FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Ausfallsicherheitsmechanismen in Datenstromverarbeitungssystemen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software-Engineering und Internet-Computing

eingereicht von

Bernhard Knasmüller, BSc BSc
Matrikelnummer 01109965

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Christoph Hochreiner

Wien, 18. Jänner 2018

Bernhard Knasmüller

Stefan Schulte

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

On Fault Tolerance in Stream Processing Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Bernhard Knasmüller, BSc BSc
Registration Number 01109965

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.-Ing. Stefan Schulte
Assistance: Christoph Hochreiner

Vienna, 18th January, 2018

Bernhard Knasmüller   Stefan Schulte

Erklärung zur Verfassung der Arbeit

Bernhard Knasmüller, BSc BSc
Pfeilgasse 4-6 / 434, 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Jänner 2018

Bernhard Knasmüller

Acknowledgements

I would first like to thank my advisors Stefan Schulte and Christoph Hochreiner for their valuable guidance and support throughout my thesis. Especially, I owe Christoph my gratitude for his continued feedback even after him having finished his own doctoral thesis. Furthermore, thanks to Michael Borkowski for his ideas on failure detection.

Thanks to Donald Knuth for inventing TeX, I still find its typesetting to be aesthetically pleasing; to Christine Kamper, Renate Weiss and Margret Steinbuch as well as Alexander Knoll, all from the Distributed Systems Group, for providing a constant flow of coffee and network packets; to the Pharmacoinformatics group at the University of Vienna, who taught me so much about scientific methodology; and to all the others who have helped me accomplishing this last step of my studies.

My deepest gratitude deserve the ABW symphonic wind orchestra for always putting music into my head; Bettina for all those helpful discussions; Thomas and Fabian for the countless coding and burger sessions during our Master's; my parents for their years of support; my sister Andrea for reminding me to leave the ivory tower from time to time; and finally, Maria-Christina, for her patience and support during these last months.

Kurzfassung

Datenstromverarbeitung (engl. “*Stream Processing*”) ist eine Methode zur Verarbeitung und Aggregation von Datenströmen, um neue Informationen zu erschließen. Darauf aufbauende Applikationen (engl. “*Stream Processing Applications*”; SPAs), die dazu verwendet werden, Datenströme zu analysieren, sind in der Regel als verteilte Systeme konzipiert. Treten in solchen Systemen Fehler oder Kommunikationsstörungen auf, werden Ausfallsicherheitsmechanismen eingesetzt, um einen unterbrechungsfreien Betrieb zu gewährleisten. Aufgrund der Nahezu-Echtzeit-Anforderungen von SPAs müssen solche im Fehlerfall eine Balance zwischen Konsistenz (d. h. korrekte Ergebnisse zu produzieren) und Verfügbarkeit (d. h. diese Ergebnisse schnell genug zu produzieren) finden, da beides zusammen nicht gleichzeitig erreicht werden kann.

Redunanz ist das zentrale Element von Ausfallsicherheit. Bestehende Ansätze aus der Literatur ermöglichen Redunanz, indem sie Operatoren (die Bausteine von SPAs) replizieren. Dies greift zu kurz, weshalb wir ein neues Ausfallsicherheitsmodell präsentieren, das auf einer höheren Abstraktionsebene angesiedelt ist und auf funktionaler Redundanz auf der Ebene von Pfaden (Abfolgen von Operatoren) beruht.

Basierend auf einem konkreten Szenario aus der Praxis identifizieren wir Anforderungen an *Pathfinder*, einem neuen Ausfallsicherheitsframework, und evaluieren es anhand eines Beispiels aus der Praxis. Pathfinder verbessert die Schwächen bisheriger Ansätze, indem SPA-Entwicklern ermöglicht wird, funktionale Redundanz zu spezifizieren. Zur Laufzeit reagiert Pathfinder auf Defekte, indem auf einen fehlerfreien Alternativpfad gewechselt wird, der eine ähnliche Funktionalität bietet. Ein Schutzschalter-ähnlicher Mechanismus dient schließlich dazu, auf geordnete Weise wieder auf den Hauptpfad zurückzukehren, sobald der Defekt behoben wurde.

Im direkten Vergleich mit einer vollständig redundanten Replizierung ist es mit unserer Lösung möglich, etwa 30% der Betriebskosten zu sparen, während eine vergleichbare Verfügbarkeit erzielt werden kann.

Abschließend wird durch einige Experimente verdeutlicht, dass die Mechanismen zur Fehlerdetektion und Ausfallsicherheit wie erwartet funktionieren und nur einen geringen Mehraufwand hinsichtlich der Performance verursachen.

Abstract

Stream processing is a practice where continuous data streams are processed and aggregated in near real-time, ultimately resulting in the discovery of new information. Stream processing applications (SPAs) are used to analyse data streams and are often deployed in a distributed manner for performance reasons. When faced with partial failures or network communication outages, fault tolerance mechanisms must ensure a continuous operation. Due to the near-real-time requirements, these mechanisms have to balance the need for consistency (i.e., producing correct results) and availability (i.e., producing results fast enough) in case of failures since fulfilling both at the same time is impossible.

The key concept of fault tolerance is redundancy. Existing fault tolerance approaches for SPAs implement redundancy by replicating operators, the building blocks of an SPA. We argue that this approach is not sufficient and present a novel fault tolerance model which focuses on functional redundancy on the level of paths (sequences of operators).

Based on a concrete motivational scenario, we identify requirements of *Pathfinder*, our new fault tolerance framework, and evaluate it based on our motivational scenario. Pathfinder addresses the shortcomings of existing approaches by allowing SPA developers to specify functional redundancy. At runtime, Pathfinder reacts to faults by switching to a fault-free path with a similar functionality. To restore the main path once the failed operator has recovered, Pathfinder uses the circuit breaker pattern which has been proven in the domain of microservices.

By comparing our approach to a fully redundant replication, we show that 30% of total operational costs can be saved while achieving a similar level of availability.

Finally, several experiments show that Pathfinder's failure detection and fault tolerance mechanisms are working as expected and only add a minimal performance overhead.

Contents

CHAPTER 1

Introduction

The *Internet of Things* (IoT) — a paradigm where everyday objects are adapted to communicate with each other and the outside world — is past its initial hype and about to revolutionise the way humans interact with cities, transportation vehicles, infrastructure, or commodity objects [1]. With the ever-growing number of connected devices, the amount of generated data explodes as well, leading towards what is known as *big data*: data sets too big for traditional solutions to handle [2]. In addition, vast amounts of data on human individuals are collected by smartphones and wearable devices such as smartwatches that are readily adopted by customers [3]. The rise of these devices does not only increase the total data volume but in many cases also provides a data series of updates, allowing, e.g., the constant monitoring of human activities including their location, health status, communication patterns and more.

This work focuses on stream processing technologies that have been developed to analyse, transform and process data from such sources and in particular some of their associated technological challenges.

1.1 Problem Statement

We consider a concrete scenario from the advertisement domain where the analysis of geographical user activity data is used to maximise the revenue of advertisement (ad) campaigns (a more detailed description of the scenario is provided in Section 1.2.2). Each participant of a social network continuously sends their location to a company's analytics system. By analysing a user's location in combination with previously acquired personal data, only the most relevant ad campaigns should be displayed to the users.

Gathering and analysing such data series, i.e., streams from millions of users, is a non-trivial task and poses significant technological hurdles (e.g., space and memory limits, latency and bandwidth constraints, data heterogeneity and fault tolerance [4]). While this scenario might as well be tackled using traditional software architectures, the nature of continuously produced data flowing through several transformation steps requires a more adequate solution. *Stream processing* refers to the paradigm of processing such continuous data streams in near real-time and offers solutions to the aforementioned hurdles (for details, see Section 2.2). *Stream processing applications* (SPAs) are basically software systems where data items flow through several building blocks named *operators*. A stream processing *topology* is a graph that describes how the operators are connected. Multiple operators that are directly connected to each other are called a *path*. The software that is used to execute an SPA is called a *Stream Processing Engine* (SPE). It takes care of aspects such as guaranteed message delivery, scalability and fault tolerance [4]. All definitions are discussed in more detail in Chapter 4.1.

Stream processing at big data scale is often implemented in a distributed manner where multiple, often heterogeneous computing nodes are involved in order to distribute the computational burden to multiple machines in a parallel way. *Distributed SPEs* (DSPEs) such as VISP [5] or Apache Storm [6] have been developed to run such SPAs.

DSPEs depend on a potentially large number of operators that perform the actual processing. Each of these operators can fail at any time or become unreachable due to network problems. Since this could, in the worst case, cause the whole SPA to become unavailable, employing fault tolerance mechanisms is one of the most important features for DSPEs.

Any system composed of multiple components is said to be *fault-tolerant* (see Section 2.5) if that system continues to deliver a correct service even in the presence of faults (such as a malfunctioning hard drive) [7]. Fault tolerance for distributed systems in general and specifically for DSPEs has already received a lot of attention [8,9,10,11]. Nevertheless, it is still an active research topic.

The key concept of fault tolerance is *redundancy* [12]. Physical redundancy (which is considered in this work) basically translates to having multiple components that provide the same functionality. If one of them fails, the other(s) can take over and the system remains operable. This principle can be applied at many different levels. In the motivational scenario, there could be multiple hosts that execute the same program. In case one of them crashes (e.g., due to a hard disk error), the others can take over. However, there are cases where this is not sufficient. For example, the program could contain a bug causing it to crash when faced with some specific input or the license period of a commercial software package could have ended. In such cases, all hosts executing the same program would be unavailable at the same time, despite the redundancy on the hardware level.

To overcome this problem, redundancy can be implemented on the software level as well. One could develop multiple software components that provide the same functionality with different implementations. These can be developed by different teams in different programming languages. The idea here is that it is unlikely that both

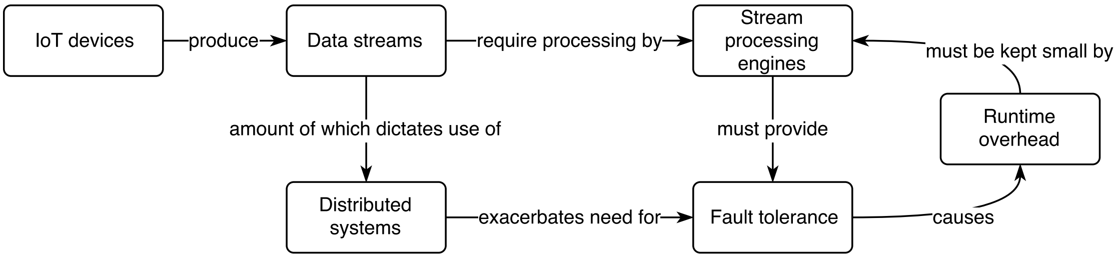


Figure 1.1: At a glance: Relation of the different concepts connected to fault tolerance in SPEs. The large number of IoT devices and similar data sources leads to an increase in both quantity and velocity of data streams. SPEs that process such streams therefore need to operate in a distributed way to handle such large computational burdens. Fault tolerance is a bigger issue in distributed systems and must be properly handled by DSPEs. This however conflicts with the goal of providing processing results in near-real-time. This thesis focuses on resolving this conflict.

implementations would contain the same bugs and at least one of the implementations would give correct results. This technique is known as *n-version programming* [13] and has the obvious disadvantage of a very costly development as well as a significant runtime overhead. Furthermore, if external APIs are used in the processing steps that are billed per invocation, the continuous operation of redundant copies will also result in high costs.

Example 1.1

As an example, consider an operator that internally invokes a third-party API. If that API becomes unavailable, the operator becomes unavailable as well. This lack of availability would not change even if the DSPE were to deploy a second redundant copy of that operator since that copy would also need to invoke the same (unavailable) API.

In summary (as depicted in Figure 1.1), the requirements of near-real-time processing combined with extensive amounts of data dictate the use of distributed systems in stream processing due to the large computational burden that cannot be taken upon one single machine. Established approaches towards fault tolerance in DSPEs are unsatisfactory and there is the need for an easy and cost-efficient solution to cope with operator faults in such contexts.

1.2 Use Cases

Many real-world use cases profit from the advancement of stream processing engines. This section introduces some of them, before discussing a single use case in more detail which serves as a motivational example throughout this thesis.

1.2.1 Areas of Application

- **eHealth:** Health care processes can benefit from near-real-time monitoring of health parameters. In nursing, correlating motion sensor data with a heart rate monitor and a gyroscope embedded in wearable devices might help detecting dangerous falls [14]. Diagnostic procedures can also include automatically collected health data like sleep patterns, nutrition, environmental influences or medication [15].

- **Industry:** In modern industry factories, machines are expected to constantly communicate with each other. A wide array of sensors allows analytics processes to support decision-making and improve the overall productivity [16]. Fault-tolerant stream processing systems play a vital role in the success of such factories since even a few minutes of unavailability could result in expensive downtimes.

- **Military:** Using wireless sensors in battlefields to track the movement of military vehicles and soldiers can be crucial to avoid enemy attacks or ambushes. A high mobility in combination with decentralised processing (and in this case, adding heavy weaponry and explosions to the list of possible causes of hardware faults) requires sophisticated fault tolerance mechanisms [17].

- **Information security:** Intrusion detection systems based on stream processing may protect company networks from computer-related criminal activities by analysing data streams from a wide array of sources such as network traffic, shell commands and system calls to detect unauthorised activities [18].

- **Financial industry:** High-speed trading and stock analysis naturally requires highly performing real-time systems in order to outperform competitors [19]. Stream processing systems can be useful to detect circumstances that may lead to stock price changes in the near future (e.g., political events that may affect the price of oil).

1.2.2 Motivational Scenario

This section introduces our motivational scenario. It serves multiples purposes. First, the framework's functional requirements are derived on the basis of this example (see Section 4.2). Second, it motivates the development of a fault tolerance framework for SPAs by highlighting the shortcomings of established fault tolerance solutions. The scenario is based on a real-world scenario [20]. An overview of the scenario is presented in Figure 1.2.

Description

Consider the following scenario from the advertisement domain. A fictional company named *Lifeinvader* allows registered users to connect with friends, share content with them and stay in contact via a text-based chat.

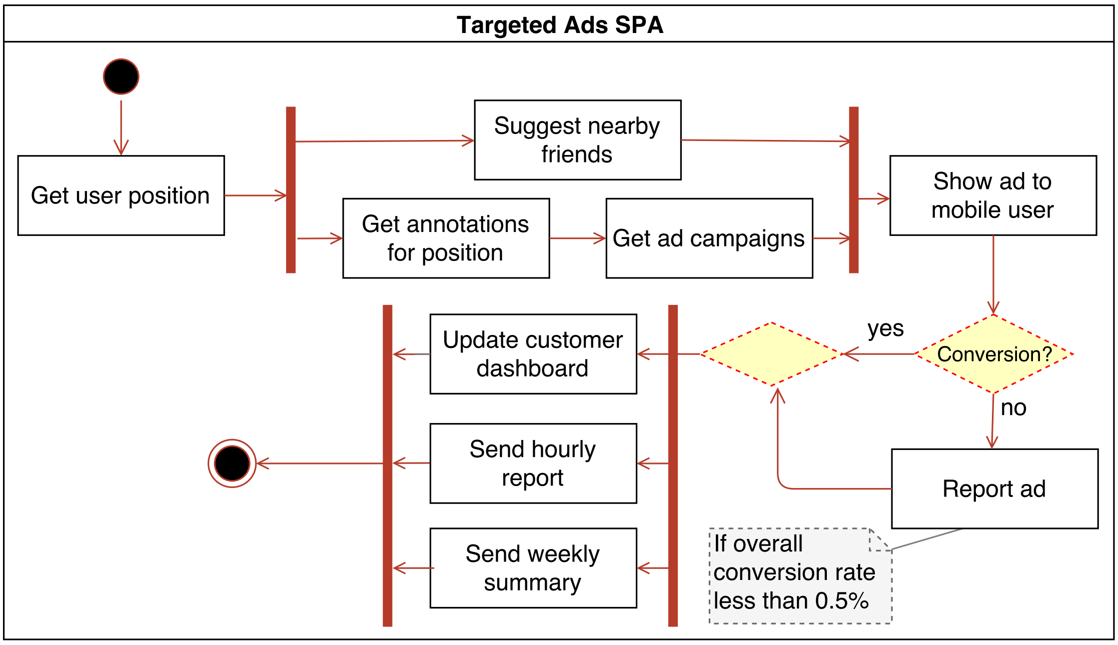


Figure 1.2: Targeted Ads SPA. This activity diagram presents an overview of the motivational scenario. After the user's location data is collected, the SPA has to suggest nearby friends and find suitable ads for that user. If an ad's overall conversion rate is less than 0.5%, it is automatically reported. Finally, the appropriate updates to the dashboard and the contributions to the hourly and weekly reports are computed.

One of Lifeinvader's major sources of revenue is targeted ads that are shown to their users on their mobile phones. Any company who wants to launch an ad campaign can specify a particular focus group (e.g., "males aged 18–25 in Western Europe who are interested in video games") and Lifeinvader will show the ad to that group as well as track the users' interactions with the ad. Traditionally, users are tracked via their browsing patterns [21].

In order to improve customer satisfaction[1], Lifeinvader decides to start a new project. By tracking each user's location via their mobile phone's GPS or with indoor positioning systems (IPS), ads ought to become even more targeted. A user travelling to a hospital to visit a relative in the oncology department might be shown ads from the local gift shop or for over-the-counter drugs to quit smoking.

Furthermore, Lifeinvader wants their users to spend more time interacting with their mobile phone app in order to make their platform more attractive for new ad campaigns. A mechanism that suggests friends based on the time spent simultaneously at the same location should be implemented to reach this goal.

[1] The customer being the party that is willing to pay for a service, i.e., the businesses launching ad campaigns — in contrast to the social network user, whose data is the product.

In a board meeting discussing these ideas, the Lifeinvader CTO suggests to design an SPA. Since Lifeinvader already has more than 500 million active users around the world, no centralised system could bear the resulting load.

While user activities are expected to vary throughout the day (people sleeping at night and being active during the day) and throughout the week (different peak times at workdays and weekends), exceptional peak loads are expected to be caused by sport events, holiday seasons or social conventions. Furthermore, the computational effort of computing friend suggestions increases exponentially with the number of people having a common location at the same time (since for each additional person to join a location, relations to all the other users in proximity have to be checked).

A very critical aspect of the whole endeavour is the mapping of GPS locations to different ad targets. Directly sending the GPS position to the businesses launching ad campaigns would be a privacy violation. The cartography startup *Atlas* offers an artificial intelligence based service that maps positions to a set of semantic annotations. For example, it maps the location of a football stadium to the keywords "football", "sports", "beer", "hot dog" and "shoes". Lifeinvader negotiates a contract that allows them to use the API for the next 6 months in order to evaluate the service. As a backup mechanism, they still have a database of cached data containing the locations of important companies and public areas that is running on their infrastructure. Finally, Lifeinvader has its own system of letting users assign tags to places they have visited. It solely relies on user contributions and has therefore a rather low accuracy. Also, there are many places that have not yet been tagged.

In order to detect ad campaigns that might disturb users, ads with a conversion rate of less than 0.5% should automatically be reported.

Lifeinvader's marketing department is instructed to analyse the revenue generated by the new project. Therefore, it needs to be informed about each user conversion (i.e., users watching and/or clicking the ad) including information about the ad campaign, the user profile, their location and the current time. It is settled that marketing will receive a full statistics report once a week. To generate the weekly report, a junior developer is assigned to develop an automated analysis script. From past experience, it is known that such a task is quite error-prone, which is why they decide to use a commercial statistics API in addition to the in-house approach. Considering all needed functions, it is estimated that it would cost about 600 EUR to generate a full report using that API. In both approaches, a correlation analysis is performed to determine the factors that caused high conversion rates depending on the stored information about the user.

For their customers, Lifeinvader provides a real-time dashboard where they can observe the success of their current ad campaigns. To support this dashboard, each user interaction must be processed and visualised within one minute. By contract, Lifeinvader guarantees a maximum downtime of one hour per month for the dashboard and agrees to pay a penalty fee of 500 EUR for each additional hour of unavailability.

SPA Requirements

Based on the scenario description, one can derive a number of requirements the newly developed SPA must fulfil.

R1 **A DSPE forms the architectural basis of the project.** Since the number of users is easily imaginable to reach millions (not every social network user provides position data), no centralised system can be used. Also, the need for a rapid software development requires an environment that allows a fast and straightforward way to implement the required functionality where developers do not have to deal with low-level details.

R2 **Availability is more important than consistency.** Assuming users react to the shown ads, it is crucial to deliver the right ads at the right point in time. Once a user has left a particular location, showing them ads for that location will not result in a successful conversion. In contrast, it is not considered harmful if unrelated ads are shown once in a while[2].

R3 **Scalability is required.** Since user activities vary throughout the day as well as throughout the week, only a cloud-based solution is expected to deliver an adequate performance at peak loads without having to overprovision resources at low loads.

R4 **Geographical effects are expected.** Both user activity and ad campaigns are expected to show geographical patterns. The SPA is required to recognise these patterns and optimise the placement of operators such that latency and network communication are minimised.

R5 **Available redundancy must be exploited.** The task of mapping GPS locations to the distance to different ad targets can be accomplished in three different ways: (1) using a commercial third-party API, (2) using a local database and (3) using user-generated content. By incorporating all three methods into the system, a high degree of redundancy can be achieved. A similar level of redundancy is also available for the data analytics report. The SPA is required to make use of these redundancies while also considering the different invocation costs and levels of data quality in the different methods.

R6 **Minimise downtimes.** By contract, each hour of downtime requires Lifeinvader to pay a penalty fee. Therefore, the company has a strong incentive to operate a highly-available software.

R7 **Suitable results must be produced.** The basic functional requirement of the SPA is that suitable ads are shown to the users since this generates revenue for Lifeinvader.

[2] With the exception of adult content reaching underaged users which must be filtered in any case.

Requirements R1, R3 and R4 show the need for using a cloud-based DSPE in order to provide the necessary elasticity. R2 and R6 demand sophisticated fault tolerance mechanisms that minimise total application downtime and favour availability over consistency. Finally, R5 is the main reason why established fault tolerance mechanisms are not adequate for this scenario. In order to avoid downtime in the face of third-party services like "Atlas" becoming unreachable, redundancy at the operator level is futile because the fault is external to the controlled environment. Furthermore, the available in-house services (such as the database of company locations) cannot be effectively utilised by n-version programming of individual operators because the end result depends on the consecutive data processing of multiple operators in a path (in addition to the potentially high operational costs of pay-per-use APIs discussed earlier).

1.3 Research Questions

The following research questions should be addressed in this work:

RQ1: Which fault tolerance mechanisms for stream processing systems are known and how can they be applied to the motivational scenario?

In order to answer this question, a thorough literature review is conducted. Selected concepts and their applicability to the motivational scenario are then discussed. Gaps arising with the respective solutions are pinpointed as well. The research focuses on mechanisms with the following properties: a) acting at runtime and not requiring a restart of the DSPE, b) supporting fallback mechanisms to be defined at design-time, c) being able to determine from the deployment context when to assume an operator is experiencing a fault and d) working on the level of processing paths, thereby utilising redundancy.

RQ2: How can data streams be processed in a fault tolerant manner while still guaranteeing a high availability?

Inspired by existing solutions, a new framework named *Pathfinder* is designed and implemented that addresses the requirements of the motivational scenario and surpasses existing solutions.

RQ3: How does the framework perform in terms of applicability and performance?

The Pathfinder system design developed for RQ2 is evaluated in a qualitative and quantitative manner using simulations, case studies and different kinds of performance measurements such as time-to-recover and runtime overhead. The real-world usage is evaluated by integrating Pathfinder into the VISP ecosystem [5]. VISP's capability to dynamically change the processing topology at runtime is crucial to developing a framework with the envisioned requirements.

1.4 Methodology

The methodological approach of this research consists of the following four phases.

1. **Literature research**. In order for this thesis to have a strong theoretical foundation, a review of introductory and related work in the fields of distributed stream processing and fault tolerance is conducted.
2. **Design**. Based on the initial concepts and insights gained during the literature research, the design for the Pathfinder framework (as described in RQ2) is created in this phase. Another outcome of this phase is a set of requirements that needs to be fulfilled by the implementation.
3. **Implementation**. A prototypical implementation of Pathfinder is created in this phase. The choice of the exact programming language, used tools, frameworks and libraries is made according to the previous phases' decisions.
4. **Evaluation**. The evaluation phase consists of two parts. First, Pathfinder is tested in isolation by setting up an artificial testbed and running simulations. Second, the VISP ecosystem is used to evaluate the performance of Pathfinder under realistic conditions.

1.5 Structure

The remainder of this thesis is structured as follows:

- **Chapter 2** provides background information about the fundamental concepts this thesis is based on. The covered subjects include stream processing, service composition, microservice architectures, fault tolerance and the circuit breaker pattern.
- **Chapter 3** focuses on state-of-the-art solutions to the problem of fault tolerance in DSPEs and related areas. By comparing their features and shortcomings, the need for the development of a new framework is motivated.
- **Chapter 4** introduces a terminology based on mathematical concepts from graph theory to provide proper definitions for the required stream processing concepts as well as a set of requirements for the framework to be developed. The remaining chapter is devoted to the architectural design of the Pathfinder framework and introduces the paradigm of *fault tolerance*-oriented development.
- **Chapter 5** discusses a prototypical implementation of Pathfinder and its integration with VISP.

- **Chapter 6** focuses on the evaluation of the framework both by quantitative measurements as well as by a qualitative case study. It then discusses the evaluation results and how well Pathfinder is able to fulfil the requirements derived in Chapter 4.
- **Chapter 7** finally concludes the outcomes of this thesis and outlines plans for future research.

CHAPTER 2

Background

This chapter provides an introduction on the fundamental concepts used in this thesis. First, the need for stream processing technologies is motivated. The concept of stream processing is then introduced from a historical perspective while also presenting technological advances that have lead to today's modern SPEs. Furthermore, an analogy from the choreography of stream processing operators to a similar endeavour in service composition and microservice architectures leads to the concept of fault tolerance. This topic is discussed regarding centralised as well as distributed systems. Then, two sections present fault tolerance in stream processing systems and the circuit breaker pattern, the concepts that form the basis of this thesis. Finally, message-oriented middlewares and VISP are discussed to familiarise the reader with the technical basis for the implementation of our fault tolerance framework.

2.1 Big Data and the Internet of Things

The Internet is constantly evolving and the amount of data produced per day is rapidly increasing. With ubiquitous Internet access provided by wireless and cellular networks, the number of connected devices increases massively as well. However, it is not only the number of human participants but also the growth in connected machines that causes this expansion [1]. The vision of direct machine-to-machine communication over the Internet drives the development of the Internet of Things (IoT), a new paradigm where the idea of a global system of interconnected devices that is the Internet is extended to include all kinds of commodity objects [22]. By equipping *things* like cars, watches, industrial machines or furniture objects with network interfaces, they can interact with each other and of course with their human users. While machine-to-machine communication is nothing new per se (after all, each router and server is a machine), it is the inclusion of commodity machines that were *not* originally intended to be connected to it that makes the IoT revolutionary [1].

Big data is the term frequently used to describe the kind of data that is generated by IoT devices in an overwhelming velocity and amount. So called *traditional* methods of data processing are no longer applicable to such volumes of data due to a lack of space, computing power and time. There is no hard limit for when to consider a set of data items as *big data*. Instead, the literature uses a set of characteristics for its definition [23]:

1. **Volume**. The quantity of generated data and its size.

   Example 2.1

   A CSV file of ten Terabyte poses great difficulties due to its sheer size alone.

2. **Variety**. The types and sources of the collected data. They may be structured (each data item having the same set of attributes) or unstructured.

   Example 2.2

   A relational database is always structured since the number and type of a data item's attribute is fixed. A document-oriented database such as MongoDB[a] however can contain data items with different attributes or even no obvious internal structure at all.

   [a] `https://www.mongodb.com`

3. **Velocity**. The speed of data generation.

   Example 2.3

   Network traffic analysis in a large company network has to deal with all packets that are generated by all the company users combined. While the total data volume might be manageable, the speed of data generation is challenging.

4. **Veracity**. The uncertainty due to a varying data quality.

   Example 2.4

   The results of big data analyses are only as good as the quality of the input data. Data quality mainly depends on the data source: internal enterprise data that is stored in a standardised way is expected to have a higher quality than data extracted from social media sources.

5. **Value**. The presence of a (hidden) value in the data set that is to be discovered by the analysis.

   Example 2.5

   A telecommunications provider analyses the interactions between their customers. The purpose of this task is to get insights into otherwise hidden relationships (namely, their behaviour).

Data sets meeting the above criteria to varying extents can be considered as being *big data*.

Example 2.6

According to prevailing opinion, examples for big data include: experimental data from the Large Hadron Collider [24]; security camera footage [24]; or experimental genomics data sets [25].

In many areas of applications related to big data, the importance and value of data items decreases rapidly the longer it takes to process them [26].

Example 2.7

Sensor data indicating car movements that originate from several locations in a smart city can be very useful to guide traffic lights. However, once a few minutes have passed, the actual traffic situation may already look completely different and the analysis of the sensor data has become obsolete. Analysing and reacting to the collected data is therefore a very time-critical endeavour.

An important classification of big data can be made into two categories: batch and real-time processing [27]. In *batch* processing, the whole data set is available at the beginning and the analytics system can access it in any arbitrary way. In *real-time* processing, the data is gathered piece by piece during the analysis. This work focuses on the latter class of big data and especially on the software that is used to process such real-time streams of data.

While the architecture of IoT applications tends to vary considerably, one common pattern is the *observe, orient, decide, act* (OODA) loop as illustrated in Figure 2.1 [28]. This model originates from the military but is nowadays used to describe many phenomena. Its application to the IoT domain requires an IoT system to have various sensors that *observe* something. The sensor data is then sent to an analytics system that aggregates all input data (*orientation*) and comes up with a *decision*. Finally, the decision result is realised by *interacting* with the real-world.

Example 2.8

A temperature sensor *observes* the room temperature and sends this information to an analytics system. This system *orients* itself by aggregating the temperature data and comparing the temperature to reference values. If certain thresholds are met, a *decision* is made to increase the temperature in a certain room. It then *acts* and turns this decision into reality by turning on a heater or switching it off.

While the IoT is certainly an important driving force, stream processing is not restricted to it (see Section 1.2). However, with most IoT devices being able to transmit continuous data streams and geographic considerations getting more important, the number of IoT use cases that allow an integration into an SPA is abundant.

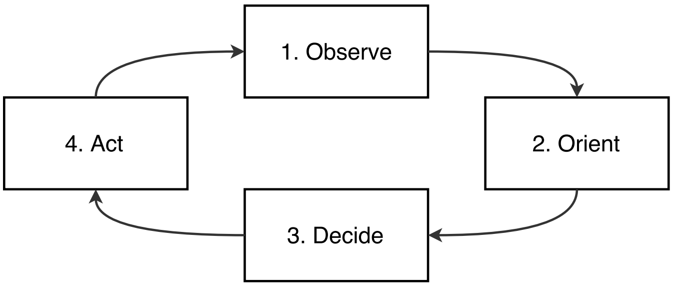


Figure 2.1: **The OODA loop**.

2.2 Stream Processing

There is a wide array of data sources that produce continuous data streams (as motivated in Section 2.1).

Example 2.9

Users adding content to social media platforms [29] and machine sensors [30] are examples for the production of real-time data.

Properly defining stream processing and differentiating it from similar technologies is not straightforward and is discussed in Section 2.2.4 in more detail. We initially consider the following definition from Andrade et al. [4]:

> *"The fundamental goal of stream processing is to process live data in a fully integrated fashion, providing real-time information and results to consumers and end-users, while monitoring and aggregating new information for supporting medium- and long-term decision-making."*

With this definition in mind and having discussed the motivation for such technologies, the next section introduces stream processing from a historical context.

As an orientation, Figure 2.2 shows an overview of the most important entities in stream processing and their relations to each other. The shown entities are successively explained in this chapter.

2.2.1 Historical Development

Early stream processing systems were not as sophisticated as the systems in use today. The original intent, the processing of large amounts of data in real time, was implemented using the following systems as classified by Andrade et al. [4].

Traditional databases are filled with data by one application and used by others. It is easy to imagine how one could implement an SPA by having several "operator" applications

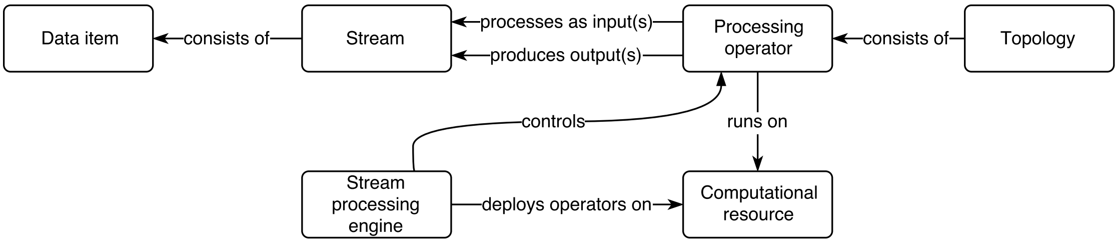


Figure 2.2: Entities in stream processing. Data streams consist of continuously produced data items of the same type. Operators use one or more streams as an input and produce one or more output streams. The set of all operators with their connections is known as the topology. An SPE controls the operators and their deployment to the available computational resources.

that continuously query a central database for new entries and process them. This "store-then-process" model however does not scale very well and is not suitable for high-throughput and low-latency applications [31].

Active databases (e.g., *Snoop* [32]) have been developed by adding features to conventional database systems in order to make them more suitable for stream processing. The basic idea is to manually define actions that are automatically triggered when a specific condition occurs. Each change to the database would trigger this mechanism, allowing a direct reaction to such changes. Certain active database systems (e.g., HiPac [33]) only allow database-internal changes to trigger actions (*closed*) whereas others (e.g., Snoop [32]) also allow external events (*open*). Since active databases still depend on the underlying database architecture, they are not able to handle data sources generating huge amounts of data with a high velocity due to the *store then process* nature of databases. Also, the programming model is limited and software developers are restricted in designing their application code [4,32].

Continuous Query (CQ) systems (e.g., NiagaraCQ [34]) can be seen as a special type of databases where persistent queries act on a continuously changing set of data [4,35]. Users can specify a query and the CQ system will provide the results as a continuous stream, thereby changing the traditional "pull" model of databases to a "push"-based mechanism.

Publish-Subscribe systems allow data consumers to subscribe to topics of interest [4,36]. Data producers then publish data to a *broker network* that decides which messages are routed to which subscribers, thereby decoupling the sender and the receiver from each other. The algorithms used by the broker network can be very sophisticated and allow subscriptions to depend on the evaluation of rules on the published data.

Example 2.10

A producer from a phone company may publish a data item for each phone conversation occurring in its network. A consumer can specify a subscription for all publications with a conversation duration of over 10 minutes where the participants are separated by at least 10 km. It is then the responsibility of the broker network to forward only those data items to the consumer that fulfil these conditions.

Modern stream processing systems share the use of messages to represent data items (in contrast to databases for example, where data is represented as static tuples). However, publish-subscribe systems are typically deployed on high-performance computer systems with high-speed connections where communication and fault tolerance is less of an issue because tendentially, the infrastructure is more homogenic and the network is more stable. Additionally, there is another difference between the two approaches: In stream processing, the main "actors" are operators and the data items passively flow through them. In publish-subscribe systems on the other hand, the messages are the important part and the surrounding infrastructure is just the means to an end [4].

Finally, *complex event processing* systems such as Esper[1] and Oracle CEP[2] have been developed specifically for the *detection* of complex events based on rules and patterns [4]. They natively support analysis of temporal aspects (e.g., event A was produced *after* event B). In contrast to stream processing and the other techniques investigated so far, these systems lack the ability to actively modify or transform the incoming data. Thus, they are mainly used for the detection of events and patterns.

2.2.2 Towards Stream Processing

Having discussed the predecessors of modern stream processing, this section deals with the evolution that finally resulted in today's DSPEs. In general, the two areas these newer frameworks excel at are *performance* (resulting from a distributed deployment) and *fault tolerance* (the importance of which directly results from that distributed deployment) [4].

Modern stream processing systems include for example Aurora [17], Borealis [37], Apache Storm [6], Apache Spark [38] and VISP [39]. Instead of going into the details of each of those systems, this section discusses their common features.

Modern SPEs allow sophisticated *data models*. Unlike database-inspired technologies such as CQ-systems, data is explicitly modelled as streams. A *stream* is an infinite list of data items of the same type (similar to object types in high-level languages such as Java). Additionally to all items of a stream having the same type, they are also *ordered* by having some kind of increasing index number (such as timestamps or sequence numbers) [4].

[1] http://www.espertech.com/products/esper.php

[2] http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html

Once a data item has entered the SPE, it is processed by a sequence of *operators*. Operators are the primary building blocks of an SPA and they can apply arbitrary functions to data items (a more detailed definition follows in Chapter 4.1).

An important mechanism in DSPEs is data parallelism. This means that different data items can be processed in parallel by different instances of the same operator. This allows DSPEs to scale efficiently by partitioning the data items to different operator instances.

Example 2.11

An operator is deployed to four virtual machines. Each incoming data item is first sent to a queue that can be accessed by all four operators. The four operators take items from the queue and process them. Each individual data item is processed only once by one of the four operators.

Finally, in contrast to related technologies such as service composition, there are usually stringent temporal requirements for the operators. In the long run, the rate of incoming data must not be higher than the rate of data processing, otherwise the input queues will overflow.

Example 2.12

A temperature sensor transmits a new data item once per second. The processing operators take 10 seconds to analyse each incoming data item and will inevitably be too slow in the long run which results in an overflow on the queue.

2.2.3 Distribution-Related Aspects

SPEs are inherently distributed since no centralised system can provide the computational power necessary to process today's data streams [31].

Operator Deployment

Early publications in the field of stream processing already anticipated that it would be beneficial if the way users interact with DSPEs was not very different compared to non-distributed systems [4]. In other words, it is the responsibility of the DSPE to transparently deploy operators and route data items depending on the available infrastructure [30].

As a consequence, many DSPEs (e.g., VISP and Apache Storm) distinguish two views of a topology [4]. The *logical view* (usually created manually by the SPA developer) shows the connections between operators. The *physical view* maps operators to specific hosts. A single operator in the logical view can consist of multiple operator instances on different hosts in the physical view (if the operator is replicated). The data flow between two operators in the logical view can either be shown between two different hosts or on the same host in the physical view (depending on whether the two operators are

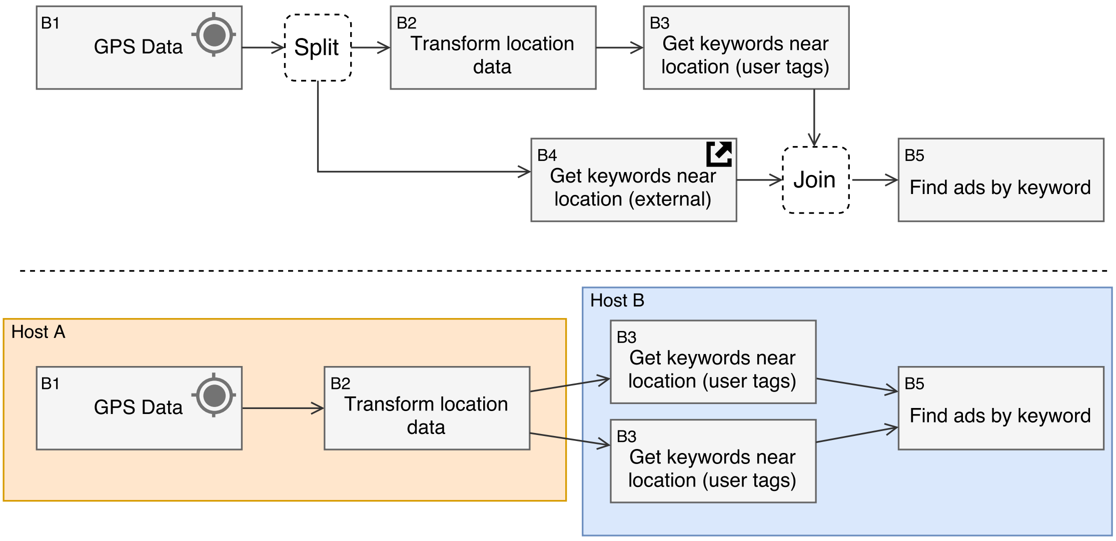

B1
GPS Data
Split
B2
Transform location data
B3
Get keywords near location (user tags)
B4
Get keywords near location (external)
Join
B5
Find ads by keyword
Host A
B1
GPS Data
B2
Transform location data
Host B
B3
Get keywords near location (user tags)
B3
Get keywords near location (user tags)
B5
Find ads by keyword

Figure 2.3: **Logical and physical view of an SPA**. The upper part of the figure shows the logical view, the lower part the physical view.

deployed on the same host or not). Figure 2.3 depicts both the logical and the physical view of an SPA. The physical view contains the additional information that operator B3 is deployed in two instances and shows which operator is deployed on which host. The logical view instead shows alternative fallback operators that could be used in case of operator failure (see Section 4.1.5 for the *split/join* notation).

Both views are useful and deserve to coexist. An SPA developer who does not care about the distributed deployment may favour the logical view, while a DevOps engineer who needs to find a communication bottleneck will rather depend on the physical view.

The initial deployment of operators to a set of hosts is called *placement* [4]. A simple placement algorithm might just randomly assign operators to hosts, while a more sophisticated one can base this decision on host capacity, communication latency or bandwidth. *Dynamic placement* can react to runtime changes in either the set of available hosts, performance indicators of individual operators or the rate of input data production in order to optimise the overall resource consumption or QoS aspects.

Scalability

One central requirement of distributed systems in general, but especially for DSPEs is scalability [4]. Due to the increasing prevalence of cloud computing, it is easier to make applications scale. The term *elasticity* refers to the ability of a software system to adapt its utilisation of resources to changing requirements in a rapid manner. In stream processing, elasticity most importantly translates to the dynamic adaptation of computational resource usage [40]. Since operators can be replicated, scaling is straightforward and can be fully automated using a state-of-the-art cloud infrastructure such as OpenStack [41] or Amazon's Elastic Compute Cloud (Amazon EC2) [42].

Message Flow

A DSPE — by definition — runs on multiple hosts that communicate over a network. It is the task of a DSPE to automatically take care of the deployment but also to set up the communication infrastructure between those communicating hosts. For example, VISP is based on a RabbitMQ messaging infrastructure (as discussed in Section 2.7). Therefore, VISP needs to take care of that all the queues and exchanges are set up appropriately.

Some SPLs (e.g., VTDL, see Section 2.2.5) allow the topology designer to place constraints on the placement of operators. This may be useful if two operators are to be deployed on the same host (e.g., because they require large amounts of data exchange), on a specific host (e.g., due to special purpose hardware only available at that host) or because of legal reasons.

Runtime Migration

By using elastic computational resources, DSPEs have many degrees of freedom regarding the deployment of individual operators. Due to dynamic changes (e.g., in user activity, traffic volume or resource availability), a DSPE can decide at runtime to migrate operators between different (cloud) resources. Ottenwälder et al. [43] present a placement and migration method for such use cases that also consider the migration costs.

2.2.4 Related Technologies

There are many technologies related to stream processing in addition to the ones that historically paved its way as depicted in Figure 2.4. *Real-time* systems include all systems that have real-time constraints (not restricted to the processing of data streams). In *big data batch processing*, there are no real-time requirements at all but instead, the challenge is to handle huge amounts of data. *Business process management* systems are also similar to SPAs because their application structure also consists of smaller modules like operators [44]. While they in general do not tolerate the loss of data, they also have less severe real-time restrictions. Figure 2.5 shows the same technologies on the two axes *amount of data* and *real-time requirements*. Stream processing tends to share the data volume of big data batch processing with real-time requirements that are between those of real-time systems and service composition.

It is also important to realise that stream processing is not a direct competitor to the mentioned technologies in this section but rather tries to complement their shortcomings. A software system is most likely still having relational databases in addition to a streaming part of their application [4].

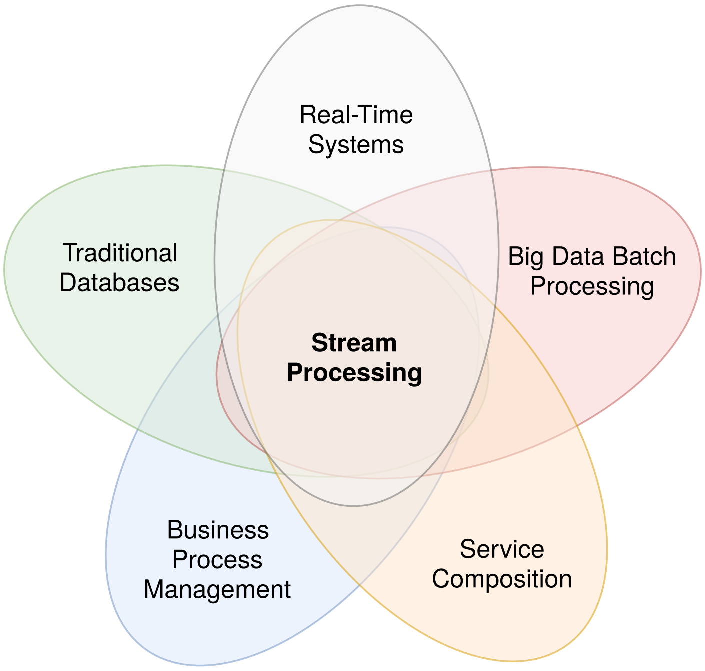


Figure 2.4: **Stream processing — Related technologies**.

2.2.5 Stream Processing Languages

Developing SPAs can be quite challenging using traditional programming languages. Since it is very cumbersome to express fault tolerance, distributed communication and parallel programming directly at the operator level, several languages have been developed to make this task easier. Their ultimate goal is that the developer does not have to worry about *how* and *where* their code is executed [4].

Most SPLs share common features that altogether separate them from other programming languages [4]. Since SPAs are composed of operators, the first commonality is *composability* [4]. This means it must be possible to separate an application into multiple operators and connect them in a straightforward way because a distributed deployment is not possible otherwise.

Example 2.13

Instead of developing a single Java application that would fetch data items, process and analyse them and generate a report, in stream processing it is more common to have individual operators for fetching, processing, analysis and report generation. Notice that composability is not to be confused with the internal structure of an application (i.e., the separation of code into different methods and classes) but rather focuses on the creation of truly independent pieces of software that can be deployed individually.

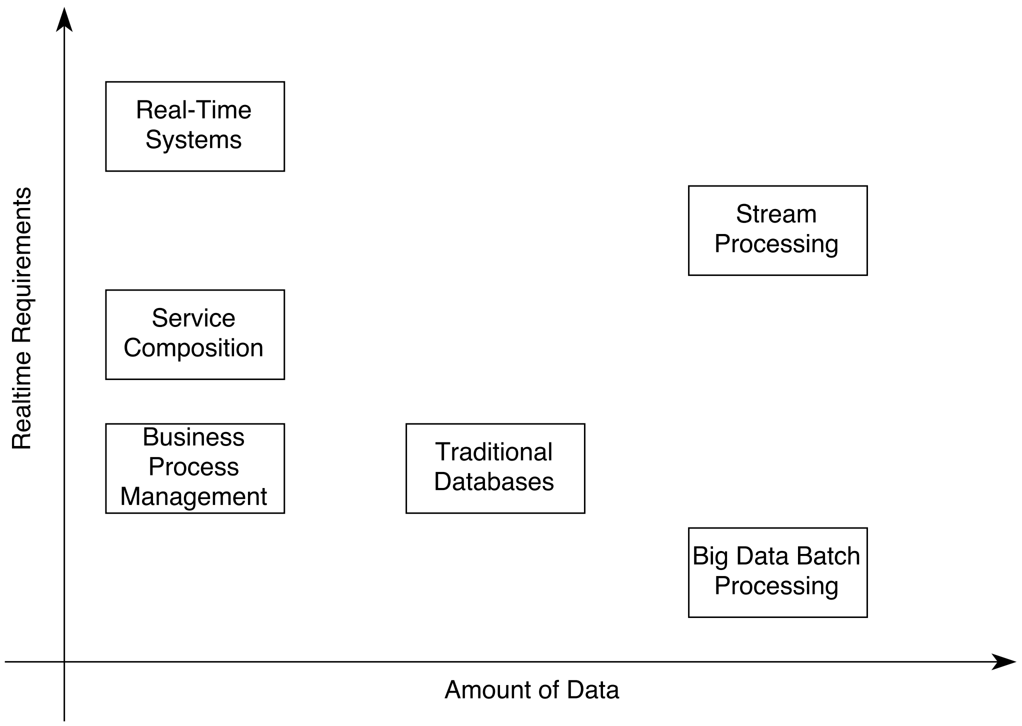


Figure 2.5: **Stream processing — Real-time requirements and amount of data**. Stream processing is distinguished by the combination of strict real-time requirements and large amounts of data to be processed. Similar technologies do not share *both* of these aspects.

Operators should also be *declarative*, meaning that the operators' interfaces are easy to understand and parameterisable [4].

Example 2.14

A stateful operator that receives individual sentences as inputs and counts the number of occurrences of a specific string should have a parameter specifying the search term to allow operator reuse.

An *expression language* should be part of an SPL in order to easily modify the operators' behaviour based on arbitrary conditions. This facilitates operator reuse and allows their application in customised use cases. As most object-oriented programming languages, SPLs often have a *type system* that guarantees only matching operators can be connected to each other [4]. Such a type system guarantees compatibility and allows the replacement of operators by other ones with the same types.

Example 2.15

Consider an operator that receives temperature data from a number of machines and computes whether any of them is above a certain threshold. The output type of this operator would be `boolean` and the operator can be connected to any other operator that accepts a `boolean`.

Windowing is also a very common feature among SPLs: it allows the execution of functions on a specific subset of the incoming continuous data stream [4]. A *time window* bases the subset selection on the time where the data items have been received, but customised windowing policies can be created by the topology designer.

Example 2.16

An operator A continuously produces data items containing the current GPS position of a monitored animal. Operator B that receives A's data items as its input can use a time window of, e.g., one minute in order to estimate the current speed of the animal.

Another important aspect of DSPEs is the mapping of an abstract topology to a concrete system-level deployment. This process is named *grounding* and an SPL must also provide means to affect for example the deployment location [45].

Example 2.17

A government law requires that confidential health data may not be transferred overseas. By restricting the deployment to locations within the origin country of the data, the DSPE is able to comply with this regulation.

The following paragraphs address stream processing languages from a historical context.

SQL-based Languages

As already discussed in Section 2.2.1, the first stream processing systems have emerged from traditional database systems. It is therefore consequential that the first stream processing languages were also heavily inspired by the languages used to query those databases.

The Continuous Query Language (CQL) was developed by researchers at Stanford to serve as an SQL-based declarative language [46]. Instead of finite sets of data, CQL uses continuous data streams and can therefore be considered as one of the first stream processing languages [4]. Creating this language enabled many developers to benefit from their experiences with SQL. The similarities between relation-based and stream-based data processing are striking. The obvious problem is: How can blocking operations (like `GROUP BY`) be performed on never-ending streams? In a `GROUP BY` operation, *all* input data must be processed before a result can be returned. This is of course not possible since a stream, by definition, is endless. The solution of

SQL-based languages is to apply such operators on *windows*. A window is a subset of data items that is selected based on the window's type. Temporal windows for example may consist of all data items of the last X seconds. While sufficient for most basic applications, SQL-based languages lack many of modern SPE's features such as fault tolerance [4].

IBM SPL

IBM has specified the *Stream Processing Language* (IBM-SPL) as a programming language for high-performance DSPEs [47]. This language is no longer based on database queries but on *data flow graphs* where operators are represented by vertices connected by edges. IBM-SPL is based on the *SPADE* language from IBM's *System S* [4] but was developed with a focus on large-scale processing [4]. It hands the control over most performance-critical options with respect to deployment, threading model, data representation etc. back to the user. For example, it provides compile-time customisation for the built-in operators to allow high performance while still being flexible [47].

Despite the focus on performance, IBM-SPL also aims to make it easier for a user to design applications. IBM-SPL is divided into a *composition language* (describing the large-scale topology) and *statement language* (which is used to define new operators or change existing ones). Having a separate composition language facilitates loose coupling between operators and easily allows changing their connectivity. This separation is relevant for this work as well because it is also implemented that way in VISP.

VISP Topology Description Language

DSPEs that take the burden of deploying operators and managing the underlying infrastructure from the topology designer require more information than only the topology itself. This meta-data may include geographical deployment restrictions, QoS-guarantees, data format descriptions and — the core topic of this work — fault tolerance measures. Therefore, it makes sense that a computer system needs such information to make informed decisions about when to scale, redeploy or switch processing paths.

The VISP Topology Description Language (VTDL) has been introduced by Hochreiner et al. [45] as a reaction to today's requirements on DSPEs. It supports the following next-generation features:

- **Deployment preferences**. This feature allows the SPA developer to define constraints for automated operator deployment mechanisms.
- **QoS compliance**. By defining QoS requirements on an operator level, a more fine-grained scaling is possible where less resources are needed.
- **Operator composability**. By using semantic annotations, the compatibility of operators can be checked and suitable operators can be found automatically.

Listing 2.1: Exemplary VTDL File. This VTDL file specifies the topology used for computing the distance of a user to a certain location and updating a dashboard afterwards.

```
$gpsPositionData    = Source() {
  concreteLocation = "::::ffff:8083:c001/cpu",
  type             = "source",
  outputFormat     = "positiondata"
}
$transform         = Operator($gpsPositionData) {
  allowedLocations = "::::ffff:8083:c001"
                     "::::ffff:8083:c002",
  concreteLocation = "::::ffff:8083:c001/cpu",
  inputFormat      = "positiondata",
  type             = "transformData",
  outputFormat     = "machinereadablePositionData",
  stateful         = "false",
  maxResponseTime  = "0.5"
}
$computeDistance    = Operator($transform) {
  allowedLocations = "::::ffff:8083:c001"
                     "::::ffff:8083:c002",
  inputFormat      = "machinereadablePositionData",
  type             = "computeDistance",
  outputFormat     = "distanceMeasure",
}
$updateDashboard    = Sink($computeDistance) {
  concreteLocation = "::::ffff:8083:c002/general",
  inputFormat      = "distanceMeasure",
  type             = "updateDashboard"
}
```

- **Runtime modifications**. This allows to change the structure of the SPA at runtime.
- **Different data transfer modes**. This allows to change between a one-at-a-time streaming mode and microbatch processing where multiple data items are transferred through the Internet as a group.

Listing 2.1 shows an exemplary VTDL file. Basically, operators are defined one after another. Each operator definition starts with a dollar sign ($) followed by the unique operator ID. The type (e.g., source, operator or sink) is stated before naming all direct input sources in brackets. In the operator definition's body, additional attributes are defined. For example, the `concreteLocation` attribute in the first operator's entry is defined as the location `::::ffff:8083:c001/cpu`.

The most important attributes are *type* (operator class), *concreteLocation* (the location the operator is deployed at), *input and output format* (the type of the input and output streams, respectively) and *stateful* (whether the operator has an internal state).

VTDL was chosen as the basis for the topology description efforts needed in this work (as discussed in more detail in Chapter 4). It needs to be noted that the VTDL only describes the *topology* of the SPA but does not directly specify the internal functionality of the operators.

2.3 Service Composition

In the *service-oriented computing* (SOC) paradigm, complex applications are built by connecting existing, independent services at runtime. A *service* in this context refers to a "*container of related capabilities consisting of [. . .] a body of logic designed to carry out these capabilities and a service contract that expresses which of its capabilities are made available for public invocation*" [48]. We discuss service composition due to its similarity to the composition of operators in DSPEs and because many fault tolerance approaches for SOC are also applicable to stream processing[3].

There are multiple advantages of such a *service composition*. First, a large program can be decomposed into a set of services that, when invoked in the right order, performs that same functionality (i.e., there are no immediate benefits for the application). In further consequence however, the same pieces can later be reused to create a completely different application, thereby speeding up the overall development time. Second, the smaller services of a composition can be replaced independently without affecting the overall application as long as the service contracts (i.e., the interface(s) the service binds itself to) stay the same. This allows multiple teams of developers to simultaneously work on different services without interfering with each other [48].

2.3.1 Comparison to Stream Processing

SOC shares many similarities with stream processing. While *services* are the central unit in SOC, *operators* have the same role in DSPEs: they are the independent building blocks that are connected in order to realise a new functionality. The same way services in SOC use contracts, operators also specify their interface via input and output types to allow easy replaceability. Operators are reusable as well and can also be shared [5].

However, there is a fundamental difference between the two concepts. SOC is focused on individual service invocations. Each invocation is guaranteed to be processed (or at least any exceptions are handled properly) and a sophisticated SOC framework will try to optimise various QoS requirements such that the request can be completed as fast and accurately as possible (e.g., by selecting the services with the lowest latency available). It may well be the case that two consecutive invocations of the same SOC application may lead to two different sets of services being invoked (e.g., because the latency of a service has changed between the two invocations).

[3] This section is partly based on unpublished work by Olena Skarlat, Stefan Schulte, Christian Inzinger, Schahram Dustdar and Philipp Leitner.

Stream processing is not just SOC with much more invocations per second, but there is a fundamental difference in what the outcome of the whole endeavour should be: Stream processing is focused on transforming, analysing and acting on the (passive) input data (often by some means of aggregation). By definition, this means that individual items are not too important since the aggregations are built from hundreds or thousands of items. If a few of them may get dropped on their way, this will most probably not affect the outcome of the process.

Figure 2.6 illustrates this example by considering data from a single temperature sensor. Assume this sensor measures the temperature of some critical manufacturing machine in an industrial facility. If it exceeds a certain threshold, safety measures must be initiated. A DSPE is used to monitor the temperature and the sensor broadcasts the current value every second. For temperature, historical data is largely irrelevant. Therefore, it does not matter if some of the input data items go missing since the succeeding items are sufficient to regain full knowledge about the state of interest. But also variables where historical data items are useful are relatively unaffected by missing data items as long as the total number of missing items is relatively small and the trend can still be captured.

Example 2.18

Consider a smartphone that transmits its GPS location every 5 minutes. With an average walking speed of 5 km/h, a pedestrian carrying that smartphone would move, on average, 416 m in between two status updates. If two consecutive transmissions are missing, one can still track the person to a region less than 1 km away from the previously measured location. This error would linearly increase with the number of missing data items. Once a new position update is transmitted successfully, the current position can be determined with absolute accuracy while the user's exact route to that location still contains an error.

2.3.2 Fault Tolerance in SOC

As motivated in the previous section, fault tolerance measures have a different focus in SOC than in stream processing. While in stream processing it is the ultimate goal to achieve a high availability (accepting small errors since results are often an approximation anyway), SOC strives for high accuracy. Basically, all fault tolerance techniques discussed in Section 2.5 are also applicable to SOC systems except those that would compromise on accuracy. For example, similar to the fault handling mechanisms a DSPE would engage in when faced with operator failures, SOC frameworks apply similar methods at the service composition level in case of service failure. Often however, SOC goes one step further. With technologies such as *WS-Transaction* [49] for example, a two-phase commit is used to enforce consistency throughout a service invocation. Such sophisticated (and time-consuming) measures would not be appropriate in a real-time stream processing scenario since it would cause a tremendous overhead.

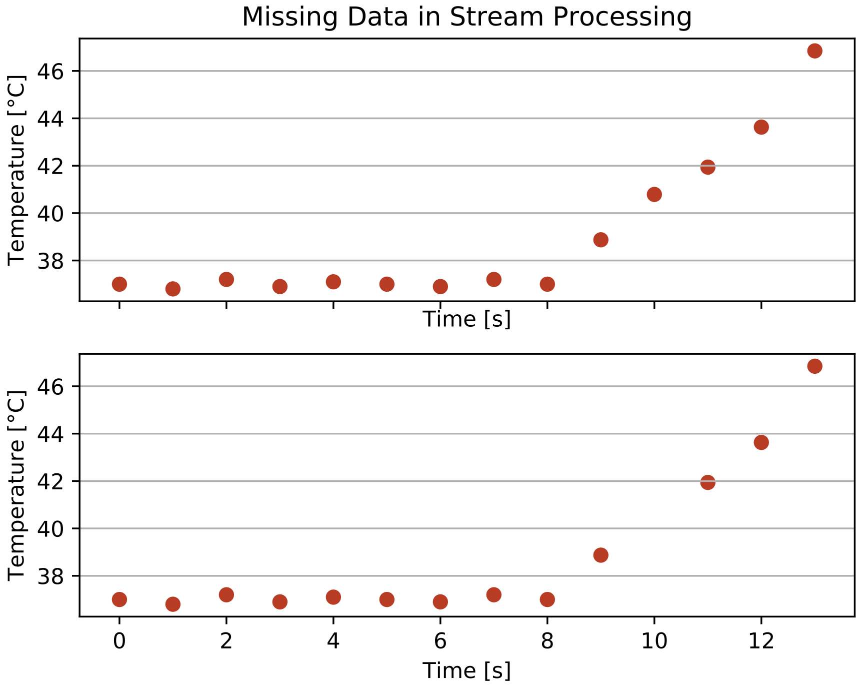


Figure 2.6: Missing Data in Stream Processing. Two plots visualise the potential effects of a data loss caused by a fault. In the first case, a temperature sensor reliably transmits the current temperature data every second. In the second case, no temperature data is available around second 10.

Furthermore, QoS attributes (such as cost, execution time, reputation, availability and reliability) [50] tend to play a higher role in SOC than in stream processing (however with modern DSPEs such as VISP, this is no longer true in general). Often, such QoS requirements are defined by legally binding service level agreements (SLAs). While this is in theory also possible for stream processing systems, SOC frameworks tend to put a greater emphasis on SLAs.

Leitner et al. [51] suggested to use machine learning approaches to predict QoS violations before they actually happen in order to initiate runtime adaptions. Such *proactive* measurements are difficult in stream processing systems due to the runtime overhead that would be introduced.

2.4 Microservice Architectures

Microservice architectures have proven themselves to be a useful software architecture strategy in the recent years. We mention them here because they share some similarities with SPAs (as discused in the next paragraphs). Microservice architectures mainly offer solutions for problems that often arise with *monolithical* software systems. Such systems are characterised by a lack of modularisation which prevents parts of the monolith to be easily replaceable [52]. Microservices tackle this by forcing developers to reduce the dependencies between software modules (thereafter named microservices). Benefits of such a microservice architecture do not only include increased maintainability but also better scalability and a more agile engineering approach [53].

Wolff defines microservice architectures by a set of criteria [53]. Instead of a single large software system, there are smaller modules (microservices), each of which can be deployed independently. There are no restrictions regarding programming languages or platforms that can be used for each microservice and no data storage is shared between them. Microservices are completely self-contained and finally, communication between microservices must be loosely coupled and most commonly uses REST.

It is apparent that microservice architectures share many similarities with SPAs.

1. **Small modules**. DSPEs are inherently composed of small operators that resemble microservices.
2. **Independent deployment**. At least some DSPEs support the dynamic deployment of operators at runtime.
3. **Platform independence**. Like microservices, operators can be developed in any programming language for some DSPEs (e.g., VISP).
4. **No shared data storage**. While operators can have state information, this state is never shared directly between operators.
5. **Loose coupling**. The operators interact only via well-defined interfaces.

Due to these similarities, the opportunity arises to apply microservice-specific solutions to SPAs as well. This work in particular focuses on the circuit breaker pattern — a pattern commonly used in microservice architectures to deal with calls to failed services — and investigates its applicability to SPAs.

2.5 Fault Tolerance

In order to properly discuss fault tolerance, a coherent terminology is first introduced in Section 2.5.1. Afterwards, the characteristics of centralised and distributed systems with respect to fault tolerance are addressed before focusing on its application in stream processing systems.

2.5.1 Terminology

Avizienis et al. [7] have published a fault tolerance taxonomy in 2004. This work uses their vocabulary definitions unless stated otherwise. The most important ones are listed in Definitions 1 to 10. All definitions are taken directly from Avizienis et al. [7].

Definition 1 (*System*)

A system is an entity that interacts with other entities, i.e., other systems including hardware, software and humans.

Systems can be identified on different levels of abstraction. One can view a whole SPA as a system that interacts with humans (e.g., by providing data to a dashboard), but one can also view an individual operator as a system that interacts with other operators. In this work, we focus on the SPA as the system.

Definition 2 (*Service*)

The service delivered by a system is its behaviour as it is perceived by its users.

Definition 3 (*Correct service*)

A software system is said to deliver a correct service if it implements the required system function.

Definition 4 (*Service failure*)

A service failure (abbreviated to failure) is an event that occurs when the delivered service deviates from correct service.

Definition 5 (*Error*)

An error is the deviation of at least one external state of a system from the correct service state. An error is detected if its presence is indicated by an error message or error signal. Errors that are present but not detected are latent errors.

Definition 6 (*Fault*)

A fault is the (hypothesised) cause of an error and can be internal or external of a system. A fault is active when it produces an error; otherwise, it is dormant.

Definition 7 (*Availability*)

Availability is the readiness for correct service.

Definition 8 (*Reliability*)

Reliability is the continuity of correct service.

The difference between availability and reliability is quite subtle. While availability can be understood as an average over time as in "*service X is available Y% of the time*", reliability refers to a probability as in "*what is the probability that service X will be continuously available for the next Y hours*".

Example 2.19

One can see this difference more clearly when considering a system that is unavailable for two minutes of maintenance each day at midnight. This corresponds to an availability of $1 - \frac{2}{24 \cdot 60} = 99.8\%$ but the system can only be considered reliable for less than one day.

Definition 9 (*Fault prevention*)

Fault prevention means to prevent the occurrence or introduction of faults.

Fault prevention is the much more common way of dealing with faults in traditional software engineering projects — it involves the use of highly reliable libraries, thorough software testing and a sophisticated software architecture. However, as experience has shown, software projects exceeding a certain scope and size are almost guaranteed to contain bugs, so fault prevention is expected to miss some of them.

Definition 10 (*Fault tolerance*)

Fault tolerance means to avoid service failures in the presence of faults.

Since it is at the heart of this work, the following section takes a more detailed look at fault tolerance and typical mechanisms used to achieve it.

2.5.2 Fault Tolerance Fundamentals

Avizienis et al. [7] classify faults by considering eight elementary attributes:

1. **Phase of occurrence** (development or operational),
2. **System boundaries** (internal or external),
3. **Phenomenological cause** (natural or human-made),
4. **Dimension** (hardware or software),
5. **Objective** (malicious or non-malicious),
6. **Intent** (deliberate or non-deliberate),
7. **Capability** (accidental or lack of competence), and
8. **Persistence** (permanent or transient).

Based on those attributes, they come to the conclusion that there are three major fault groupings: 1. *Development faults* that occur during development, 2. *physical faults* that include all faults affecting hardware, and 3. *interaction faults* that include all external faults [7].

Example 2.20

A developer accidentally failing to initialise a loop's counter variable correctly is causing a *development fault* (which is also internal, human-made, in the software dimension, non-malicious, non-deliberate and permanent).

In distributed systems, failures in one part of the system can in turn cause faults in other parts of a system. This *causality relationship* between faults, errors and failures is illustrated by the following example.

Example 2.21

A software developer makes a (human) error and writes code that divides by zero for some input cases. This error is initially undetected and results in a dormant fault in the software. Once a certain input to the software is provided, the fault becomes active and causes an error. Only if that error also affects the delivered service (e.g., by aborting a computation), a failure occurs.

By this mechanism, a single fault can lead to multiple errors — this is called *error propagation*. When multiple components are involved, failures in one component can of course affect other components when the failing components provide (incorrect) inputs to them.

Fault tolerance now tries to avoid *failures* of the system in the presence of faults by using *error detection* and *system recovery* [7]. By using these two mechanisms, failures are prevented from becoming failures. Error detection takes place either concurrently or while the normal system service is suspended. Once the presence of an error is detected, *recovery* consists of two steps. First, the *error* activated by the fault must be dealt with. The goal here is to eliminate the error from the system state completely by transforming the system state. *Rollback*, *rollforward* and *compensation* are three ways to deal with the error. Rollback and rollforward try to reach an error-free state by transformation to an earlier, error-free state and to a newly created, error-free state, respectively.

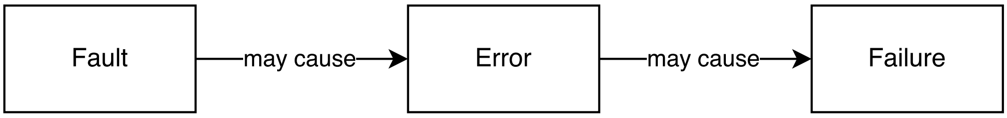


Figure 2.7: Relation between the terms *fault*, *error* and *failure*. Terminology from Avizienis [7].

Example 2.22

An operator A assigns registration numbers to new students. Its internal state stores the last registration number that has been assigned successfully. While assigning number `9965`, the operator crashes and must be restarted.
Without any recovery strategy, its internal counter would resume at `0`.
By using rollback recovery, the operator would recover the last internal state of `9965` and would next assign number `9966`.
If rollforward recovery is used instead, the internal state can also be set to `10000` (skipping some registration numbers, thereby making sure that no number is assigned twice). The crucial difference is that this state has never been reached in the past and has newly been created during recovery.

Compensation is a different approach where the error is not removed but rather masked by redundancy (for example by relying on fallback hardware). Second, the fault itself is handled and should be prevented from becoming active again. This is called *fault handling* and consists of four steps. *Diagnosis* tries to identify the likely cause of the error, *isolation* aims to exclude the faulty components from further participation in service delivery, *reconfiguration* tries to switch in spare components or reassigns tasks and *reinitialisation* updates the new configuration and system tables [7]. Ultimately, while fault handling prevents the same fault from causing another error, *error handling* corrects the system state by undoing the effects of the error.

2.5.3 Fault Tolerance in Stream Processing

Stream processing systems have some unique characteristics that require special attention regarding fault tolerance. On the one hand, due to their continuous nature of operating, fault tolerance is even more crucial than in applications that run once only [4]. On the other hand, individual data items tend to have less importance in SPAs than for example in business process management systems (as seen in Figure 2.6).

A challenging aspect of fault tolerance in stream processing system is that there may be different levels of fault tolerance required for different parts of the application.

Example 2.23

An operator responsible for computing "recent trends" in a social network can easily deal with being unavailable for a short time. In contrast, it may cost millions if an algorithm that selects appropriate advertisement clips for social media users experiences a downtime.

It is also important to distinguish between stateful and stateless operators. Stateful operators have a local state that is potentially changed by incoming stream tuples (as shown in Example 2.21) and fault tolerance mechanisms need to deal with this and guarantee a reliable state even when faced with failures.

SPAs that tolerate at least a small amount of data and/or state loss are called *loss tolerant*, whereas *loss intolerant* SPAs are unable to cope with losses [4].

Andrade et al. [4] list three basic mechanisms of fault tolerance in stream processing systems: *cold restart*, *checkpointing/restart* and *replication*. Both *cold restart* and *checkpointing/restart* assume that there is only a transient failure that can be corrected by restarting the same software on a different host. This can happen either directly or via a checkpointing mechanism. There, snapshots of the operator's internal state are stored periodically while operating correctly and are recovered on a restart. This type of fault tolerance is not considered further in this work since we do not solely focus on transient faults.

Replication is redundancy in its most apparent form. Andrade et al. [4] describe *active replication*, where multiple copies of the same operator are running concurrently. Once a copy of the operator fails, one of the others can become active and replace the failed one.

Finally, it is also challenging to define what exactly constitutes a failure in SPAs. When an operator stops working completely, this is definitely a failure. There is however a grey area when QoS violations occur ("When is an operator *too slow*? What if an operator gives inaccurate results?"). Deciding when an operator has failed is a context-dependent decision and cannot be answered for all possible use cases. For the sake of general applicability, this work treats operators as black boxes. Therefore, their inner workings are ignored and failures are only detected by observing their input and output behaviour as well as rudimentary statistics gained by monitoring their hardware usage (see Section 4.3 for more details).

The Need for Sophisticated Fault Compensation

With the continuously rising importance of stream processing, more and more software engineers will use DSPEs to tackle new projects. At the same time, SPAs are composed of many operators. The more complex an SPA grows, the more operators are required. An increasing number of operators goes hand in hand with a higher probability that one of them becomes unavailable.

Example 2.24

An SPA that is composed of seven operators, each of which has a 0.999 availability, will have an average availability of $0.999^7 = 0.993$. One that is composed of 30 operators will only reach 0.97. These scores are equivalent to a downtime of 5 and 21 hours per month.

Since it is infeasible to reach a 100% availability for each operator, it is crucial for the DSPE to mitigate the effects of such faults.

2.6 The Circuit Breaker Pattern

Applications that are executed in a distributed way on multiple machines have to communicate with each other. In each communication step, failures may arise. Even if a failure for any given service is highly unlikely, a complex system might use dozens or even hundreds of such services. If no fault tolerance method is applied, failure of even a single service will inevitably mean failure for the whole system. Instead of allowing such failures to cause unavailability, it has been common practice to retry communication attempts. However, in large-scale applications, such retry mechanisms can cause resource depletion (i.e., each connection requiring some amount of memory that accumulates) and may unintendedly act as a denial-of-service attack towards the non-responsive service (i.e., causing even more stress on the failing service by flooding it with more input). Furthermore, if a service truly is unavailable, it is counterproductive to even spend time trying to contact it [54].

2.6.1 Mechanism

Nygard et al. [55] have discussed this problem and proposed a solution in the form of the *circuit breaker* pattern to deal with failing services. The concept of a circuit breaker originally comes from electronics where it is used to stop current flow in case of a malfunctioning electrical device. Analogously, a circuit breaker in a software system would *trip* (i.e., stop the control/data flow to a service) when a failure is detected, thereby preventing future calls to that service from being made at all. Instead, either a fallback solution is used or the failure is immediately propagated to the caller as soon as possible.

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

A circuit breaker can be viewed as an automaton with three states `OPEN`, `HALF-OPEN` and `CLOSED` as depicted in Figure 2.8. Being closed by default, the circuit breaker will act as a proxy and forward all requests to the appropriate service and in turn

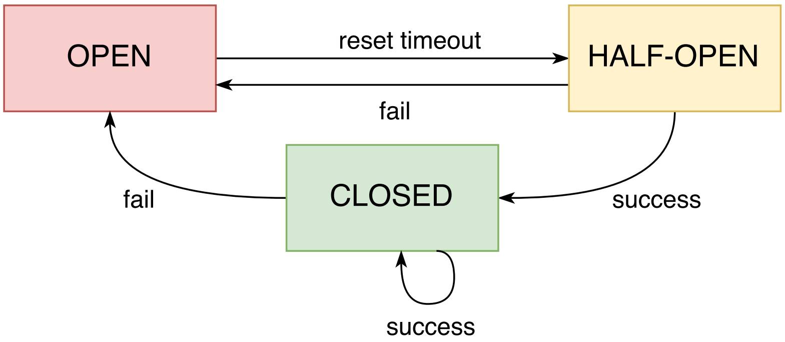


Figure 2.8: States of a circuit breaker. Figure adopted from [54].

forward the service's reply to the caller. If a service failure is detected, the circuit for the respective service is opened. In this state, no further calls are forwarded to the external service. Instead, either the fallback is used or an error is returned. After a certain time, the circuit breaker switches to the half-open state and forwards a small fraction of the requests to the service to see whether it has recovered. If it has, it returns to the closed state and the service is fully operable again.

2.6.2 Opening the Circuit

When designing a circuit breaker, one of the most important questions is when to open it. In a very simple approach, one can open it once the first request times out or fails. Such a strict policy would probably produce a large number of false positives since network errors can indicate failure while the service is correctly functioning.

Fowler [54] discusses an elegant model for circuit breakers in asynchronous communication environments. Instead of monitoring each request individually, a queue is set up where all outgoing requests are put into. The asynchronous service then consumes requests from this queue and the circuit breaker monitors the queue's status. This model is suitable for distributed stream processing for two reasons. First, the communication between operators is not only asynchronous but does not produce a response at all. Second, having queues where incoming messages are stored is very natural and many systems (e.g., VISP [5] using RabbitMQ) are already implemented this way.

When monitoring the queues, the circuit breaker must have some kind of policy about when to open the circuit. This policy can depend on factors including the queue's input and consumption rate, as well as the current and past number of items.

2.6.3 Fallbacks

It has already been mentioned that with circuit breakers, services can immediately throw an exception if they are about to call a failed service, thereby saving time. However, software developers can also implement a *fallback* method that can still provide a valid result even in case of service failure. Using such a fallback method does not necessarily have to be detrimental, it may just get the necessary answer from a different service with the same quality. Another strategy would be to rely on cached values if the domain context allows for that. This is not the same as calling another service since the cache is created and updated locally by the caller.

Example 2.25

Service A offers users the capability to buy train tickets. Its responsibility is handling the payment process. Since users are usually interested in knowing the exact departure time, it relies on service B for that information. A uses a circuit breaker for calls to B. If the circuit is open (indicating a failure in B), it invokes a fallback method that returns cached data from the previous invocation. The user is therefore unaware of the service failure. Additionally, A might make the user aware of the fact that its result is based on historical data.

2.7 Message-Oriented Middlewares

Many stream processing systems (including VISP [5]) depend on an infrastructure called *message-oriented middleware* (MOM) to send and receive messages. The *advanced message queuing protocol* (AMQP) has been developed as an open standard for enterprise-scale asynchronous messaging to replace proprietary protocols such as IBM Websphere MQ, Microsoft Message Queuing (MSMQ) and the Java Message Service (JMS) [56]. Being a binary protocol, it is much more space-efficient than a text-based format such as JSON or XML. AMQP does not only reach a very high performance, it also includes sophisticated mechanisms for reliability, message queuing and routing.

As its name suggest, the central object in MOMs is the *message*. It consists of the *bare message* (the actual payload) and optional annotations (metadata) that can be added and removed during processing. The AQMP specification also defines certain *distribution nodes* [56] that are able to filter, distribute and reject certain messages (e.g., queues, exchanges in RabbitMQ).

AMQP implementations include Apache ActiveMQ[4], Apache Qpid[5] and RabbitMQ[6]. Since VISP is based on RabbitMQ, its features are discussed in more detail.

The basic actors in RabbitMQ are message *producers*, message *consumers*, *queues* and *exchanges*[7]. A queue is a buffer that stores messages originating from a producer and allows consumers to take and process them. An exchange sits between a producer and one or more queues and decides the destination queue of a certain incoming message as depicted in Figure 2.9. Exchanges are useful because the producer does not have to

[4] http://activemq.apache.org/

[5] https://qpid.apache.org/

[6] https://www.rabbitmq.com/

[7] https://www.rabbitmq.com/tutorials/tutorial-three-python.html

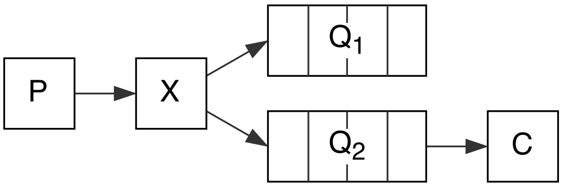


Figure 2.9: Actors in RabbitMQ. A message producer (P) sends messages to an exchange (X) that decides to which queue (Q_1 and/or Q_2) a message is forwarded to. This procedure is completely transparent to the producer and the consumer (C), the latter of which then receives messages from only the queue it is subscribed to (Q_2).

be aware of the queues it sends its messages to. Instead, it just sends them to a queue and the messaging infrastructure creates the routing from the exchange to one or more queues. At runtime, this routing can be changed without the producer's knowledge.

Since RabbitMQ clients are available for all major programming languages including Java, C++, Ruby and C#, it is an excellent choice for a DSPE to base its communication on since operators written in any of those languages can directly interact with the MOM.

2.8 VISP

The *VIenna ecosystem for elastic Stream Processing* (VISP) has been proposed as a next-generation DSPE by Hochreiner et al. [5]. As suggested by its name, one of VISP's main goals is to enable the usage of elastic resources (i.e., cloud computational resources) in stream processing to adapt to changes in data volume at runtime. Such changes occur in many important use cases (e.g., those involving the monitoring of human activities).

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 prototypical implementation of the fault tolerance framework developed in this work.

2.8.1 Architecture

Figure 2.10 shows VISP's software architecture as presented by Hochreiner et al. [45]. The main component is the *VISP Runtime* that is based on a RabbitMQ messaging infrastructure. It receives data from various data sources and other VISP runtime instances and uses operators that are run on computational resources (e.g., virtual machines on a private or public cloud) to process them. An operator repository can be used to obtain the images for the operators (e.g., Docker images) that are then deployed by VISP. Another important part of VISP is the parsing of topologies in the VTDL format (see Section 2.2.5). This is not only necessary during initialisation but also when

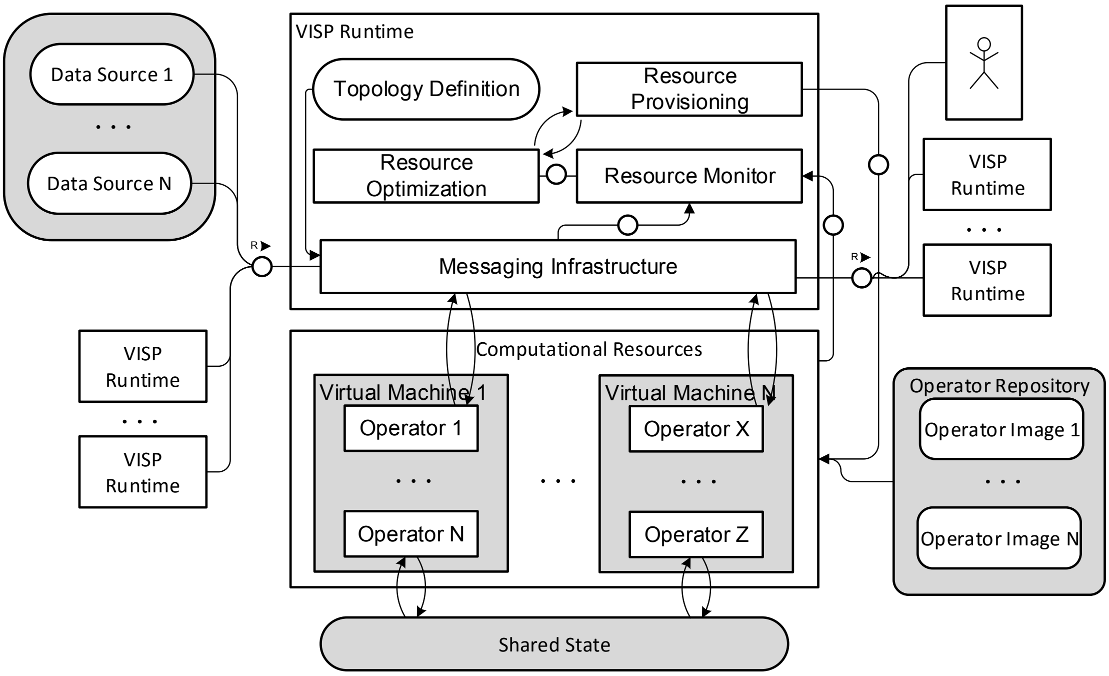


Figure 2.10: **VISP Stream Architecture**. Figure from [45].

the topology is changed at runtime. Since topology changes at runtime are excessively used for this work, this mechanism is discussed in more detail.

2.8.2 Topology Updates

Each topology change is triggered by the upload of a VTDL file to the VISP runtime (for simplicity, we do not consider the case where multiple VISP runtimes are involved — for details see [45]). The next steps are the same regardless whether another topology has already been deployed before or not. The VTDL file is parsed and all operators are assigned to a concrete location to be deployed (this may be already defined by the `concreteLocation` attribute in the VTDL file or determined automatically using the `allowedLocations` attribute). Then, the RabbitMQ infrastructure is updated to match the new topology (see the next section for details). This leads to a situation where a subset of the operators are still working while newly added operators are not yet deployed. This is not detrimental since RabbitMQ queues can store the produced messages until the new operators are ready as well. Finally, the new operators are deployed and start processing messages that are already waiting for them on their respective queues.

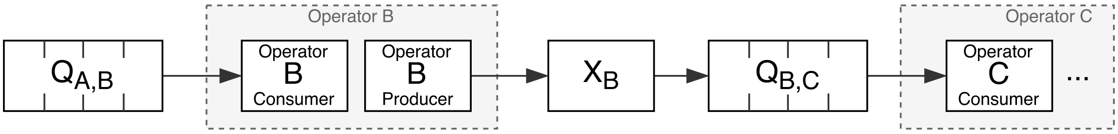


Figure 2.11: VISP's RabbitMQ architecture. Each VISP operator simultaneously acts as a consumer and a producer of messages. Messages are consumed from the queue populated by the input operator (in this case, $Q_{A,B}$) and are then processed. The resulting data items are then pushed to the operator's exchange (here, X_B) and are then distributed to the downstream operators' queues (here, $Q_{B,C}$). Using exchanges has the advantage that is is very easy to add another downstream operator simply by reconfiguring the exchange's queue bindings which can be done at runtime.

2.8.3 Messaging Infrastructure

In VISP, operators are deployed as Docker[8] containers in the programming language of the developers choice. To facilitate a reliable messaging infrastructure, VISP uses RabbitMQ. Deployed operators interact directly with RabbitMQ which guarantees a high performance.

A VISP operator B receiving input data from an operator A would simply subscribe to the queue named after those two operators (in this case, $Q_{A,B}$), process them and then push the resulting messages to the exchange X_B. The operator B itself is not aware of what happens at exchange X_B and which subsequent operator is involved in the processing flow. VISP configures each exchange such that it forwards incoming messages to the right subsequent operator's queue. In this case, an operator C can be downstream (an operator A is downstream of an operator B if there is a transitive data flow from A to B) of B and therefore, X_B is configured to forward incoming messages to the queue $Q_{B,C}$ where the messages are fetched by C. This scenario is depicted in Figure 2.11.

[8] https://www.docker.com/

CHAPTER 3

State of the Art

One of the goals of this thesis is to provide an overview of fault tolerance mechanisms for stream processing systems (see RQ1 in Section 1.3). This chapter identifies mechanisms that were *specifically* developed for stream processing systems (Section 3.1) but also for related areas such as service composition (Section 3.2) and microservice architectures (Section 3.3). Finally, this chapter is concluded by a summary of different approaches which form the foundation for our approach.

3.1 Fault Tolerance in Stream Processing

Fault tolerance in stream processing can be classified into two classes based on the kind of fault they address [31]. *Communication* faults are caused by lost packages due to the underlying network infrastructure. Especially when computational resources are distributed over a large geographical area, network communication failures are common [57]. Fault tolerance on the *operator level* deals with the failure of whole operators that may be caused e.g., by underlying hardware problems or software bugs.

3.1.1 Communication Faults

Network problems are a very common cause of faults in all distributed systems and DSPEs are no exception. DSPEs secure their communication in different ways depending on their underlying communication model.

As a concrete example, we consider the handling of network errors in RabbitMQ[1] that forms the basic communication infrastructure of the VISP DSPE [39] (as discussed in Section 2.7). The connections between the server and any clients can be interrupted by network errors at any time, potentially compromising the integrity of the messages

[1] https://www.rabbitmq.com/reliability.html

that are being sent at that time. RabbitMQ uses *acknowledgements* to confirm that a message has been received. While RabbitMQ is based on the Transmission Control Protocol (TCP) [58] which already guarantees reliable transfers, these additional acknowledgements are necessary because RabbitMQ wants to also ensure that the receiver has actually processed the message. If no acknowledgement message is received, the sender will attempt a retransmission. The receiver must be able to identify such retransmitted messages to avoid duplicates when an acknowledgement message is lost.

Example 3.1

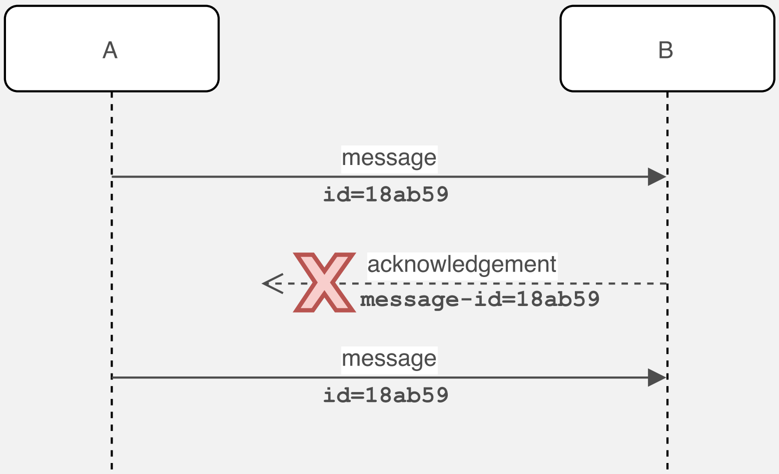


Figure 3.1: Message acknowledgement.

A sends a message to *B* and waits for an acknowledgement. *B* receives the message and sends the acknowledgement message to B. The acknowledgement message is lost. *A* cannot distinguish whether the original message to B was lost or just B's acknowledgement message. Therefore, *A* retransmits the original message. From *B*'s perspective, this second message from *A* is a new message. However, by using unique identifiers in the messages themselves, *B* can identify the second message from being a duplicate of the first one.

Finally, in order for the queues and exchanges to avoid losing messages, they are regularly persisted on disk.

For some requirements of SPAs (e.g., real-time processing), this messaging model may cause a too high latency, especially due to the disk persistence of data items [4]. Other approaches are conceivable where no such persistence is necessary in exchange for less consistent results (see, e.g., *approximate fault tolerance* in Section 3.1.2).

3.1.2 The Operator Level

The long running nature of SPAs requires proper fault handling mechanisms on the operator level. Specifically, both the operator's internal state (if present) and the stream data items themselves need to be considered when discussing such mechanisms [4].

Failed operators themselves can be recovered quickly by spawning a new instance of the operator.

Example 3.2

In VISP, each operator type is provided as a Docker image and each instance is run as a Docker container. Therefore, spawning a new operator instance corresponds to spawning a new Docker container from the same Docker image.

The more challenging problem is then to restore the operator's internal state that has accumulated until its crash.

Example 3.3

A topology may consist of two operators. The first one outputs a data item for each car that passes by a sensor. The second one counts the number of passed cars in the last 60 minutes.
If the first operator crashes and recovers, it can immediately continue emitting data items without any complications. A crash affecting the second operator however would also lead to losing its internal state, i.e., the total number of passed cars.

A common solution to this problem is *(state) checkpointing* [4]. An external data store is used to backup a snapshot of the internal state in a regular time interval. During a recovery, the state can then be recovered from that store. What exactly triggers a backup and how to incorporate the data store into the DSPE differs between implementations. The use of checkpointing in DSPEs is important since they are often long-running applications and recreating a consistent state from the beginning is not straightforward.

As postulated by the *CAP-theorem*, it is not possible to simultaneously achieve availability, consistency and partition (fault) tolerance in a distributed system [59]. Therefore, there is always a trade-off between consistency and availability for fault-tolerant distributed systems. The following section discusses mechanisms that handle this trade-off in different ways.

There are two extreme cases of this trade-off [11]. First, there is the strategy where the DSPE waits for a potentially indefinite amount of time for operators to recover from an error. This approach also uses strict backup policies for the internal states of the operators. This *"error-free"* mechanism provides a high level of consistency, but due to the lack of an upper bound on the recovery time and the large overhead of the backups, the availability of an SPA is low. Second, the DSPE can just discard all data items that are affected by a failed operator. This *"best-effort"* strategy would lead to a high availability but also to a low consistency due to the lack of backup mechanisms.

Balazinska et al.

Balazinska et al. [31] introduce a fault tolerance approach implemented for Borealis. It allows the SPA developer to change the tradeoff between availability and consistency by specifying a *maximum waiting time*. The DSPE will wait as long as the predefined amount of time for operators to recover. If they do not recover in time, *tentative results* are produced. Tentative results serve as approximations for the missing results for subsequent operators and allow a continuous output of data items even in the case of operator failures. Once the failed operators are successfully recovered, the tentative results are corrected and all computations based on the tentative results are repeated with the corrected ones. This process of correction is named *stabilisation*. It starts when the previously failed operator has recovered. Then, the downstream operator that produced tentative results (because of the failing upstream operator) recovers its state to a pre-failure checkpoint. It then reprocesses all data items since then (this time with the data items from the failed operator), allowing it to emit data items that are no longer tentative. Finally, it sends special *undo* data items to the downstream operators that received tentative results previously (to signal them that the previous results should be removed) and sends them the corrected data items.

For this mechanism to work, operator failures must be detected quickly. *Heartbeat messages* are therefore regularly sent to all operators. If an operator fails to reply to such a heartbeat, it is considered as failed.

Balazinska et al.'s approach promises *eventual consistency*. This means that results may be tentative at first in order to meet the availability demands, but ultimately, all results are correct as long as no external services are invoked during the processing of tentative results. The idea is that it is more important to have at least some results in time than to have 100% correct results that are delayed.

Example 3.4

The operators A and B are both needed for C to produce its results. If both A and B are operational, correct results are sent to D.

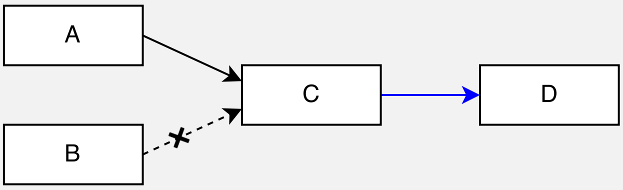


Figure 3.2: Production of tentative results.

Now consider the case where operator B crashes and is no longer able to provide input to operator C. If C was waiting for B to recover, it may violate the maximum waiting time. Instead, tentative results (shown in blue) are produced that are not fully correct (because B's input is missing) and sent to D.

Once B recovers, the missing data items are sent to C. Furthermore, C sends corrected versions of the already sent data items to D, guaranteeing eventual consistency.

A restriction of Balazinska et al.'s approach is the need for customised code to implement the generation of tentative results. The SPA developer needs to specify exactly how an operator should generate tentative results based on some operators' results missing. In some cases, such a generation may even be impossible. Another restriction is the need for large buffers for data items since they might need to be resent.

Example 3.5

Generating tentative results for an operator that just transforms data items into a different format is impossible if the one and only input operator stops producing data items. Conversely, an operator that only depends on the current weather condition as an input may be able to produce tentative results by just assuming an "average" default weather.

Another disadvantage of this method is the large computational overhead that is required to cope with failures. In effect, all computations are then performed two times, first with tentative and then with the real data items.

MillWheel

Akidau et al. [60] describe MillWheel, a DSPE developed at Google, along with its fault tolerance model. MillWheel provides very strong consistency guarantees by supporting exactly-once delivery and error-free state checkpointing. A unique feature is its *low watermarks* mechanism where time of data arrival and data generation can be distinguished for each data item. The *low watermark* is basically a timestamp that signals the SPA that all data up to that timestamp has been successfully received. It allows the SPA developer to make sure that they have a complete picture of the data up to that time [60].

MillWheel promises consistency even if faced with arbitrarily many hosts crashing and an infinite amount of data item loss. These guarantees are implemented using an acknowledgment mechanism to prevent message loss and a fine-grained checkpointing mechanism that periodically saves each operator's state to an external BigTable [61] data store.

The exactly-once delivery is ensured by combining mechanisms for at-least-once delivery (the acknowledgment of successfully received messages) and at-most-once delivery (unique identifiers for each outgoing data item).

There is a restriction in MillWheel regarding the invocation of external services: Since exactly-once delivery is based on the acknowledgement of received data items, a data item can be processed multiple times if acknowledgement messages are lost. This is harmless in general since the duplicate results can be detected by having the same identifier. However, if the external service itself is stateful, it may receive the same data item multiple times. Therefore, no external services may be invoked by the operators in order to avoid such side effects. This restriction can be evaded by making the external service idempotent, i.e., guaranteeing that multiple invocations with the same parameters have no additional effects.

Approximate Fault Tolerance

Huang and Lee [11] present an approach on *approximate fault tolerance*. They argue that providing perfect fault tolerance is not only very expensive but also not needed in general. Especially for scenarios where streaming data is only processed to identify trends, it is tolerable to lose some of the data items to improve availability (as argued in Section 2.5.3). Again, the trade-off between consistency and availability is configurable. Three user-specifiable parameters, θ, l and γ, can be tuned in order to specify how many errors are tolerated.

Their approach uses a state backup that is only triggered once the deviation from the current state and the most recent backup is greater than a maximum of θ. Unlike MillWheel and Balazinska et al.'s approach, this mechanism tolerates a small amount of errors that can remain uncorrected. By changing θ, the SPA developer can specify how much state deviation is acceptable.

Example 3.6

An operator X counts occurrences of temperature measurements above 35° C. Its state divergence function is defined in such a way that it returns the difference between the current counter value and the most recent backed up one. The user defines the maximum state divergence to be 10.
Each time the counter is changed, the state divergence function is used to compute whether it returns a number greater than 10. Once it does, a new backup is issued.

Similar to the state backup, there is also a backup mechanism for the yet unprocessed data items. The variable l defines the maximum number of such unprocessed items. Once it exceeds l, a backup of the data items is issued. Finally, γ is used to denote the maximum number of unacknowledged items in the queue of an operator. If the queue length reaches γ, no more items can be forwarded by that operator until previously sent items are acknowledged.

The authors admit that the manual fine-tuning of these variables is difficult for SPA developers. It is also questionable if defining them globally for the whole SPA is appropriate, because some operators are likely to be more tolerant in terms of state divergence than others. Furthermore, the authors provide theoretical guarantees about the error bounds depending on the user-specified parameters [11].

Table 3.1: Comparison of the three introduced fault tolerance mechanisms on the operator-level. Table adapted from [62].

	Balazinska et al. [31]	MillWheel [60]	Huang and Lee [11]
Fault tolerance for state	✓	✓	✓
Fault tolerance for data items	✓	✓	✓
Tradeoff availability & consistency	✓	✓	✓
Tradeoff configurable	✓		✓
Maximum inconsistency			✓
Theoretical error bounds			✓
Tentative results	✓		
User has to tune parameters			✓

3.1.3 Comparison

Table 3.1 shows an overview of the three discussed fault tolerance mechanisms on the operator-level. All three mechanisms provide fault tolerance for state and data items with a tradeoff between availability and consistency. As discussed, such a tradeoff is necessary due to the CAP theorem [59]. This tradeoff is configurable for the approaches by Balazinska et al. and Huang and Lee by specifying a maximum waiting time and a maximum backup divergence, respectively. This makes those two solutions more flexible and applicable to a wider array of use cases.

All three approaches focus on fault tolerance on the level of individual operators. This is not feasible since in reality, a functionality is often implemented by a composition of multiple operators [4]. This limitation is especially apparent for the approach of Balazinska et al. [31]. Here, tentative results are generated at the level of processing operators. This leads to the already discussed case where multiple operators depending on only one input stream are unable to create meaningful tentative results if that stream becomes unavailable. In reality, those two operators would have to be considered as a single functional unit where either both of them are available or an (approximate) replacement is installed for both of them. The idea of such a *path-level* mechanism is further discussed in Chapter 4 and forms the basis of the approach created in this thesis.

Tolerating a maximum level of inconsistency (i.e., not requiring 100% consistency) is unique in the approach of Huang and Lee [11] and allows flexibility for SPAs where data is only mined to identify trends. These inconsistencies are however theoretically bounded depending on the values of user-defined parameters.

Tentative results (as introduced by Balazinska et al. [31]) allow a high availability when faced with failing operators while still guaranteeing eventual consistency of the final results. However, this mechanism requires the user to implement functions for generating such tentative results and needs to spend a considerable amount of time for reprocessing data items once operators become available again.

Conclusion. All three discussed fault tolerance approaches have the common goal of providing a suitable balance of availability and consistency. However, they reach this goal with very different mechanisms. Comparing their features tells only part of the truth since no benchmark exists that directly compares their performance in different use cases. The benchmarks provided by the respective authors differ in their methodology and lack the applicability to a real-world scenario.

3.2 Fault Tolerance in Service-Oriented Computing

SOC has already been introduced in Section 2.3. There is a large body of literature on fault tolerance in SOC. As discussed, fault tolerance is even more important in SOC since, for these systems, perfect consistency is usually expected and every single service invocation must succeed [63]. It is therefore already clear at this point that one cannot just copy the SOC mechanisms and directly apply them to stream processing since SPAs usually do not have such strong demands for consistency [4]. It is however possible to apply some methods in the same way.

WS-ReliableMessaging has been introduced to reliably communicate over an unreliable network in SOC [64]. Different communication modes are available where data items are transmitted *at least once*, *at most once* or *exactly once*. Similar messaging mechanisms are used for DSPEs (e.g., MillWheel [60]) and MOM (e.g., RabbitMQ[2]).

WS-Replication is used to forward service calls to different replicas, guaranteeing fault tolerance when one of them becomes unavailable [65]. This approach also enables n-version programming when different operator implementations are used as replicas. Again, similar mechanisms can be applied for DSPEs where multiple operator instances are used as replicas for a single operator type.

With *WS-Transaction*, a coordination between services is possible [49]. It can be guaranteed that an invocation involving multiple operators is carried out atomically or not at

[2] https://www.rabbitmq.com/

all. This mechanism's application to SPAs is not advisable in general since the resulting computational overhead is not justifiable by the marginal gains in consistency.

Another interesting approach in SOC fault tolerance is the treatment of *non-functional faults*. Here, quality of service expectations (e.g., a certain throughput) are considered. Their violation is then treated as a fault that can be corrected [66]. Handling QoS faults is also an interesting challenge in SPAs. It is however out of scope of this work.

3.3 Hystrix

In 2011, Netflix has introduced Hystrix, a fault tolerance library for distributed systems based on the circuit breaker pattern (see Section 2.6) [67]. Since the major revenue source of Netflix is streaming video content, it has considerable interests in a high availability of their Web services.

For each incoming user request, multiple calls to subsystems are realised. With both a huge number of user requests and subsystems, a single failure in one of them would lead to downtimes.

Hystrix solves this problem by applying three basic mechanisms:

1. **Custom fallback.** If possible, the service developer should specify a *fallback mechanism* in case of a service failure. Fallbacks may include the use of cached values, reasonable defaults or alternative ways of providing meaningful results. Importantly, fallbacks must not depend on external services themselves since their invocation may not itself fail.

2. **Fail silently.** In case no fallback is available, it is still acceptable to return *null* or a proper exception. If the failed service is not essential to the overall outcome, this failure can be hidden from the user.

3. **Fail fast.** Instead of wasting time waiting for a server response, services should either return an answer or an exception *quickly*.

Hystrix is implemented as a Java library. Each callable service is wrapped in a `HystrixCommand` object and must implement a `run()` as well as a `getFallback()` method as shown in Listing 3.1. The `run()` method is invoked when the circuit is closed and the service is expected to be available.

Hystrix continuously monitors the amount of failed service calls in relation to the total number of calls. Once a certain threshold is reached, it transitions the circuit breaker to the *open* state (as discussed in Section 2.6.2) and the `getFallback()` method is used instead. After some time, the circuit breaker is switched to the *half-open* state and the availability of the service is checked. Once the service has recovered, the state is set to *closed* again and the `run()` method is executed again.

Listing 3.1: Basic `HystrixCommand` implementation. Source: https://github.com/Netflix/Hystrix/wiki/How-To-Use.

```
public class CommandHelloWorld extends HystrixCommand<String> {
  private final String name;

  public CommandHelloWorld(String name) {
    super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
    this.name = name;
  }

  @Override
  protected String run() {
    return "Hello " + name + "!";
  }

  @Override
  protected String getFallback() {
      return "Hello Failure " + name + "!";
  }
}
```

Hystrix is successfully used for microservice architectures (see Section 2.4). Its application to stream processing is not straightforward since the operator calls are made by the DSPE. In other words, the developer of the individual operators would only be able to use Hystrix for the invocation of external services it uses, but the invocation of operators itself cannot be influenced on that level. To the best of our knowledge, there is no DSPE that actively uses Hystrix or the circuit breaker pattern in general directly.

3.4 Inspirations

This work aims not only to create a mechanism that unifies features of existing approaches. The unique feature to provide redundancy at the level of processing paths is not considered in the literature to the best of our knowledge. Compared to the related work investigated in this chapter, we can conclude the following inspirations:

- **Circuit breaker**. The usage of the circuit breaker pattern to dynamically switch between a main function and fallbacks is inspired by Hystrix from the context of microservices.
- **Tradeoff between consistency and availability**. The publications of Balazinska et al. [31], Akidau et al. [60]. and Huang and Lee. [11] motivate the use of a compromise between availability and consistency to deal with faults in distributed stream processing systems.

- **Operator-level fault tolerance.** The established mechanisms to guarantee fault tolerance at the operator level (such as checkpointing and recovery and active replication) have been kept in mind when designing our new approach. Although this work does not implement them, such mechanisms can be added to our framework to further enhance fault tolerance.

CHAPTER 4

Design of a Fault Tolerance Framework for Stream Processing

This chapter presents the Pathfinder framework for fault tolerance in distributed stream processing systems. First, terminological definitions are introduced in Section 4.1. Then, the requirements for Pathfinder are discussed in Section 4.2. Based on those requirements, Pathfinder's fault tolerance model is discussed in Section 4.3. Section 4.4 then introduces Pathfinder together with its system design. Finally, Section 4.5 concludes this chapter by introducing fault-oriented development, a set of software development guidelines for SPAs to fully utilise the capabilities of Pathfinder.

4.1 Terminology

An overview about stream processing terminology has already been given in Section 1.1. In this section, we replace those definitions in a more formal way. Since, to our knowledge, there exists no literature dealing with fault tolerance in stream processing on a language level, we need to derive a new nomenclature. This is especially true for those constructs that have been newly introduced by this work. Where possible, this nomenclature is inspired by informally used concepts (such as [4]).

4.1.1 Operators and Data Flow

An *operator* is the most basic building block of an SPA [4]. Its input are zero or more input streams. It applies processing logic for each incoming data item and outputs zero or more output streams. We subdivide operators into three classes: Operators with zero input streams are called *source*, those with zero output streams *sink* and all others *processing operator*. Data items enter the SPA through sources and leave it through sinks (e.g., by persisting them into a database or updating a dashboard).

In the literature, the *data flow* representation has been used model the flow of data items between operators [4]. In this representation, operators are modelled as boxes and the data flow between them as a directed edge. It is therefore easy to comprehend how a data item is passed through individual operators in the SPA.

4.1.2 Topology

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 [39]. The topology therefore defines which operators exist in an SPA and how they are connected to each other. The topology is not only a formal way to view an SPA but also exists as a file that is created by the SPA developer (e.g., in the VTDL format).

When considering an SPA, we can differentiate between a logical and a physical topology view. Both views use a similar notation, i.e., edges to denote the data flow. However, only in the physical view, the edges correspond directly to the physical flow of data in the SPA. In the logical view, edges only indicate the *possibility* of a data flow (e.g., the data flow from a join operator to alternative paths depends on which of them is currently active).

A data flow in general refers to data moving from one processing stage to another [4]. We concretise this definition the following way:

A *logical data flow* from operator A to another operator B ($A \neq B$) exists if the *logical* topology view has an edge from A to B. We denote such a data flow with: $\mathrm{df}_l(A, B)$.

A *physical data flow* from operator A to another operator B ($A \neq B$) exists if the *physical* topology view has an edge from A to B. We denote such a data flow with: $\mathrm{df}_p(A, B)$.

Since the logical data flow is a prerequisite of the physical data flow, the following condition must hold:

$$\neg\mathrm{df}_l(A, B) \Rightarrow \neg\mathrm{df}_p(A, B)$$

The *logical topology view* only considers the logical data flow while the *physical topology view* only considers the physical data flow.

4.1.3 Operator Connections

We now put our attention to the way operators are connected to each other. We define the set of *upstream* operators of an operator A as the set of operators that have a direct or indirect logical data flow to A. Similarly, the set of *downstream* operators of A is the set of operators that have a direct or indirect logical data flow from A.

We say an operator A *has an input* (operator) B if there is a logical data flow from B to A. Similarly, A *has an output* (operator) C if there is a logical data flow from A to C.

The indegree of an operator A is denoted as $\deg^+(A)$ and represents the number of input streams of A. Similarly, the outdegree represents the number of output streams of A and is denoted as $\deg^-(A)$.

As already stated above, we define a source to be an operator A where $\deg^+(A) = 0$ and a sink to be an operator B where $\deg^-(A) = 0$.

Example 4.1

The figure below shows the physical view of a topology. It consists in total of four operators. A is a source (since it has no input streams), C and D are sinks (since they have no output streams) and B is a processing operator (since it has both input and output streams). B has exactly one input (A) and two outputs (C and D). Therefore, $\deg^+(B) = 1$ and $\deg^-(B) = 2$.

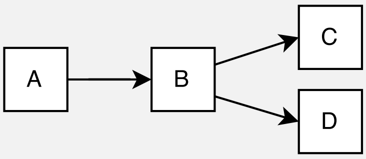


Figure 4.1: Physical topology view.

4.1.4 Paths

Operator paths are a crucial concept in SPAs. Similar to service composition in SOC, operator paths provide functionality by combining individual operators in the right order [4].

We consider a topology $T(O, E)$ that consists of a set of operators O and a set of edges that denote data flow E. A *path* π in the topology T is defined as an ordered sequence of operators such that there is always a logical data flow between consecutive operators in that sequence. More formally, a path is a sequence of operators $(A_1, A_2, \dots A_n)$ such that:

$$\bigwedge_{i=1}^{n-1} \mathrm{df}_l(A_i, A_{i+1})$$

We define an index-based access to the elements of a path π by referring to the i-th operator in π using the notation π^i (starting at index 0).

We denote that an operator A is part of the path π with $A \in \pi$.

Example 4.2

There are multiple paths in the topology below. Let us define the path π as the operator sequence A, B, C. π is a path because there is a logical data flow between each two consecutive operators of π. The first operator in that path, denoted as π^0, is A.
We define another path ρ as the sequence A, B, D. In this particular example, it holds that $\pi^0 = \rho^0$, i.e., π and ρ share a common operator at the first position. In other words: $A \in \pi \wedge A \in \rho$.

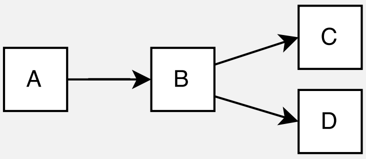


Figure 4.2: Physical topology view.

A path π is said to be *active* if there exists an operator $A \notin \pi$ where $\mathrm{df}_p(A, \pi^0)$ (there is a physical data flow from A to the first operator of π). A path is *inactive* if there does not exist an operator $A \notin \pi$ where $\mathrm{df}_p(A, \pi^0)$ (there is no operator with a physical data flow to the first operator of π).

Example 4.3

In the following physical topology view, we define the path π_x to consist of the operators B, C and D. Since $\exists n \in (N - \pi_x) : \mathrm{df}_p(n, \pi_x^0)$, π_x is an active path.

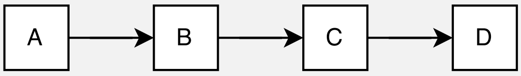


Figure 4.3: Physical view of a path composed of four operators.

A path is said to contain a *failure* if one or more of its operators contain a failure.

4.1.5 Split/Join Operators

For the sake of completeness, we present split and join operators in this section although their use becomes apparent only after reading Section 4.4.

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 available. We name an alternative path π by its first operator (π^0) since their beginning and end are unambiguously defined by the split and join operators, respectively. For split operators, there is a discrepancy between the physical and the logical topology view: although a logical data flow is set up from the split operator to multiple operators, there is always only one physical data flow.

The *path order* ρ defines the order in which Pathfinder falls back on the alternative paths in case of operator failures. The main path has a path order of $\rho = 1$, i.e., it is used with the highest priority. There must exist at least one other alternative path with $\rho > 1$. No two alternative paths of a split operator can have the same path order (i.e., for a split operator S, $\mathrm{df}_l(S, A) \land \mathrm{df}_l(S, B) \Rightarrow \rho_A \neq \rho_B$).

The counterpart to the split operator is the *join* operator. It indicates the place where the alternative paths come together again. In other words, the operator directly downstream of the join will receive input data items either from the main path or from any of the fallback paths.

Example 4.4

In the below logical topology view, operator C has failed. It is part of the main path (as denoted by the path order “1” assigned to that path next to the split operator). Since operator C has failed, the whole path (consisting of B and C) is considered failed. Therefore, fallback path D will be used instead.

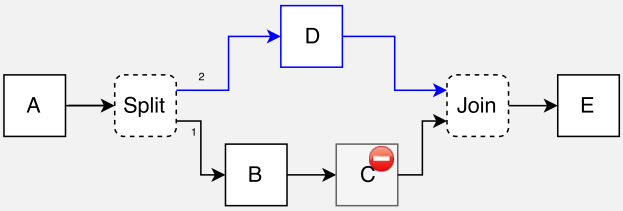


Figure 4.4: Path failures.

4.2 Requirements

Section 1.2.2 introduces our motivational scenario and presents the requirements for this scenario. Chapter 3 concludes that current fault tolerance methods for stream processing lack certain features and are therefore not applicable for our motivational scenario. This section will discuss the framework requirements (*FR*) needed to support the fault tolerance of the motivational scenario.

FR1 **Scalability**. Scalability is usually implemented at the DSPE level (e.g., by replicating operators). While it is not the fault tolerance framework itself that needs to handle scalability, it must at least be compatible with the scaling measures of the DSPE.

FR2 **Exploit functional redundancy**. This is the central requirement that cannot be fulfilled by any other mechanism presented in Chapter 3. Having redundancies at the functionality level (i.e., being able to use different functions to achieve the same or a similar result) that must be exploited by the fault tolerance framework necessarily requires input from the user. Specifically, the SPA developer must specify which functionalities are redundant and how the redundant resources can be accessed. Furthermore, the user must provide an order in which the redundant functionalities should be accessed if more than one is available at the same time.

FR3 **Minimise downtimes**. While already hinted in R2, this requirement explicitly states that the total SPA downtime should be minimised. Specifically, redundant resources should be utilised whenever failures occur to keep the SPA's period of unavailability as short as possible.

FR4 **Reliability of results**. This requirement states that the absolute limit of the downtime minimisation is the production of results with a *low* quality that are still usable — the fault tolerance framework must not go further than that. Producing results that are completely unsuitable for the SPA while still guaranteeing availability would violate this requirement.

The framework requirements above are mainly caused by the SPA requirements R3, R5, R6 and R7 from Section 1.2.2. The remaining SPA requirements do not directly affect the fault tolerance framework but rather the DSPE itself (we refer to this DSPE as the *slave DSPE* since it will be controlled by the fault tolerance framework). R1 states that indeed a *distributed* SPE must be used to cope with the expected data volume. R3 concretises this requirement and states that the DSPE must also be able to scale using cloud resources. Furthermore, R4 requires the DSPE to optimise operator placement with respect to geographical patterns. All DSPE requirements (R1, R3, R4) are fulfilled by VISP which is therefore used primarily in this work.

4.3 Fault Tolerance Model

Based on the requirements from the previous section as well as existing concepts from the related work (discussed in Section 3.4), a new fault tolerance model for distributed stream processing is presented in this section.

4.3.1 Language-Level Fault Tolerance

Initially, the most important design decision is: On what level should fault tolerance be dealt with? While this framework addresses fault tolerance at the *language-level* (i.e., by explicitly defining fault tolerance fallback actions via the topology description language), there are several other choices.

1. **Operator level.** For the first approach, the operators are responsible for their own fault tolerance mechanisms. While this looks promising, it is too short-sighted. First, it is not possible to address inter-operator communication problems at this level. Second, such an approach faces restrictions since fallback mechanisms must be created for each operator in isolation, i.e., it would be very cumbersome to

replace the functionality of a whole processing path by a different one. The lack of reusability of fault tolerance mechanisms is another disadvantage.

2. **DSPE level.** Here, a mechanism integrated directly into the DSPE performs a similar failure detection mechanism as our approach and redeploys failed operators once a failure has been detected. While this does not require user-intervention, it would not work for failures caused by long-term errors (e.g., network problems of a third-party service, revoked software licenses, bugs for certain input data).

3. **Infrastructure level.** Fault tolerance at the level of the computing infrastructure can for example include automatic restarts of failed virtual machines in a cloud environment. However, since these approaches do not consider the application logic, the same problems as on the DSPE level are to be expected.

Considering the requirements from the previous section, it is, to the best of our knowledge, not feasible to deal with fault tolerance for SPAs anywhere else than on the topology description language level without facing significant limitations.

Furthermore, language-level fault tolerance has the advantage of being both easy to change (i.e., simply by changing the topology file) and allowing customisations via configurational parameters (e.g., *lazy deployment* as introduced in Section 5.5.1).

4.3.2 Path Redundancy

As stated by Gärtner et al. [12], there can be no fault tolerance without redundancy. Redundancy in SPAs does not necessarily imply the use of multiple instances of the same operator in case one of them crashes. In our fault tolerance model, we rather consider *paths* as the fundamental unit of fault tolerance. Instead of having multiple instances of the *same* path, our model allows the SPA developer to design 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.

- **Flexibility**. Since paths can consist of arbitrarily many operators, SPA developers can define fault tolerance actions for specific sets of operators. Furthermore, also the length and number of alternative paths can be chosen by the SPA developer to reflect the level of availability needed for specific functionalities. All but the simplest SPAs are likely to exhibit some functionality where failures can be tolerated more easily and others where faults are intolerable.

Example 4.5

An SPA is used to analyse network traffic and includes two functionalities. First, it informs the responsible person about intrusion events that need to be investigated. Second, it changes the scheduling of several maintenance tasks based on the current and predicted amount of traffic. However, if no scheduling suggestions are made by the SPA, all maintenance tasks are still completed with a minimal loss of efficiency.

Since network intrusions can be an existential threat to a company whereas scheduling optimisation is dispensable, failures in the latter are more tolerable than those in the former. Therefore, a responsible SPA developer would create more alternative paths covering the intrusion detection functionality than for the scheduling optimisation.

By giving the topology designer this powerful tool, they can create fault tolerance mechanism that are tailored to their specific needs and can be as fine-grained as desired.

- **Operator composability**. Operator reusability dictates to decompose a functionality into a set of operators that are then connected by data flow (thus allowing reuse of the operators in later projects). A path is exactly such a composition of operators. If no path-level redundancy was available, SPA developers would hesitate to decompose functionality into multiple operators since fault tolerance mechanisms for all operators would have to be created.
- **Simplicity**. SPA developers do not need a thorough knowledge on fault tolerance or distributed systems to implement alternative paths. From their perspective, adding fault tolerance measures is as simple as writing more application code — the fault tolerance framework is then responsible for deployment and activation. Furthermore, the code of existing operators does not need to be modified in order to add fault tolerance which makes it less likely to introduce any new bugs.

4.3.3 Error and Fault Handling Approach

Avizienis et al. [7] divide recovery from faults into the two stages *error handling* and *fault handling*. Our model makes the same distinction and performs a two-stage recovery from faults (as depicted in Figure 4.5). Applying the terminology introduced in Section 2.5.1, we consider the *failure* of an operator (i.e., it stopping working correctly). The same event can also be seen as a *fault* from the view of the DSPE. If no countermeasures were taken, this *fault* could cause the *failure* of the SPA. Therefore, fault tolerance is used to recover from the fault and prevent a failure.

First, the faulting operator itself is handled. In the diagnosis step it is *detected* which operator causes the error The whole path containing the faulty operator is then *isolated* (i.e., by blocking the physical data flow to that path), to *mask* the fault. In a subsequent *reconfiguration* step, an alternative path with similar functionality is activated. Finally,

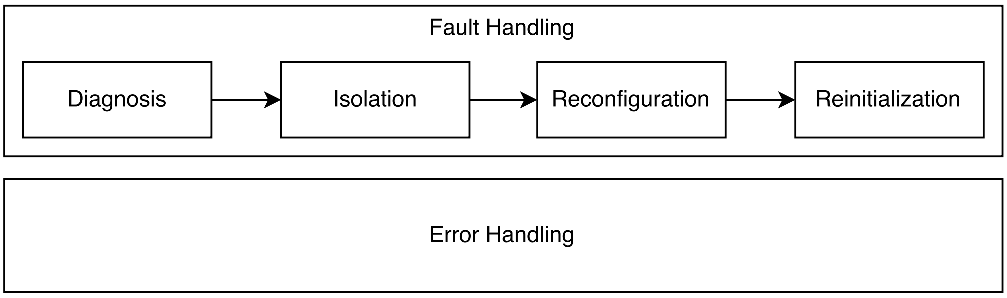


Figure 4.5: Pathfinder's fault tolerance model.

reinitialisation of the system takes place and the data item routing is updated such that the newly activated processing path is used instead of the faulty one.

In contrast to the model of Avizienis et al. [7], our model's fault handling does not stop after the faulty processing path has been deactivated. While it is acceptable to use a fallback path while the main path is unavailable, there is also a need for some kind of mechanism to get back to the main path once it is available again. A probing mechanism is used to continuously check whether a previously failed path could recover (see Section 4.4.4).

Recovery is the process of creating an error-free state. While recovery is essential in many software systems, DSPEs often have specific requirements. As discussed in Section 2.5.3, streaming data is often valuable only immediately after its production. For example, in the introduced motivational scenario, there is no use in dealing with past location data since the user has probably moved somewhere else in the meantime.

Therefore, our model does not include a dedicated recovery mechanism. It would however be possible to add a checkpointing mechanism for stateful operators (for example similar to the one of MillWheel [60]). However, this is out of the scope of this work.

Furthermore, no attempts are made to reprocess 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.

4.4 Pathfinder — A Fault Tolerance Framework for Stream Processing

We now present Pathfinder, a fault tolerance framework for distributed stream processing. Before going into the details of the framework, this section will give a simplified overview of how Pathfinder works.

4.4.1 Basic Functionality

While fault tolerance mechanisms could be directly included into the DSPE, Pathfinder is an independent system that closely interacts with an associated DSPE. Thereby, Pathfinder is not restricted to a specific DSPE. Instead, it can be used to support any DSPE that is able to provide the needed information and obeys the commands by Pathfinder. Pathfinder controls the slave DSPE's data flow and operator deployment based on the monitoring of operational statistics.

Basically, Pathfinder

- *monitors* the slave DSPE and continuously analyses 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. This step is essential since fault tolerance mechanisms can only be initiated when the framework knows about the faults. *Commanding* the DSPE is realised because Pathfinder is not able to directly influence the DSPE's operators or its data flow. For this step, Pathfinder needs to know about all available fallback paths. This knowledge needs to be added by the SPA developer via an extension to the VTDL that includes special operators for fault tolerance. These special operators are named *split* and *join* (see Section 4.1.5). Downstream of a split and upstream of a join operator are at least two alternative paths that provide a similar functionality with different implementations. Pathfinder decides which of the alternative paths is active (i.e., receives the physical data flow from the operator that is directly upstream of the split operator).

Similar functionality means that the SPA developer is free to specify an arbitrary path as an alternative path as long as it has the same input and output stream types as the main path.

Figure 4.6 shows a simple topology on top. Below is a different version of the topology where split and join operators are added. Primarily, the main path (i.e., B2, B3) is used (identifiable by the path order "1" shown next to the split operator) and a physical data flow only exists to and from the main path. If Pathfinder detects a failure in the main path, it switches to the fallback path (for details see Section 4.4.4).

4.4.2 System Design

Pathfinder consists of three core modules: the *Circuit Breaker*, the *Nexus* and the *Communicator* module. Figure 4.7 shows an overview of the system design that is explained in detail in the next paragraphs.

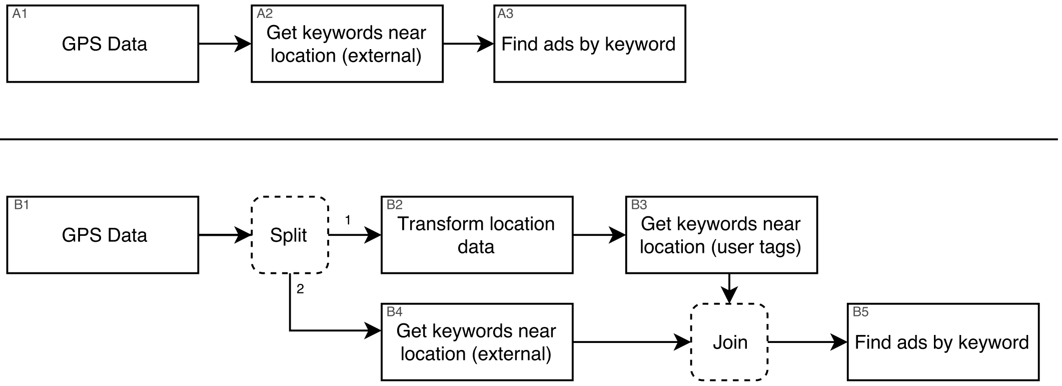

A1
GPS Data
A2
Get keywords near location (external)
A3
Find ads by keyword
B1
GPS Data
Split
1
2
B2
Transform location data
B3
Get keywords near location (user tags)
B4
Get keywords near location (external)
Join
B5
Find ads by keyword

Figure 4.6: **Top:** A simple SPA for fetching advertisement campaigns for a given GPS position. It consists of three operators. First, a GPS position item enters the system (A1). An intermediate step is then used to map that GPS position to a list of keywords that seem relevant to that location (A2). Finally, all currently available campaigns that are relevant to the set of keywords are fetched (A3). **Bottom:** The same topology with a fault tolerance mechanism. Under normal circumstances, the main path will be used and data processing will take place using the same operators as above (B1, B4, B5). If step B4 fails, the fallback path is activated and an external service (B3) with a prior transformation step (B2) will be used for the same task instead.

- *Communicator*. Pathfinder needs to communicate with other Pathfinder instances and instances of the slave DSPE. Bundling all communication aspects in a single module enables loose coupling since this module can be easily exchanged when a different DSPE should be controlled.

- *Circuit Breaker*. The circuit breaker module (CBM) is aware of the topology that is currently active in the slave DSPE. Internally, Pathfinder, and in particular this module, always uses the VTDL to describe a topology. If a different DSPE needs to be addressed, a conversion step from and to VTDL can be added to the communicator module.

  A circuit breaker object is created and maintained for each path where at least one alternative is available. By querying the Nexus component, operator failures are detected that are then translated into state transitions of the circuit breaker objects.

  In case of topology changes initiated by the slave DSPE, the local topology representation of the CBM is discarded and new circuit breaker objects are created for all paths of the updated topology of the SPA.

- *Nexus*. The Nexus component is responsible for analysing statistical data and classifying operators into *working* and *failed*. Each operator is classified individually only based on current and historical statistics collected from the slave DSPE.

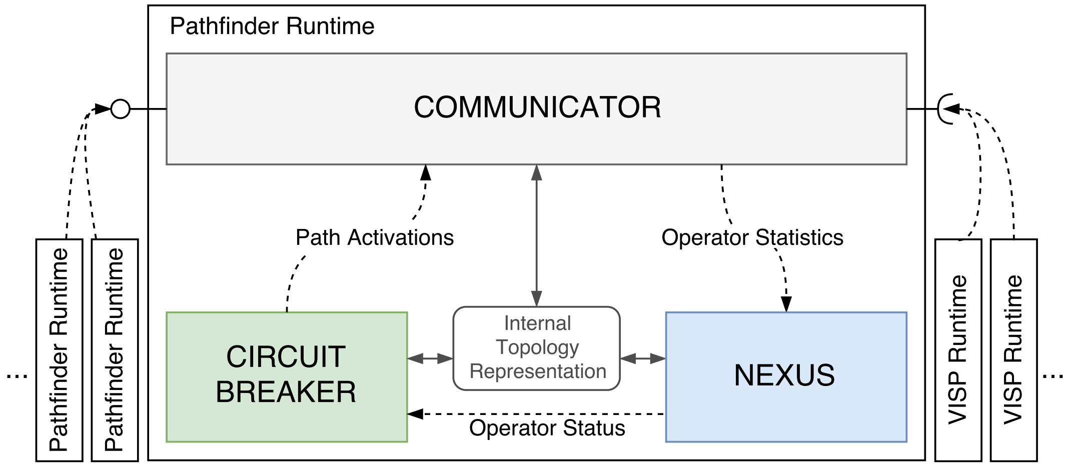


Figure 4.7: Pathfinder's system design.

The concrete Nexus implementation can make use of different technologies such as manually created rules (e.g., considering operators as failed if their CPU usage is less than 1% for more than 30 seconds) or machine learning-based identifications.

4.4.3 Distributed Deployment

It is desirable for many reasons not to have a central component that is responsible for fault tolerance. First of all, what if the host crashes where that central component is deployed? In many cases this would also mean a crash of the whole DSA or at least heavy limitations in terms of fault tolerance if the component is not properly recovered. Introducing such a single point of failure into a DSPE is just not an option.

To solve these issues, Pathfinder is deployed in a distributed manner itself. Individual Pathfinder instances communicate with each other via the communicator module. This architecture not only enhances availability but is also expected to have a beneficial effect on overall performance. Since every Pathfinder instance can be restricted to query only a subset of the DSPE's operators, distributed deployment allows easy scaling.

In a simple yet effective scenario shown in Figure 4.8 where each DSPE instance is queried by exactly one Pathfinder instance that resides on the same host, a very high performance is expected since the frequent statistic requests are performed on the same host and do not require communication over the Internet.

The only exchange via the Internet happens horizontally (i.e., between Pathfinder instances). Horizontal communication is mainly necessary when an operator fails or recovers, which is assumed to be a relatively rare event. The main portion of the communication however does not leave the respective hosts and is therefore very efficient.

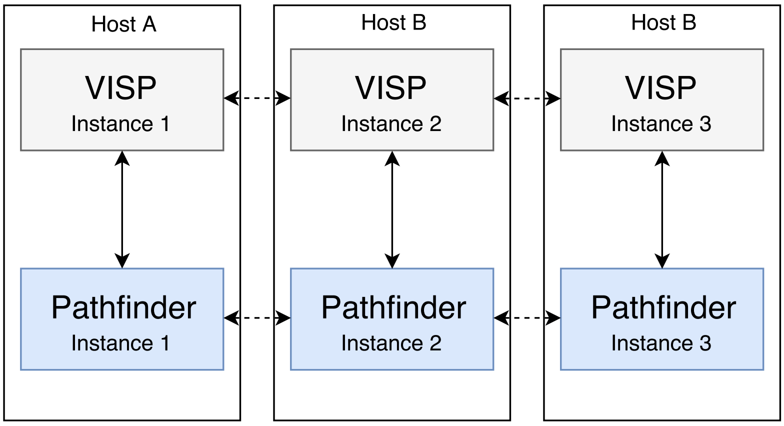


Figure 4.8: Possible distributed deployment of Pathfinder. Three VISP instances (1 to 3) are deployed on the three hosts A, B and C. By deploying a Pathfinder instance on each host, the communication overhead to the VISP instances is minimised.

Example 4.6

There are three DSPE instances on hosts A, B and C, located in three different countries. In scenario 1, a single Pathfinder instance queries all of the instances for operational statistics. Since these queries are very frequent and there is a considerable delay per query due to the large geographic distance, a low performance is expected. In scenario 2, there is a single Pathfinder instance for each DSPE instance that is deployed close to the respective instance. Each Pathfinder instance now only queries the DSPE instance that is closest to it and therefore achieves a very high performance. The occasional communication between different Pathfinder instances happens less frequently and does not impact the overall performance.

4.4.4 Circuit Breakers in Pathfinder

Pathfinder's fault tolerance mechanism is influenced by the circuit breaker pattern (see Section 2.6). The pattern suggests to fail fast — namely, not to unnecessarily wait for responses to services that have already failed.

For each topology where all operators are working properly, there is a constant data flow. If one of the operators fails for any reason, this data flow is interrupted. Moreover, the failing operator's queue would fill up since the data sources would continue to send new data items. This is problematic for two reasons. First, the queue might eventually reach its maximum capacity and overflow. Second, it may take an infinite amount of time until the operator can recover from the failure and the messages may thus suffer an infinitely long delay. Both consequences are harmful to the SPA and need to be dealt with.

The basic idea of the circuit breaker pattern (as discussed in Section 2.6) 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 approach would be the implementation of the circuit breaker pattern on the level of individual operators. This would mean that topology designers would have to implement a fallback solution for each operator, resulting in a very inflexible and work-intensive programming model that results in n-version programming. Instead, Pathfinder implements circuit breakers on the level of paths that can consist of arbitrarily many operators.

The huge advantage of this approach is that the topology designer can construct redundancy at a high 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.

Example 4.7

A complex path (consisting of dozens of operators) with the purpose of creating some statistical analysis can be backed up by a redundant path consisting of only one simple operator that creates a very superficial analysis only.
If fault tolerance was implemented at the operator level, this would not be possible and each operator would need a fallback action, making the overall procedure much more complicated since it is highly unlikely that there are replacement services covering exactly the same functionality as a single operator).

Since circuit breaker states and transitions have already been examined in Section 2.6, only the application of the circuit breaker pattern to Pathfinder is left to be discussed at this point.

The *open* state does not allow any physical data flow through the circuit breaker. In the *closed* state, physical data flow is not interrupted. A probing mechanism is invoked in the *half-open* state where only a small fraction of data items is allowed to pass the circuit breaker. Additionally, there must exist state transitions in such a way that the circuit breaker can change its state based on whether its path is operational or not.

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 Section 4.4.2) provides information about the current health of each operator. Using that information, the circuit breakers are updated accordingly (i.e., it is opened if at least one operator failure in that path is detected). On each circuit breaker transition, the DSPE is contacted and advised to update the physical data flow (i.e., stop it when the circuit breaker is opened and resume it once it is closed).

Example 4.8

The following figure depicts the data flow through a split operator *Split*. Each of the outgoing paths (P_1, P_2 and P_3) is assigned its own circuit breaker and the state of each circuit breaker is indicated by a traffic light symbol.

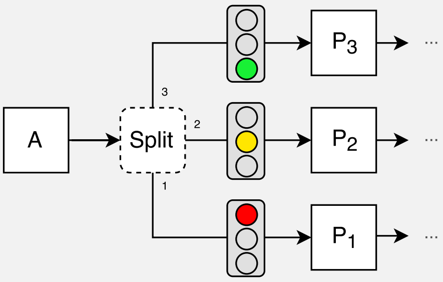


Figure 4.9: Representation of circuit breakers in Pathfinder.

Since P_1's circuit breaker is in the *open* state, there is no physical data flow from A to P_1. By means of the probing mechanism, a small fraction of data items is forwarded to P_2 due to its circuit breaker being *half-open*. P_3 is the active path because it is has the lowest path order (as indicated by the numbers next to the edges) of all available paths (due to its circuit breaker being *closed*).

The following circuit breaker transitions are used in Pathfinder (as depicted in the state diagram in Figure 2.8):

- **Closed to open.** The most important transition is arguably the opening of the circuit breaker (which corresponds to blocking the physical data flow). An implementation is conceivable where there are no other transitions and once opened, the circuit breaker would stay open. This would imply that the main path would no longer be usable and all subsequent data items are sent to the fallback path.

  While being in the closed state, the data flow is uninterrupted. The transition to the open state causes the flow to stop immediately. More precisely, Pathfinder advises the DSPE to stop the data flow by invoking an exposed REST endpoint. 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 mechanism takes place. Probing is the procedure to determine whether a previously failed operator has already recovered.

- **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 closed.** If the probing attempt succeeds and all operators of the paths are fully operational again, the circuit breaker transitions to the closed state and the normal data flow is restored.

4.5 Fault Tolerance-Oriented Development

Pathfinder is only moderately useful if it is solely used as a tool for supporting already existing SPAs. This is comparable to introducing object-oriented programming into a project after it has been implemented using imperative paradigms. Although there may be some places in the code where it is beneficial to upgrade, it is too late in general. Likewise, Pathfinder is most useful when new SPAs are developed.

Yang et al. [68] have coined the term "fault tolerance-oriented programming" where they advocate the integration of fault tolerance into CPU multi-core architectures. In a similar way, this work introduces *fault tolerance-oriented development* as a set of design principles that help dealing with potentially failing software systems in a distributed stream processing context.

4.5.1 Development Guidelines

There are several design and implementation guidelines one should follow in order to maximise the benefits of an SPA supported by Pathfinder.

G1 **Each operator can fail at any time.** There are many reasons why operators might fail, including hardware failures, operating system crashes, meteor strikes and – last but not least – software bugs in the operators themselves. By accepting that failures will eventually happen (since most SPAs run continuously for very long periods of time), they become less frightening. This can be compared to exception handling in ordinary programming languages. It is therefore recommended to treat each operator as if it could fail at any time. In particular, one should not rely on all computations to complete uninterruptedly.

G2 **Availability first.** More than anything else it is Pathfinder's goal to maximise an SPA's availability. This will inevitably conflict with the goal of high consistency in the cases of failure. One cannot argue that this design decision is suitable for all kinds of applications, but there are already approaches available focusing on consistency (see Section 3.1.2). There are many scenarios where the focus on availability is suitable that can substantially profit from our approach. To utilise this behaviour, SPA developers need to design their applications in such a way that a few missing results will not disturb aggregated end results. In return, high availability can be expected.

G3 **Utilise functional redundancy.** From the first two recommendations, it also follows that the application designer must come up with a plan B in case of operator failures, because otherwise the high availability cannot be guaranteed. As a rule of thumb, each operator should be at least replaceable by a single fallback path. Availability in general increases with decreasing fallback path length and with an increasing number of fallback paths per operator (see Section 4.5.3).

Example 4.9

We consider the following topology:

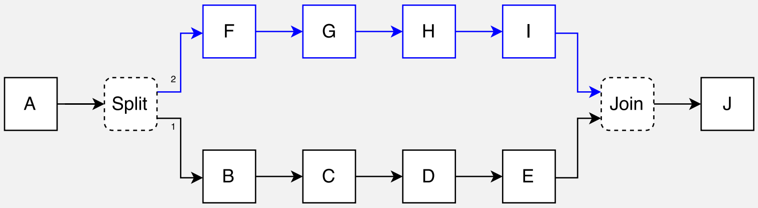


Figure 4.10: A topology of 10 processing operators that also contains a split and a join operator.

By adding a single fallback path (F, G, H, I) that provides a similar functionality as the main path (B, C, D, E), failures in those operators can be compensated. However, the fallback path itself may also suffer from faults. Having multiple fallback paths would guarantee availability even in the unlikely event of multiple failures.
Now we consider an alternative topology that uses two smaller fallback paths instead of one to cover the same path as before:

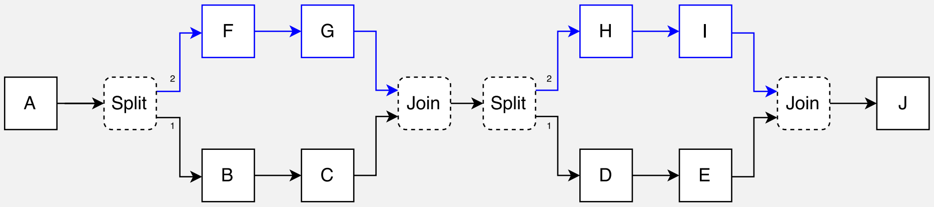


Figure 4.11: A topology of 10 processing operators with two split and two join operators.

This approach is much safer. We consider the case of one single failing operator in the original main path. This will either activate the fallback path (F, G) or (H, I). However, the availability of the whole application then only depends on the active fallback path consisting of *two* operators rather than on four as in the previous example.
While designing smaller fallback paths is safer, it may also be more difficult and can converge to *n-version programming* for paths of length one.

4.5.2 Allocation of Labour

There is another interesting advantage of a fault tolerance-oriented development. We consider the case where a team of ten software developers is given the task to design and implement a mission-critical SPA. We further assume that their SPA consists of twenty operators and Pathfinder is not used. Since there is no redundancy, extreme care must be taken to identify and fix all bugs in the software by exhaustive testing techniques. Nevertheless, if one of the operators fails, the whole SPA becomes unavailable.

Now we consider the same scenario with the same team of developers but this time, they are using the Pathfinder framework to support path-level redundancy. By initially defining fallback paths for distinct tasks, the development can be better split into independent tasks and less care must be taken to identify bugs since in the case of crashes or exceptions, the fallback path can be used.

4.5.3 Estimating Topology Resilience

To give developers concrete advice for their SPA, it is possible to analyse topologies with respect to their level of resilience.

Laprie [69] defines resilience as *"the persistence of service delivery that can justifiably be trusted, when facing changes"*. This definition is also applicable to SPAs that are controlled by Pathfinder. Service delivery is present if at least one alternative path of a split operator does not contain any failing operators. Whether service delivery *continues* in the future facing changes (i.e., failing operators) is dependent on the number and availability of alternative paths: service delivery in the future is more likely if faults can be tolerated.

One can try to quantify resilience of a particular SPA topology by considering

- the number of operators for which no alternative paths are available at all,
- the average length of all alternative paths, and
- the average number of alternative paths.

For each operator where no alternative path is available, the whole SPA fails if a single operator fails. Conversely, an SPA's overall availability increases with the number of alternative paths and their shortness.

We consider a theoretical scenario where the probability of failure in the next 24 hours is p for each operator.

First, we assume an extreme case where no alternative paths are available at all and n operators are connected linearly (that is, there is a single path of length n that contains all operators). The probability of failure is then $1 - (1 - p)^n$.

Next, we consider a case where each of those n operators has one alternative path of length one (therefore, having a total of $2n$ operators in the topology now). The probability of failure for each pair of operators is then p^2. The probability of failure for the whole topology is then $1 - (1 - p^2)^n$.

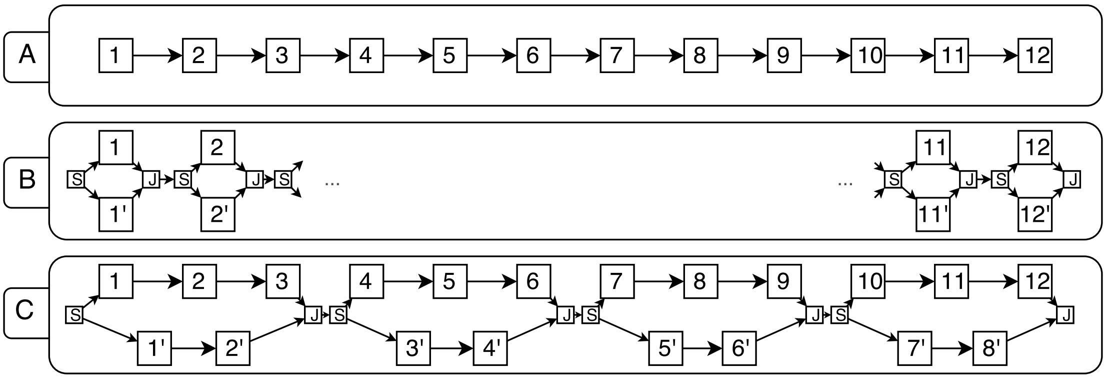


Figure 4.12: Topologies for computing probability of failure with fallback paths..

Example 4.10

For example, we consider a topology of $n = 12$ operators (shown in Figure 4.12 topology A) and a probability of failure of $p = 0.05$ (i.e., if the operator is run for a hundred days, it will be free of failures in 95 of those days). Without alternative paths, its probability of failure is $1 - (1 - 0.05)^{12} = 0.4596$. In contrast, with the alternative paths of length one (shown in Figure 4.12 topology B), the probability of failure is just $1 - (1 - 0.05^2)^{12} = 0.0296$.

As an alternative, we consider a topology with four alternative paths that each cover 25% of the operators (i.e., the first 25% of the operators have a single fallback path of length two, the second 25% have one, and so on). For each of the four main paths, the probability of failure is $1 - (1 - p)^{n/4}$. Since each main path has a fallback path with the probability of failure $1 - (1 - p)^2$, the overall probability of each 25% segment is $(1 - (1 - p)^{n/4})(1 - (1 - p)^2)$. The whole SPA then consists of four such blocks one after another. The overall probability of failure is therefore $1 - (1 - (1 - p)^{n/4})(1 - (1 - p)^2)^4$.

Example 4.11

For a topology with four segments of alternative paths (Figure 4.12 topology C) where $n = 12$ and $p = 0.05$, the probability of failure is $1 - (1 - (1 - 0.95^3)(1 - 0.95^2))^4 = 0.0545$.

The lesson here is that the design of shorter alternative paths enhances overall availability while having a comparable development effort. This of course restricts the structure of alternative paths since they need to produce intermediary results that are compatible for all succeeding alternative paths.

Figure 4.13 shows a topology and uses colour codes to represent resilience. Sources (A and K) and sinks (Ω) are shown in grey. The operators in the first split/join segment (B to I) are shown in green since they are very resilient due to a large number (three) of short (average length 2.67) alternative paths. Operators J, L and M show very little resilience since a failure of one of those operators would lead to a failure for the whole

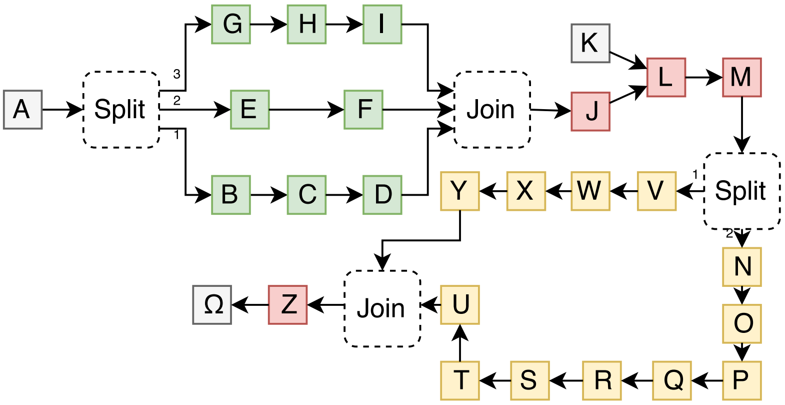


Figure 4.13: Colour coded resilience. Green, yellow and red depict a high, medium and low level of resilience, respectively.

SPA. The operators of the next split/join segment show a comparatively low resilience due to the length of the two alternative paths (6.0 on average). Long alternative paths are unfavourable since the failure probabilities in one alternative path accumulate. Finally, operator Z shows again little resilience due to a lack of alternative paths.

Colour coding the topology this way can be a very useful tool for SPA developers since it allows them to directly identify weak portions of the topology that need attention. With this figure in mind, it is straightforward to prioritise the development of alternative paths for the operators J, L, M and Z.

CHAPTER 5

Implementation

This chapter discusses the implementation of the Pathfinder fault tolerance framework. It is based on the software architecture presented in Chapter 4.

Following a quick overview of the technology stack in Section 5.1, the implementation details of the three Pathfinder modules — the *Communicator* (Section 5.2), *Nexus* (Section 5.3) and *Circuit Breaker* (Section 5.4) — are discussed. Then, Section 5.5 addresses the changes that were made to VISP in order to enable its interoperability with Pathfinder. Finally, Section 5.6 introduces Pathfinder's Web front-end.

5.1 Technology Stack

Pathfinder is developed as a standalone application that communicates with VISP via REST calls. Figure 5.1 shows an overview of Pathfinder's technology stack. The implementation uses *Java 8*[1] based on the *Spring Boot* framework[2]. It uses *Apache Maven*[3] as a build automation tool (thereby enabling reuse of VISP data structures from the *VISP Common* module[4]). For object persistence, the Hibernate[5] library is used on top of a MySQL database[6].

Persistence is used mainly for storing and retrieving operational statistics gathered from VISP. To make Pathfinder accessible, a Web-based front-end has been developed using the *jQuery*[7] JavaScript library for making asynchronous REST calls to the back-end

[1] `https://www.java.com/`
[2] `https://projects.spring.io/spring-boot/`
[3] `https://maven.apache.org/`
[4] `https://github.com/visp-streaming/common`
[5] `http://hibernate.org/`
[6] `https://www.mysql.com/`
[7] `https://jquery.com/`

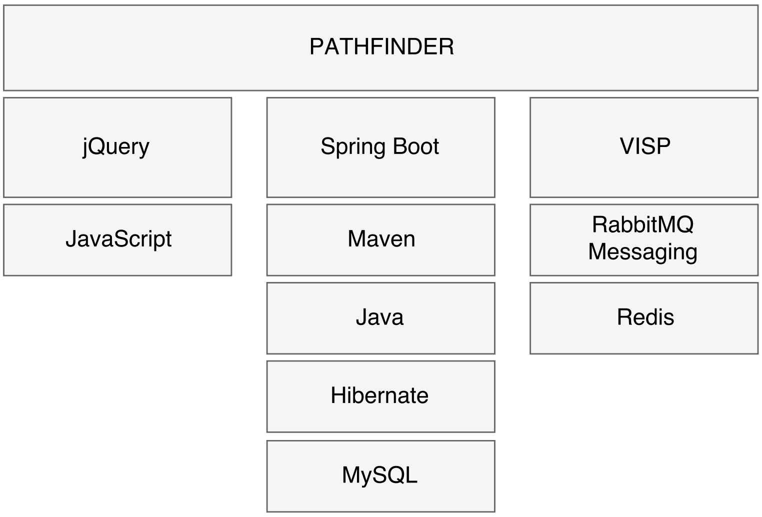


Figure 5.1: Pathfinder's technology stack.

server. Asynchronous JavaScript enables the inclusion of a dashboard into the front-end that automatically refreshes and shows topology performance indicators for human operators.

Pathfinder is provided as a Docker[8] image which allows the easy deployment on any host supporting Docker. Such a deployment has the advantage that no dependencies need to be installed manually.

Although Pathfinder can be used with any DSPE that provides suitable interfaces, this implementation is tailored specifically to work with VISP [5]. VISP is also based on Spring Boot, Hibernate and MySQL. Additionally, it uses RabbitMQ[9] as a communication infrastructure and Redis[10] as a data structure store. Figure 5.2 shows the full deployment of Pathfinder with VISP. Pathfinder is executed on its own host and communicates with VISP by means of REST calls via the *Communicator* which in turn manages the operators on its resource pools.

5.2 Communicator

Pathfinder needs to connect to VISP in order to query it for operational statistics and to enable or disable alternative paths. Furthermore, communication to other Pathfinder instances must be handled (e.g., to share DSPE statistics and to reach consensus about operator availability).

[8] http://docker.com/
[9] https://www.rabbitmq.com/
[10] https://redis.io/

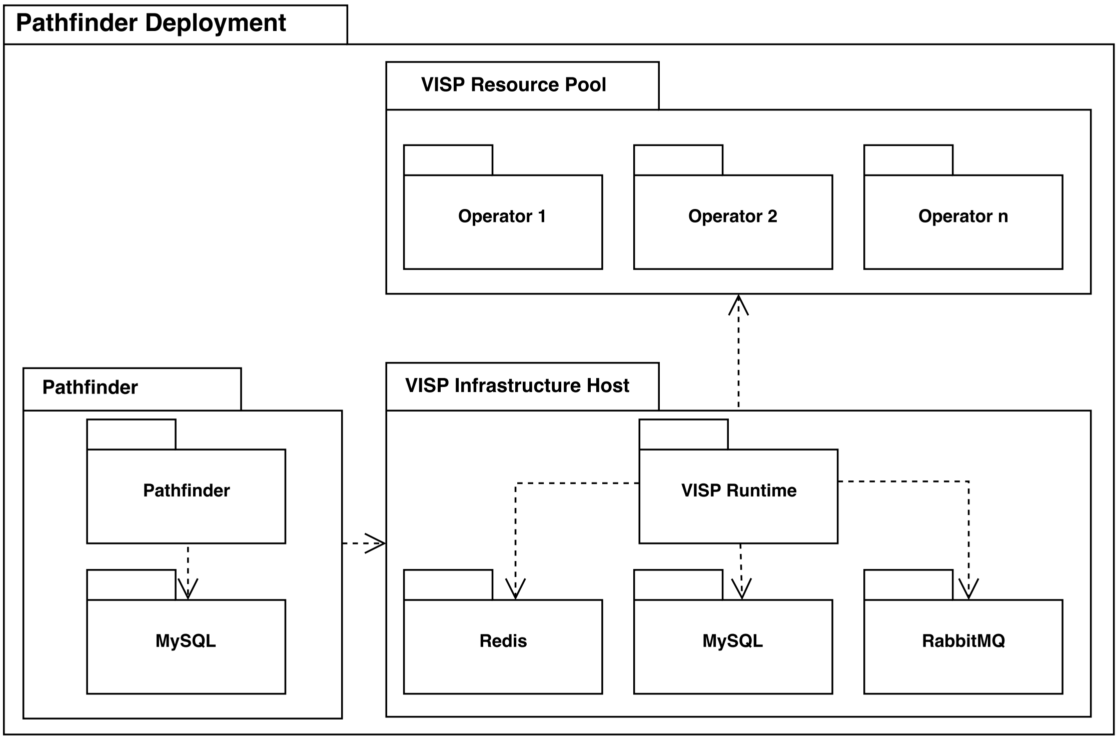


Figure 5.2: **Deployment diagram of Pathfinder and VISP.**

5.2.1 Scheduler

Pathfinder uses a pull-based information gathering mechanism, i.e., it needs to actively request statistical data from VISP. Therefore, the `Scheduler` service regularly invokes the `getStatisticsFromAllRuntimes()` method to trigger a topology retrieval (if it has changed; see Section 5.2.2) and to fetch operational statistics. Using the `@Scheduled(fixedDelay = 10000)` annotation provided by Spring Boot, it is specified that such retrievals happen with pauses of 10 seconds between two invocations.

Example 5.1

The first invocation of `getStatisticsFromAllRuntimes()` occurs at $t = 0$ and takes 8 seconds. At $t = 8$, the 10 second waiting interval starts. The next method invocation occurs at $t = 18$.

There are also two other scheduled methods: `updateTopologyStability()` recomputes the topology stability (an indicator describing the topology's stability displayed in the dashboard) every 15 seconds and `updateCircuits()` computes the new circuit breaker states every second. The latter function is called more frequently than new statistical data is retrieved. The reason for this difference in frequency is that circuits

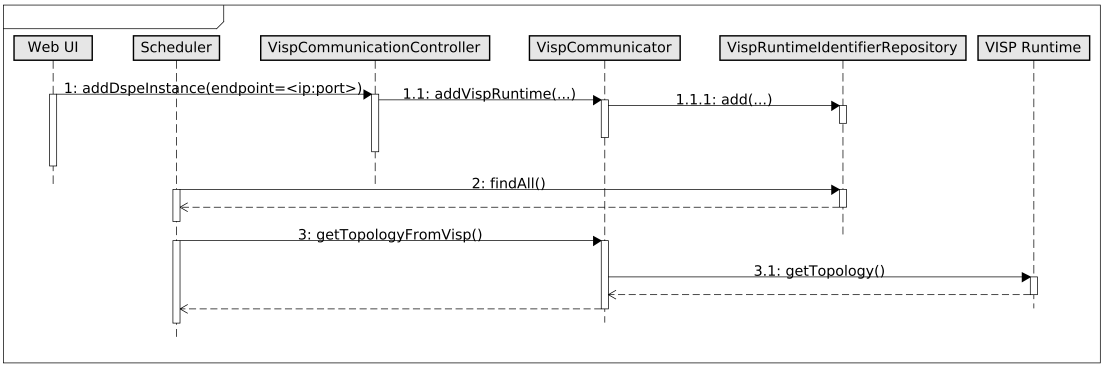


Figure 5.3: Retreiving VISP's topology.

in the half open state have a cooldown period and the `updateCircuits()` must regularly check whether that period is already over.

5.2.2 Topology Retrieval

The `DSPECommunicationController` interface is provided by Pathfinder to communicate with VISP. Specifically, the `VispCommunicationController` implements this interface to coordinate the communication to VISP. Once the `/addDspeInstance` API is invoked with the appropriate IP and port of the VISP Runtime, the `VispCommunicator` service queries the VISP runtime for its current topology. Figure 5.3 shows a sequence diagram depicting this procedure. First, the Pathfinder user accesses the Web front-end and inputs the IP and port of the VISP runtime. The `VispCommunicationController` then persists this runtime identifier to the local repository that is then automatically queried by the `Scheduler` service. By accessing VISP's `/getTopology` REST endpoint, the `VispCommunicator` finally fetches the topology in the VTDL format and stores it locally for later use by the *Nexus* and *Circuit Breaker* modules.

If the newly fetched topology differs from the locally stored one, a topology update took place. This indicates that at least some operators have been changed by VISP. As a reaction, Pathfinder resets all circuit breakers and creates new ones if necessary if new alternative paths have been added.

5.2.3 Fetching Operational Statistics

The *Nexus* module detects faults based on statistical data about operators. It is therefore the responsibility of the *Communicator* to regularly fetch these statistics provided by VISP.

In VISP, statistical data is available at the `/pathfinder/getAllStatistics` endpoint. The `Scheduler` service invokes this endpoint automatically every 15 seconds.

Listing 5.1: Operational statistics as returned by VISP's `/pathfinder/getAllStatistics` endpoint.

```
[
  "position_to_keywords_user_tags": {
    "network_out": 1932,
    "network_in": 2430,
    "delivery_rate": 1,
    "cpu_now": 0.3019249969849246232,
    "ram_now": 303,
    "items_waiting": 0
  },
  "position_to_companies": {
    "network_out": 0,
    "network_in": 0,
    "delivery_rate": 0,
    "cpu_now": 0.0006677819095477388,
    "ram_now": 303,
    "items_waiting": 24
  },
  //...
]
```

Internally, VISP stores operational statistics from Docker for each operator instance in a local MySQL database. The statistics include CPU usage, memory usage, network utilisation, as well as information about the RabbitMQ infrastructure such as queue size and data item throughput. VISP's statistics response consists of a JSON-encoded map where the current values of those statistics are provided for each operator. Listing 5.1 shows an exemplary response for two operators `position_to_keywords_user_tags` and `position_to_companies`. While the former one is operating normally, the second operator is experiencing a failure as can be seen by the low CPU utilisation and the high number of items waiting in the queue. The *Communicator* persists this information with the request's timestamp. This way, historical data about the operators is available that can be used by the *Nexus* component for classification purposes.

5.2.4 Path Commands

VISP needs to provide interfaces for Pathfinder to allow the execution of several path-related commands:

- **Add data flow to an alternative path.** This command is needed whenever an alternative path is activated (either because it is a fallback path and the main path has failed or due to probing attempts). VISP offers the `/pathfinder/switchAlternative` endpoint where the data flow of a specific split operator can be set to one of its alternative paths.

- **Deploy alternative paths' operators**. For VISP, no specific command is needed to deploy the operators of an alternative path since this is also handled by the `/pathfinder/switchAlternative` invocation.

- **Probing**. By invoking VISP's `/pathfinder/probe` endpoint, a data flow to a specific alternative path is initiated for a short window of time in order to find out whether previously failed operators have recovered (see Section 5.4.2).

5.3 Nexus

The *Nexus* component is responsible for classifying operators. The `INexus` interface specifies the `predict()` method that takes a `SingleOperatorStatistics` object as its argument and returns an `OperatorClassification` enum (`WORKING` or `FAILED`).

5.3.1 Operational Statistics

The `SingleOperatorStatistics` object contains statistics about an operator instance. The data is collected by VISP and is persisted as a JSON object when the *Communicator* module queries VISP. However, not all attributes are necessarily used in the classification process.

5.3.2 Rule-Based Classification

The `RuleBasedNexus` class implements the `INexus` interface and classifies operators based on their CPU and memory usage as well as the number of items in the RabbitMQ queue. In particular, an operator is considered as failed if either

1. more than `MAX_QUEUE` data items are in the operator's queue and the CPU usage is below `MIN_CPU`,

2. the memory usage is below `MIN_MEMORY` MB and the CPU usage is below `MIN_CPU`,

3. the memory usage is above `MAX_MEMORY` MB, or

4. the rate of data item processing is lower than the rate of incoming data items.

Cases 1 and 2 strongly indicate that a failure occurred in the operator (in the first case because the operator is not processing data while items are available to fetch and in the second case because the low memory and CPU usages are uncharacteristic for Java-based operators). In case 3, an unusually high amount of memory is used by the operator which also suggests a malfunction. Finally, case 4 does not identify a functional failure but rather the inability to process data items fast enough. This is a problem as well since it leads to an overflow in the operator's queue in the long run. The concrete values for the variables MAX_QUEUE, MIN_CPU, MIN_MEMORY and MAX_MEMORY can be set by the user according to the particular topology.

These rules are in no way universally valid and need to be adapted to the specific operators that are used. Creating a customised *Nexus* can be accomplished by implementing the INexus interface (e.g., for environments where less memory usage or a higher incoming data item rate is expected). Future work in Chapter 7 contains suggestions for more sophisticated ways to implement this module.

5.4 Circuit Breaker

A basic CircuitBreaker class has been implemented that stores the active state (closed, open or half open) as well as the last time a probing attempt has been made, and that allows transitions between states. The SplitDecisionService creates and manages a map that assigns a circuit breaker object to each alternative path (see Figure 4.9). After the operator classification took place, the circuit breakers' states are changed according to the availability of the operators.

5.4.1 Circuit Breaker Management

Once a new topology is fetched from VISP, new circuit breaker objects are created for each alternative path by the SplitDecisionService. The identifiers of the alternative paths are derived directly from the VTDL file by the *VISP TopologyParser*[11]. Each circuit breaker starts in the closed state since initially, it is assumed that all operators are operating correctly.

Each time the operator availability changes, the ProcessingOperatorHealth service reassesses whether this change affects any of the alternative paths. 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, it transitions that path's circuit breaker into the open state and activates an alternative path. 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 path order. It then advises VISP to change the physical 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.

[11] https://github.com/visp-streaming/topologyParser

Example 5.2

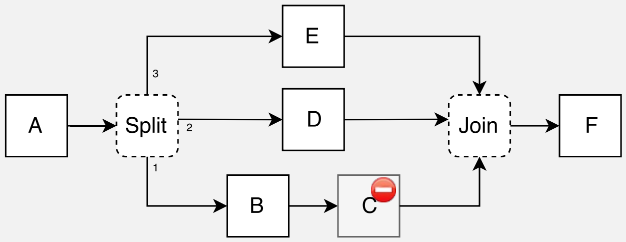


Figure 5.4: Selection of the alternative path.

In the logical topology view above, the main path contains a failing operator. Therefore, Pathfinder tries to find a suitable alternative path. There are two such paths, both of which are available. Pathfinder therefore picks path *D* since it has the lowest path order.

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. In this state, probing attempts are initiated.

5.4.2 Probing

Figure 5.5 shows a sequence diagram of the probing mechanism. First, the *Circuit Breaker* module determines that an alternative path with a circuit breaker in the open state is to be probed again. Then, the circuit breaker's state is transitioned to the half open state and VISP's `/pathfinder/probe` endpoint is invoked to start the probing procedure. Next, VISP adapts the RabbitMQ infrastructure by creating a queue-exchange binding that causes physical data flow to the alternative path (while leaving the data flow to the main path intact). After a short amount of time, this data flow is removed again. All the data items that have been produced between the activation and the deactivation are processed by the probed path (unless it is still failing to do so). 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 amount of time (cooldown) has passed. This mechanism ensures that no resources are wasted for an inactive path (especially in long paths containing many operators). The data items that have been produced during the probing attempt are automatically removed from the operator's queue after 10 minutes (see Section 5.5.1).

5.5 VISP Contributions

The fact that VISP is used as the slave DSPE of Pathfinder is reflected in several ways:

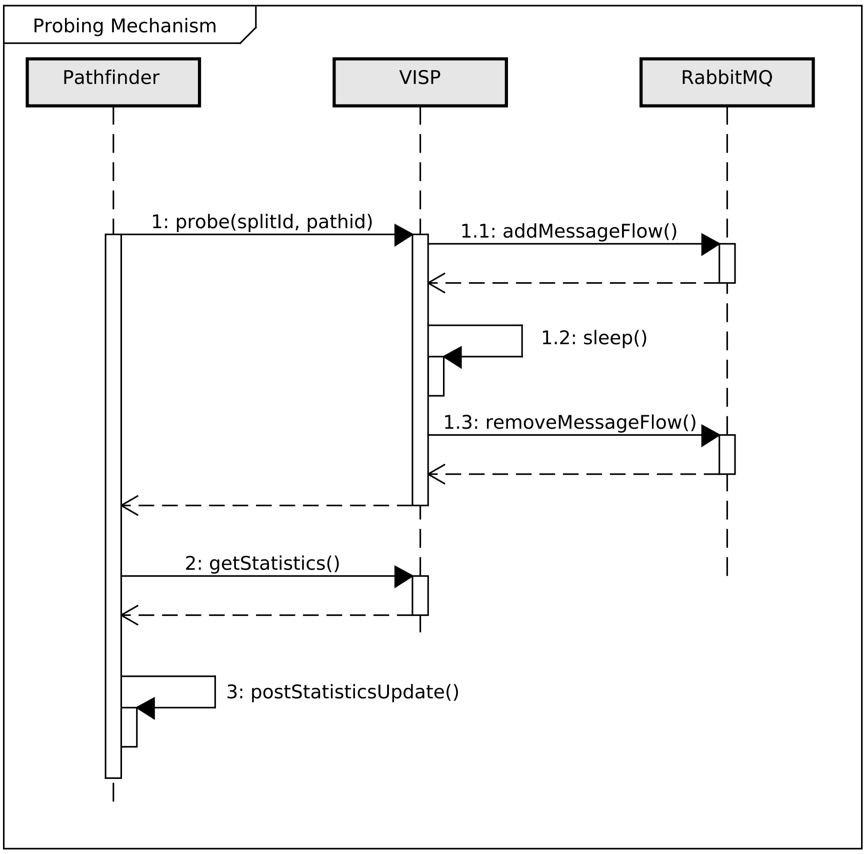


Figure 5.5: Probing mechanism.

1. **Receiving the current topology**. In order to fetch the currently active topology, Pathfinder calls VISP's REST API which returns the topology in the VTDL format. It then uses the *VISP TopologyParser* to convert the VTDL file into a machine-readable format that is then stored locally. Since the topology can be changed at runtime by the slave DSPE, Pathfinder periodically checks for updates.

2. **Checking operator health**. In order to determine whether an operator is working correctly or has failed, Pathfinder queries VISP for operational statistics.

3. **Switching paths**. Once Pathfinder has made the decision to change the currently active path of a split operator, it must communicate this change to VISP. For this purpose, it makes a REST call specifying the split operator's ID and the new active path. This invocation is idempotent, i.e., Pathfinder does not need to wait for an acknowledgement of the message but can rather re-transmit it in the next scheduling interval if it has not detected an according topology change in the meantime.

4. **Probing**. When in the half open state, the circuit breaker of an alternative path must enable data flow from time to time to see whether the operators have recovered (see Section 4.4.4). With VISP, this mechanism is implemented by changing the binding of queues to exchanges (see Section 5.4.2).

Again, making these decisions is DSPE-specific and is not to be seen as a restriction to Pathfinder in general.

Example 5.3

To operate Pathfinder with another DSPE that does not support the VTDL format, one needs to add customised code to Pathfinder that translates the topology. Similarly, if a DSPE does not directly allow the switching of alternative paths, one needs to perform this action indirectly using the topology modification mechanisms that are available.

5.5.1 Lazy Deployment

When an alternative path is activated by Pathfinder, VISP must ensure that all of its operators are deployed. One possible approach for VISP is to deploy every operator of the topology at initialisation regardless of its membership in fallback paths. This way, operators are continuously ready for service and the path switching process happens very quickly. However, this comes with the caveat that resources must be reserved for operators that are not used until faults occur. If this strategy is chosen, it is questionable why not to simply run a replica of the first system on different resources and use that system as an active-standby backup (although this would ignore the benefit of having different implementations that can fill in for each other).

Therefore, another approach is to delay operator deployment until it is necessary. This *lazy deployment* approach requires less computational resources at initialisation time but also during runtime since it is highly unlikely that all fallback paths are activated at the same time. Obviously, the caveat of this approach is the time it takes until an alternative path is ready for processing. During this time, the SPA does not produce results and would therefore possibly violate availability constraints.

Listing 5.2: VTDL file showing the usage of the lazyDeployment parameter.

```
$split = Split($step1) {
  lazyDeployment = true,
  pathOrder = $step2 $step3
}
```

Example 5.4

If lazy deployment is used with the topology below, only operators *A*, *B*, *C* and *E* are deployed at initialisation while operator *D* is not. Only when the failure of operator *C* is detected, Pathfinder commands VISP to deploy *D* in order to use it as a fallback. While this saves resources, it causes an additional delay since no results are produced between the failure of *C* and the complete deployment of *D*.

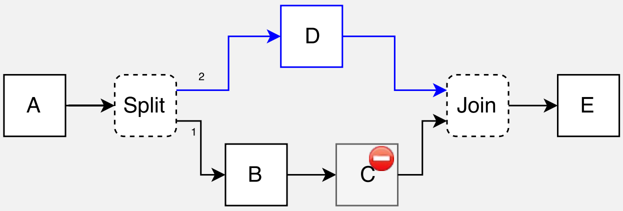


Figure 5.6: Lazy deployment.

To find a tradeoff between availability and resource consumption, Pathfinder allows the user to adjust its behaviour by using configuration parameters in the VTDL file. Specifying such parameters directly by language-level annotations comes natural for the SPA developer and is already best practise in other fault tolerance approaches [70]. Using the `lazyDeployment` parameter, one can specify for each split operator whether the alternative path's operators' deployment must already happen at topology initialisation (*false*; default) or not until the activation of the fallback path (*true*). Listing 5.2 shows an excerpt of a VTDL file that shows the usage of the `lazyDeployment` parameter. There, the split operator `split` has two alternative paths. Only the operators of the main path are deployed at runtime while the operators of the fallback path `step3` are deployed only when a fault in `step2` is detected.

When lazy deployment is used, several additional steps occur after the activation of an alternative path. At first, the messaging infrastructure is set up by creating queues and exchanges. As soon as the first queue for the new alternative path has been created, the data flow is redirected from the failing main path to the fallback path. Data items pile up in the operator's queue until the operator deployment has been finished and the newly started operator is able to process them. By automatically discarding data items older than a certain time interval (e.g., 10 minutes), it is assured that the alternative path does not waste time processing obsolete data. Listing 5.3 shows how such a mechanism is implemented using RabbitMQ's time-to-live feature. By specifying the

Listing 5.3: **Using RabbitMQ's TTL feature to automatically discard old data items**.

```
1  Connection connection = createConnection();
2  Channel channel = connection.createChannel();
3
4  //declare the exchange:
5  channel.exchangeDeclare("exchange-name", "fanout", true);
6  Map<String, Object> args = new HashMap<String, Object>();
7  int maxTtl = 10 * 60 * 1000;
8  args.put("x-message-ttl", maxTtl);
9
10 //declare the queue:
11 channel.queueDeclare("queue-name", true, false, false, args);
12
13 //bind queue to exchange
14 channel.queueBind("queue-name", "exchange-name", "");
```

`x-message-ttl` argument (line 8) when declaring a queue, RabbitMQ automatically discards data items that have been on that queue for a longer time than the value of that parameter.

While a fallback path is in use, attempts are made to recover the failed main path (see Section 5.2.4). Once Pathfinder reports that the main path is fully operational, the physical data flow is redirected to the main path again and the operators of the fallback path are removed if lazy deployment is used.

5.5.2 Path Switching

In order for Pathfinder to be able to switch between different alternative paths, the RabbitMQ message distribution mechanisms must be adapted. There are two requirements that must be fulfilled:

1. **Switching paths must be quick**. The probing mechanism used by Pathfinder (see Section 4.4.4) determines whether a path is able to correctly process data items. A probing attempt forwards data items only for a small time window in order to avoid putting additional load on an already overloaded system and save costs (e.g., in cases where an external API is invoked). The creation and removal of physical data flows is very time-efficient in VISP due to the underlying RabbitMQ infrastructure (as demonstrated in the evaluation experiment in Section 6.2.2).

2. **Data items must be duplicated.** 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 both paths successfully process the same data items is tolerated since Pathfinder only guarantees *at-least-once* and not *exactly-once* delivery.

5.5.3 Split and Join Operators

In contrast to other operators, split and join operators are not deployed by VISP on computational resources. Instead, they are just a semantic entity that allows a simple notation of fallback paths for SPA developers. Whenever a split/join pair is identified during the VTDL parsing process, the following rules are applied:

- a data flow from an operator A to a split operator S is replaced by a data flow from A to all operators directly downstream of S,
- a data flow from an operator B to a join operator J is replaced by a data flow from J to all operators directly downstream of J,
- a queue-exchange binding from the exchange of A is only created for the queue of the main path's first operator.

Example 5.5

Below the physical and logical views of a topology consisting of four processing operators A, B, C, D that are mapped to the messaging infrastructure are shown. While the *Split* operator is no longer existing as a separate entity at the infrastructure level, its semantic information is captured in operator A, i.e., there is a queue for each alternative path in A that can be bound to A's exchange. Therefore, there are two separate queues (Q_C and Q_B) for A.
In the default state, there is only a physical data flow between the exchange (X) of A to queue Q_B since B is the alternative path with the lowest path order (1). If B fails, the physical data flow from X to Q_C is established while the one to Q_B is removed and all further data items are sent only to Q_C. Once in a while, the probing mechanism briefly activates the data flow to Q_B again in order to check whether B has recovered.

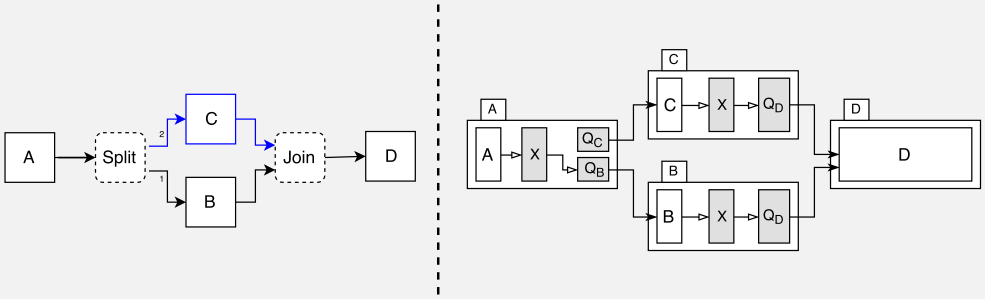


Figure 5.7: Left: Logical view of a topology consisting of four operators. **Right:** Physical view of the same topology showing the data flow as well as the created exchanges and queues. Since operator C is only part of a fallback path, there is no physical data flow from A to C unless operator B fails.

While initially only the main path is active, the sequence diagram in Figure 5.8 shows how Pathfinder can change which alternative path is active. Once the `SplitDecisionService` opens an active path's circuit breaker, it invokes `VispCommunicator`'s `switchSplitToPath` method which contacts the VISP Runtime via a REST endpoint

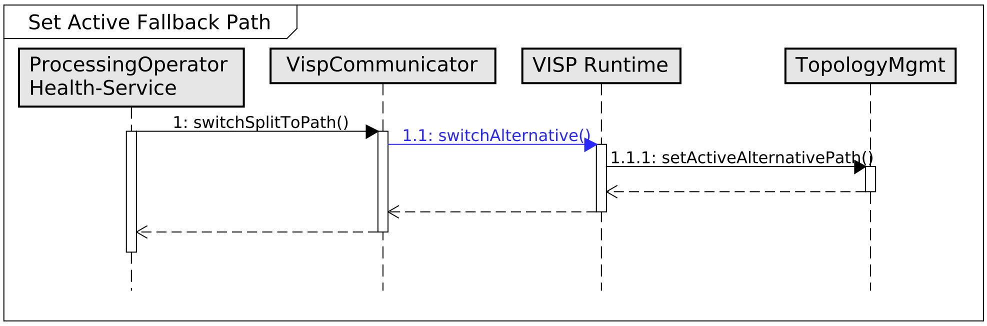


Figure 5.8: Setting the active alternative path.

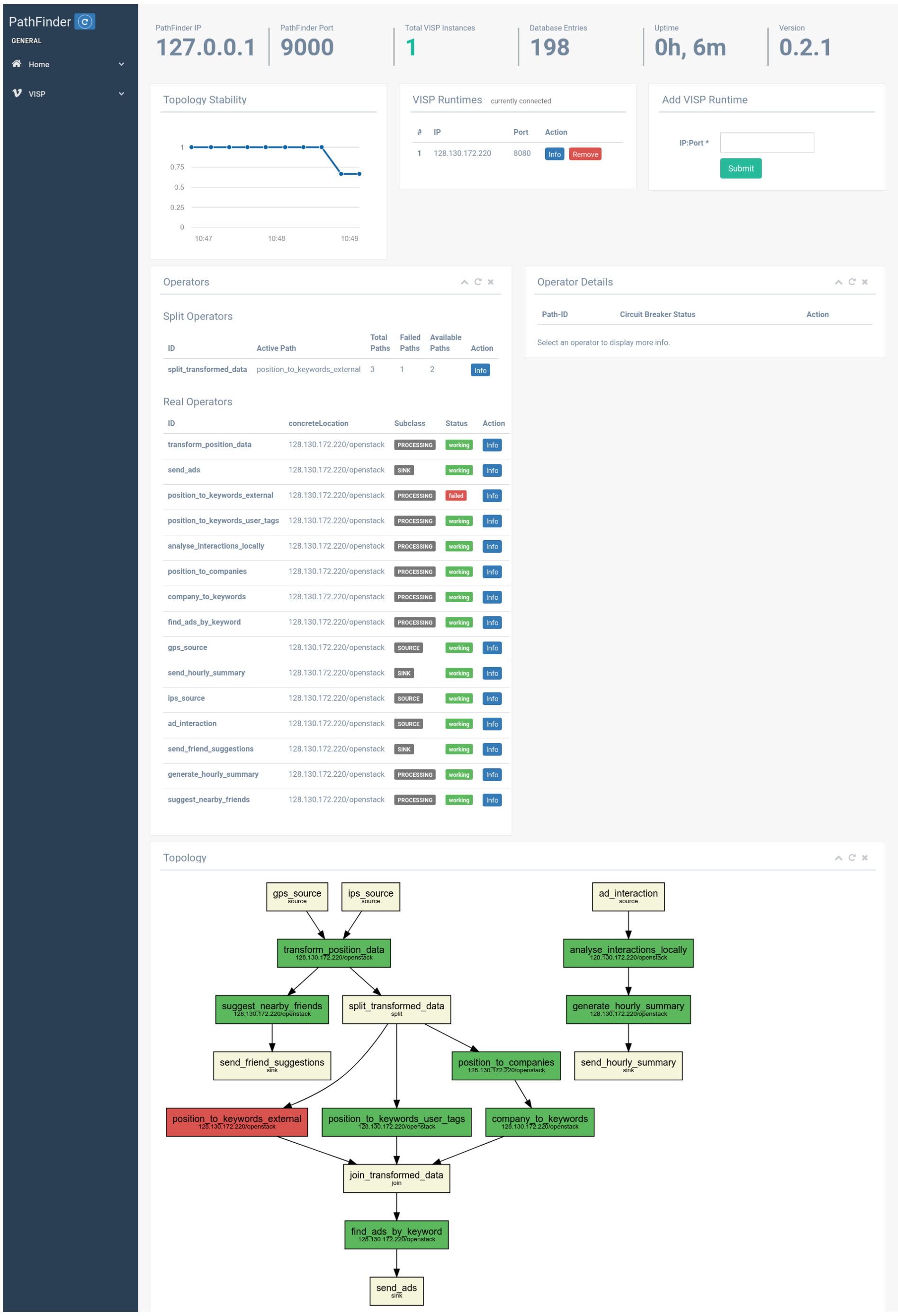
invocation (message 1.1 depicted in blue) and instructs it to change the active alternative path by changing the queue-exchange bindings appropriately using the `TopologyManagement` service as shown in Figure 5.7.

5.6 Front-End

A Web-based front-end has been created to allow the monitoring of the topology's stability and to add or remove VISP Runtime instances. Figure 5.9 shows a screenshot of the front-end. On top, key figures (IP, port, the number of connected instances, the number of accumulated database entries, uptime and software version) are shown. A text input form can be used to enter the IP and port of a new VISP Runtime instance Pathfinder connects to. The front-end also allows the removal of VISP Runtimes.

In the left part of the first row, a continuously refreshed diagram shows the current topology stability. Topology stability is a number between 0 (low stability) and 1 (high stability) and is computed as an average over the fraction of active alternative paths over all split operators.

Below the topology stability, an overview over all operators with their locations and types are shown. Additionally, the operators' health status are indicated. On the very bottom of the Web page, the topology is depicted using colour codes (green and red for working and failed operators, respectively).



PathFinder

GENERAL

Home

VISP

PathFinder IP	PathFinder Port	Total VISP Instances	Database Entries	Uptime	Version
127.0.0.1	9000	1	198	0h, 6m	0.2.1

Topology Stability

VISP Runtimes currently connected

#	IP	Port	Action
1	128.130.172.220	8080	Info Remove

Add VISP Runtime

IP:Port *

Submit

Operators

Split Operators

ID	Active Path	Total Paths	Failed Paths	Available Paths	Action
split_transformed_data	position_to_keywords_external	3	1	2	Info

Real Operators

ID	concreteLocation	Subclass	Status	Action
transform_position_data	128.130.172.220/openstack	PROCESSING	working	Info
send_ads	128.130.172.220/openstack	SINK	working	Info
position_to_keywords_external	128.130.172.220/openstack	PROCESSING	failed	Info
position_to_keywords_user_tags	128.130.172.220/openstack	PROCESSING	working	Info
analyse_interactions_locally	128.130.172.220/openstack	PROCESSING	working	Info
position_to_companies	128.130.172.220/openstack	PROCESSING	working	Info
company_to_keywords	128.130.172.220/openstack	PROCESSING	working	Info
find_ads_by_keyword	128.130.172.220/openstack	PROCESSING	working	Info
gps_source	128.130.172.220/openstack	SOURCE	working	Info
send_hourly_summary	128.130.172.220/openstack	SINK	working	Info
ips_source	128.130.172.220/openstack	SOURCE	working	Info
ad_interaction	128.130.172.220/openstack	SOURCE	working	Info
send_friend_suggestions	128.130.172.220/openstack	SINK	working	Info
generate_hourly_summary	128.130.172.220/openstack	PROCESSING	working	Info
suggest_nearby_friends	128.130.172.220/openstack	PROCESSING	working	Info

Operator Details

Path-ID	Circuit Breaker Status	Action

Select an operator to display more info.

Topology

gps_source
source
ips_source
source
ad_interaction
source
transform_position_data
128.130.172.220/openstack
analyse_interactions_locally
128.130.172.220/openstack
suggest_nearby_friends
128.130.172.220/openstack
split_transformed_data
split
generate_hourly_summary
128.130.172.220/openstack
send_friend_suggestions
sink
position_to_companies
128.130.172.220/openstack
send_hourly_summary
sink
position_to_keywords_external
128.130.172.220/openstack
position_to_keywords_user_tags
128.130.172.220/openstack
company_to_keywords
128.130.172.220/openstack
join_transformed_data
join
find_ads_by_keyword
128.130.172.220/openstack
send_ads
sink

Figure 5.9: Pathfinder's Web front-end.

CHAPTER 6

Evaluation

In order to investigate how well Pathfinder performs, we have conducted several experiments. First, this chapter introduces the experimental methodology and setup in Section 6.1. Then Section 6.2 describes three preliminary experiments that are conducted to investigate different aspects of Pathfinder's fault tolerance mechanism in isolation. Section 6.3 evaluates Pathfinder itself by applying it to the motivational scenario introduced in Section 1.2.2. The evaluation results are presented and interpreted in Section 6.4. Furthermore, we discuss the general applicability of Pathfinder to the motivational scenario as well as its limitations in Section 6.5. Finally, the fulfilment of Pathfinder's initially defined requirements is evaluated in Section 6.6.

6.1 Evaluation Setup

All of the experiments are conducted using the following experimental setup:

- **Test infrastructure**: Experiments are performed on a private OpenStack[1]-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 the VISP Runtime and Pathfinder, respectively. Additionally, there are four more `m1.large` instances serving as a computational resource pool for spawning the operator instances used for data processing.
- **Evaluation topology**: An evaluation topology has been created based on the motivational scenario introduced in Section 1.2.2. It is used in all experiments and its implementation is described in detail in Section 6.1.1.

[1] `https://www.openstack.org/`

- **Data generation**: We use the VISP DataProvider[2] to generate data items in a reproducible way. We use the *constant* pattern, i.e., the production of new data items at a constant rate. We furthermore specify that a new data item is generated every two seconds. The DataProvider is deployed on the same virtual machine as the VISP Runtime to minimise network communication overhead and thereby to allow more accurate measurements of the troughput.
- **Introduction of faults**: Since naturally occurring faults (e.g., a failing hard drive) are rare, the behaviour of Pathfinder in case of faults is analysed by causing them artificially. We introduce three different kinds of failures to show how Pathfinder detects them and whether it reacts accordingly.

6.1.1 Evaluation Topology

A topology has been developed based on the motivational scenario. The VTDL file defining the topology is included in the Appendix (Listing A.1) and Figure 6.1 shows its logical topology view.

Every sink in the topology only consumes the data without any further action. For the remaining operators, Table 6.1 lists how much work they need to perform for each data item for the following tasks:

- `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.
- `S3_Ad_interaction`: Each time a social network user interacts with an ad campaign (e.g., by clicking on a banner), an ad interaction data item is produced.
- `P1_Transform_location_data`: This operator transforms data items from the sources S1 and S2 into a common format for further processing.
- `P2_Suggest_nearby_friends`: Based on the current position of the social network participants, 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_to_location_external`: The external *Atlas* service is used to retrieve a set of keywords for a specific location.
- `P4_Find_companies_near_distance_to_location`: Based on a location, a set of companies located in proximity is fetched from a database.

[2] `https://github.com/visp-streaming/dataProvider`

Listing 6.1: Computing Fibonacci numbers to simulate load.

```
public long _fibonacci(int n) {
    if (n <= 1) return n;
    else return _fibonacci(n-2) + _fibonacci(n-1);
}
```

- `P5_Get_keywords_by_company`: A company is transformed into a set of keywords based on what they sell.

- `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 all ad campaigns by restricting them to certain keywords.

- `P8_Analyse_interaction_data_locally`: Uses statistical methods to find ad campaigns with a low number of user interactions.

- `P9_Generate_hourly_summary`: Based on the analysis outcomes, a report is sent to a human operator every hour to inform them about ads with low conversion rates.

- `P10_Analyse_full_statistics_for_weekly_report`: A local system performs a correlation analysis to identify factors causing a high conversion rate depending on the stored information about the customers.

- `P11_Get_full_statistics_from_3rd_party_API`: An external third-party service is used to generate the raw data for the weekly report.

- `P12_Transform_statistics_response`: The response from the third-party service is transformed into a suitable format and annotated with data from the local customer database.

- `P13_Generate_PDF_report`: The weekly PDF report for the marketing department is created.

- `P14_Count_interactions_by_campaign`: Interaction numbers for each ad campaign are generated in order to display them in a customer dashboard.

Since fully implementing the motivational scenario is out of scope of this thesis, we used individual operators that just simulate a predefined load. In particular, the `SimulatedLoadController` class in VISP's *ProcessingNodes* module has been created. For each data item arriving at an operator's queue, the processing of that item is simulated by computing the Fibonacci sequence (as shown in Listing 6.1) until

Operator	*n*	*Avg. proc. time [s]*
P1_Transform_location_data	37	0.199
P2_Suggest_nearby_friends	43	2.115
P3_Get_keywords_near_distance_to_location_external	40	0.841
P4_Find_companies_near_distance_to_location	40	0.841
P5_Get_keywords_by_company	40	0.841
P6_Get_keywords_near_distance_to_location_by_user_tags	40	0.841
P7_Find_ads_by_keyword	40	0.841
P8_Analyse_interaction_data_locally	48	18.188
P9_Generate_hourly_summary	40	0.841
P10_Analyse_full_statistics_for_weekly_report	49	38.880
P11_Get_full_statistics_from_3rd_party_API	43	2.115
P12_Transform_statistics_response	37	0.199
P13_Generate_PDF_report	37	0.199
P14_Count_interactions_by_campaign	43	2.115

Table 6.1: Load simulation. This table shows how the different operators were assigned different simulated loads in order to reproduce a real-world scenario.

n where n is configured according to the operator type (the higher n, the longer the process takes and the less data items per second can be processed by the operator). The values of n for the different operators are shown in Table 6.1 and 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 shown in the last column.

6.2 Preliminary Experiments

In this section, we demonstrate Pathfinder's capabilities in isolation. First, we show that Pathfinder is in fact able to detect whether an operator is experiencing a failure by evaluating operational statistics provided by VISP. Then, the path switching mechanism introduced in Section 5.5.2 is evaluated. Finally, a theoretical experiment compares Pathfinder to a full replica setup regarding their operational cost.

6.2.1 Fault Detection

Pathfinder delegates the task of detecting faults to the Nexus component. The `RuleBasedNexus` class is the default Nexus implementation and works by comparing several operator statistics to predefined thresholds.

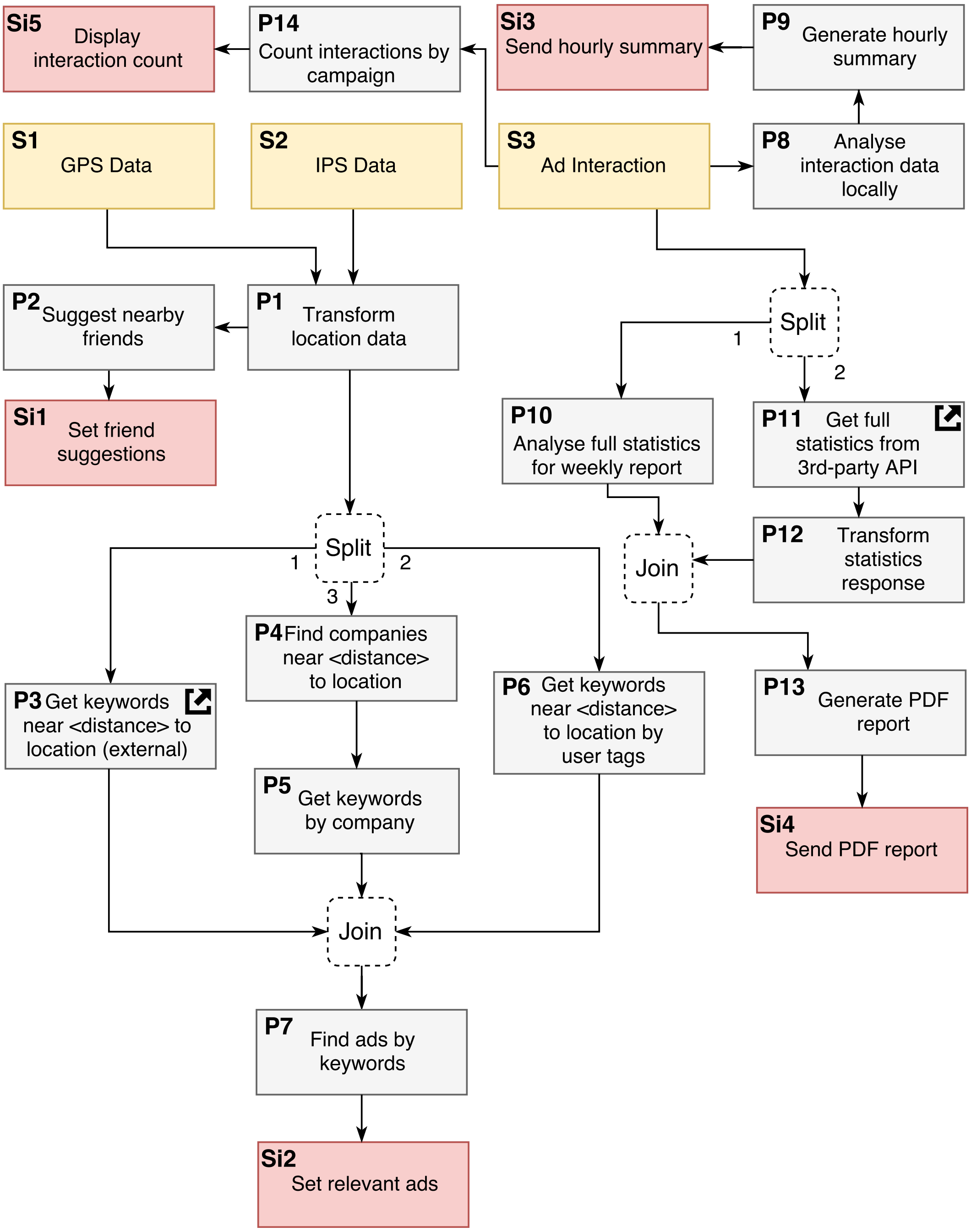

Si5 Display interaction count
P14 Count interactions by campaign
Si3 Send hourly summary
P9 Generate hourly summary
S1 GPS Data
S2 IPS Data
S3 Ad Interaction
P8 Analyse interaction data locally
P2 Suggest nearby friends
P1 Transform location data
Split
1
2
Si1 Set friend suggestions
P10 Analyse full statistics for weekly report
P11 Get full statistics from 3rd-party API
P12 Transform statistics response
Join
Split
1
2
3
P4 Find companies near <distance> to location
P3 Get keywords near <distance> to location (external)
P6 Get keywords near <distance> to location by user tags
P13 Generate PDF report
P5 Get keywords by company
Si4 Send PDF report
Join
P7 Find ads by keywords
Si2 Set relevant ads

Figure 6.1: **Topology for the motivational scenario.**

While the thresholds can be manually adjusted to the actual operators, this experiment shows that our general approach is valid. For the sake of clarity, we assume that there is exactly one operator instance per operator type (i.e., there is no scaling).

Listing 6.2: Obtaining CPU statistics from Docker. This bash script continuously queries the docker stats API to obtain the current CPU load of a specific container.

```
while true
do
  a=`date +%s`
  b=`docker stats $1 --no-stream | awk '{print $2}'`
  a+="\t"
  a+=$b
  echo -e $a
done
```

The goal of this experiment is to show that the CPU usage can be properly detected by Pathfinder using the statistics obtained by Docker. Listing 6.2 shows the code that is used to obtain the statistics (executed on the Docker host). It relies on the `docker stats` command to retrieve the CPU usage each second.

Figure 6.2 shows the CPU usage of a single Docker container that runs a processing operator. There are two phases that can be distinguished. From time $t = 0$ to $t = 205$, the CPU activity is constantly low. This corresponds to an inactive operator. At $t = 205$, the data flow is activated and the CPU activity fluctuates between 40% and 100%. This clearly indicates that an inactive operator can be detected using this mechanism.

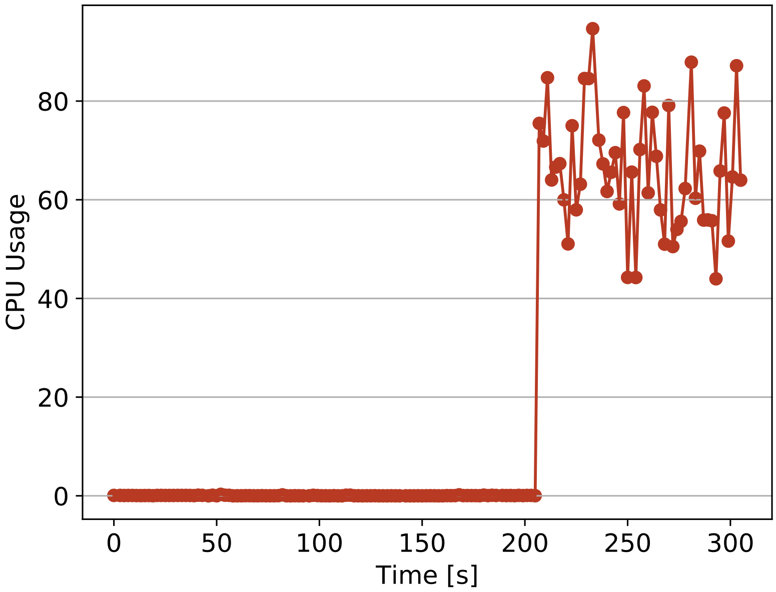


Figure 6.2: Docker container CPU usage over time.

6.2.2 Switching Alternative Paths

When Pathfinder detects a failing operator and an alternative path is available, it attempts switching the data flow from the failing path to the available path. This experiment evaluates a) how long it takes to switch to another path and b) whether a constant data flow can be accomplished throughout this process.

In this experiment, three operators named operator 0, 1 and 2 are set up as alternative paths in a split/join segment. In order to measure operator usage, each operator instance transmits its operator ID to a REST endpoint as soon as a data item has been processed successfully. A Python application is used to collect these requests and log the cumulative invocations over time. Based on those logs, a detailed time series can be created to deduce the operator usage.

The experiment is fully automated and repeated 16 times to exclude indeterministic effects due to the cloud usage of other applications.

6.2.3 Comparison to Active Replication

In the following theoretical experiment, we want to show how Pathfinder compares to setting up an active replication where each operator instance in a split/join segment is assigned a replica that takes over once a failure occurs.

We consider the topology in Figure 6.1 where two split/join segments are included. For each segment, we assume there is only one path (i.e., we ignore the functional redundancy in the form of alternative paths) in setup A. In setup B, we assume all alternative paths are active at all times.

Setup A

In setup A, each operator instance is deployed twice and both instances (*replicas*) process incoming data items in parallel and emit results. Directly after each pair of replicas, a *duplication detection* operator must be installed that decides which of the results is forwarded to the next operator according to the topology. If both replicas are working correctly, both of them will produce the same results and the duplicate detection operator only needs to make sure that no duplicate results are forwarded. If one of the replicas fails, the duplicate detection operator either must be able to discern this based on the emitted data items (e.g., if the failed operators sends a specific error code for each data item) or the failing operator ceases to produce any data items at all.

For the topology in Figure 6.1, this means

- two additional operator instances must be deployed as active replicas (P3 and P10),
- there must be two additional duplicate detection operators,

- operators P4, P5, P6, P11 and P12 are no longer used.

While this might look like superior from an arithmetic point of view (adding four operators while removing five), the reality is more complicated. First, Pathfinder would use the operator instances in the fallback paths (P4, P5, P6, P11 and P12) only in case there are failing operators in the main path. Most of the time, they are not deployed and therefore do not take up any resources.

Second, the functional redundancy in Pathfinder does not only protect from failures that can be corrected by spawning a new operator instance. For example, a bug in the operator itself that is caused by specific input data can persist even after a redeployment. Using Pathfinder, it is highly unlikely that all fallback paths show the same flaws.

However, active replicas have the advantage of a very fast adaptation rate, i.e., no time is wasted between the occurrence of a failure and further data item processing (provided the other replica is still available).

Setup B

In setup B, only one instance per operator is deployed. However, if there is functional redundancy, all alternative paths are deployed at all times. There is also a need for duplicate detection operators to make sure that different alternative paths do not produce duplicate results.

For the topology in Figure 6.1, this means

- there must be two additional duplicate detection operators,
- operators P4, P5, P6, P11 and P12 are deployed at all times.

Compared to setup A, this setup does not need the replicas of individual operators while having the same number of duplicate detection operators. However, it actually takes more operator instances in total since (1) there is more than one alternative path for P3 and (2) the length of the alternative paths exceeds the length of their respective main paths in both split/join segments.

Since the redundant instances have different implementations in this setup, it provides the same level of protection against faults as the one where Pathfinder is used. However, compared to Pathfinder, the resource usage of this setup is higher because the alternative paths are permanently in use (regardless of the presence of an operator failure).

Comparison

To conclude this theoretical experiment, we estimate the total cost for all three deployment scenarios. We assume that each operator is deployed on a public cloud using a virtual machine with 2 virtual CPUs and 8 GiB memory. Furthermore, we assume each instance cost amount to 0.093 EUR per hour and is billed every 60 minutes. This corresponds to a `t2.large` instance at Amazon AWS[3].

We compute the cost for four weeks of operation where in each week, both operators fail two times each (once for 10 minutes and once for 2.5 hours).

- **Setup A**. In this case, there are 14 processing operators minus the five ones that are only part of fallback paths, plus two extra instances being active replicas and two duplicate detection operators. This sums up to a total of 13 operators that must be deployed.

  $$13 \text{ operators} \cdot 0.093 \text{ EUR/hour} \cdot 24 \text{ hours/day} \cdot 28 \text{ days} = 812.45 \text{ EUR}$$

- **Setup B**. Since all operators are active at all times, there are 14 processing operators plus two duplicate detection operators. This sums up to a total of 15 operators that must be deployed.

  $$15 \text{ operators} \cdot 0.093 \text{ EUR/hour} \cdot 24 \text{ hours/day} \cdot 28 \text{ days} = 937.44 \text{ EUR}$$

- **Pathfinder**. Here, we have to consider all 14 operators. However, five of those are only part of fallback paths, so their deployment cost is only considered when failures occur.

  $$9 \text{ operators} \cdot 0.093 \text{ EUR/hour} \cdot 24 \text{ hours/day} \cdot 28 \text{ days} = 562.46 \text{ EUR}$$

  is the cost for deploying all the main paths. We then need to add that cost that is caused by deploying fallback path operators in case of failures. Since we know that there are four failures of 10 minutes (however, we have to bill a full hour) and four failures of 2.5 hours (billed for three hours) for each of the two split/join segments, we have to add

  $$\begin{aligned} &2 \text{ operators} \cdot 4 \text{ failures} \cdot 0.093 \text{ EUR/hour} \cdot 1 \text{ hour} + \\ &\quad 2 \text{ operators} \cdot 4 \text{ failures} \cdot 0.093 \text{ EUR/hour} \cdot 3 \text{ hour} = 2.98 \text{ EUR} \end{aligned}$$

  In total, the cost for the Pathfinder case accumulates to 565.44 EUR.

In summary, setup B has the highest cost of the investigated scenarios. The cost of setup A is about 15% lower since only 13 instead of 15 operators are deployed. This decrease in cost comes with the price of less redundancy since the alternative implementations are not used at all. Pathfinder comes with the lowest cost (about 40% less than setup B and 30% less than setup A) while achieving a similar level of redundancy as setup A. Figure 6.3 visually depicts its cost.

[3] `https://aws.amazon.com/ec2/instance-types/`

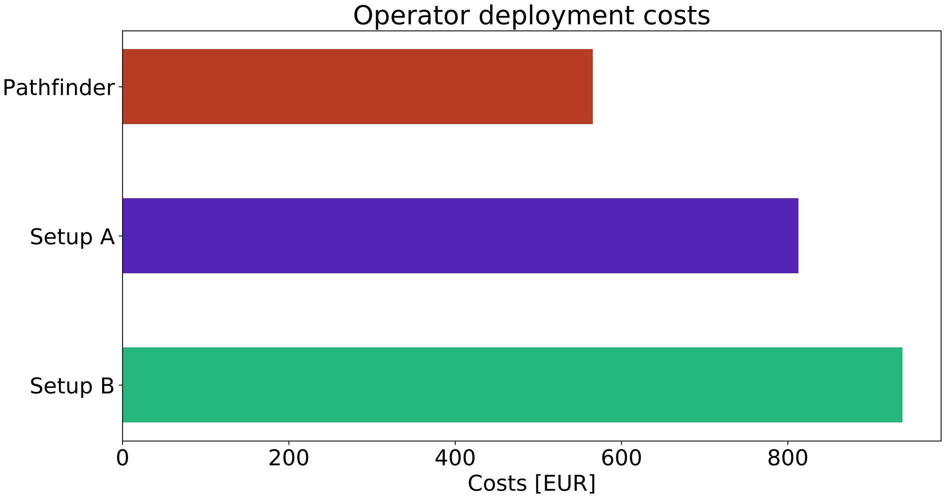


Figure 6.3: **Deployment cost for active replicas and Pathfinder**.

6.3 Motivational Scenario

Evaluating Pathfinder in the context of a real-world scenario is necessary to emphasise the benefits for the SPA users. This experiment investigates how much time it takes Pathfinder to activate a fallback path once an operator fails.

Figure 6.1 shows an SPA topology that implements the motivational scenario. It consists of three sources: S1 and S2 provide the positions of social network users and S3 creates data items whenever a user interacts with an ad. The processing operators P1 to P14 implement the functionality described in the introduction of the motivational scenario. The focus of the evaluation is on operators P1, P3, P4, P5, P6 and P7 due to the three alternative paths that lead from P1 to P7. Primarily, P3 is used to get a list of keywords for a certain GPS location by invoking the external *Atlas* API. In case of failure, P6 is used as a fallback. Finally, if both P3 and P6 are unavailable, a third alternative path consisting of P4 and P5 can be used. Finally, the keywords arrive at P7 and are used to find suitable ads.

6.3.1 Operator Failures

We simulate operator failures in three different ways: (1) exhaustive memory consumption, (2) total suspension of execution and (3) low processing throughput. Each failure is triggered manually or automatically based on a scripted scenario after a certain number of data items have been successfully processed.

Total Suspension of Execution (SLP)

There are multiple reasons why an operator can completely suspend its execution in real-world use cases. Examples include application code exceptions, hardware failures, host operating system failures and unavailability of external services. All those examples have in common that the operator is no longer emitting results. Pathfinder's Nexus component detects these faults by observing the rate of the item output and classifies an operator as failed if it is zero despite data items are waiting on the input queue(s).

This experiment simulates such cases by invoking the `Thread.sleep()` method to suspend further execution in the operator implementation after a certain number of data items has been produced.

Exhaustive Memory Consumption (MEM)

Under normal operating conditions, each operator uses a certain amount of system memory. The exact amount may fluctuate, but one would not expect sudden spikes. This observation is used to classify an operator as failed if its memory usage is above or below predefined thresholds. In this experiment, an operator is programmed to suddenly use more memory than usual.

Again, Pathfinder's Nexus component is programmed to detect this increase and classifies the operator as failed once the predefined threshold of 1024 MB is reached.

Low Processing Throughput (THR)

An operator does not necessarily have to fail completely to be classified as failed from the DSPE perspective. On the contrary, it is sufficient that the operator is not able to cope with the velocity of its input data streams. In this experiment, an operator is slowed down artificially.

Pathfinder's Nexus component 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. If no fallback were to be initiated, the operator's queue would eventually overflow.

6.3.2 Experimental Procedure

Each experiment starts by uploading the VTDL file to the VISP Runtime. As a result, the operators are deployed and the main path is activated once the setup is complete. Then, the VISP data provider is configured to continuously produce data items at the `gps_source` source. The split operator `split_transformed_data` describes three alternative paths `P3_Get_keywords_near_distance_to_location_external` (P3), `P4_Find_companies_near_distance_to_location` (P4) and `P6_Get_keywords_near_distance_to_location_by_user_tags` (P6). Initially, P3 is active

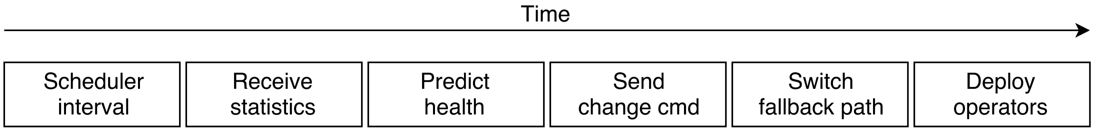


Figure 6.4: Time to response. The time it takes between an operator failing and the appropriate replacement being deployed consists of six phases.

and receives data items from P1. After about 300 seconds, an artificial fault is deliberately introduced into P3 based on the symptoms discussed in the previous section.

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 (P6). The time between the beginning of the failure and the activation of P6 is named *time to adapt*. It consists of six parts (as depicted in Figure 6.4): the time between two scheduling intervals, the time it takes to receive the statistics from VISP, the time it takes for the Nexus component to make a prediction for an operator, the time it takes until Pathfinder can communicate its decision to VISP, the time it takes VISP to switch paths and the time it takes until a newly activated operator is deployed (only when lazy deployment is activated).

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

6.4 Evaluation Results

6.4.1 Fault Detection

The preliminary experiment regarding fault detection (Section 6.2.1) aims to show that statistics from Docker containers can be used to detect operator failures. As shown by the data in Figure 6.2, there is a clear distinction between phases of activity where the CPU usage fluctuates between 40% and 90%, and phases of inactivity with a CPU usage near zero. Docker limits queries for container statistics using the `stats` command to once per second. This is not an issue for the use cases addressed by Pathfinder where unavailability even for a few minutes is acceptable.

Fault detection using Docker statistics is of course limited to those cases where the operator's failure results in abnormal CPU or memory usage.

Example 6.1

Examples for operator failures that affect CPU and memory usage include: (1) a crashing JVM (e.g., due to an exception), (2) hardware failures, (3) operating system crashes, and (4) memory leaks.

For those cases, this experiment shows that the general detection approach is working.

6.4.2 Path Switching

Section 6.2.2 introduced an experiment to measure how long it takes to switch between alternative paths and whether the downstream data flow is interrupted during this process.

Figure 6.5 depicts the experimental results. The plot shows the cumulative number of operator invocations over time for operators A, B and C in blue, orange and green, respectively. Red lines indicate the times when the message flow has been changed (at 120, 240, 360, 480 and 600 seconds). The plot shows the median over 16 individual measurements and their standard deviation in lighter colours by averaging the cumulative number of invocations for each second. This high number of experiments was conducted to exclude potential indeterministic effects due to the cloud usage of other applications.

Initially, only operator A is invoked since it is the main path. At $t = 120$ seconds, the active path is switched to operator B (as shown by the red vertical line). Almost instantly, no more invocations occur for operator A but for operator B instead. After another 120 seconds, the data flow is redirected to operator C, which shows the same instant rise in activity as operator B before. The whole procedure is then repeated for a second time with another three 120 second cycles.

As shown by the standard deviation, there is a certain degree of random fluctuation in the data flow throughput. First, this can be attributed to the usage of cloud resources. Since other applications are also running on the same infrastructure, their resource usage (e.g., network and hard disk usage) can influence the actual processing capabilities. Second, the measurement of the invocations itself can affect the overall performance since for each processed data item, an HTTP request is sent to a central monitoring component. This puts additional load on the network infrastructure and might also be the reason for the high fluctuations.

Nevertheless, the experimental results clearly show that the path switching mechanism itself is operating quickly and is therefore suited for the use in Pathfinder.

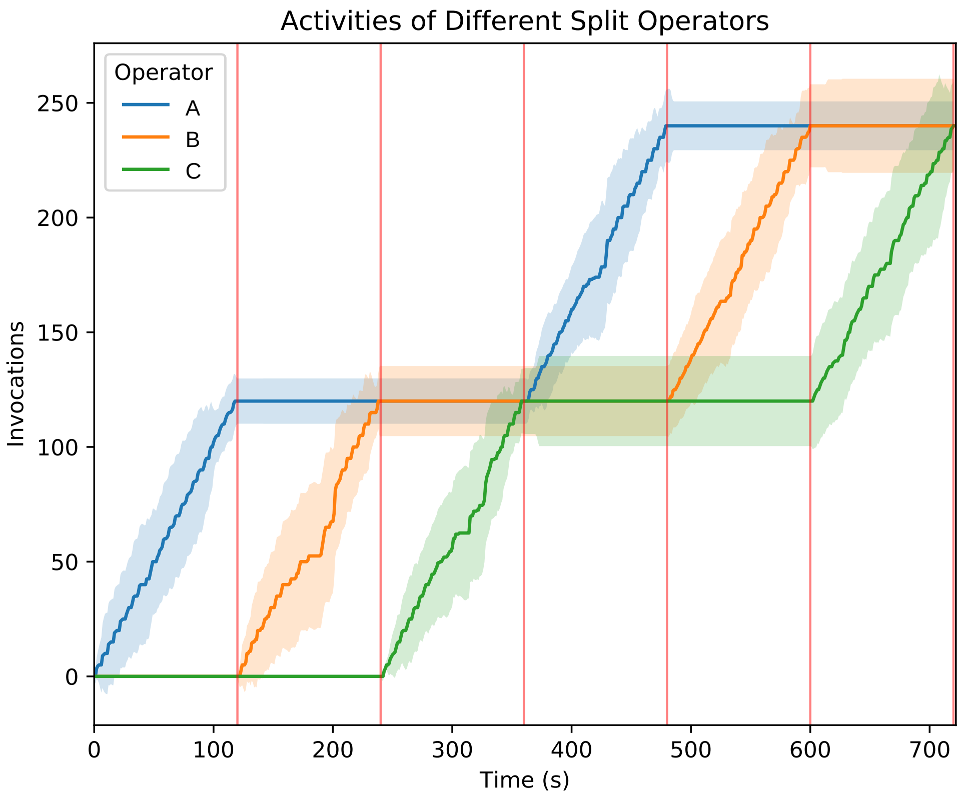


Figure 6.5: Activating different fallback paths ($n = 16$).

6.4.3 Motivational Scenario

In Section 6.3, Pathfinder was applied for a topology derived from the motivational scenario. The time to adapt was measured for three different failure types in order to evaluate how well Pathfinder performs with respect to availability and resource usage.

Discussion of Results

Figure 6.6 shows a plot that visualises the temporal connection between the different experimental phases. Different phases are depicted by horizontal bars in different colours. The width of each bar corresponds to its duration and the position along the x-coordinate shows its beginning and end.

Starting at time $t = 0$, the initial topology setup takes about 57 seconds. This phase includes (1) uploading the VTDL file to VISP (and all subsequent initialisation steps performed by VISP), (2) adding the IP of the VISP Runtime instance to Pathfinder and (3) starting the VISP DataProvider.

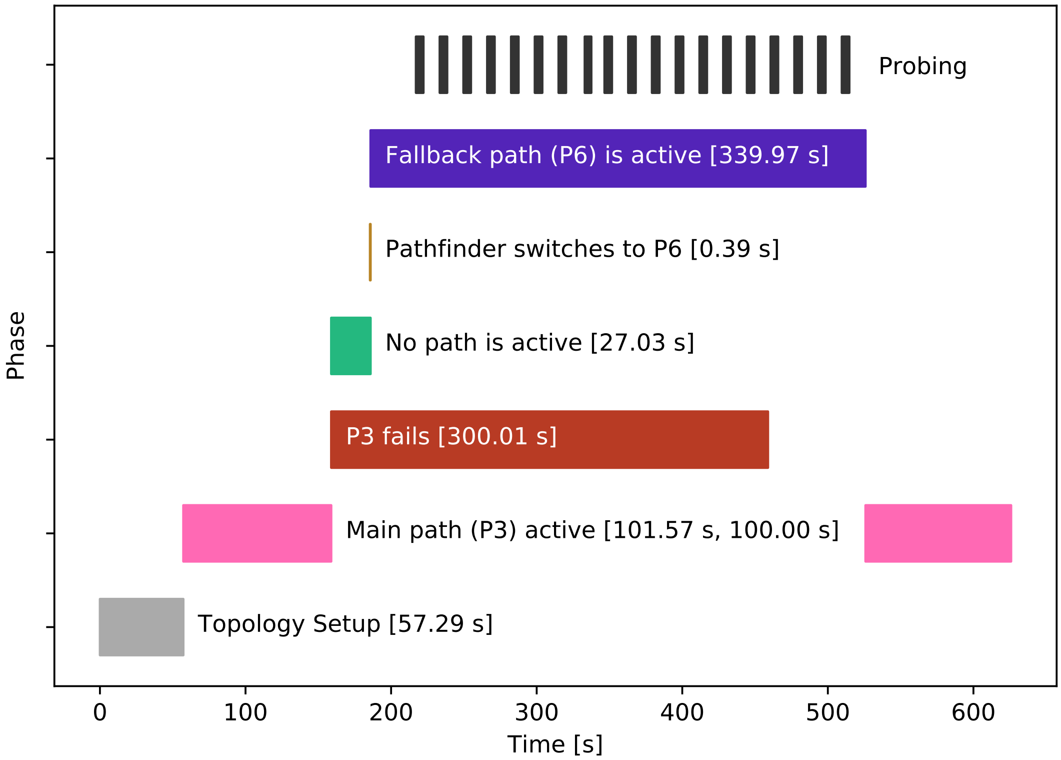


Figure 6.6: Plot showing Pathfinder's actions in case of failure.

The second phase starts when the first data item from the VISP DataProvider was successfully processed by the `P1` operator (approximately at $t = 58$). In this phase, the main path (P3) is active and is successfully processing incoming data items. While it is not explicitly shown in the plot, Pathfinder continuously queries the VISP Runtime instance 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, this plot shows the data for the total suspension of execution (SLP), but it looks similar for all failure types. The total length of the failure is 300 seconds. During this time, the main path 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 commands VISP to switch to P6. As discussed in the previous section, switching paths is very time-efficient and after 0.39 seconds, P6 is active.

While 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. In this implementation, the time between two probing attempts is always the same. In a production environment, one can increase this time after each failed attempt to generate less resource usage.

Once P3's failure stops after 300 seconds, Pathfinder switches back to P3 after a few probing attempts and the period of P6 activity ends after a total of 340 seconds. Without Pathfinder, this period would have equalled to unavailability of the whole SPA.

In summary, the experiment has successfully shown that Pathfinder is able to maximise the SPA's availability in the presence of operator failures by utilising functional redundancy at the path level.

Shortcomings

There are a few more things to discuss at this point. First, the time it takes to detect the failure is quite high (27 seconds). This is the price to pay for the very generalised approach where absolutely no implementation details of the operators are shared with Pathfinder and the detection relies solely on statistics provided by Docker and the performance indicators of the messaging infrastructure. To reduce this time, one can query statistics from VISP more frequently at the price of a higher performance overhead.

Second, the time it takes to detect the recovery. As can be seen in the plot, there are three probing events between the end of P3's failure and the activation of the main path. One would assume that a single probing event should suffice, but it is more complicated. 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 produced data items, it can happen that the pause is too short and no data items are produced during this time span. This can again be fine-tuned by elongating the pause at the cost of increased resource usage. Also, the post-probing operator classification uses the same mechanism as the normal operator classification and therefore also relies on VISP's statistics. If the probing window opens immediately before the last statistics has been fetched, it takes another 15 seconds until the probing results show up in the next request.

Different Failure Types

Figure 6.7 shows the time to adapt for the three different operator failure types discussed in Section 6.3.1. The y-axis shows the average time to adapt as measured in four experiments (as those shown in Figure 6.6) for each failure type. The total suspension of execution (SLP) is detected on average after 30 seconds. It is the quickest to be detected since its effects are reflected immediately by the lack of new data items. The first rule of the `RuleBasedNexus` implementation (see Section 5.3.2) interprets both a low CPU utilisation and a high number of enqueued data items as a sign for a failure and therefore classifies this operator as failed.

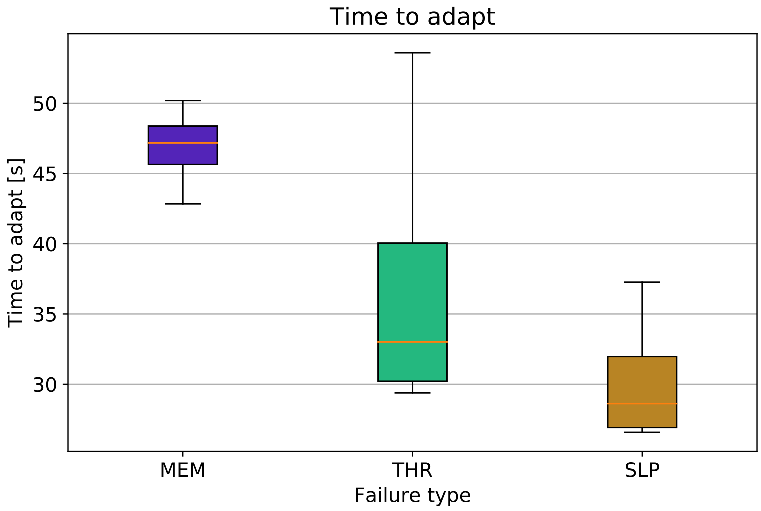


Figure 6.7: Time to adapt.

The exhaustive memory consumption (MEM) takes between 45 and 50 seconds to be detected. Once the failure starts, more and more memory is used, but it takes some time until the threshold in rule 3 is reached and the operator is classified as failed.

The low processing throughput (THR) failure's detection time varies much more than that of the others. This is due to its dynamic nature. Since the failure only causes a processing delay, the failure classification (rule 4) is only triggered when the incoming data item rate is large enough. As discussed in Section 6.4.2, this depends on multiple factors including the current utilisation of the cloud infrastructure.

6.5 Limitations of Applicability

Based on the discussion of the evaluation results, this section focuses on the limitations of Pathfinder's applicability.

First and most important, there is the time to react to failures. The experiments in the previous section have already shown how much time it takes to detect failures and recoveries and switch between different paths. Another factor that was not even considered here is the time it takes to deploy new operator instances if lazy deployment is used (which however heavily depends on the underlying DSPE and is therefore not evaluated in this thesis).

While this implementation is production ready, one can think of improvements to reduce the time to adapt. However, it cannot be expected to reduce it to the millisecond range, for example. Use cases where such a level of availability is needed (e.g., in critical software of self-driving vehicles) can therefore not be covered by Pathfinder.

Second, there is a limitation regarding the use of external services. Since Pathfinder uses probing to temporarily activate the data flow to another path, there can be the situation where two or more alternative paths are active simultaneously. This can be problematic due to side-effects (i.e., outside state modifications) if the SPA developer does not anticipate this behaviour and there is communication to external services in those paths.

Example 6.2

An operator of type *counter* counts how many of its input data items match a certain criterion. Once its internal counter reaches 10, it sends a notification to a customer. We consider a split/join segment consisting of two alternative paths, both of which contain a (unique) instance of type *counter*. During a probing event, both of those instances count *the same* data items which will ultimately result in inaccurate notifications because items are counted twice.

Another limitation is Pathfinder's own resource usage. It needs to be deployed as a standalone service and requires a certain amount of resources. In cases where such resources are scarce (e.g., in IoT environments), the usage of Pathfinder might not be feasible.

6.6 Requirements

In this section we finally discuss whether Pathfinder's initially defined requirements (see Section 4.2) are fulfilled.

FR1 **Scalability**. Pathfinder is based on VISP which supports scalability by relying on cloud resources for the deployment of operators and contains automatised mechanisms to utilise them when faced with varying system load. Pathfinder itself is also scalable by allowing a distributed deployment across multiple locations as discussed in Section 4.4.3. This requirement is therefore fulfilled.

FR2 **Exploit functional redundancy**. Functional redundancy can be defined by split/join segments in the form of alternative paths with a similar functionality. Pathfinder then exploits this redundancy in case of operator failures by switching the data flow to an alternative path that does not contain any failures. Pathfinder furthermore uses the circuit breaker pattern to store a path's state and to initiate recovery detection attempts (probing) in a controlled way. This requirement is fulfilled, which has also been demonstrated by the experiments in Section 6.3.

FR3 **Minimise downtimes**. Pathfinder's fault tolerance model minimises downtimes by switching to alternative paths. The conducted experiments have shown that the time to adapt for different failures is in the range of 30 to 50 seconds (Section 6.4.3). We therefore consider this requirement as fulfilled as well and refer to Section 6.5 for the limits of applicability.

FR4 **Reliability of results**. Pathfinder only produces results via the alternative paths specified by the user. If all alternative paths are unavailable due to operator failures, no more results are produced. Pathfinder instead continues its recovery detection attempts until one of the paths becomes available again. This final requirement is therefore fulfilled as well.

In summary, all the framework requirements have been fulfilled by Pathfinder. Furthermore, Pathfinder can be used to solve the challenges posed by the motivational scenario as shown in the experiments.

CHAPTER 7

Conclusion

This last chapter concludes the contributions of this work in Section 7.1, puts the initially defined RQs into context in Section 7.2 and provides an outlook on future work in Section 7.3.

7.1 General Conclusion

Fault tolerance is a key aspect for DSPEs due to their near-real-time requirements, their long-running nature and the resulting high probability of faults to occur. Based on the shortcomings of established solutions and on the requirements raised by a motivational scenario, a new fault tolerance framework for DSPEs named Pathfinder has been designed that uses a novel approach based on functional redundancy. By allowing users to define concrete fallback functionalities for faults at design-time, Pathfinder can react to those faults at runtime and thereby maximise the availability of the SPA.

A prototypical implementation of Pathfinder was then developed and evaluated using the VISP DSPE. Several experiments were conducted to show that Pathfinder's failure detection and fault tolerance mechanisms are working and only add a negligible performance overhead.

7.2 Research Questions

Section 1.3 introduced three RQs that remain to be answered:

RQ1: Which fault tolerance mechanisms for stream processing systems are known and how can they be applied to the motivational scenario?
During the literature review, several fault tolerance mechanisms for DSPEs have

been identified. As summarised in Chapter 3, none of them can be used for the motivational scenario due to a lack of support for functional redundancy at the level of paths. Also, none of the approaches from the literature are able to detect failing operators based solely on information from the deployment and messaging infrastructure (e.g., Docker and RabbitMQ in the case of VISP).

RQ2: How can data streams be processed in a fault tolerant manner while still guaranteeing a high availability?
Since RQ1 showed that none of the existing approaches fulfils the requirements of the motivational scenario, a new fault tolerance framework has been designed. It addresses the exact shortcomings identified in RQ1 and solves them by allowing SPA developers to define functional redundancy at the level of paths to achieve a high availability at runtime by dynamically switching between functionally redundant paths when faults are detected.

RQ3: How does the framework perform in terms of applicability and performance?
Pathfinder's performance has been evaluated both in isolation (focusing on its failure detection and path switching capabilities) and by integration into VISP. All experiments show satisfying results both in terms of performance and applicability to the motivational scenario with respect to the initially derived requirements.

7.3 Future Work

Although the implementation of Pathfinder is already fully functional, there are several future research possibilities.

1. **Failure detection**. Detecting whether an operator is still functioning correctly challenging, particularly since it is (by reduction from the Halting problem) provably undecidable in general [71]. Since operators can be implemented using any programming language, platform and framework, defining universal thresholds for metrics like CPU and memory usage is next to impossible. A better idea would be to use data collected during normal (failure-free) operation and compare that to the current operator metrics using statistical and machine learning methods. Such methods may also be enhanced by language-level annotations about the expected behaviour (e.g., RAM usage) provided by the SPA developer [70].

2. **Compatibility to other DSPEs**. As described in Section 2.2.2, the DSPE market provides many different frameworks from several vendors. While compatibility with VISP was the primary goal of this work, Pathfinder's architecture is not restricted to VISP. Instead, compatibility with other DSPEs should be prioritised in future work to reach a larger target audience.

3. **Lazy deployment**. The prototypical Pathfinder implementation allows the SPA developer to choose between lazy and eager deployment of fallback path operators.

Neither solution is optimal since one either has to waste resources on deploying operators that are not actually needed or waste time waiting for the spawning of needed operators. Future work may focus on this dilemma by investigating other approaches in addition to the lazy deployment. One idea is to predict faults using statistical and machine learning methods and proactively deploy fallback operators when faults seem likely.

4. **Human feedback**. Failure detection can rely on human users for parts of the DSPE where user interaction is required (e.g., considering a whole path as failed if at least five end-users report that some functionality is not working correctly). Such reporting could also happen implicitly (e.g., by observing that many users cancel their activities once they reach a certain task like submitting a form that seems to be failing).

APPENDIX A

Evaluation Topology

Listing A.1: Evaluation topology. This topology in the VTDL format is used to evaluate Pathfinder's performance. It is based on the motivational scenario introduced in Section 1.2.2.

```
$gps_source = Source() {
  concreteLocation = 128.130.172.220/openstack,
  type             = gps_source,
  outputFormat     = "Position data",
}

$ips_source = Source() {
  concreteLocation = 128.130.172.220/openstack,
  type             = ips_source,
  outputFormat     = "Position data",
}

$transform_position_data = Operator($gps_source, $ips_source) {
  allowedLocations = 128.130.172.220/openstack,
  concreteLocation = 128.130.172.220/openstack,
  inputFormat      = "Position data",
  type             = transform_position_data,
  outputFormat     = "Standardized position data",
  size             = small,
  stateful         = false
}

$suggest_nearby_friends = Operator($transform_position_data) {
  allowedLocations = 128.130.172.220/openstack,
  concreteLocation = 128.130.172.220/openstack,
  inputFormat      = "Standardized position data",
  type             = suggest_nearby_friends,
  outputFormat     = "user",
  size             = small,
  stateful         = false
}

$send_friend_suggestions = Sink($suggest_nearby_friends) {
  concreteLocation = 128.130.172.220/openstack,
  inputFormat      = "user",
  type             = "send_friend_suggestions",
}

$split_transformed_data = Split($transform_position_data) {
  pathOrder = $position_to_keywords_external $position_to_keywords_user_tags
}

$position_to_keywords_external = Operator($split_transformed_data) {
  allowedLocations = 128.130.172.220/openstack,
  inputFormat      = "Standardized position data",
  type             = "position_to_keywords_external",
  outputFormat     = "keyword",
  size             = small,
  stateful         = false,
}

$position_to_keywords_user_tags = Operator($split_transformed_data) {
  allowedLocations = 128.130.172.220/openstack,
  inputFormat      = "Standardized position data",
  type             = "position_to_keywords_user_tags",
  outputFormat     = "keyword",
  size             = small,
  stateful         = false,
}

$join_transformed_data = Join($position_to_keywords_external,
  ↪  $position_to_keywords_user_tags) {}

$find_ads_by_keyword = Operator($join_transformed_data) {
  allowedLocations = 128.130.172.220/openstack,
  concreteLocation = 128.130.172.220/openstack,
  inputFormat      = keyword,
  type             = "find_ads_by_keyword",
  outputFormat     = "ad",
  size             = small,
  stateful         = false
}

$send_ads = Sink($find_ads_by_keyword) {
  concreteLocation = 128.130.172.220/openstack,
  inputFormat      = "ad",
  type             = "send_ads",
}
```

Bibliography

[1] A. Whitmore, A. Agarwal, and L. Da Xu, "The Internet of Things—A survey of topics and trends," *Information Systems Frontiers*, vol. 17, no. 2, pp. 261–274, 4 2015.

[2] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," *McKinsey Global Institute*, 2011. [Online]. Available: http://www.mckinsey.com/insights/mgi/research/

[3] K. J. Kim and D.-H. Shin, "An acceptance model for smart watches," *Internet Research*, vol. 25, no. 4, pp. 527–541, 8 2015.

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

[5] C. Hochreiner, M. Vögler, P. Waibel, and S. Dustdar, "VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things," in *20th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 2016, pp. 1–11.

[6] 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 International Conference on Management of Data*, 2014, pp. 147–156.

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

[8] M. Anis and U. Nasir, "Fault Tolerance for Stream Processing Engines," *arXiv preprint arXiv:1605.00928*, 2016.

[9] L. Su and Y. Zhou, "Tolerating correlated failures in Massively Parallel Stream Processing Engines," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 517–528.

[10] B. Balasubramanian and V. K. Garg, "Fault tolerance in distributed systems using fused state machines," *Distributed Computing*, vol. 27, no. 4, pp. 287–311, 8 2014.

[11] Q. Huang and P. P. C. Lee, “Toward High-Performance Distributed Stream Processing via Approximate Fault Tolerance,” *VLDB Endowment*, vol. 10, no. 3, pp. 73–84, 11 2016.

[12] F. C. Gärtner, “Fundamentals of fault-tolerant distributed computing in asynchronous environments,” *ACM Computing Surveys*, vol. 31, no. 1, pp. 1–26, 1999.

[13] C. Exton, “Distributed fault tolerance specification through the use of interface definitions,” in *Technology of Object-Oriented Languages. TOOLS 24.* IEEE, 1997, pp. 254–259.

[14] C.-L. Fok, C. Julien, G.-C. Roman, and C. Lu, “Challenges of Satisfying Multiple Stakeholders: Quality of Service in the Internet of Things,” in *2nd Workshop on Software Engineering for Sensor Network Applications*. ACM, 2011, pp. 55–60.

[15] K. Wang, Y. Shao, L. Shu, C. Zhu, and Y. Zhang, “Mobile big data fault-tolerant processing for ehealth networks,” *IEEE Network*, vol. 30, no. 1, pp. 36–42, 2016.

[16] J. Lee, H. A. Kao, and S. Yang, “Service innovation and smart analytics for Industry 4.0 and big data environment,” in *Procedia CIRP*, vol. 16, 2014, pp. 3–8.

[17] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management,” *The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003.

[18] W. Wang, X. Guan, and X. Zhang, “Processing of massive audit data streams for real-time anomaly intrusion detection,” *Computer Communications*, vol. 31, no. 1, pp. 58–72, 2008.

[19] J. H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, “High-availability algorithms for distributed stream processing,” in *21st International Conference on Data Engineering*, IEEE, Ed. IEEE, 2005, pp. 779–790.

[20] M. Couceiro, D. Suarez, and D. Manzano, “Data stream processing on real-time mobile advertisement: Ericsson research approach,” in *2011 IEEE 12th International Conference on Mobile Data Management*. IEEE, 2011, pp. 313–320.

[21] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas, “Adnostic : Privacy Preserving Targeted Advertising,” in *2010 Network and Distributed System Security Symposium*. Internet Society, 2010, pp. 1–21.

[22] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A survey,” *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[23] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, “The rise of "big data" on cloud computing: Review and open research issues,” *Information Systems*, vol. 47, pp. 98–115, 1 2015.

[24] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future*, vol. 2007, no. 2012, pp. 1–16, 2012.

[25] V. Marx, "Biology: The big challenges of big data," *Nature*, vol. 498, no. 7453, pp. 255–260, 2013.

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters." in *4th USENIX Conference on Hot Topics in Cloud Ccomputing*. Boston, MA: USENIX Association, 2012, pp. 10–10.

[27] A. Y. Zomaya and S. Sakr, *Handbook of big data technologies*. Springer, 2017.

[28] A. Shukla and Y. Simmhan, "Benchmarking Distributed Stream Processing Platforms for IoT Applications," in *Technology Conference on Performance Evaluation & Benchmarking*. Springer, 2016.

[29] G. J. Chen, S. Yilmaz, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, and T. Williamson, "Realtime Data Processing at Facebook," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1087–1098.

[30] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *Innovative Data Systems Research Conference*. VLDB, 2003, pp. 277–289.

[31] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-Tolerance in the Borealis Distributed Stream Processing System," *ACM Transactions on Database Systems*, vol. 33, no. 1, pp. 1–44, 3 2008.

[32] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data & Knowledge Engineering*, vol. 14, no. 1, pp. 1–26, 1994.

[33] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, and others, "The Hipac project: Combining active databases and timing constraints," *ACM Sigmod Record*, vol. 17, no. 1, pp. 51–70, 1988.

[34] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A scalable continuous query system for internet databases," in *ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 379–390.

[35] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," in *1992 ACM SIGMOD international conference on Management of data*. ACM, 1992, pp. 321–330.

[36] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[37] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The Design of the Borealis Stream Processing Engine." in *Second Biennial Conference on Innovative Data Systems Research*, 2005, pp. 277–289.

[38] Apache Software Foundation, "Apache Spark - Lightning-Fast Cluster Computing," 2016. [Online]. Available: https://spark.apache.org/

[39] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things," *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pp. 100–107, 2016.

[40] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the Cloud," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 3, no. 5, pp. 333–345, 2013.

[41] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an Open-Source Solution for Cloud Computing," *International Journal of Computer Applications*, vol. 55, no. 03, pp. 38–42, 2012.

[42] Amazon Web Services LLC, "Amazon Elastic Compute Cloud (EC2)." [Online]. Available: http://aws.amazon.com/ec2/

[43] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, "MigCEP: operator migration for mobility driven distributed complex event processing," in *7th ACM international conference on Distributed event-based systems*. ACM Press, 2013, p. 183.

[44] M. Weske, *Business Process Management: Concepts, Languages, Architectures*, 2nd ed. Springer Publishing Company, 2012.

[45] C. Hochreiner, M. Nardelli, B. Knasmueller, S. Schulte, and S. Dustdar, "VTDL: A Notation for Stream Processing Applications (accepted for publication)," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. NN, 2018.

[46] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[47] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, M. Mendell, H. Nasgaard, K.-l. Wu, T. J. Watson, and Y. Heights, "SPL Stream Processing Language Specification," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24897, 2009.

[48] T. Erl, *Service oriented architecture: principles of service design*. Prentice Hall, 2008.

[49] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, and S. Thatte, "Web services transaction (WS-transaction)," *Specification, BEA, IBM, Microsoft and TIBCO*, 2002.

[50] J. M. Ko, C. O. Kim, and I.-H. Kwon, "Quality-of-service oriented web service composition algorithm and planning architecture," *Journal of Systems and Software*, vol. 81, no. 11, pp. 2079–2090, 2008.

[51] P. Leitner, W. Hummer, and S. Dustdar, "Cost-based optimization of service compositions," *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 239–251, 2013.

[52] G. Mazlami, J. Cito, and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 524–531.

[53] E. Wolff, *Microservices*. Heidelberg: dpunkt.verlag GmbH, 2016.

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

[55] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[56] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006.

[57] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., 2006.

[58] J. Postel, "Transmission control protocol," 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[59] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[60] 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," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[61] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[62] B. Knasmueller, "Fault Tolerance for Distributed Stream Processing Engines," *Unpublished Seminar Work*, 2017.

[63] A. Liu, Q. Li, L. Huang, and M. Xiao, "FACTS: A Framework for Fault-Tolerant Composition of Transactional Web Services," *IEEE Transactions on Services Computing*, vol. 3, no. 1, pp. 46–59, 1 2010.

[64] R. Bilorusets, D. Box, L. F. Cabrera, D. Davis, D. Ferguson, C. Ferris, T. Freund, M. A. Hondo, J. Ibbotson, L. Jin, and others, "Web services reliable messaging protocol (WS-ReliableMessaging)," *Specification, BEA, IBM, Microsoft and TIBCO*, 2005.

[65] J. Salas, F. Perez-Sorrosal, M. Patino-Martinez, and R. Jimenez-Peris, "WS-replication: a framework for highly available web services," in *15th international conference on World Wide Web*. ACM, 2006, pp. 357–366.

[66] A. Erradi, P. Maheshwari, and V. Tosic, "Recovery policies for enhancing Web services reliability," in *2006 IEEE International Conference on Web Services*. IEEE, 2006, pp. 189–196.

[67] B. Schmaus, "The Netflix Tech Blog: Making the Netflix API More Resilient," 2011. [Online]. Available: http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html

[68] L. Yang, Z. Cui, and X. Li, "A Case Study for Fault Tolerance Oriented Programming in Multi-core Architecture," in *11th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2009, pp. 630–635.

[69] J.-C. Laprie, "From dependability to resilience," in *International Conference on Dependable Systems and Networks*, 2008, p. G8–G9.

[70] G. Jacques-Silva, B. Gedik, H. Andrade, and K.-L. Wu, "Language level checkpointing support for stream processing applications," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. IEEE, 2009, pp. 145–154.

[71] M. Davis, *Computability & Unsolvability*. Dover, 1958.