Pathfinder: Fault Tolerance for Stream Processing Systems

Bernhard Knasmüller, Christoph Hochreiner, Stefan Schulte
Distributed Systems Group, TU Wien
Email: e1109965@student.tuwien.ac.at, {c.hochreiner|s.schulte}@infosys.tuwien.ac.at

***Abstract*—Data from Internet of Things devices such as sensors often need to be processed in (near) real-time. One common approach to do so is the usage of stream processing. According stream processing systems are able to integrate data from various sources, and to invoke self-hosted and external operators. In case of faults, such systems usually rely on redundancy of single stream processing operators, while the relationship between the single operators is not taken into account. Hence, there is a lack of approaches for fault-tolerant distributed stream processing which consider that stream processing applications are often composed of different operators.**

Within this paper, we present the Pathfinder framework which overcomes these shortcomings by enabling functional redundancy at the level of stream processing operator paths. During system runtime, Pathfinder reacts to operator failures in the main path by switching to a fault-free path with a similar functionality. To restore the main path once a failed operator has recovered, Pathfinder uses the circuit breaker pattern.

1. Introduction

With the ever-growing number of sensing devices in the Internet of Things (IoT), the amount of generated data increases as well, leading towards what is known as *big data*: data sets with a volume, variety, and velocity which prevent the utilization of traditional processing solutions [1].

Gathering and analyzing IoT data series, i.e., *data streams*, from a very large number of devices and users, poses significant technological hurdles, e.g., space and memory limits, latency and bandwidth considerations, data heterogeneity, and fault tolerance [2]. Processing continuous data streams in (near) real-time and offering solutions to the aforementioned hurdles is also known as *data stream processing* (DSP) [3]. DSP at big data scale is usually implemented in a distributed manner where multiple computing nodes are involved to divide the computational burden [4]. To run such distributed stream processing applications (SPAs), distributed stream processing engines (SPEs) like Apache Storm [5] are applied.

Distributed SPEs depend on a potentially large number of *operators* that are composed to perform data processing tasks such as data aggregation, filtering or transformation. These operators can be internally hosted, e.g., using cloud-based computational resources [6], but can also be invoked via the Internet from external service providers in a Software-as-a-Service manner [7], [8].

Each internal and external operator can fail at any time or become unreachable due to communication problems. Since this could, in the worst case, render a complete SPA unresponsive, employing fault tolerance mechanisms is essential for distributed SPEs [9] and has therefore gained a lot of attention recently, e.g., [10], [11]. Notably, today's approaches to fault-tolerant DSP mostly rely on redundant single operators applying an active replication approach [12], while there is a lack of approaches which take into account the fact that SPAs are often implemented by a composition of multiple operators, which together form an operator *path* in a stream processing topology [2]. Within this paper, we argue that by taking advantage of functional redundancy on the level of operator paths, the interplay between operators can be handled while taking care of faulty operators, and it is possible to achieve a higher level of flexibility, compared to redundancy on the level of single operators.

Therefore, we present the *Pathfinder* framework for fault tolerance in distributed DSP. Pathfinder is based on the notion of operator paths and functional redundancy and allows SPA developers to define fallback paths, which are activated if a fault occurs. Based on this information, Pathfinder is able to react to faults at runtime and thereby increases the availability of an SPA. Once the faulty operator has been recovered, a switch back to the main path can be conducted. To achieve this, Pathfinder regularly tests the availability of the failed main path by applying the circuit breaker pattern.

Pathfinder is an independent system that closely interacts with an associated SPE. Therefore, Pathfinder is not restricted to a particular SPE such as Apache Storm. Instead, it can be integrated into any distributed SPE providing the necessary data and control APIs. To achieve fault-tolerant DSP, Pathfinder is provided with functionalities to monitor the distributed SPE, to continuously analyze operational statistics, to classify each operator into being free from failures (or not) based on those statistics, and to initiate the use of fallback paths if failures are detected.

The remainder of this paper is organized as follows: In Sect. 2, we introduce some background information and further motivate the need for path-based fault tolerance. Afterwards, we discuss the functionalities and implementation of the Pathfinder framework in Sect. 3. We evaluate the framework in terms of applicability and performance in Sect. 4. In Sect. 5, we discuss the related work. Afterwards, we conclude the paper in Sect. 6.

2. Background

2.1. Fault Tolerance for Stream Processing

In general, fault tolerance is aiming at avoiding "service failures in the presence of faults" [13]. With regard to fault-tolerant DSP, this means that the service (i.e., generating data output) should still be operational even if there is a fault on the communication or operator level. Within this paper, we are focusing on faults on the operator level. In distributed systems, failures in one part of the system can cause faults in other parts of a system. This error propagation is also frequent in DSP: If one operator needs as input the outcome of another operator, a failure of the former operator will necessarily lead to the failure of the whole SPA.

DSP systems have some unique characteristics that require special attention regarding fault tolerance. First, due to their continuous nature of operation, fault tolerance is even more crucial than in batch processing applications [2]. Second, individual data items tend to have less importance in SPAs than in other kinds of systems, since the loss of single data items can be mitigated by interpolation and does not necessarily prevent service delivery [9]. Another important aspect in fault-tolerant DSP is the differentiation between stateful and stateless operators. As the name implies, stateful operators have a local state that is constantly updated by incoming data items. Therefore, fault tolerance mechanisms need to deal with this and guarantee a reliable state even when faced with failures. In general, the basic methods introduced within the work at hand can be applied to both stateful and stateless operators. While in the following, we will limit our discussion to stateless operators, means for the handling of faulty stateful operators could be directly integrated into our solution approach, e.g., by implementing the mechanisms presented in [9], [11], [14] (see Sect. 5).

In general, there are three basic mechanisms of fault tolerance in DSP systems [2]: (i) cold restart, (ii) checkpointing/restart, and (iii) replication. Both cold restart and checkpointing/restart assume that there is only a transient failure that can be corrected by restarting an operator, e.g., by deploying an operator on a different host. This restart can happen either directly or via checkpointing. Replication is redundancy in its most apparent form. In the case of active replication, multiple instances of the same operator are running concurrently [2]. Once a particular operator instance fails, one of the other instances replace it.

The limitation to fault tolerance on the level of single operators may become problematic for several reasons: First, restart and replication only lead to a failure-free system if the underlying fault is transient. If the fault occurs because there is, e.g., a permanent software bug in an operator, the failure will persist. Second, replication and restart do not take into account the interplay between different operators in an SPA. In the case of a common-mode failure (i.e., a permanent, deterministic failure [13]), where it is necessary to replace an operator by another operator with a similar functionality, the replacement operator may not possess the same interface as the faulty one or may require a different data schema for its input. Hence, it is necessary to adapt the SPA, e.g., by adding another operator which adapts the data stream. This is not foreseen in fault tolerance mechanisms at the level of single operators.

Third, if external operators are invoked within an SPA, operator instances are not under control of the SPE. Instead, an SPE invokes an externally-hosted operator. In this case, the SPE may not be able to simply select another instance of the same operator, because such an instance does not exist. Hence, a different type of operator may have to be invoked, leading again to the problem that the data streams may have to be adapted. Fourth, a different operator type may also be invoked for other reasons, e.g., an external operator may be invoked on an interim basis, because an internally hosted operator is overloaded. Again, this may require adaptations in the stream processing topology. Fifth, in replication-based approaches, fallback solutions need to be created for each operator in isolation, which makes it cumbersome to replace the functionality of a complete processing path (i.e., a combination of individual operators) by a different one.

Instead of focusing on the level of operator instances, functionalities for fault tolerance could also be applied on the infrastructure level or on the level of the distributed SPE. On the infrastructure level, the application logic is not considered, but it is, e.g., possible to restart a failed host. Fault tolerance mechanisms on the level of distributed SPEs provide the means to control the single operator instances in an efficient way and are therefore a promising approach to redeploy failed operator instances only if necessary, instead of fully replicating operators. However, these approaches suffer from the same drawbacks regarding missing awareness of operator interplay, as discussed in the paragraph above.

Due to the shortcomings of fault tolerance mechanisms on the level of single operators and on the infrastructure level, we propose to provide *path redundancy*. Redundancy in SPAs does not necessarily imply the use of multiple instances of the *same* operator in case of an operator crash. Instead of having multiple instances of the same operator path, we argue that SPA developers should be allowed to define different alternative paths that provide the same (or a similar) functionality and can replace each other. Compared to redundancy at the operator level, this approach has several advantages:

Functional Redundancy. Fault tolerance on the level of operator paths allows to use an alternative path (with different operators) in case of a failure in the main path. This resembles the idea of *N-version programming* [15]. Functional redundancy is especially helpful if it is not possible to deploy a replica of a failed operator, e.g., because the operator is externally hosted, or because of a permanent software bug which cannot be solved immediately.

Notably, applying functional redundancy on the level of operator paths does not prohibit that the same operator type is used within different paths, however unique instances of these operators per unique path.

Flexibility. Since paths can consist of arbitrarily many operators, SPA developers can define fault tolerance

actions for specific sets of operators. In addition, the length and number of alternative paths can be chosen by the SPA developer to reflect the level of availability that needs to be achieved. Naturally, there are SPA functionalities for which failures can be tolerated more easily, while for others, faults are intolerable [2].

Mitigation of Fault Propagation. The handling of faults on the level of single operators does not allow to define how to avoid data congestion since data items cannot be processed any longer. By defining fault tolerance on the path level, i.e., by defining how to redistribute data items in case a particular operator fails, faults can be intercepted before such data congestions occur.

Simplicity. Pathfinder allows to define fault tolerance on the level of a stream processing topology definition, as presented in [6]. Hence, no thorough knowledge about fault tolerance or distributed systems is required to define alternative paths. Furthermore, the code of existing operators does not need to be modified in order to add fault tolerance. Instead, a distributed SPE takes the information from the topology definition to mitigate failed operators.

2.2. Terminology

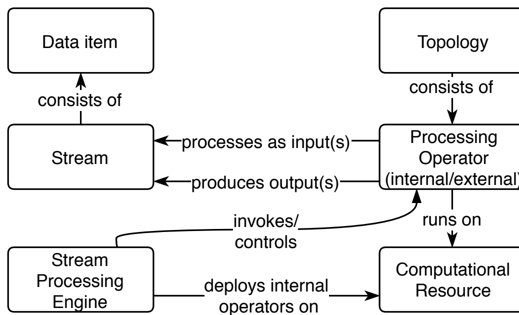

Figure 1: Entities in DSP

Fig. 1 provides an overview of the entities in DSP most important for the work at hand. Following the definition by Andrade et al. [2], data streams consist of a continuous sequence of data items of the same type. Processing *operators* consume one or more streams as an input and produce one or more output streams. The composition of all operators with their connections is known as a *topology*. Usually, a data flow representation is used to model the flow of data items between operators [2]. An SPE manages the internal operators and their deployment on computational resources, and invokes external operators, respectively.

We define the *topology* of an SPA as a directed acyclic graph $T(O, E)$ that consists of a set of vertices representing operators O and a set of edges that denote a data flow E [16]. The topology, therefore, defines which operators exist in an SPA and how they are connected to each other. A data flow from operator A to operator B exists if there is an edge from A to B. We denote such a data flow as $df(A, B)$. An operator A has an input (operator) B if there is a logical flow from B to A. Analogously, A has an output (operator) C if there is a logical flow from A to C. The input degree of an operator A is denoted as $deg^+(A)$ and represents the number of input streams of A. Analogously, the output degree $deg^-(A)$ represents the number of output streams of A.

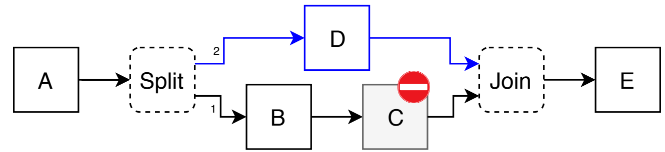

Figure 2: Example Topology with Split/Join Operators

Operator paths provide functionality by combining individual operators in the right order [2]. A *path* π in a topology T is defined as an ordered sequence of operators such that there is always a data flow between consecutive operators in that sequence. We refer to the i-th operator in path π using the notation π^i (starting at index $i = 0$). An operator A is part of path π if $A \in \pi$. A path π is *active* if there exists an operator $A \notin \pi$ where $df_p(A, \pi^0)$, i.e., there is a data flow from A to the first operator of π. A path is *inactive* if there is no operator $A \notin \pi$ where $df_p(A, \pi^0)$, i.e., there is no operator with a data flow to the first operator of π. A path is said to contain a *failure* if one or more of its operators contain a failure.

Topologies may contain *split* and *join* operators. A *split* operator S is an operator with $\deg^-(S) > 1$. Semantically, a split operator defines several *alternative paths*: a *main path* and one or more *fallback paths*. The main path is the alternative path that is active by default if all alternative paths are fault-free. We name an alternative path π by its first operator (π^0) since the beginnings and ends of alternative paths are unambiguously defined by the split and join operators, respectively. An alternative path's *path order* ρ defines the order in which Pathfinder falls back on the alternative path in case of operator failures. The main path has a path order of $\rho = 1$, i.e., it is used with the highest priority. Path orders are defined by an SPA developer. There must exist at least one other alternative path with $\rho > 1$. Two alternative paths of a split operator must not have the same path order (i.e., for a split operator S, $\mathrm{df}_l(S, A) \wedge \mathrm{df}_l(S, B) \Rightarrow \rho_A \neq \rho_B$). The counterpart to the split operator is the *join* operator. It indicates the place where the alternative paths come together again. Notably a split operator will only forward data items to the active path, and if applicable, to a path which is currently *probed* (see Sect. 3.4). Join operators only receive data items from the active path.

Fig. 2 shows an example topology with a split and a join operator. In this representation, operators are modeled as boxes and the data flow between them as directed edges. In the example, operator C, which is part of the main path π^B ($\rho = 1$), has a failure. Therefore, the complete main path has failed, and fallback path π^D ($\rho = 2$) is activated by Pathfinder through the split operator; the active fallback path is highlighted in blue colors. As can be seen in the example

topology, functional redundancy as applied in Pathfinder does not necessitate the availability of the same operator types in a path, even though this *could* be the case. In the example, the composed operators B and C are offering a functionality which is similar to the functionality of the single operator D. Therefore, the alternative path π^D can take over data processing if the main path π^B fails.

3. Pathfinder – A Fault Tolerance Framework for Stream Processing

3.1. Error and Fault Handling Approach

Following a common approach, Pathfinder performs a two-stage recovery from faults [13]. For this, first, the faulting operator itself is handled. In a *diagnosis* step, it is detected which operator caused the error. The whole path containing the faulty operator is then *isolated* to mask the fault. This is done by blocking the data flow to that path. In a *reconfiguration* step, an alternative path is activated. Finally, *reinitialization* of the SPA takes place and the data item routing is updated such that the newly activated path is used instead of the faulty one.

Pathfinder's fault handling does not stop after the faulty processing path has been deactivated. While it is acceptable to use a fallback path as long as the main path is unavailable, there is also a need for a mechanism to get back to the main path once it is available again, if it is possible to restore the main path. If this is not possible, the other operators in the main path are shut down, unless they are needed in a different path. In the case of external operators in the failed main path, the operators are not invoked any longer. A probing mechanism is used to continuously check whether a previously failed path was able to recover. This probing mechanism is based on the circuit breaker pattern and will be discussed in detail in Sect. 3.4.

Notably, no attempts are made at reprocessing data items by the main path after they have already been processed by a fallback path. This is because it is assumed that the fallback path produces results of a similar quality and the benefits of reprocessing by the main path are small compared to the computational overhead of reprocessing.

3.2. Framework Functionality

Pathfinder is an independent framework that closely interacts with an associated SPE. Thereby, Pathfinder is not restricted to a specific distributed SPE. Instead, it can be used to support any distributed SPE that is able to provide the needed information and to obey the commands by Pathfinder. Pathfinder controls the associated SPE's data flow and operator deployment based on monitoring operational statistics. For this, Pathfinder

- *monitors* the associated SPE and continuously analyzes operational statistics,
- *classifies* each operator into being free from failures or not based on those statistics, and
- *commands* the use of fallback paths if failures are detected.

Monitoring and *classifying* are necessary to detect faults. These steps are essential since fault tolerance mechanisms can only be initiated when Pathfinder knows about the faults. *Commanding* a distributed SPE is needed because Pathfinder does not reimplement control features to influence the distributed SPE's operators or its data flow. Thus, separation of concerns between Pathfinder's functionalities and an SPE's usual mode of operation is provided.

For the commanding, Pathfinder needs to know about available fallback paths. This knowledge needs to be added by the SPA developer. For instance, a topology definition language like VTDL [6] could be extended accordingly. For the definition of fallback paths, *split* and *join* operators as introduced in Sect. 2.2 are used. As it can be seen in the example in Fig. 2, downstream of a split and upstream of a join operator are at least two alternative paths that provide functional redundancy through different operators. Pathfinder decides at runtime which of the alternative paths is active, i.e., receives the data flow from the operator that is directly upstream of the split operator. In Fig. 2, this is operator A. If Pathfinder detects a failure in the main path (identifiable by the path order $\rho = 1$ shown next to the split operator), it applies the circuit breaker pattern by switching to a fallback path and to observe the recovery of the main path (see Sect. 3.4).

3.3. System Design

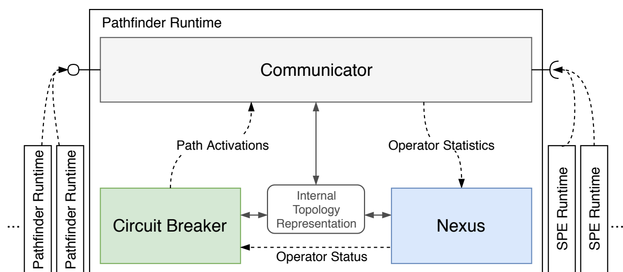

Figure 3: Pathfinder's System Design

Pathfinder's basic functionalities are provided through three core modules: The *Circuit Breaker*, the *Nexus* and the *Communicator*. Fig. 3 shows the system design.

A critical component such as the Pathfinder framework should not become a single point of failure. Hence, Pathfinder is deployed in a distributed manner itself, as can be seen by the presence of different *Pathfinder Runtimes* in Fig. 3. Since every Pathfinder Runtime can be restricted to query only a subset of a distributed SPE's operators, distributed deployment of Pathfinder Runtimes also allows easier scaling based on the partitioning of the topology.

In the following paragraphs, the three core modules of Pathfinder are explained in more detail:

(i) Since Pathfinder can be deployed in a distributed manner, a Pathfinder Runtime instance needs to communicate with other Pathfinder Runtime instances and also with

instances of the associated SPE, e.g., in order to actively request operational statistics. Bundling all communication aspects in the single module *Communicator* enables loose coupling since this module can be easily exchanged when a different distributed SPE should be controlled. The Communicator module is also responsible for updating the states of the circuit breakers (see Sect. 3.4) and for retrieving topology information from the associated SPEs.

(ii) The *Circuit Breaker* module is aware of the topology that is currently active in the associated SPE. To describe a topology, e.g., a description language needs to be used, as mentioned above. In order to address arbitrary distributed SPEs, a conversion step from and to the description language is added to the Communicator.

A *circuit breaker object* is created and maintained for each alternative path at a split operator. By continously querying the Nexus component, operator failures are detected that are then translated into state transitions of the circuit breaker objects. In the event of topology changes initiated by the associated distributed SPE, the internal topology representation of the Circuit Breaker is discarded and new circuit breaker objects are created for all paths of the SPA's updated topology.

(iii) The *Nexus* component is responsible for analyzing statistical data and classifying operators into *working* and *failed*. Each operator is classified separately based on current and historical statistics collected from the associated SPE. The concrete Nexus implementation can make use of different technologies such as manually created rules or machine learning-based profiles. Particular example rules which have already been implemented in Pathfinder are: (i) Not more than a maximum number of data items are allowed to be buffered for processing at a specific operator while the CPU utilization is below a particular minimum threshold, (ii) the memory and CPU utilization is below or above minimum/maximum thresholds, and (iii) the rate of data item processing is lower than the rate of incoming items.

3.4. Circuit Breakers in Pathfinder

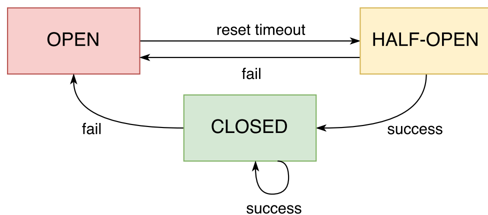

Figure 4: States of a Circuit Breaker (Adopted from [17])

A circuit breaker in a software system "trips" (i.e., stops the control/data flow to a service) when a failure is detected, thereby preventing future calls to that service from being made [17]. Instead, either a fallback solution is used or the failure is propagated to the caller as soon as possible.

Using a circuit breaker has two major benefits. First, the failing service (here: an operator) itself will experience less load which might be beneficial for a quick recovery, depending on the failure's cause. Second, the other operators as well as the SPE will not waste time waiting for responses from the failed operator and can instead instantly provide a fallback solution or throw an exception.

The basic idea of the circuit breaker pattern in Pathfinder is to avoid invoking a failed operator's service. Instead, a fallback solution is used to replace the operator as soon as that failure is detected. One solution approach would be the implementation of the circuit breaker pattern on the level of individual operators, but as discussed in Sect. 2.1, this may not be the best solution in all scenarios. Instead, Pathfinder implements circuit breakers on the level of paths that can consist of arbitrarily many operators. The major advantage of this approach is that a topology designer can construct functional redundancy at the topology level and is not restricted by the subdivision of functionality into multiple operators. However, redundancy can still be implemented at the operator level simply by considering paths of length one (i.e., containing only one single operator) in Pathfinder.

A circuit breaker can be viewed as an automaton with three possible states: *open*, *half-open* and *closed* [17], as shown in Fig. 4. Being closed by default, a circuit breaker acts as a proxy and forwards all requests to a software component and in turn forwards the component's reply to the caller. With regard to distributed SPEs, this means that a circuit breaker forwards the data flow to an operator if it is closed. If an operator failure is detected, the circuit for the respective operator is opened. In this state, no further data is forwarded to the failed operator. After a configurable amount of time, the circuit breaker switches to the half-open state and forwards a small fraction of data items to the presumably failed operator to see whether it has recovered. If it has (indicated with "success" in Fig. 4), it returns to the closed state and the operator is fully functional again. For this, Pathfinder invokes a *probing* mechanism in the half-open state where only a small fraction of data items is allowed to pass the circuit breaker (see below).

To implement this pattern in Pathfinder, a circuit breaker object is created for every alternative path of every split operator. Pathfinder's *Nexus* component (see Sect. 3.3) provides information about the current health of each operator. Using that information, the circuit breakers are updated accordingly (i.e., circuit breakers are opened if at least one operator failure in that path is detected). On each circuit breaker transition, the distributed SPE is contacted and advised to configure the data flow, i.e., stop it when the circuit breaker is opened and resume it once it is closed.

An example is shown in Fig. 5. The figure depicts the data flow through a split operator. Each of the outgoing paths ($\pi^{P_{1-3}}$) is assigned its own circuit breaker and the state of each circuit breaker is indicated by a traffic light symbol. Since the circuit breaker of π^{P_1} is in the *open* state, there is no data flow from operator A to operator P_1. By means of the probing mechanism, a small fraction of data items is forwarded to P_2 due to the circuit breaker of π^{P_2} being *half-open*. π^{P_3} is the active path due to its circuit breaker being *closed*.

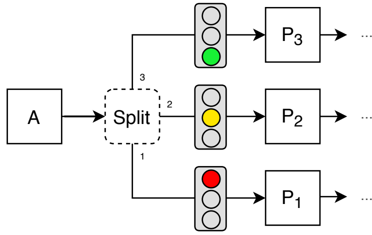

Figure 5: Representation of Circuit Breakers in Pathfinder

As can be seen in Fig. 4, the following circuit breaker transitions are used in Pathfinder:

Closed to open. The most important transition is arguably the opening of the circuit breaker, which corresponds to blocking the data flow to an operator. While being in the closed state, the data flow is uninterrupted. The transition to the open state causes the flow to stop immediately, i.e., Pathfinder commands the SPE to stop the data flow by invoking a specific API of the SPE. This transition is caused by the Nexus component providing information about an operator failure.

Open to half-open. After a configurable time period, the circuit breaker automatically transitions from the open to the half-open state. In this state, the probing takes place, i.e., a small amount of data items is forwarded to the first operator in the probed path in order to assess if the path is fully operational again.

Half-open to open. If the probing has indicated that the path still contains failed operators, the circuit breaker transitions back to the open state until another probing attempt is started.

Half-open to close. If the probing attempt succeeds and all operators of the probed path are fully operational again, the circuit breaker transitions to the closed state and the normal data flow is restored, i.e., the probed path is activated again.

In addition to these transitions, Pathfinder allows the execution of several path-related commands. These commands are triggered via Pathfinder's Communicator module at the distributed SPE if the circuit breaker of a particular path changes. If an operator recovery is detected, the circuit breaker is changed to the closed state and the path is available again. However, if an operator of an active path becomes unavailable, that path's circuit breaker is transitioned into the open state and an alternative path is activated. To find a suitable alternative path, Pathfinder first considers all alternative paths of the split operator the failing path was part of. From those alternative paths, it further considers only those paths with a closed circuit breaker. If more than one path with a closed circuit breaker is available, Pathfinder uses the one with the lowest ρ. It then commands the associated SPE to change the data flow of all operators directly upstream of the split operator to that alternative path. However, if no such path is available, Pathfinder is not able to correct the operator failure at that point in time and waits for the next scheduled retrieval of new operational statistics. In any case, Pathfinder then transitions the circuit breaker of the failed path into the open state to signal that it is no longer available. Also, probing attempts are initiated.

For this, the Circuit Breaker component determines that an alternative path with a circuit breaker in the open state is to be probed. Then, the circuit breaker's state is transitioned to the half-open state and an endpoint of the distributed SPE is invoked to start the probing procedure. In the meantime, Pathfinder continues gathering operational statistics from the operators and detects whether the probed path has recovered in which case its circuit breaker is transitioned to the closed state. However, if no recovery took place, no further probing attempts are made until a certain user-defined amount of time has passed. This cool-down mechanism ensures that no resources are wasted on an inactive path, especially in long paths containing many operators. Also, the topology designer can define how many probing attempts are conducted before a path is marked as permanently failed. This is done in order to be able to shut down internal operators and to stop the invocation of external operators if they are not needed any longer. Notably, during a probing attempt, the data flow to the currently active path must not be interrupted. Otherwise, data item loss may occur. The rare case where active and probed both paths successfully process the same data items are tolerated since Pathfinder follows an *at-least-once* delivery approach.

4. Evaluation

4.1. Experimental Setup

Pathfinder has been implemented as a standalone framework that communicates with associated SPEs via REST APIs. For the purposes of the evaluation, the *Vienna Ecosystem for Elastic Stream Processing* (VISP) [18] has been used, but in general, Pathfinder could be integrated into any arbitrary SPE, as long as this SPE is able to deliver the needed information and can execute commands from Pathfinder (see Sect. 3.2). VISP and Pathfinder are available on Github[1].

VISP is a fully-fledged research SPE, which allows the utilization of cloud-based computational resources in DSP to adapt to changes in data volume and velocity at runtime. In VISP, Docker containers are utilized to host the single operators. VISP allows *lazy deployment* of operators. This means that only the operators in the main path are activated at initialization, while operators in alternative paths are not deployed yet. This saves computational resources and therefore cost, especially if compared to the usually applied active replication of operators in SPAs. However, it also causes an additional startup time since no results are produced between the failure of the main path and the complete deployment of a fallback path.

Notably, VISP provides a DataProvider, which is applied to generate input data in a reproducible way. In the experiments, a *constant* pattern is applied, i.e., the production of new data items is done at a constant rate. The DataProvider

1. https://github.com/visp-streaming

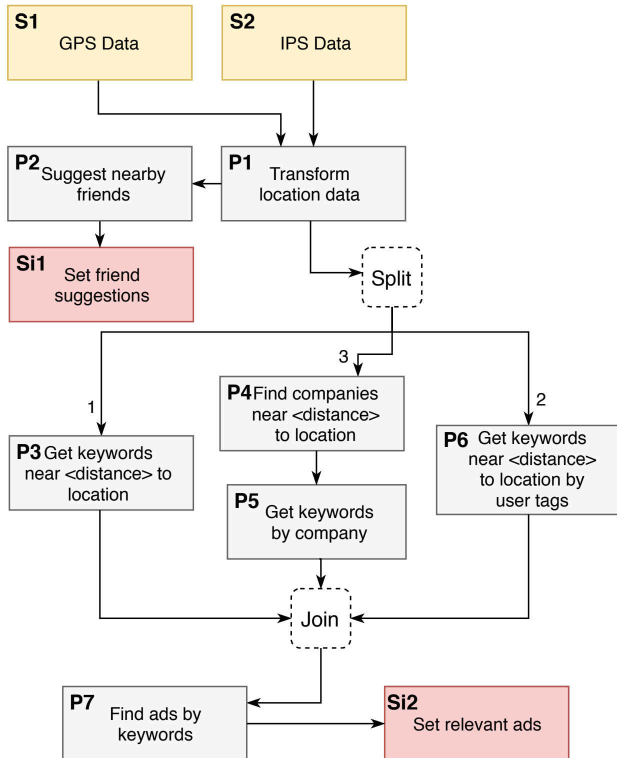

Figure 6: Evaluation Topology

is deployed on the same virtual machine as VISP in order to minimize network communication overhead. Another key feature of VISP is the ability to change the topology at runtime. Since this ability is important for fault tolerance mechanisms, VISP has been chosen as the basis for the evaluation.

In order to evaluate the performance and functionality of Pathfinder, a number of experiments have been conducted. The experiments are performed in a private OpenStack-based cloud. An `m1.large` instance (7 GB memory, 4 VCPUs) and an `m1.medium` instance (3 GB memory, 2 VCPUs) are used for the deployment of VISP and Pathfinder, respectively. Additionally, four `m1.large` instances serve as computational resource pool for spawning the operator instances used for data processing.

Fig. 6 describes the topology used for the evaluation of Pathfinder. The topology is based on a real-world scenario presented by Couceiro et al. [19]. As can be seen in the figure, it provides an SPA example from the marketing domain, aiming at targeted advertisements based on the user location. This could be applied, e.g., in order to provide personalized ads to users on their smartphones, e.g., in a mall.

There are two data sources $S1$-2, supplying the necessary data input, two data sinks $Si1$-2, and seven operators $P1$-7, which provide the following functionality:

- `S1_GPS_Data`: User positions derived from the GPS module of mobile devices enter the SPA via this source.
- `S2_IPS_Data`: User positions derived from indoor position systems enter the SPA via this source.
- `P1_Transform_location_data`: This operator transforms data items from $S1$ and $S2$ into a common format for further processing.
- `P2_Suggest_nearby_friends`: Based on the current position of the users, this operator suggests connections with other users based on geographical proximity and other matching criteria (e.g., common friends, workplaces or interests).
- `P3_Get_keywords_near_distance...`: An external service is used to retrieve a set of keywords for a specific location.
- `P4_Find_companies_near_distance...`: Based on a location, a set of companies located in proximity is fetched from a database.
- `P5_Get_keywords_by_company`: A company is transformed into a set of keywords based on what it sells.
- `P6_Get_keywords_near_distance_-to_location_by_user_tags`: Keywords for a specific location are fetched from a database containing user annotations.
- `P7_Find_ads_by_keywords`: Filters the set of advertisement campaigns by restricting them to certain keywords.

In order to generate reproducible results, we intentionally did not fully implement every single operator. Instead, $P1$-7 simulate a predefined computational load. For each data item to be processed by an operator, the processing of that item is simulated by computing the Fibonacci sequence until n. Tab. 1 lists how much work the operators need to perform for each data item. The n values have been chosen to approximate the expected work done by each operator. The average time it takes an operator to process a *single* data item for a specific n is also shown in the table. The sinks $Si1$-2 consume the data without any further action.

As shown in Fig. 6, the topology features a split/join segment that has been defined by the topology designer to achieve fault tolerance. In the case of active replication, all paths π^{P3}, π^{P4}, and π^{P6} would be active at all times. However, when applying Pathfinder, one path is selected at a time, as described in Sect. 3. As it can be seen in the figure, functional redundancy of the different operators in the alternative paths is assumed, i.e., the different paths provide a comparable functionality, but the single operators are not identical. We assume that all operators are self-

TABLE 1: Load Simulation

Operator	*n*	*Avg. proc. time [s]*
P1	37	0.199
P2	43	2.115
P3	40	0.841
P4	40	0.841
P5	40	0.841
P6	40	0.841
P7	40	0.841

hosted on the abovementioned computational resources.

In order to produce deterministic results, we artificially inject faults at pre-defined points of time during the experiments. We simulate operator failures in three different ways: (i) exhaustive memory consumption (MEM) of an operator, (ii) total suspension of execution (SLP), and (iii) low processing throughput (THR). Pathfinder's Nexus is able to identify these different failures as follows: (i) For MEM failures, an operator is classified as failed if its memory usage is above or below predefined thresholds. These thresholds can be defined by the topology designer or can be derived from historical operator performance data. Nexus identifies failures by comparing thresholds and monitored memory consumption. (ii) SLP failures are detected by observing the rate of item output. An operator is classified as failed if this rate is zero despite data items waiting in the input queue of an operator. (iii) THR failures occur if an operator is slowing down with regard to the number of data items processed in a particular timeframe, e.g., because the operator is not able to cope with the velocity of its input data streams. To identify such failures, Nexus compares the rate of incoming data items to the rate of their consumption and classifies an operator as failed if the incoming rate is larger.

4.2. Experiments

For each of the three mentioned operator failure types (MEM, SLP, and THR), four experiments are carried out at different times of the day and on different days. This is done in order to cater for "background noise", e.g., the load on the OpenStack-based cloud testbed by other applications. Each experiment starts with deploying the topology in the VISP Runtime. Once the setup is complete, the main path π^{P3} is activated. Then, the VISP DataProvider is configured to continuously produce data items at $S1$ and $S2$ at a constant rate. Initially, $P3$ is active and receives data items from $P1$. After another 100 seconds, an artificial MEM, SLP, or THR failure is deliberately introduced in $P3$.

In the meantime, Pathfinder continuously queries VISP for operator statistics. Initially, all statistics are expected to show normally working operators. Once the artificial failure has been introduced, Pathfinder is expected to detect the failure of $P3$ and react by switching to the next alternative path π^{P6} (as indicated by $\rho = 2$). The time between the beginning of the failure and the activation of π^{P6} is named *time to adapt*, which is equal to the time where no path is active. The time to adapt consists of six parts: (i) the time between two scheduling intervals, (ii) the time it takes Pathfinder to receive the statistics from VISP, (iii) the time it takes for the Nexus component to make a failure prediction for an operator, (iv) the time it takes until Pathfinder can communicate its decision to VISP, (v) the time it takes VISP to switch paths, and (vi) the time it takes until a newly activated operator is deployed.

Once the artificial failure is removed and $P3$ recovers, Pathfinder is expected to detect this recovery and switch back to the main path π^{P3}. Regular probing events are used to verify whether $P3$ has already recovered.

4.3. Results and Discussion

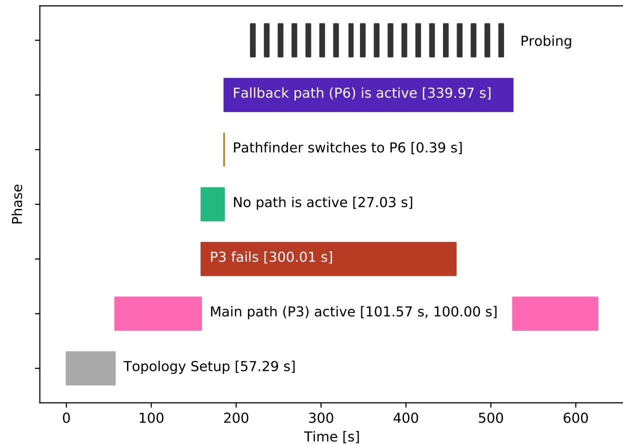

Figure 7: Pathfinder's Action in Case of Failure

Fig. 7 shows a plot that visualizes the temporal connection between the different experimental phases described above. Different phases are depicted in a GANTT-inspired style by horizontal bars in different colors. The width of each bar corresponds to the duration of the phase, and the position along the x-axis shows its beginning and end. Notably, the figure presents one representative evaluation run (out of twelve runs) to avoid cluttering the figure with too much information. In fact, for each of the twelve evaluation runs, a similar plot has been produced. While the plots of all twelve evaluation runs show similar results for most of the phases, there is one notable exception, namely the time to adapt, i.e., the timespan where no path is active. Therefore, this aspect will be discussed separately below.

Fig. 7 shows that, starting at time $t = 0$, the initial topology setup takes about 57 seconds. This phase includes (1) uploading a topology description to VISP and all subsequent initialization steps, (2) connecting VISP with Pathfinder, and (3) starting the DataProvider which emits the data for $S1$ and $S2$. This is not influenced by Pathfinder, since it simply sets up the topology within VISP.

The second phase starts when the first data item from $S1$ or $S2$ was successfully processed by $P1$ (approximately at $t = 58$). In this phase, the main path π^{P3} is active and is successfully processing incoming data items. While it is not explicitly shown in the plot, Pathfinder continuously queries VISP for operational statistics to detect potential failures but does not detect anything yet. At $t = 159$, an artificially generated operator failure of $P3$ is triggered. Specifically, Fig. 7 shows the data for one run of the SLP failure type. The total length of the failure is 300 seconds. During this time, the main path π^{P3} is not available.

Since Pathfinder queries VISP in intervals of 15 seconds, some time passes where no path is active. This phase takes about 27 seconds and ends when Pathfinder recommends VISP to switch to π^{P6}. Reasons for the duration until the switch-over is recommended are discussed below.

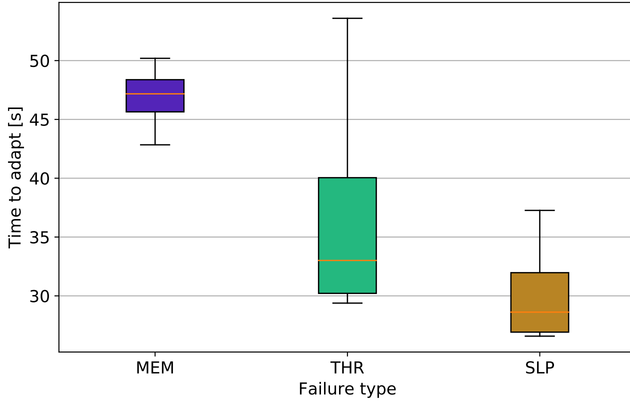

Figure 8: Time to Adapt

Switching paths is very time-efficient – after 0.39 seconds, π^{P6} is active. While π^{P6} is active, probing attempts are made where the data flow to $P3$ is restored for a few seconds to see whether the operator has recovered. Once $P3$'s failure stops after 300 seconds, Pathfinder switches back to π^{P3} after a few probing attempts and the period of π^{P6} activity ends after a total of 340 seconds. Without Pathfinder, the whole SPA would have been unavailable during this time. In summary, the experiment shows that Pathfinder is able to maximize the SPA's availability in the presence of operator failures by utilizing functional redundancy at the path level.

Nevertheless, there are some shortcomings that need to be discussed. First, the time it takes to detect the failure is quite high (around 27 seconds). This can be traced back to the fact that Pathfinder's Nexus component solely relies on statistics and performance indicators provided by the execution environment (see Sect. 3.3), instead of relying on implementation details of the single operators. While this allows using Pathfinder without development efforts on the operator level, deriving a failure obviously takes additional time. Second, the time to detect the recovery is also quite long. As can be seen in Fig. 7, there are three probing events between the end of $P3$'s failure (i.e., the end of the red bar) and the re-activation of the main path (i.e., the second start of the pink bar). Each probing event consists of three stages: (1) the activation of data flow, (2) a short pause, and (3) the inactivation of data flow. Due to the fluctuations in data volume, it can happen that the pause is too short and no data items are produced during this time span. Therefore, one probing event might not be sufficient. This could be fine-tuned by elongating the pause at the cost of increased resource usage.

In order to detect the influence of the three different operator failure types regarded by Pathfinder, the average time to adapt for the failure types has also been evaluated. Fig. 8 shows the outcomes of these experiments. As mentioned above, experiments were conducted four times for each failure type. SLP is the quickest to be detected (on average around 30 seconds) since its effects are reflected immediately by the lack of new data items. The MEM failure takes longer to be detected (45-50 seconds) since it takes some time until the memory utilization threshold is reached and the operator is therefore classified as failed. The THR failure's detection time varies much more than that of MEM and SLP, as can be seen from the high spread depicted in Fig. 8. Since THR failures cause a processing delay, the failure is only triggered when the incoming data item rate is large enough. This depends on multiple factors, including the current utilization of the cloud infrastructure. The identification of solutions to reduce the time to adapt for the different failure types remains part of our future work.

5. Related Work

To the best of our knowledge, no approaches to path-level fault tolerance for distributed SPEs have been proposed so far. Nevertheless, there is some relevant related work in the field of fault-tolerant SPEs.

To start with, Balazinska et al. [11] introduce a fault tolerance approach for Borealis, aiming at eventual consistency. It allows SPA developers to change the trade-off between availability and consistency by specifying a maximum waiting time. The distributed SPE will wait as long as the predefined recovery time for operators has not passed yet. If an operator does not recover in time, tentative results are produced, serving as approximations for the missing results. This allows a continuous output of data items. Once a failed operator is successfully recovered, the tentative results are corrected. Akidau et al. [14] present the distributed SPE MillWheel, along with a fault tolerance model. MillWheel promises consistency even if faced with arbitrarily many hosts crashing and an infinite amount of data items lost. These guarantees are implemented using an acknowledgment mechanism to prevent message loss, and a fine-grained checkpointing mechanism. Huang and Lee [9] present an approach to approximate fault tolerance for distributed SPEs. The authors argue that especially for scenarios where streaming data is only processed to identify trends in the data, it is tolerable to lose some of the data items to improve availability. Hence, the authors tolerate a small number of errors that can remain uncorrected.

All three discussed fault tolerance mechanisms focus on individual operators. As has been argued in Sect. 2.1, Pathfinder is based on the assumption that fault tolerance should also be provided on the level of operator paths, since operators should not be seen as atomic units. Hence, our work enables functional redundancy on the level of operator paths. This has not been foreseen in the related work so far.

It should be noted that the work presented in [9], [11], [14] explicitly aims at stateful operators, while we have not discussed operator state in Pathfinder so far. Within the scope of the paper at hand, the inclusion of operator state was only of secondary interest, since the goal was to present the feasibility and benefits of path-based fault tolerance in DSP. In order to support stateful operators, the according methods from [9], [11], [14] could be integrated into Pathfinder.

In their seminal work on Spark Streaming [12], Zaharia et al. argue that active replication is too expensive and

therefore novel solutions should be applied in fault-tolerant DSP. The authors propose discretized streams, allowing parallel recovery, which is also able to tolerate stragglers. In contrast to most of the related work and Pathfinder, the authors abstain from a continuous operator model. Instead, stream processing is done in micro-batches. This is surely a very interesting approach, but not compatible with the operator-based streaming model applied in the work at hand.

Apart from the abovementioned approaches which have been specifically developed for distributed SPEs, there are also related approaches from the field of service-oriented computing. In fact, the roles of services in service-oriented computing and of operators in SPEs are very similar, since these entities are independent building blocks that are composed to realize a functionality. While there are also significant functional differences between services and operators, the commonalities allow us to adopt some basic approaches to fault tolerance from the field of service-oriented computing. Especially, fault-tolerant frameworks for microservice architectures like Netflix' Hystric [20] are of interest, since the underlying principles of *custom fallback* and *fail fast* are adopted in Pathfinder.

6. Conclusion

DSP has become a commodity in many application areas, with distributed SPEs like Apache Storm being used widely by the industry. Despite the significant advances in distributed SPEs, there is nevertheless a lack of sophisticated fault tolerance mechanisms in state-of-the-art SPEs. Today, most existing SPEs rely on active replication in order to ensure continuous operation. However, active replication of single operators features a number of shortcomings since the interactions between single stream processing operators are not taken into account.

In order to mitigate this, we have presented Pathfinder, a framework to introduce functional redundancy on the level of operator paths into distributed SPEs. At runtime, Pathfinder reacts to faults by switching to a fault-free path with a similar functionality. To restore the main path once a failed operator has recovered, Pathfinder applies the circuit breaker pattern. While we have evaluated Pathfinder in the research SPE VISP, integrating and testing it with further distributed SPEs such as Apache Storm or Kafka would surely also be an interesting exercise. As a second major part of our future work, we aim to improve the presented failure detection mechanisms in order to decrease the time to adapt. We are currently working on a solution that compares data from a failure-free operation with the current metrics of an operator. For this, statistical and machine learning methods are applied. Last but not least, we plan to integrate existing mechanisms for the handling of faulty stateful operators.

Acknowledgments

This work is partially funded by COMET K1, FFG – Austrian Research Promotion Agency, within the Austrian Center for Digital Production (Contract No. 854187).

References

[1] A. McAfee and E. Brynjolfsson, "Big Data: The Management Revolution," *Harvard Business Review*, vol. 90, no. 10, pp. 61–67, 2012.

[2] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing*. Cambridge University Press, 2014.

[3] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 Requirements of Real-Time Stream Processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[4] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[5] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm @Twitter," in *2014 ACM SIGMOD Int. Conf. on Management of Data*, 2014, pp. 147–156.

[6] C. Hochreiner, S. Schulte, S. Dustdar, M. Nardelli, and B. Knasmüller, "VTDL: A Notation for Data Stream Processing Applications," in *12th IEEE Int. Symp. on Service Oriented Syst. Engineering*. IEEE, 2018, pp. 76–85.

[7] B. Allen, J. Bresnahan, L. Childers, I. T. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a Service for Data Scientists," *Comm. of the ACM*, vol. 55, no. 2, pp. 81–88, 2012.

[8] Z. Zheng, J. Zhu, and M. R. Lyu, "Service-Generated Big Data and Big Data-as-a-Service: An Overview," in *IEEE Int. Cong. on Big Data*. IEEE, 2013, pp. 403–410.

[9] Q. Huang and P. P. C. Lee, "Toward High-Performance Distributed Stream Processing via Approximate Fault Tolerance," *Proc. of the VLDB Endowment*, vol. 10, no. 3, pp. 73–84, 2016.

[10] L. Su and Y. Zhou, "Tolerating correlated failures in Massively Parallel Stream Processing Engines," in *2016 IEEE 32nd Int. Conf. on Data Engineering*. IEEE, 2016, pp. 517–528.

[11] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-Tolerance in the Borealis Distributed Stream Processing System," *ACM Trans. on Database Syst.*, vol. 33, no. 1, pp. 1–44, 2008.

[12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *24th ACM Symp. on Operating Syst. Principles*. ACM, 2013, pp. 423–438.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE T. on Dependable and Secure Comp.*, vol. 1, no. 1, pp. 11–33, 2004.

[14] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. Mcveety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," *Proc. of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[15] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE T. on Software Engineering*, vol. 11, pp. 1491–1501, 1985.

[16] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things," *2016 IEEE 9th Int. Conf. on Cloud Comp.*, pp. 100–107, 2016.

[17] M. Fowler, "Circuit Breakers," 2014. [Online]. Available: https://martinfowler.com/bliki/CircuitBreaker.html

[18] C. Hochreiner, M. Vögler, P. Waibel, and S. Dustdar, "VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things," in *20th Int. Enterprise Distributed Object Comp. Conf.* IEEE, 2016, pp. 1–11.

[19] M. Couceiro, D. Suarez, and D. Manzano, "Data stream processing on real-time mobile advertisement: Ericsson research approach," in *IEEE 12th Int. Conf. on Mobile Data Management*. IEEE, 2011, pp. 313–320.

[20] Netflix Inc., "Hystrix," 2013. [Online]. Available: https://github.com/Netflix/Hystrix