-monotonic mapping.

Figure 11 Monotonic and non-monotonic mapping of configurations.

Properties of Observational Explainability. Observability functions are required to be monotonic, since observations should be regarded as stable. That is, once a value has been observed, then it should remain observable in the future. The system should not be able to undo something that has been observed, otherwise the observer would not be able to rely on the output. The reason for this is twofold. First, an observer may observe the system multiple times, and newer observations should provide more up-to-date views. Second, the sequence of observations should correspond to a valid explanation with respect to the higher-level specification, this is explored next.

In the general case, it is desirable to have a monotonic mapping of configurations between the abstract-level and implementation-level executions. Figure 11a shows a monotonic mapping of configurations between an implementation (top) and a specification (bottom). What makes the mapping monotonic is that each subsequently mapped configuration of the implementation is mapped to a configuration with a monotonically growing index. Figure 11b, on the other hand, shows a non-monotonic mapping, as indicated by the red dashed line. Non-monotonic mappings, however, are not considered valid explanations. For example, if the specification consists of the sequence a followed by b, then an implementation which produces b followed by a is not considered a valid implementation thereof. Thus, we should not use non-monotonic mappings for the explainability of executions. We capture this notion in the definition of monotonic observational explanation.

▶ **Definition 5.6** (Monotonic Observational Explanation)**.** *An observational explanation is* monotonic *if it is a monotonic mapping of configurations. That is,* $[C_i]_i^n$ *is* monotonically *explained by* $[C'_j]_j^{n'}$ *w.r.t.* O *and* O' *if:*

$$\exists \big[h_k\big]_k^n.\ (\forall k < n.\ \forall k' \leq k.\ h_{k'} \leq h_k)\ \wedge\ (\forall m < n.\ \exists m' = h_m < n'.\ O(C_m) = O'(C'_{m'}))$$

The following lemma explicitly shows that our definition of *observational explainability* is equivalent to the definition of *monotonic observational explainability*. That is, our definition does not have the problem with non-monotonic mappings of configurations since the observability functions are required to be monotonic. For this reason, we do not distinguish between the two definitions in the following sections.

▶ **Lemma 5.7.** *If* R *is observationally explainable by* R' *w.r.t.* O, O', T, *then it is also* monotonically *observationally explainable:*

$$\forall\, c' \in \mathrm{dom}(T).\ \forall c.\ c'Tc \implies \forall C \in \mathbb{E}^R_c.\ \exists C' \in \mathbb{E}^{R'}_{c'}.$$
$$C \text{ is monotonically explained by } C' \text{ w.r.t. } O \text{ and } O'$$

Proof. The complete proof is available in the companion technical report [62]. ◀

To further aid the use of these definitions within proofs, we also show that the definition of observational explainability is transitive, as well as a compositionality lemma on the observability functions. The parametrization of the observable explainability enables reasoning

about models which differ in their initial states, and for which we want to apply different observability functions at the different levels. That is, it can be used for reasoning about sets of rules which differ in their initial states, and for which we want to apply different observability functions at the different levels.

▶ **Lemma 5.8** (Transitivity). $R \overset{O}{\underset{\leftarrow}{\overset{T}{\rightleftharpoons}}} O' R' \wedge R' \overset{O'}{\underset{\leftarrow}{\overset{T'}{\rightleftharpoons}}} O'' R'' \implies R \overset{O}{\underset{\leftarrow}{\overset{T \circ T'}{\rightleftharpoons}}} O'' R''$

Proof. The complete proof is available in the companion technical report [62]. ◀

▶ **Lemma 5.9** (Composition). $\forall O''.\ R \overset{O}{\underset{\leftarrow}{\overset{T}{\rightleftharpoons}}} O' R' \implies R \overset{O'' \circ O}{\underset{\leftarrow}{\overset{T}{\rightleftharpoons}}} O'' \circ O' R'$

Proof. The complete proof is available in the companion technical report [62]. ◀

5.4 Defining Failure Transparency

The general goal of failure transparency is to provide an abstraction of a system which masks failures from the users. We express this notion using observational explainability between the implementation and its failure-free part. That is, the implementation should be observationally explainable by the implementation without failures. By explicitly separating the set of failure-related rules F, it is easy to define the two systems: namely, the implementation system with all rules, *i.e.,* R; and another system with all rules except the failure-related rules, *i.e.,* $R \setminus F$. To fully instantiate the observational equivalence, we further use the same observability function O on both the low and high levels, and as a translation relation we use the identity relation on the set of initial configurations.

▶ **Definition 5.10** (Failure Transparency). *A set of rules R is* failure-transparent *with respect to failure rules $F \subseteq R$ for a monotonic observability function O and a set of initial configurations K, this is denoted as $R \lVert\!\lVert^{O}_{K} F$, iff:*

$$R \overset{O}{\underset{\leftarrow}{\overset{\{(c,c) \mid c \in K\}}{\Longrightarrow}}} O (R \setminus F)$$

6 Failure Transparency of Stateful Dataflow

In this section, we show that the presented implementation model (Section 4) is failure transparent (Definition 5.10) for the observability function out (Definition 4.3). In order to prove this, instead of reasoning about executions directly, we reason about the traces of steps which are performed to obtain these executions. This simplifies the proof, enabling us to reorder and remove specific steps in and from a trace; in contrast, doing the same with a configuration from an execution affects all following configurations. In this section, we first define traces and a causal order relation on traces, and then prove the failure transparency of the implementation model by manipulating traces. Finally, we complete our analysis of the model by formulating and proving its liveness, showing that the implementation model eventually produces outputs for all epochs in its input.

6.1 Traces and Causality

A trace is a sequence of steps, for which each step is a compact representation of the derivation of a transition from one configuration to another.

▶ **Definition 6.1** (Trace). *A trace Z is a sequence of trace steps. A trace step z is one of:* $\langle$I-Event$, p, N_p, X\rangle$*;* $\langle$I-Border$, p, N_p, X\rangle$*;* $\langle$F-Fail$, p\rangle$*;* $\langle$F-Recover$\rangle$*. Here* I-Event*,* I-Border*,* F-Fail*, and* F-Recover *play the role of the discriminant, where the trace step is a tagged union.*

For example, if in the derivation tree of an execution step from the ith to the $i+1$th configuration, *i.e.,* of $R \vdash C_i \Rightarrow C_{i+1}$, F-Recover was the root rule, then this execution step corresponds to the step $\langle$F-Recover$\rangle$ in the trace. To link traces with executions, we use the following definition of trace application.

▶ **Definition 6.2** (Trace Application). *A trace Z of length n applied to a configuration c results in a sequence of configurations C of length $n+1$,* i.e., *$Z(c) = C$, if, for all steps Z_i, the represented derivation of an execution step can be applied to the ith configuration producing the $i+1$th configuration.*

Traces can be generated from executions; however, not every trace corresponds to an execution. This may be the case if a trace has been constructed incorrectly, or reordered in some way. For this reason, we define valid traces, which are traces that correspond to executions.

▶ **Definition 6.3** (Valid Trace). *A trace Z is valid from configuration c if it is applicable to it,* i.e., *if there exists an execution $C \in \mathbb{E}_c^I$ such that $Z(c) = C$.*

As the proof reasons about the reordering of steps in a trace, it is important to formulate which reorderings of steps preserve the validity of the trace. To handle this, we define a causal order relation on trace steps similar to the happens-before relation [35], and show how it can be used to reason about traces.

▶ **Definition 6.4** (Causal Order). *(See technical report [62] for the formal definition) A step Z_i happens before Z_j with $i < j$ if:*

1. *One of them is an* F-Recover *step (global recovery)*
2. *They both occur on the same processor (intraprocessor order)*
3. *If Z_i produced a message which is consumed by Z_j (interprocessor order)*
4. *If there exists some step Z_k such that Z_i happens before Z_k and Z_k happens before Z_j (transitivity)*

Finally, we state a lemma that causality-preserving permutations, *i.e.,* permutations that preserve the causal order relation [62, Definition B.5], also preserve the validity and the end result of their application. Intuitively, it follows from the fact that causally unrelated steps should not influence each other.

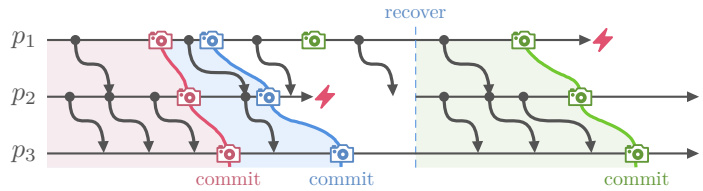
▶ **Lemma 6.5** (Application of Causality-Preserving Permutations). *For a trace Z valid from c with size $|Z| = n$, if Z' is a causality-preserving permutation of Z, then: Z' is valid from c; Z and Z' end in the same configuration after application to c,* i.e., *$Z(c)_n = Z'(c)_n$.*

Proof. The complete proof is available in the companion technical report [62]. ◀

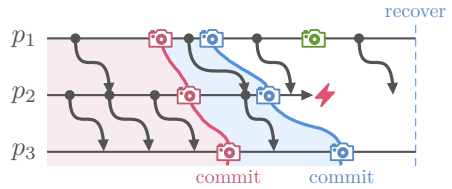
6.2 Proving Failure Transparency

As it is required by the definition of failure transparency, we first define the sets of rules, namely I, F, and (I $\setminus$ F); and the set of valid initial configurations K.

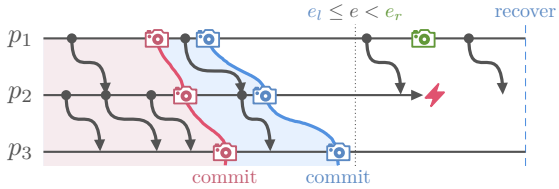
The semantics of the model consist of seven rules, defining two separate sets of rules. The set of rules with failures I consists of all seven rules that have been defined for the stateful dataflow implementation model; it corresponds to the implementation model presented in



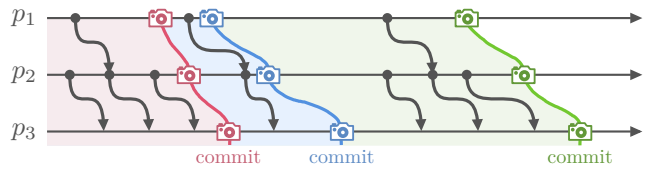
(a) Original faulty execution.



(b) Separated generations.



(c) Reordered generations.



(d) Merged failure-free execution.

Figure 12 The step-wise construction of a failure-free execution trace from an execution with failures.

Section 4. The set of failure-related rules F within the implementation model consists of the two rules F-Fail and F-Recover. This way, the rules without failures are defined as the set $(\mathrm{I} \setminus \mathrm{F})$.

▶ **Definition 6.6** (Implementation Model Rules). $\mathrm{I} = \{\text{S-Step}, \text{S-AbsX}, \text{S-AbsP}, \text{I-Event}, \text{I-Border}\} \cup \mathrm{F}$

▶ **Definition 6.7** (Failure-Related Rules). $\mathrm{F} = \{\text{F-Fail}, \text{F-Recover}\}$

The sets of initial configurations which are considered are any acyclic graph structures which are properly initialized.

▶ **Definition 6.8** (Valid Initial Configurations). *$K = \langle\, \Pi,\, \Sigma,\, N,\, M,\, M_0 \,\rangle$ such that: the graph defined by Π is acyclic, and the tasks' functions f do not output infinite sequences; Σ are the initial well-formed states; N are sequence numbers initialized to 0 for the streams; M consists of the well-formed inputs to the streams; $M_0 = M$.*

▶ **Theorem 6.9** (Failure Transparency of the Implementation Model). *$I \lessdot\!\!\!\cdot^{\text{out}}_{K} F$,* i.e., *the set of rules $\mathrm{I} = \{\text{S-Step}, \text{S-AbsX}, \text{S-AbsP}, \text{I-Event}, \text{I-Border}\} \cup \mathrm{F}$ is failure transparent with respect to the failure rules $\mathrm{F} = \{\text{F-Fail}, \text{F-Recover}\}$ for the observability function* out *and the set of initial configurations K.*

Before proceeding with the proof itself, we provide a sketch of it. The proof idea is to construct a failure-free observational explanation of an arbitrary execution in the implementation model.

The construction is done using traces; we reorder and manipulate the original trace so that failures, recoveries, and discarded trace steps are removed from it. Figure 12 illustrates the construction: (1) first, we split the trace by the recovery steps into generations; (2)

next, the trace steps are reordered such that all discarded steps are moved to the end of the generation; (3) then, these steps are safely discarded; (4) finally, we concatenate the generations to get the final trace.

Next, we have to show that: (i) the constructed trace is valid, *i.e.,* it corresponds to a failure-free execution; and (ii) that the execution is an observational explanation of the original execution. We do so by reasoning about the preservation of validity and observable outputs in each step of the construction. For trace validity, the most complicated step is the reordering (step 2 of the construction). We show that the reordering is causality-preserving and thus, by Lemma 6.5, it produces a valid trace. For observational explanation, throughout the construction we maintain a mapping of observations from the steps of the original trace to the steps of the constructed trace. The challenge lies in the reordering of steps (step 2 of the construction) and the fusion of generations (step 4 of the construction). For the reordering, we show a lemma that the observable output is not changed by the discarded steps; and, for the fusion, we show that the latest common snapshot of a generation is exactly the configuration obtained by the reordering and removal of the discarded steps. This, accompanied by an analysis of the rules, lets us show that the sequence of observable outputs is the same for the original and the failure-free traces.

Proof. Expanding the definitions, we need to prove that, for all executions in I with potential failures, there is an observational explanation in the failure-free model ($\mathrm{I} \setminus \mathrm{F}$). Given an arbitrary execution C of length n in I from initial configuration $c \in K$, *i.e.,* $\left[C_i\right]_i^n \in \mathbb{E}_c^{\mathrm{I}}$, the goal is to construct a failure-free execution $\left[C'_j\right]_j^{n'}$ such that:

$$\left[C'_j\right]_j^{n'} \in \mathbb{E}_c^{\mathrm{I}\setminus\mathrm{F}} \wedge \forall m < n.\ \exists m' < n'.\ \mathrm{out}(C_m) = \mathrm{out}(C'_{m'})$$

This execution is constructed indirectly, by first constructing a trace Z' which then generates it. First, we need to prove that the constructed failure-free trace Z' is valid from c, *i.e.,* $Z(c)$; next, we need to show that the corresponding execution $C' = Z'(c)$ is an observational explanation of the original execution, that is, for each configuration in the original execution, we have to provide an observationally equal configuration in the constructed execution. From the original trace Z, for which $Z(c) = C$, we construct the failure-free trace Z' in four steps as outlined in the proof sketch and illustrated in Figure 12.

(1) First, the trace is split by the recovery steps into generations, giving us a sequence of generations G (Figure 12b). Each generation is a sequence of S-Steps ending with an F-Recover step; in the case of the last generation it may not necessarily end with an F-Recover step. By construction, each generation is a valid trace as each of them is a contiguous part of a valid trace. We construct the observability mapping by mapping the configurations of the original trace to their closest preceding committing border steps. A *committing border step* is an I-Border step which changes the greatest common epoch number, gce, and thus also the observed output, out; such steps are labeled with "commit" in Figure 12. The equality of observations holds, since, by inspection of the rules, only a committing border step can change the observable output [62, Lemma B.6].

(2) Next, from each generation $g = G_i$, we construct a new reordered trace $g' = G'_i$ so that all the steps of epochs above the greatest common epoch of the generation are placed after the steps of epochs below it (Figure 12c). In effect, this moves all the discarded steps to the end of the generation, since they are discarded by the recovery, which in turn is done to the greatest common epoch of the generation. In other words, $g' = \mathrm{filter}(x \in g.\ \mathrm{epoch}(x) \leq e) : \mathrm{filter}(x \in g.\ \mathrm{epoch}(x) > e)$, where $e = \mathrm{gce}(g_{|g|-1})$ is the epoch number to which the recovery is done. The new traces are still valid, as the reordering is causality preserving [62,

Lemma B.7], and thus the validity follows from Lemma 6.5. The mapping of observations is kept intact, since the outputs of the committing border steps are not changed by the reordering [62, Lemma B.8]. This follows from the fact that the observable output is only changed by committing border steps [62, Lemma B.6], that causality-preserving permutations result in the same configuration (Lemma 6.5), and that the reordering is preserving causality.

(3) Then, from each G'_i we construct a new trace G''_i by removing the discarded steps and the recovery step. That is, the suffix consisting of the failure steps, recovery steps, and any steps of epochs greater than the greatest common epoch of the generation are removed. The new trace is a prefix of G'_i, and is thus still a valid trace [62, Lemma B.2]. We keep the same mapping of observations for the steps that were not removed. As, within a generation, only the suffix is removed, it does not affect the observed outputs of the remaining steps, and thus the mapping of observations is kept unchanged.

(4) Finally, we concatenate all stripped generations G''_i to get the merged trace Z' (Figure 12d). We show that the last configuration of each of the generations G''_i is exactly the latest common snapshot of the original generation G_i [62, Lemmas B.10-11], in other words, the latest common snapshot is a view of a configuration as if only the committed steps occurred. Since the recovery is done to the latest common snapshot, it is also the same configuration as the first configuration of the following generation G''_{i+1}. For this reason, the concatenation of all generations forms a trace Z' valid from c. The observed outputs are not changed by the merge, and we maintain the same mapping.

By these four steps we have constructed a failure-free observational explanation of the faulty execution, which means that the implementation model is observationally explainable (Definition 5.5) by its failure-free version, or, in other words, it is failure transparent (Definition 5.10). ◀

6.3 Liveness

The proposed definition of failure transparency is a safety property [3, 34], *i.e.,* it prohibits the implementation from reaching invalid states. Being as such, failure transparency does not require the implementation to take any observable execution steps; an implementation that never takes a step would trivially satisfy the property. In contrast, ensuring that the implementation eventually does something is a *liveness* property [3, 34]. To complete our analysis, we would like to show that the implementation model eventually produces outputs for all epochs in its input. This is a liveness property which, consequently, does not concern itself with the correctness of the outputs. However, in combination with the failure transparency property, the properties ensure that the presented implementation model *eventually* produces the *correct* outputs. For this reason, we prove the following theorem about the liveness of the implementation model.

▶ **Theorem 6.10** (Liveness of the Implementation Model). *For every input epoch present in the initial configuration, eventually a corresponding epoch appears in the output of a fair execution. That is:*

$$\forall k = \langle \Pi, \Sigma, M, N, D \rangle \in K.\ \forall C \in \mathbb{E}^{\mathrm{I}}_k.\ \mathrm{fair}(C) \implies$$
$$\forall (n\ s\ \langle e, d \rangle) \in M.\ \exists c \in C.\ \exists (n'\ s'\ \langle e', d' \rangle) \in \mathrm{out}(c).\ e = e'$$

where a fair *execution is maximal (*i.e., *it is not a prefix of another execution), has a finite amount of failures, and eventually executes any step which is eventually always enabled (see the technical report [62] for the formal definition of* fair *execution).* ⌟

The liveness theorem states that for any fair execution C starting from a valid initial configuration k, and for all input epochs e, eventually there is a configuration c in the execution for which the output $\text{out}(c)$ contains the epoch e.

Proof. The complete proof is available in the companion technical report [62], it is summarized as follows. First, we show that it suffices to demonstrate that, continuing from any configuration c reachable from the valid initial configuration k, one or both of the following are true: eventually there is a failure; or eventually the epoch is visible in the output. As the considered executions have only finite amounts of failures, we further simplify the proof goal: it suffices to show that eventually the epoch appears in the output under the assumption that there are no more failures. We handle this simplified case by inductive reasoning on the acyclic dataflow graph of processors. The induction's base case is the graph consisting of the source input streams but with no processors. The induction hypothesis states that all streams are well-formed and that the border message of all input epochs eventually appear on all streams; this is satisfied for the base case by validity of the initial configuration. Then, in the induction step, we construct the graph by adding one processor at a time, given that all of its input streams are already handled, as either source inputs or as outputs of other processors in the previous step's graph. The assumption of fair scheduling allows us to reason about the processor locally, since, by definition of fairness, if a message has arrived to the processor, it will eventually be consumed. As a conclusion of the induction, each processor will eventually have processed a border of each epoch present in the initial configuration; thus, eventually all processors will process a border of each initial epoch. This, in turn, by analysis of I-Border, gce, and out, shows that the border messages of the epoch will eventually be in the output. ◀

7 Related Work

Failure Transparency, Observational Explainability. There has been a significant body of research on failure transparency [40]. To our knowledge, the earliest work on failure transparency was by von Neumann in 1956 [63] on creating reliable systems from unreliable components. Later work by Wensley in 1972 [65] discussed software techniques for failure transparent computing. Lowell and Chen discussed failure transparency in the context of consistent failure recovery protocols [44]. In their work, they introduced "equivalence functions" for comparing executions, a concept which inspired the observability functions in this paper. Our work, in contrast, restricts these functions to be monotonic, and discusses their application to both levels (low and high) of the system, which facilitates the presented transitivity lemma (Lemma 5.8). Around the same time as Lowell and Chen, Gärtner discussed general models for fault-tolerant computing [26]. Similar to our work, Gärtner separated fault-tolerant programs into two separate sets of rules (actions): the rules for normal behavior; and the rules for failure behavior. With this separation, Gärtner discussed various properties and forms of fault-tolerant programs. In the context of Gärtner's work, our definition of failure transparency would be considered "failure masking", in the sense that the system can recover from failures and continue its normal operation. Whereas these works defined failure transparency as a conjunct of *safety and liveness* [34, 3], we have only considered its safety property for our definition.

The presented definition for observational explainability is closely related to previous definitions of refinement (*e.g.,* TLA [38, 36], Compiler Correctness [53]), implementation (*e.g.,* I/O Automata [45]), and simulation. In simplified terms, one set of executions implements another if it is a subset thereof (modulo stuttering and multistep executions).

Our definition of observational explainability, in some sense, extends the notion of refinement to directly include a refinement mapping [1] on both sides via observability functions. It resembles notions from related work such as observational equivalence [8] and observational refinement [30]; in contrast to these works, we provide a formal definition thereof. Different from inductive proof approaches as typical for TLA [38] and simulation proof strategies, our proof approach reasons about the whole sequence. This makes it not necessary to include notions for ghost variables [49] (also known as auxiliary variables [39]) for the purpose of reasoning about past or future events.

Failure Transparency Proofs. Failure transparency and observational explainability can be proven in various ways. For example, Burckhardt et al. [8] prove "observational equivalence" for their serverless programming model. Mukherjee et al. [51] propose a failure transparency theorem for their system of reliable state machines: an execution of the implementation is a refinement of an execution without failures "with respect to its observable behavior", reminiscent of our definition of failure transparency. Other works include models for distributed reliable actor communication [61], serverless microservices and observational refinement [30], and reliable state machines [51]. Their specific approaches may differ, some use simulation [8, 30], others model failures explicitly [30, 61, 51], and others use notions similar to observability functions [8]. Another approach is to prove the proper restoration of applications to the exact configuration as before the crash [50]. Our presented failure transparency proof shares similarities to the proof of the Asynchronous Barrier Snapshotting protocol [10], such as reasoning about causal orderings; however, our proof relies to a greater degree on abstraction in terms of refinement of models.

Distributed, Resilient Programming Models. Stateful dataflow has had a high impact [24] through systems such as: MapReduce [19], Apache Spark [67, 66], Apache Flink [12], Google Dataflow [2], IBM Streams [18], Portals [60], and others [7, 56]. However, there are other notable resilient programming models and systems, including: Pregel, a graph-based system [47], Resilient X10 [17], virtually resilient immortals [27], fault-tolerant reactives [50], thread-safe reactive programming [20], Durable Functions [8], stateful entities [54], the eXchange Calculus [6], and others [61, 30, 51, 15]. In general, these resilient programming models provide system means to recover from failures, the user does not need to implement the failure recovery mechanisms themselves. Actor models, in contrast, provide the users with manual failure-handling constructs. For example, the failure-handling constructs in Erlang, such as actor monitors and supervision [5], have been used successfully for building reliable services within the telecom industry [4]. Moreover, other programming models such as Argus [42] and transactors [22] provide constructs for transactions, which in turn can be used for building reliable services.

The formalization of distributed systems has been a long-standing research topic. Notably, formalization frameworks such as TLA [38] and I/O Automata [45], have been used to reason about distributed systems. Examples of this include a dataflow system that was formalized using I/O Automata [46]. The ABS protocol for stateful dataflow has been formalized with transition systems [10]. Recently, operational semantics have been used to model and reason about such systems [8, 61, 30, 51, 28].

Failure Recovery. A general overview of rollback-recovery protocols was given by Elnozahy et al. [21], comparing between checkpointing-based and logging-based protocols. Stateful dataflow systems use either checkpointing, or a combination of the two [7, 56, 2, 12, 64, 66,

19, 18]. The MapReduce system performs failure recovery by detecting failed nodes, and replaying the computation from sources or from persisted intermediate results [19]. Apache Spark, in contrast, improves the recovery by replaying from the sources through what is called lineage recovery [66]. A similar idea is used in a dynamic dataflow system within Ray [64]. This paper focused on the ABS protocol used in Apache Flink, which, in contrast to previous works, uses an asynchronous checkpointing technique [12]. It has been proven to provide high performance and has since been widely adopted [58]. The current version of Apache Flink's runtime offers an opt-in feature for "unaligned checkpoints", which allow the checkpoint markers to be treated at a higher priority, decreasing the end-to-end latency at the cost of some overhead as buffered events may become part of the snapshots [23]. Other adaptations of the Flink protocol include Clonos [59], which logs the nondeterminism to facilitate faster partial recovery after failures. Failure recovery remains an open research topic, as it has great impact on the performance characteristics of fault-tolerant systems [58].

8 Conclusions and Future Work

This paper studies failure transparency of stateful dataflow systems. We propose a novel definition of failure transparency for programming models expressed in small-step operational semantics. For the definition of failure transparency we introduce observational explainability, a notion which resembles refinement but on the level of observations of executions. We provide an implementation model of a stateful dataflow system using the Asynchronous Barrier Snapshotting protocol in a small-step operational semantics, and prove that the model is failure transparent and guarantees liveness.

In future work, we plan to implement a fully verified implementation of a stateful dataflow system based on the semantics presented in this paper, starting from our Coq mechanization. Furthermore, we would like to apply our definitions to existing related work.

References

1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.

2 Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015. doi:10.14778/2824032.2824076.

3 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:10.1016/0020-0190(85)90056-0.

4 Joe Armstrong. Erlang–a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, pages 16–18, 1996.

5 Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.

6 Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. The exchange calculus (XC): A functional programming language design for distributed collective systems. *J. Syst. Softw.*, 210:111976, 2024. doi:10.1016/J.JSS.2024.111976.

7 Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 13–24. ACM, 2005. doi:10.1145/1066157.1066160.

8 Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021. doi:10.1145/3485510.

9 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.

10 Paris Carbone. *Scalable and Reliable Data Stream Processing*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2018. URL: https://nbn-resolving.org/urn:nbn:se:kth:diva-233527.

11 Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 2017. URL: http://www.vldb.org/pvldb/vol10/p1718-carbone.pdf, doi:10.14778/3137765.3137777.

12 Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015. arXiv:1506.08603.

13 Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL: http://sites.computer.org/debull/A15dec/p28.pdf.

14 K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. doi:10.1145/214451.214456.

15 Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. New directions in cloud programming. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf.

16 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, 2017. doi:10.1145/3110268.

17 David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay A. Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: efficient failure-aware programming. In José E. Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 67–80. ACM, 2014. doi:10.1145/2555243.2555248.

18 Gabriela Jacques da Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sariyüce. Consistent regions: Guaranteed tuple processing in IBM streams. *Proc. VLDB Endow.*, 9(13):1341–1352, 2016. doi:10.14778/3007263.3007272.

19 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

20 Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proc. ACM Program. Lang.*, 2(OOPSLA):107:1–107:30, 2018. doi:10.1145/3276477.

21 E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002. doi:10.1145/568522.568525.

22 John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 195–208. ACM, 2005. doi:10.1145/1040305.1040322.

23 The Apache Software Foundation. Unaligned checkpoints flip-76. https://issues.apache.org/jira/browse/FLINK-14551, 2020. Accessed on 2024-03-28.

24 Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *VLDB J.*, 33(2):507–541, 2024. doi:10.1007/S00778-023-00819-8.

25 Yupeng Fu and Chinmay Soman. Real-time data infrastructure at Uber. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2503–2516. ACM, 2021. doi:10.1145/3448016.3457552.

26 Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999. doi:10.1145/311531.311532.

27 Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, 2020. doi:10.14778/3377369.3377370.

28 Philipp Haller, Heather Miller, and Normen Müller. A programming model and foundation for lineage-based distributed computation. *J. Funct. Program.*, 28:e7, 2018. doi:10.1017/S0956796818000035.

29 Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir A. Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel CoreTM i7 processor execution engine validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2009. doi:10.1007/978-3-642-02658-4_32.

30 Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. *Proc. ACM Program. Lang.*, 7(POPL):367–395, 2023. doi:10.1145/3571206.

31 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

32 Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

33 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

34 Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.

35 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

36 Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. doi:10.1145/177492.177726.

37 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

38 Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. URL: http://research.microsoft.com/users/lamport/tla/book.html.

39 Leslie Lamport and Stephan Merz. Auxiliary variables in TLA+. *CoRR*, abs/1703.05121, 2017. arXiv:1703.05121.

40 Peter Alan Lee and Thomas Anderson. *Fault Tolerance*, pages 51–77. Springer Vienna, Vienna, 1990. doi:10.1007/978-3-7091-8990-0_3.

41 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

42 Barbara Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988. doi:10.1145/42392.42399.

43 David E. Lowell. *Theory and practice of failure transparency*. PhD thesis, University of Michigan, USA, 1999. URL: https://hdl.handle.net/2027.42/132190.

44 David E. Lowell and Peter M. Chen. The theory and practice of failure transparency. Technical report, University of Michigan, 1999.

45 Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989. Also available as MIT Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.

46 Nancy A. Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. *Inf. Comput.*, 82(1):81–92, 1989. doi:10.1016/0890-5401(89)90066-7.

47 Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010. doi:10.1145/1807167.1807184.

48 Yancan Mao, Zhanghao Chen, Yifan Zhang, Meng Wang, Yong Fang, Guanghui Zhang, Rui Shi, and Richard T. B. Ma. StreamOps: Cloud-native runtime management for streaming services in ByteDance. *Proc. VLDB Endow.*, 16(12):3501–3514, 2023. doi:10.14778/3611540.3611543.

49 Monica Marcus and Amir Pnueli. Using ghost variables to prove refinement. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany, July 1-5, 1996, Proceedings*, volume 1101 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 1996. doi:10.1007/BFB0014319.

50 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. doi:10.1145/3360570.

51 Suvam Mukherjee, Nitin John Raj, Krishnan Govindraj, Pantazis Deligiannis, Chandramouleswaran Ravichandran, Akash Lal, Aseem Rastogi, and Raja Krishnaswamy. Reliable state machines: A framework for programming reliable cloud services. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPIcs*, pages 18:1–18:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ECOOP.2019.18.

52 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455. ACM, 2013. doi:10.1145/2517349.2522738.

53 Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP):85:1–85:29, 2019. doi:10.1145/3341689.

54 Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragkoulis, Guido Salvaneschi, and Asterios Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. In Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani, editors, *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, pages 15–21. OpenProceedings.org, 2024. doi:10.48786/EDBT.2024.02.

55 Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2016. doi:10.1007/978-3-319-41540-6_3.

56 Mehul A. Shah, Joseph M. Hellerstein, and Eric A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 827–838. ACM, 2004. doi:10.1145/1007568.1007662.

57 Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010. doi:10.1109/MSST.2010.5496972.

58 George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. CheckMate: Evaluating checkpointing protocols for streaming dataflows. *CoRR*, abs/2403.13629, 2024. doi:10.48550/arXiv.2403.13629.

59 Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1637–1650. ACM, 2021. doi:10.1145/3448016.3457320.

60 Jonas Spenger, Paris Carbone, and Philipp Haller. Portals: An extension of dataflow streaming for stateful serverless. In Christophe Scholliers and Jeremy Singer, editors, *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022*, pages 153–171. ACM, 2022. doi:10.1145/3563835.3567664.

61 Olivier Tardieu, David Grove, Gheorghe-Teodor Bercea, Paul Castro, Jaroslaw Cwiklik, and Edward A. Epstein. Reliable actors with retry orchestration. *Proc. ACM Program. Lang.*, 7(PLDI):1293–1316, 2023. doi:10.1145/3591273.

62 Aleksey Veresov, Jonas Spenger, Paris Carbone, and Philipp Haller. Failure transparency in stateful dataflow systems (technical report), 2024. arXiv:2407.06738.

63 John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34(34):43–98, 1956.

64 Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: fault tolerance off the critical path. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 338–352. ACM, 2019. doi:10.1145/3341301.3359653.

65 John H. Wensley. SIFT: software implemented fault tolerance. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '72 Fall Joint Computer Conference, December 5-7, 1972, Anaheim, California, USA - Part I*, volume 41 of *AFIPS Conference Proceedings*, pages 243–253. AFIPS / ACM / Thomson Book Company, Washington D.C., 1972. doi:10.1145/1479992.1480025.

66 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

67 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

68 Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013. doi:10.1145/2517349.2522737.

69 Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. The NebulaStream platform for data and application management in the internet of things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf.