UiO : University of Oslo

Espen Volnes

Distributed Stream Processing: Performance Evaluation and Enhanced Operator Migration

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics

Faculty of Mathematics and Natural Sciences

2024

© Espen Volnes, 2024

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2717

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: UiO.
Print production: Graphic center, University of Oslo.

Abstract

With the advent of the Internet of Things (IoT), billions of devices will be interconnected, leading to a dramatic increase in data traffic on the Internet. These devices, ranging from high-capacity servers to highly constrained sensors, create a diverse ecosystem with varying processing capacities and requirements. Addressing this surge, distributed stream processing systems (DSPS) aim to process data where it originates, instead of storing all data in centralized databases. DSPSs filter, aggregate, join, and transform incoming data. However, the distributed nature of these systems complicates the evaluation of such systems. Another challenge lies in the management of state during operator migration, as not all state elements are equally important. The field of distributed stream processing also lacks a common language and platform, making it difficult to compare or build on existing systems.

The first key contribution of this dissertation is developing software tools to enhance the evaluation of DSPSs, introducing simulation tools for easier prototyping and benchmarking of existing systems. We have developed the experimental framework Expose to be tailored for real-world DSPS evaluation. This is implemented with real-world DSPSs, as well as being integrated with the DSPS simulator DCEP-Sim to further streamline simulation definition and execution. Significant enhancements to the DCEP-Sim simulator are introduced, encompassing simulation models for query processing operator functionality, processing delays, and operator migration mechanisms. The query processing operators include join, aggregation, filter, group by, and select. Given the requirements for simulation of delay in discrete event simulators, introducing simulation delay for each processed tuple is crucial to mirror real-world systems and avoid potential network saturation.

The second key contribution of this Thesis addresses the critical DSPS functionality of operator migration. Operator migration is explored in a comprehensive tutorial, which offers a consolidated view of the current state-of-the-art and setting the groundwork for novel contributions. Two novel migration mechanisms are introduced that address issues that are relevant for geo-distributed DSPSs. The Travel Light migration mechanism combines operator migration with load shedding to prioritize the most important states when the migration might not complete entirely, or not all the state is important. The migration mechanism Lazy Migration offers a latency mode that minimizes the downtime for operators during migration and a utility mode that maximizes the utility of the migrated state, in cases where the migration might not fully finish.

The performance evaluation tools developed in this dissertation are successfully employed to analyze and compare the efficiency of Lazy Migration against three state-of-the-art solutions: Megaphone, Rhino, and Meces, highlighting the effectiveness of Lazy Migration in the case of the aggregation and join operators. This evaluation indirectly underscores the robustness and utility of both Expose and DCEP-Sim.

Abstract

Med Internet of Things (IoT) vil milliarder av enheter kobles sammen, noe som fører til en dramatisk økning i datatrafikken på Internett. Disse enhetene, alt fra servere med høy kapasitet til svært begrensede sensorer, skaper et mangfoldig økosystem med varierende behandlingskapasitet og krav. For å adressere denne økningen, har «Distributed Stream Processing Systems (DSPS)» som mål å behandle data der de kommer fra, i stedet for å lagre alle data i sentraliserte databaser. En DSPS utfører operasjoner som «filter», «join», «group by» og «aggregation». Den distribuerte naturen til disse systemene kompliserer imidlertid evalueringen av slike systemer. En annen utfordring ligger i styringen av tilstand under operatormigrasjon, da ikke alle elementer som tilstanden består av er nødvendigvis like viktige. DSPS'er mangler også et felles språk og plattform, noe som gjør det vanskelig å sammenligne eller bygge på eksisterende systemer.

Det første nøkkelbidraget til denne avhandlingen er å utvikle programvareverktøy for å forbedre evalueringen av DSPS'er, og introdusere simuleringsverktøy for enklere prototyping og benchmarking av eksisterende systemer. Vi har utviklet det eksperimentelle rammeverket Expose for å være skreddersydd for DSPS-evaluering i den virkelige verden. Dette er implementert med DSPS'er i den virkelige verden, i tillegg til å være integrert med DSPS-simulatoren DCEP-Sim for å strømlinjeforme simuleringsdefinisjon og utførelse ytterligere. Betydelige forbedringer av DCEP-Sim-simulatoren er introdusert, som omfatter simuleringsmodeller for å behandle spørringer, introdusere behandlingsforsinkelser og støtte flere typer operatormigrasjonsmekanismer. Spørringsbehandlingsfunksjonaliteten inkluderer «join», «aggregation», «filter», «group by» og «select» operatorene. Gitt kravene til simulering av forsinkelse i diskrete hendelsesimulatorer, er det avgjørende å introdusere simuleringsforsinkelse for hver behandlet tuppel for å speile virkelige systemer og unngå potensiell nettverksmetning.

Det andre nøkkelbidraget til denne avhandlingen tar for seg den kritiske DSPS funksjonaliteten operator-migrasjon. Operator-migrasjon utforskes i en omfattende veiledningsartikkel, som gir et konsolidert syn på den nåværende «state-of-the-art» og legger grunnlaget for nye bidrag. To nye migrasjonsmekanismer er introdusert som tar for seg problemer som er relevante for geo-distribuerte DSPS'er. Travel Light-migrasjonsmekanismen kombinerer operator-migrasjon med «load shedding» for å prioritere de viktigste tilstandene til operatorene når migrasjonen kanskje ikke kan fullføres helt, eller ikke hele tilstanden er viktig. Migrasjonsmekanismen Lazy Migration tilbyr en latensmodus som minimerer nedetiden for operatorer under migrasjon og en utilitarisk modus som maksimerer nytten av den migrerte tilstanden, i tilfeller der migrasjonen kanskje ikke kan fullføres.

Ytelsesevalueringsverktøyene som er utviklet i denne avhandlingen er vellykket brukt for å analysere og sammenligne effektiviteten til

Lazy Migration med tre toppmoderne løsninger: Megaphone, Rhino og Meces, og fremhever effektiviteten til Lazy Migration når det gjelder aggregation og join operatorer. Denne evalueringen understreker indirekte robustheten og nytten til både Expose og DCEP-Sim.

Acknowledgements

This thesis is submitted in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here was conducted at the University of Oslo, under the supervision of Professor Thomas Plagemann and researcher Dr. Stein Kristiansen.

The thesis comprises six papers, arranged in chronological order based on when they were initiated. The papers are preceded by an introductory chapter that relates them to each other and provides background information and motivation for the work. Each of the papers is a joint work with multiple co-authors, including Thomas Plagemann, Vera Goebel, Stein Kristiansen, Boris Koldehofe, Morten Lindeberg, and Øystein Dale.

First and foremost, I'd like to express my profound gratitude to my main supervisor, Thomas Plagemann. Most of the papers were only possible due to the stimulating discussions we had together. Your guidance and insights have been invaluable. I also want to thank my co-supervisor Stein Kristiansen. Your assistance in the first half of my Ph.D. work laid the foundation for my subsequent endeavors, and I'm grateful for the expertise you provided before departing from the university. Vera Goebel deserves my thanks for always helping to make concepts easier to understand, in both the papers and the writing of this Thesis. I also want to thank Boris Koldehofe for his help with our operator migration research papers. Your expertise and perspective have enriched our joint endeavors.

Our research group has seen several changes throughout my Ph.D. work, and I am grateful for the lunches we have spent together. Thank you Morten, Konstantinos, Maik, Marta, He, Farzan and Tallal. I want to extend a special thank you to Maik for being helpful almost to a fault, always ready to talk about an issue or spend time to solve a problem.

Lastly, I want to thank my family for their consistent support. And to my partner Ingrid, thank you for your encouragement and belief in my endeavors.

Espen Volnes
Oslo, February 2024

List of Papers

Paper I

Espen Volnes, Stein Kristiansen, and Thomas Plagemann. 2021. Improving the accuracy of timing in scalable WSN simulations with communication software execution models. Computer Networks, volume 188, 2021, 107855. doi:10.1016/j.comnet.2021.107855

Paper II

Espen Volnes, Stein Kristiansen, Thomas Plagemann, Vera Goebel, and Morten Lindeberg. 2019. Modeling the Software Execution of CEP in DCEP-Sim. In Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (DEBS ’19). Association for Computing Machinery, New York, NY, USA, 244–247. doi:10.1145/3328905.3332508

Paper III

Espen Volnes, Thomas Plagemann, Vera Goebel, and Stein Kristiansen. 2020. EXPOSE: Experimental performance evaluation of stream processing engines made easy. Technology Conference on Performance Evaluation and Benchmarking. Cham: Springer International Publishing, 2020. doi:10.1007/978-3-030-84924-5_2

Paper IV

Espen Volnes, Thomas Plagemann, and Vera Goebel. 2023. To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing. IEEE Communications Surveys & Tutorials (under minor revision).

Paper V

Espen Volnes, Thomas Plagemann, Boris Koldehofe, and Vera Goebel. 2022. Travel light: state shedding for efficient operator migration. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS ’22). Association for Computing Machinery, New York, NY, USA, 79–84. doi:10.1145/3524860.3539638

Paper VI

Espen Volnes, Thomas Plagemann, Vera Goebel, and Boris Koldehofe. 2023. Lazy Migration: Just-In-Time State Migration For Distributed Stream Processing. Submitted to VLDB (September 2023).

Abbreviations

API Application Programming Interface.

CCA Clear Channel Assessment.

CEP Complex Event Processing.

CSE Communication Software Execution.

CSV Comma-Separated Values.

DAG Directed Acyclic Graph.

DCEP Distributed Complex Event Processing.

DS Downstream Nodes.

DSP Distributed Stream Processing.

DSPS Distributed Stream Processing System.

FSM Functional Service Model.

GUI Graphical User Interface.

HAR Human Activity Recognition.

HIRQ Hardware Interrupt.

HM Handover Manager.

IFP Information Flow Processing.

IMU Inertial Measurement Unit.

IoT Internet of Things.

IPAQ IP layer packet queue.

NH New Host.

NLP Natural Language Processing.

OH Old Host.

PID Process ID.

QoS Quality of Service.

RPI Raspberry Pi.

RQ Research Question.

RSD Relative Standard Deviation.

SBON Stream-Based Overlay Network.

SEM Service Execution Model.

SLA Service-Level Agreement.

SPE Stream Processing Engine.

SUT System Under Test.

TOSSIM TinyOS SIMulator.

TPS Tuples Per Second.

US Upstream Nodes.

VANET Vehicular Ad Hoc Network.

WSN Wireless Sensor Network.

YAML YAML Ain’t Markup Language.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

With the rise of the Internet of Things (IoT), billions of devices are now connected to the Internet, producing vast amounts of data. These devices generate immense volumes of data every moment, necessitating robust processing systems. Distributed Stream Processing Systems (DSPS) are at the forefront of this, analyzing real-time data to extract valuable insights. Cloud computing has further amplified the efficiency and affordability of deploying novel services. By centralizing resources in large data centers, it allows service providers to lease rather than purchase equipment, and strategically position themselves closer to both data producers and consumers.

Using the cloud model of sending all data to data centers is not scalable for IoT, due to the amount of data that needs to be processed. Therefore, Fog computing promises the distribution of computing resources, all the way from centralized data centers to the edge of the Internet with the data producers and consumers [9].

DSPSs are usually deployed in data centers, requiring all data to be sent to them. This occurs even when initial filter operations reveal data that does not match the filter criteria. A straightforward way to distribute DSPSs is to prioritize Quality of Service (QoS) when placing filter operators. By doing so, these operators can be strategically positioned directly at the data producer nodes, enabling source filtering.

The rapid expansion and distribution of DSPS queries have inevitably led to significant QoS challenges, including the heterogeneity of hardware resources and the spatial distances between them. Addressing these challenges requires an effective means of conducting geo-distributed experiments. However, a gap exists in the current DSPS landscape since many of these systems are designed with centralization in data centers in mind.

Adapting to changing QoS conditions is more challenging with geo-distributed DSPSs. A node might fail or the network connection quality might be reduced, causing the system to perform sub-optimally. In a data center, the primary concern is not necessarily the placement of the query operators, but how resources are allocated to them. By effectively scaling up, down, in, or out, and ensuring a balanced load, the QoS can be upheld. In a geo-distributed system, the complexity increases with various factors, including the possibility that a migration may not be successfully completed. With the added complexity of geo-distribution, advanced operator migration mechanisms are essential to ensure optimal system performance.

Given the challenges of geo-distribution, simulations serve as a valuable tool before deploying solutions on real-world systems. Simulators such as DCEP-Sim [66] offer a controlled experimental environment, ensuring consistent results and replicability. They provide a way to fine-tune parameters and evaluate various scenarios, proving essential for testing innovative algorithms for operator placement and migration.

The intricacies involved in geo-distributing DSPS queries, coupled with the essential requirement for effective experimentation, highlight the inadequacies of traditional methods. Although simulations emerge as a viable alternative, the limitations of existing tools and frameworks pose additional challenges.

1.1 Problem Statement

With the rise of large-scale DSPS applications, arises the urgent need for frameworks that can support them, especially given the challenges and resource-intensive nature of deploying these applications in real-world settings. These challenges have motivated us to explore the potential of realistic DSPS simulation as an alternative to real-world experiments. Despite the existence of DCEP-Sim, one of the few dedicated DSPS simulators, it currently operates more as a prototype, lacking many core DSPS functionalities. Therefore, we aim to bridge these gaps, with an emphasis on supporting distributed stream processing operator execution, adaptation mechanisms, and the execution of complex DSPS experiments. In particular, we aim to address the following problems (P):

- P1: In the era of distributed stream processing and fog computing, a multitude of heterogeneous devices, ranging from powerful servers to highly resource-constrained sensors, collaborate to process and manage data. Accurately reflecting the performance of these diverse devices in discrete event simulators like DCEP-Sim becomes a crucial, yet non-trivial task. The difficulty is compounded when dealing with resource-constrained devices, whose limited capabilities require unique consideration in simulations. Understanding and modeling such a varied landscape of devices is essential to accurately simulate DSPSs.

- P2: In the growing field of distributed stream processing, there is a lack of standardized terminology and unified platforms, making it challenging to compare and build upon different solutions. Decisions regarding operator migration, which is a disruptive and complex mechanism, often rely on basic thresholds, lacking comprehensive exploration. The existing literature does not adequately investigate the balance between "when" migration is warranted, "why" it is necessary, and "how" it should be conducted. This absence of standardization and understanding hinders innovation, collaboration, and the development of new solutions, limiting the expansion of state-of-the-art techniques in distributed stream processing.

- P3: In DSPSs, operator migration mechanisms often fail to account for the differing importance of state elements, instead focusing on completing the migration process as swiftly as possible. In DSPSs, while load shedding is commonly used to handle overload scenarios, the optimization of migration mechanisms for these situations remains crucial. The challenge lies in developing more adaptive migration mechanisms that align with the actual requirements of the system, recognizing the differing importance of various state elements, and employing techniques that expand the state-of-the-art in these complex and volatile contexts.

1.2 Research Questions

Each of these three research problems highlights a topic that we address in this Thesis. These problems are broad and there is not just one way of solving them. Therefore, we narrow in on the problems by deriving a research question (RQ) for each research problem.

- RQ1: Is it feasible to accurately model the timing behavior of DSPSs in a discrete event simulator like DCEP-Sim?
- RQ2: How can we identify and incorporate common tasks across different DSPSs and complex event processing (CEP) systems in a manner that allows for fair, realistic, and replicable performance evaluation?
- RQ3: How can an adaptation mechanism for DSPSs work well in geo-distributed environments while minimizing disruptions during operator migration?

1.3 Research Methods

To address our research questions, it is important to pick the right methods. The methods we choose shape how we collect and understand our data. We have picked methods that help us get clear and meaningful answers to our questions [31].

- **Literature review:** This Thesis has an overarching theme of addressing a lack of unified terminology and systems for comparing and studying DSPSs and algorithms. Therefore, a significant amount of effort has been put into studying the literature to make an attempt to understand the state-of-the-art and conceptualizing it. This approach is more comprehensive than a traditional systematic literature review, aiming to provide deeper insights beyond the standard encyclopedic knowledge. The outcome is a more holistic and rigorous understanding of DSPSs, laying the groundwork for future research in this domain.

- **Algorithmic Design**: Addressing the complex challenge of operator migration in DSPSs, this research method involves investigating existing migration strategies to identify their strengths and weaknesses. Building upon this foundational knowledge, new and more efficient algorithms are conceptualized and formulated.

- **Performance modeling:** The processing delay of DSPSs can be represented through various modeling techniques, each offering different levels of accuracy. These models are often derived by tracing actual systems to capture their run-time behavior. The complexity of integrating these models into a simulator varies, depending largely on the depth and intricacy of the models.

- **Iterative Development and Testing:** Implementing new features in a system encompasses the continuous cycle of designing, implementing, testing, refining, and re-testing. For this Thesis, it includes the development and iterative refinement of query processing functionalities, migration mechanisms, and processing delay models. Through continuous testing and feedback loops, these components are honed to meet the desired functionality and performance benchmarks within the DSPS environment.

- **Experimental evaluation:** For the evaluation of processing delay models, query processing features and migration mechanisms, real-world or simulation experiments are conducted that represent real-world scenarios. For real-world experiments, it necessitates setting up multiple DSPSs and coordinating them, while for simulation-based experiments, we define, configure, and execute multiple simulation configurations. Obtaining results and visualizing them is also an essential aspect of performing real-world and simulation-based experiments.

1.4 Contributions

To address the research questions, we used the research methods to make significant practical contributions to the field. These are detailed in six published papers. Below, we outline each of these contributions (C).

- C1: We created a tracing framework for resource constrained wireless sensor networks (WSN) devices, and demonstrated the accuracy of processing delay models of WSN devices that are based on the methodology by Kristiansen et al. [42].
- C2: We showed that it is possible to apply the methodology by Kristiansen et al. [42] to modeling DSPSs, and identified obstacles for creating comprehensive models.
- C3: We identified the common tasks of DSPSs and created an application programming interface (API) based on these tasks.
- C4: We created a framework for performing distributed stream processing experiments, applying C3. The experiments can be defined in an easy way and executed on any DSPS that implements the API.
- C5: We created a conceptual model of operator migration that can be used to understand the state-of-the-art in a more collected way, using both new and existing terminology. This conceptual model was used to survey the literature for studies that present migration mechanisms and perform migration decisions.
- C6: We combined load shedding and operator migration in Travel Light [78] to enable operator migration when the state is too big or the connection too poor. By giving higher priority to some partial states than others, the most important partial states can be moved. The rest of the states are dropped.
- C7: We extended the original DCEP-Sim [66] to DCEP-Sim 2.0 with an implementation of DSPS operators and an operator migration mechanism that exploits knowledge of the state to more efficiently handle stateful operators such as join and sliding window timed aggregation operators.
- C8: Through the successful implementation and evaluation of Lazy Migration, the capabilities and extended functionalities of both Expose and DCEP-Sim 2.0 are demonstrated. This highlights their practical applicability and efficiency in distributed stream processing applications.

Table 1.1 offers a structured representation of the relationship between the six papers associated with the Ph.D. work and the three problems they address, which are P1, P2, and P3. In this overview, Paper I addresses problem P1 with contribution C1. Meanwhile, Paper II also focuses on problem P1 but introduces contribution C2. Shifting focus to problem P2, Paper III provides contributions C3 and C4. Paper IV further contributes to problem P2 with C5. Problem P3 is explored by Paper V, which offers contribution C6, and Paper VI, which adds contributions C7. Paper VI also contributes to problem P2 with C8.

Paper/Problem	P1	P2	P3
Paper I	C1		
Paper II	C2		
Paper III		C3,C4	
Paper IV		C5	
Paper V			C6
Paper VI		C8	C7

Table 1.1: Overview of how each paper contributes to address specific research problems

Figure 1.1 describes the three main topics of the Ph.D. Thesis: operator migration, DSPS simulation, and DSPS evaluation. It also shows the overlap between the topics and the relevance of the papers to the topics. Paper I is mainly about modeling the processing delay of IP-forwarding in resource-constrained WSN devices that can be used DSPS simulation. Paper II is about modeling the processing delay of real-world DSPS systems in DCEP-Sim [66], and therefore intersects DSPS simulation and DSPS evaluation. Paper III introduces an experimental framework that makes running DSPS experiments easier, which makes it relevant for DSPS evaluation. Paper IV introduces a conceptual model about operator migration and does a tutorial on the topic of operator migration. Moreover, it includes real-world experiments that compare different operator migration mechanisms. Therefore, Paper IV involves both operator migration and DSPS evaluation. Paper V and Paper VI both introduce new operator migration mechanisms, which are implemented and evaluated in DCEP-Sim. Therefore, they combine operator migration with DSPS simulation. Chapter 4 introduces DCEP-Sim 2.0, which combines all three topics in that it discusses all the work that has been done on DCEP-Sim. This categorization acts orthogonal to the problems, research questions and contributions, and presents a different angle to the works in this Ph.D. Thesis. Later in Chapter 3, we take a closer look at the connection between these topics and papers.

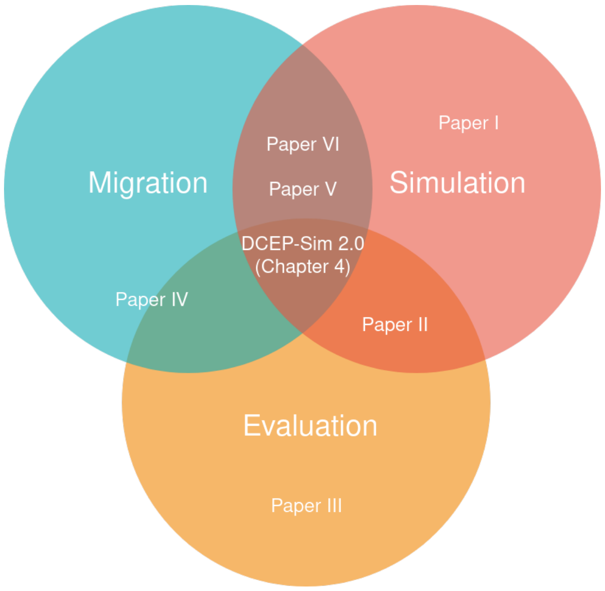


Figure 1.1: Venn diagram detailing the topics of the papers in this Ph.D. Thesis

1.5 Outline

This Thesis is based on two major parts. Part I gives a general introduction to the Thesis and presents material that is not contained in the research papers that form Part II. Chapter 2 describes the background knowledge that is needed to understand the contributions of this dissertation. The background includes a presentation of DSPS, adaptation in DSPSs, performance evaluation of DSPSs, and modeling and simulation of DSPS. Chapter 3 describes the research papers that are included in the dissertation, and how they are connected. A general overview of each paper is described, in addition to a discussion about the relevance of the paper to the Thesis. Chapter 4 details the extensions made to DCEP-Sim, which facilitated the research presented in Paper V and Paper VI. This chapter includes significant contributions to the dissertation that are unpublished, due to the lack of available time. Chapter 5 answers the research questions that were presented in this introduction in detail, describes future directions and concludes the summary of the included papers. Part II consists of six chapters, one for each of the included research papers. In Appendix A, we have collected configuration files from DCEP-Sim 2.0 that are too detailed to be included in the text.

Chapter 2

Background

This chapter aims to give the reader an understanding of the problems that are faced in this dissertation. As the topic revolves around distributed stream processing, we first start out with background on DSPSs. One of the main challenges that such systems face is that the workload for the system may vary significantly, causing the system to potentially be underprovisioned or overprovisioned at any time during execution. Therefore, DSPSs actively optimize execution through adaptations, with operator migration being one of the core mechanisms of these adaptations. Operator migration is explored extensively in this Thesis and is therefore an important topic to investigate. Further, performance evaluation of such systems is also an important topic that we study in this Thesis. Finally, modeling and simulation of DSPSs is explained and discussed.

2.1 Distributed Stream Processing Systems

A DSPS is a system that continually processes data tuples using different kinds of operators. Operators include filter, group by, join, aggregation, pattern-matching, and more. It can be used in IoT and Smart * applications where data is produced by possibly thousands of devices, and data needs to be processed on a large scale. These systems must be reliable, efficient, and be able to meet the demand of the data producers and consumers at any given time.

Numerous DSPSs exist, including but not limited to Storm[1], Flink [10], Esper[2], Siddhi [67], and T-Rex [17]. For a more comprehensive list, please refer to the survey by Isah et al. [32]. Apache Beam[3] is a system that provides a unified interface to existing stream processing systems, where each supported system needs a runner that represents the integration between Beam and a given system.

2.1.1 System Model

A data stream is an unbounded sequence of tuples that are continuously generated over time [11]. It is denoted as $S = t_1, t_2, t_3, ...$, where t_i represents the i_{th} tuple in the stream. For example, a data stream that represents

[1]https://storm.apache.org

[2]https://www.espertech.com/esper

[3]https://beam.apache.org

stock market prices could be denoted as $S = t_1, t_2, t_3, ...$, where $t_i =$ (symbol, AAPL), (price, 150.23), (time, 2022-02-14 10:30:00), representing the stock symbol, price, and timestamp of the i_{th} price update in the stream.

A tuple is an ordered list of attribute-value pairs that represents a single unit of data. It is denoted as a set of key-value pairs, where the keys represent the attribute names and the values represent the corresponding attribute values. For example, a tuple that represents a person's information could be denoted as (name, John), (age, 30), (gender, male). Instead of including the attribute names in the tuples, a data stream expects the incoming tuples to follow a schema, and thus, the attributes are inferred.

A query in a DSPS is a function that processes one or more input data streams and produces an output stream based on defined criteria. It is represented as $Q(S)$, where S denotes the input data stream and Q is the defining function. In DSPSs , the logic and the computational functions to analyze and transform data streams are given in form of operators, e.g., filter, join, group by, aggregation, and pattern-matching operators. The operators are commonly organized in a data flow graph, called the operator graph. The operator graph models dependencies between operators and data sources in receiving and producing tuples from and to specific streams. The operators are executed on hosts of the distributed infrastructure. They can also be dynamically migrated between hosts to meet the performance requirements of the application or react to other changes, such as failures.

Nodes in a DSPS can be static or mobile, and have one or more of the following roles:

- *Data producer*: Examples of this include sensors that convert analog signals into data tuples, often with a fixed sampling rate, and software monitors that might create data tuples at a dynamic rate. Crucially, the DSPS must be able to process all tuples produced by these sources.

- *Data consumer*: These are nodes that request a service, and typically have some QoS requirement, such as a bound of the tuple latency.

- *Operator host*: These nodes execute at least one operator and contribute to event forwarding in the operator network, i.e., map the input events (from upstream nodes) of the operators they execute to output events, and forward them to downstream nodes in the operator network.

Data stream processing operators may be *stateful* or *stateless*. For instance, when joining two data streams, arriving tuples are placed in a data window, where they remain and can be joined until they expire and are removed from the window. When aggregating state, such as counting words, we are typically interested in creating an aggregate per key. In concurrent systems, each key produces output separate from other keys, and as such, these aggregates can be produced by different processes. Therefore, it is common to parallelize such queries, and execute some keys on one host and other keys on another host, in a cluster.

2.1.2 Complex Event Processing

CEP is a technology that performs data stream processing to derive higher level events, also called business events [45]. Instead of simply filtering, joining or aggregating tuples, CEP introduces specialized pattern-matching operators. These operators can recognize and interpret sequences of events that meet certain conditions, representing them as a distinct event type. CEP can be applied to do anomaly detection on credit card transactions, to uncover credit card fraud. CEP queries can be used to interpret a higher level event called SuspiciousTransactions, based on a set of regular transaction events [20]. Most CEP systems have traditional DSPS features, but with more finely grained operators that can look at event sequences.

CEP can be considered as a layer on top of distributed stream processing, as it is in Apache Flink [10]. Flink has a CEP layer that uses distributed stream processing, but allows for more advanced pattern-matching operators that use non-deterministic finite automata for internal state, based on [2]. These patterns are similar to regular expression operators, and can be used to find a specific number of tuples that fulfill a given set of conditions. Thereafter, the query can produce a complex event that can trigger higher-level events. This makes CEP able to express many more types of queries compared to DSPSs, where the number of possible queries is limited to filtering, joining, aggregating, and transforming the data streams.

2.1.3 Heterogeneity of Systems

Heterogeneity is a big overarching challenge in this Ph.D. Thesis when trying to evaluate, compare, and model systems that might use different terminology to indicate the same functionality. The reason for this heterogeneity is two-fold. First, DSPSs have evolved over the last two decades from two branches [18]: the database community and the publish-subscribe community. As such, the terminology and concepts that are used differ. Cugola et al. [18] described a unified model that is called Information Flow Processing (IFP) as a way to overcome this obstacle. However, this model was intended to highlight the similarities between these communities, and not meant to be adopted and replace existing views.

The second reason for heterogeneity is due to the lack of a common standard, which means that new systems have little reason to adopt the language of existing systems. Apache Storm describes a data producing node as a *sprout*, and a data processing node as a *bolt*. In CEP, a query is typically called a pattern, which is different from queries in DSPSs in that patterns are more finely grained and tunable than DSPSs. In T-Rex [17], incoming tuples build sequences of tuples, and a tuple may contribute to multiple sequences, triggering zero or more matches.

2.2 Adaptation of DSPSs

Operator migration and load shedding are adaptation mechanisms employed by DSPSs to deal with unsustainable conditions for DSPSs.

2.2.1 Load Shedding

Load shedding is an established mechanism for operator execution to react to overload situations, e.g., as originally proposed for the data stream management system Aurora [1, 69]. In overload scenarios, part of the workload for an operator is dropped to stabilize the system. Most of the literature describes solutions where input tuples are dropped [5, 15, 21–23, 39, 59, 63, 69]. For aggregation operators, the goal is to minimize the relative error of the calculated aggregate. For join operators, the goal is to drop those tuples that, during their remaining time in the window, join with the fewest tuples. Another method is to drop windows [68] internally, which reduces the number of produced aggregates instead of reducing the accuracy of the aggregates. In pattern-matching operators, dropping input tuples is likely to distort the results completely, because individual tuples can determine whether a sequence fulfills a pattern or not. In such cases, a different state-based load shedding mechanism that drops partial states from the operator is a better option. In CEP systems, the state is often materialized as partial matches. A partial match might or might not result in a complex event. If the likelihood of the partial match in producing output is low, the entire sequence of tuples might be dropped. This is done for the pattern-matching operator in a few recent works [14, 64, 80, 81]. As a result of load shedding, the consistency may be invalidated, but the accuracy and usefulness of the query may remain high.

2.2.2 Operator Migration

Operator migration is a mechanism for exchanging operators between hosts in the DSPS. It requires organizing the state transfer between the old and new host and reorganizing the flow of data streams. A major objective of current operator migration procedures is to ensure consistency, i.e., to ensure the migration of the entire state completes and the resulting migration has no impact on the operator results.

Approaches for performing operator migration can be classified according to their stream management during the state transfer, i.e., in a *single track* or *parallel track* [82]. In single-track migration, the tuples of upstream operators are buffered (at the upstream node, new host, or old host). Therefore, the migration procedure results in a temporary downtime during the handover between the new and old host until all upstream tuples and operator state are transferred consistently.

Parallel-track migration algorithms are able to migrate state without operator downtime by upstream nodes sending tuples to the old and new host. Either the old host continues its executions until the state transfer has been completed or the old host gradually moves state to the new host. These algorithms require temporary duplication of input streams and good connectivity. Under high system dynamics, e.g., slow communication links and drastically reduced bandwidth, these mechanisms can significantly reduce the performance of the DSPS. Migration mechanisms can also split the state into multiple parts, e.g., by key [26, 30] or into static and dynamic state [19, 55].

Paper IV of this dissertation [74] consists of a tutorial of operator migration, and also discusses when migration is worth it or not. Please refer to this tutorial for an in-depth description of operator migration.

2.3 Performance Evaluation

Performance evaluation within the domain of DSPSs is a complex task, necessary for the continual refinement and optimization of these systems. As the scale, complexity, and heterogeneity of DSPS continue to expand, the ways for assessing their performance must also evolve. The necessity of evaluating aspects such as throughput, latency, resource utilization, and fault tolerance is heightened by the real-time demands placed upon DSPS across various application domains. These evaluations provide insights that drive enhancements, not only to individual system components but to the overall orchestration and efficiency of the DSPS. Tracing and measuring these factors require sophisticated techniques that can capture the intricate interplay between system elements, account for contextual variables, and deliver precise, actionable insights.

2.3.1 Benchmarks

In order to evaluate DSPSs fairly and comprehensively and free from bias, we have to rely on existing benchmarks that describe application areas, datasets and queries that are representative of distributed stream processing in general. A benchmark should describe a relevant application area, a varied and realistic dataset, and a comprehensive set of queries that can be used to stress different types of DSPSs and compare them fairly.

Multiple benchmarks and benchmark tools for stream processing exist in the literature [29, 38]. One of the earliest works introduced is the Linear Road benchmark [4], which can be used to simulate traffic in motor highways.

Systems may then achieve an L-rating that is a measure of their supported query load. Although Linear Road is relatively old, it is still implemented for new systems like Apache Flink [28], and for DSPSs written in P4 to run on ASICS [33]. Other benchmarks include [16, 29, 35, 44, 57, 61]. These benchmarks have in common that they are mainly meant for heavyweight DSPSs such as Apache Storm, Apache Samza, Apache Spark and Apache Flink.

NEXMark Benchmark Suite

NEXMark [70] is a widely recognized benchmark suite tailored for stream processing systems, particularly to evaluate their capabilities and performance under varying conditions. Originating from the realm of database systems, the suite is designed to model a real-time online auction system, featuring complex event streams representing bids, auctions, and people. Central to NEXmark is its suite of queries, each designed to stress different aspects of a stream processing system, such as windowed joins, event-time aggregation, and pattern-matching. These queries provide comprehensive coverage, testing scenarios that these systems may encounter in real-world applications. NEXmark, with its diverse set of queries, offers valuable insights into the performance of DSPSs. However, to ensure a complete understanding of system performance, it is essential to complement it with other benchmarks and real-world datasets.

2.3.2 Experimentation

Running experiments with DSPSs is cumbersome, due to the complexity of running distributed experiments, lack of a standard for DSPSs, and lack of experimental frameworks. Running distributed experiments is in and by itself a complex task, because of the unpredictability of running and synchronizing multiple machines. This makes it hard to replicate results, which makes it hard to obtain insights from running the experiments. With the lack of a common standard for DSPSs, it is hard to compare them. They use different APIs, different terminology and architectures, and emphasize different concepts.

Few experiment frameworks for DSPSs exist that aim at providing a user-friendly experience. The PEEL experiment framework is one of them [8]. It enables users to define experiments, execute them, and repeat them. Runtime logs from the running systems are collected, and so the experiments can be used to benchmark systems. Their experiment definitions need special treatment for each DSPS. FINCoS [51] is another experiment framework, which is an extended version of the benchmarking framework in [50]. They enable users to use their own datasets and can communicate with different DSPS engines.

The unification of the use of DSPS is an ongoing effort. A Stream SQL standard is recently proposed in [6] and is in the process of being implemented for existing DSPSs. Apache Beam[4] is a framework that attempts to unify DSPSs by providing a unified interface for writing distributed stream processing applications.

2.3.3 Tracing and Measuring

In the evolving landscape of DSPSs, ensuring accurate and comparable performance metrics is a complex matter. Various systems offer different metrics and tools to trace run-time performance, yet the heterogeneity in these tools poses significant challenges. While metrics such as tuple processing time, input rate, output rate, selectivity, and backpressure are generally agreed-upon, their precise definitions can differ subtly across systems. Such disparities pose challenges for the analysis, making fair comparisons challenging.

Systems like Flink [10] come equipped with built-in metric tools, offering users insights into the performance of the application and QoS. These tools, while undeniably valuable, often tailor their metrics to the internal design and architecture of the system. If relied upon exclusively, these metrics might not present a comprehensive view, potentially missing certain aspects of the performance of the DSPS. While built-in tools might highlight if the system performance has deteriorated or improved relative to a previous time frame, they offer little insight into how the DSPS performs in an absolute sense or in comparison to its peers.

2.4 Modeling and Simulating

The final topic of this background addresses the modeling and simulation of DSPSs. Proper modeling requires a comprehensive understanding of the fundamental operations of the DSPS, including query processing and adaptation, as well as the capability for performance evaluation. In this thesis, we primarily use DCEP-Sim as our simulation platform. However, the original version of this platform demonstrated significant shortcomings in these aforementioned areas.

2.4.1 Discrete Event Simulation

Discrete event simulation provides a method for modeling the evolution of a system over time. Unlike continuous simulation, which processes events throughout the entire time-frame, discrete event simulation focuses on events at specific moments when the system undergoes changes. This approach

[4]https://beam.apache.org

allows the use of a global clock, with each event assigned a specific execution time. Such events are then placed into a priority queue, ensuring that the next event to be processed is the earliest one. Notably, while time progresses between events, it remains static during the execution of each event, setting discrete event simulation apart from hardware emulators.

A significant application of discrete event simulation is for the simulation of network communication. Tools like ns-3 [58], OMNeT++ [72], and OPNET [13] leverage discrete event simulation, focusing on the logical aspects of communication and do not model the physical elements involved in data transmission. Instead, they efficiently schedule the sending and receiving of packets based on factors like bandwidth availability and packet size.

2.4.2 Simulators

The field of fog and edge computing simulation has witnessed the development of numerous simulators that primarily model generic services. Examples include CloudSim [25], iFogSim [27], iFogSim2 [46], EdgeCloudSim [65], FogNetSim++ [56], and others. These tools largely emphasize interactions between edge, fog, and cloud nodes using generic services, rather than the specialized operators vital for data stream processing. Singh et al. [62] study the simulation and emulation tools for fog computing, and conclude that iFogSim [27] is the most popular simulator and EmuFog [49] is the most popular emulator for fog computing.

Simulation of DSPS is a practical and simple way of trying out applications and topologies before applying them in real-world systems. If a company plans to deploy a DSPS for data collection, transformation, and monitoring, they benefit from understanding what the workload might become and how many hardware resources might be necessary. They also benefit from testing correctness of data stream queries in an isolated and simulated environment. One of the strongest benefits of a simulator over a real-world DSPS is that it can easily replicate executions using the same dataset as previously. Monitoring the behavior of a simulated distributed system is much simpler than monitoring a real distributed system. Research has been performed for decades on the topic of monitoring distributed systems [34, 41, 43, 47, 48, 60, 79], and it is still no easy task, even with modern and more user-friendly systems.

In DSPS simulation, DCEP-Sim [66] and ECSNeT++ [3] are two prominent tools. While ECSNeT++ is built upon OMNeT++, and therefore, inherits network simulation functionality, it primarily simulates an abstract representation of operator processing tasks. Each operator in ECSNeT++ is defined through parameters like selectivity and productivity ratio. In contrast, DCEP-Sim embeds an authentic DSPS system in a simulation framework.

DCEP-Sim 1.0

Implemented on top of the ns-3 network simulator [58], DCEP-Sim was designed to streamline evaluations of Distributed CEP systems. Its architecture allows for the simulation of distributed experiments within one program that runs on a single CPU thread. This design not only simplifies evaluations, but also enables concurrent simulations under varied scenarios. By utilizing the object aggregation feature of ns-3, the simulator can seamlessly integrate new selection or placement policies without altering the existing classes.

While DCEP-Sim embodies significant promise for DSPS simulations, its initial version, DCEP-Sim 1.0, exhibits several shortcomings. Its foundational architecture and modules, though in place, offer limited functionality in areas like placement, adaptation, and query processing. Dynamic placement and adaptation are rudimentary, and the query processing operators are restricted to basic CEP operations like AND and OR. For comprehensive evaluations, there is a need to incorporate advanced DSPS features. Without these features, the scope of the simulator remains constricted to a few specialized cases. Given this gap in current simulation tools, this dissertation aims to enhance DCEP-Sim [66] in version DCEP-Sim 2.0 (see Chapter 4).

Chapter 3

Overview of Research Papers

This Thesis is composed of six papers. Each paper is listed below, accompanied by an explanation of its relevance to the Thesis. Figure 3.1 describes the relationship between the papers and shows their topic area. The papers are either centered around performance modeling and evaluation, or operator migration. To start, Paper I [73] and Paper II [77] are about modeling the temporal behavior of resource-constrained WSN devices and DSPSs, respectively. Paper III [75] describes an experimental framework that can be used to run real-world DSPS experiments. We later apply this to DCEP-Sim, and details of this are described in Chapter 4. Paper IV [74] lays the foundation for us to understand and compare existing operator migration mechanisms. It also enabled us to extend DCEP-Sim 1.0 with advanced operator migration mechanisms to DCEP-Sim 2.0. Therefore, Paper IV and DCEP-Sim 2.0 became the foundation for Paper V [78] and Paper VI [76]. Paper VI [76] also took significant inspiration from Paper V with the state shedding mechanism that it presents.

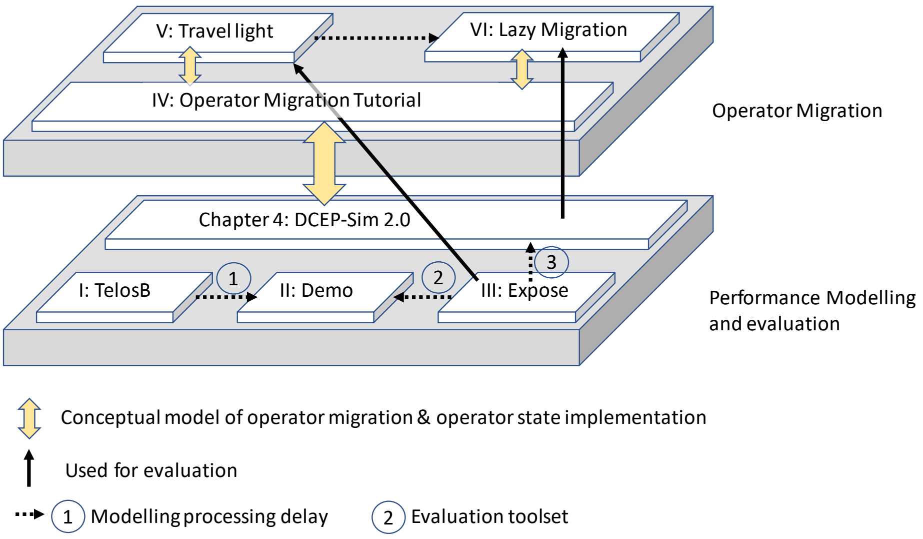

V: Travel light
VI: Lazy Migration
IV: Operator Migration Tutorial
Operator Migration
Chapter 4: DCEP-Sim 2.0
3
I: TelosB
1
II: Demo
2
III: Expose
Performance Modelling and evaluation
Conceptual model of operator migration & operator state implementation
Used for evaluation
1 Modelling processing delay
2 Evaluation toolset

Figure 3.1: Relationship between the papers

3.1 Paper I

Title: Improving the accuracy of timing in scalable WSN simulations with communication software execution models

Authors: Espen Volnes, Stein Kristiansen and Thomas Plagemann

Status: Espen Volnes, Stein Kristiansen, and Thomas Plagemann. 2021. Improving the accuracy of timing in scalable WSN simulations with communication software execution models. Computer Networks, volume 188, 2021, 107855. doi:10.1016/j.comnet.2021.107855

Abstract: Emerging infrastructure-less network architectures such as WSNs consist of devices that perform packet processing in software. General-purpose network simulators do currently not possess models to simulate the intra-node delay of such devices. For example, a TelosB mote with TinyOS spends seven ms on processing packets with a size of 36 bytes and fifteen ms on packets of 124 bytes. The core problem addressed in this work is that simulation does not include such delays, and therefore, the results are inaccurate. To overcome this problem, we create a communication software execution model of TelosB that accounts for its temporal behavior to enable more accurate WSN simulations in the ns-3 simulator. A challenge is to create a tracing framework for TinyOS that can be used to accurately and reliably trace the behavior of a very resource-constrained system. By analyzing the software execution of TelosB running TinyOS in the emulator Cooja/MSPSim and on a real device, we discover discrepancies in the temporal behavior. The evaluation of our model shows that it is scalable and accurate; the simulated intra-OS delay deviates at most 5% from the intra-OS delay in the real mote. When we include the model in simulations, the forwarding capacity of a mote is decreased by 36%. The WSN community can use this model for more realistic simulations, and future WSN mote models will be easier to make with it as a foundation.

Relevance for the Thesis: This paper provides DCEP-Sim with simulation models that can be used to simulate accurately the processing delay of resource-constrained nodes. This way, it helps to answer RQ1. Such models are crucial for two reasons: for the system to function properly without overflowing queues, and to reflect the heterogeneous environment that DSPSs often operate in, with many kinds of devices with different capacities.

3.2 Paper II

Title: Demo: Modeling the Software Execution of CEP in DCEP-Sim

Authors: Espen Volnes, Stein Kristiansen, Thomas Plagemann, Vera Goebel, and Morten Lindeberg

Status: Espen Volnes, Stein Kristiansen, Thomas Plagemann, Vera Goebel, and Morten Lindeberg. 2019. Modeling the Software Execution of CEP in DCEP-Sim. In Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems (DEBS ’19). Association for Computing Machinery, New York, NY, USA, 244–247. doi:10.1145/3328905.3332508

Abstract: DCEP-Sim facilitates simulation of distributed CEP where the latency and bandwidth limitations in the network are well reflected, but it currently lacks models to simulate the temporal behavior of event processing. In this demonstration, we use a modeling methodology to model the software execution of a CEP system called T-Rex. We instrument and trace T-Rex to parameterize a software execution model that is integrated into DCEP-Sim. Furthermore, we use this instance of DCEP-Sim to run simulations and see how significant the processing delay introduced by the model is compared to the transmission delay.

Relevance for the Thesis: This paper partially answers RQ1 with regard to modeling the temporal behavior of DSPSs and was crucial in highlighting the need for DSPS experiment frameworks. We constructed detailed processing delay models for both Siddhi and T-Rex. However, due to space constraints, only the T-Rex model is presented in this paper. Although we were successful in applying the modeling methodology by Kristiansen et al. [42] to model the temporal behavior of DSPSs in the DCEP-Sim simulator, there are some limitations. Originally, this methodology was applied to IP forwarding, and since DSPS behavior is significantly more complex, there are many more possible behaviors. The modeling methodology is based on obtaining traces, which means we need to trace all the scenarios that we want to reflect in the models. This includes for DSPSs all kinds of queries and datasets. This requires an experimental framework that enables us to perform distributed experiments in an easy way, which is what Paper III is about.

3.3 Paper III

Title: EXPOSE: Experimental Performance Evaluation of Stream Processing Engines Made Easy

Authors: Espen Volnes, Thomas Plagemann, Vera Goebel, and Stein Kristiansen

Status: Espen Volnes, Thomas Plagemann, Vera Goebel, and Stein

Kristiansen. 2020. EXPOSE: Experimental performance evaluation of stream processing engines made easy. Technology Conference on Performance Evaluation and Benchmarking. Cham: Springer International Publishing, 2020. doi:10.1007/978-3-030-84924-5_2

Abstract: Experimental performance evaluation of stream processing engines (SPE) can be a great challenge. Aiming to make fair comparisons of different SPEs raises this bar even higher. One important reason for this challenge is the fact that these systems often use concepts that require expert knowledge for each SPE. To address this issue, we present Expose, a distributed performance evaluation framework for SPEs that enables a user through a declarative approach to specify experiments and conduct them on multiple SPEs in a fair way and with low effort. Experimenters with few technical skills can define and execute distributed experiments that can easily be replicated. We demonstrate Expose by defining a set of experiments based on the existing NEXMark benchmark and conduct a performance evaluation of Flink, Beam with the Flink runner, Siddhi, T-Rex, and Esper, on powerful and resource-constrained hardware.

Relevance for the Thesis: This paper is important for this Thesis because it contributes to answering RQ2. Expose laid the foundation for how the distributed experiments are conducted in the subsequent papers. The API it provides makes it easy to express on a high level what is done in an experiment or algorithm. The experiment framework makes it easy to define and execute distributed experiments, and therefore, none of the remaining papers would be possible without it. This paper focused on real-world distributed experiments, but we later applied the concepts of Expose to DCEP-Sim because it makes it easier to define and run simulations. Conveniently, an experiment configuration that was initially designed and executed in a real-world environment required only the addition of network link information to be compatible with DCEP-Sim. Moreover, although Paper IV is a tutorial and survey paper, Expose tasks are used to model existing migration mechanisms and in experiments for demonstrating different migration mechanisms.

3.4 Paper IV

Title: To Migrate or Not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing

Authors: Espen Volnes, Thomas Plagemann and Vera Goebel

Status: Espen Volnes, Thomas Plagemann, and Vera Goebel. 2023. To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed

Stream Processing. IEEE Communications Surveys & Tutorials (under minor revision).

Abstract: One of the most important issues in distributed data stream processing systems is using operator migration to handle highly variable workloads cost-efficiently and adapt to the needs at any given time on demand. Operator migration is a complex process involving changes in the state and stream management of a running query, typically without any data loss, and with as little disruption to the execution as possible. This tutorial aims to introduce operator migration, explain the core elements of operator migration, and provide the reader with a good understanding of the design alternatives used in existing solutions. We developed a conceptual model to explain the fundamentals of operator migration and introduce a unified terminology, leading to a taxonomy of existing solutions. The conceptual model separates mechanisms, i.e., how to migrate, and policy, i.e., when to migrate. This separation is further applied to structure the description of existing solutions, offering the reader an algorithmic perspective on various design alternatives. To enhance our understanding of the impact of various design alternatives on migration mechanisms, we also conducted an empirical study that provides quantitative insights. The operator downtime for the naïve migration approach is almost 20 times longer than when applying an incremental checkpoint-based approach.

Relevance for the Thesis: This paper is fundamental for the operator migration part of this Thesis. It identifies existing operator migration mechanisms, how they work and when operator migration is usually triggered. Since operator migration is a core feature of DSPSs, this work helps to answer RQ2. Papers V and VI would not be possible, if this paper did not highlight the state-of-the-art migration mechanisms and highlight limitations in them. Moreover, this paper contrasts the migration mechanisms with the migration goal and investigates when migration is worth it.

3.5 Paper V

Title: Travel Light - State Shedding for Efficient Operator Migration

Authors: Espen Volnes, Thomas Plagemann, Boris Koldehofe, and Vera Goebel

Status: Espen Volnes, Thomas Plagemann, Boris Koldehofe, and Vera Goebel. 2022. Travel light: state shedding for efficient operator migration. In Proceedings of the 16th ACM International Conference on Distributed and

Event-Based Systems (DEBS ’22). Association for Computing Machinery, New York, NY, USA, 79–84. doi:10.1145/3524860.3539638

Abstract: Operator migration is a crucial concept to adapt event processing systems to dynamic changes. When the placement of a stateful operator changes, the operator state must be migrated to the new host. However, operator state size and time constraints can make it impossible to migrate the operator without severe Quality of Service (QoS) degradation. As a relief, we propose to perform state shedding in such a situation. The core idea of state shedding is to partition the operator state, assign a utility to each partial state, and use the utility and size of each partial state to identify the most useful partial states that can be migrated in a given time frame. Thus, state shedding can maintain a substantially higher QoS with a lower impact on query results than state-of-the-art solutions targeting consistent state at the old and new host. In this paper, we define this novel approach and in a simulation environment evaluate state shedding in migration scenarios with pattern-matching queries.

Relevance for the Thesis: This paper is significant for this Thesis since it connected the two major contributions of this Thesis. First, it applies the operator migration concepts from Paper IV to create a novel migration mechanism that is the first of its kind, addressing RQ3. Previous migration mechanisms aimed to transfer all state, while Travel Light migrates the state in order of expected utility, ensuring that the most important state is migrated in case the migration cannot be fully completed. Second, in order to evaluate Travel Light, we needed to extend DCEP-Sim with query processing features and integrate it with Expose, thus making significant steps towards DCEP-Sim 2.0.

3.6 Paper VI

Title: Lazy Migration: Just-In-Time Fragmented State Migration For Distributed Stream Processing

Authors: Espen Volnes, Thomas Plagemann, Vera Goebel and Boris Koldehofe

Status: Espen Volnes, Thomas Plagemann, Vera Goebel, and Boris Koldehofe. 2023. Lazy Migration: Just-In-Time State Migration For Distributed Stream Processing. Submitted to VLDB (September 2023).

Abstract: Operator migration is an essential adaptation mechanism in distributed stream processing systems. The main challenge is how to

perform adaptations without disruption of the system, i.e., to minimize the experienced latency for the data consumers and to ensure that the data consumers get the correct data. Existing solutions, on the other hand, only focus on minimizing the latency caused by input tuples waiting in queue, which is different from output tuple latency. To do this, we present Lazy Migration, an operator semantic aware migration mechanism that applies the lazy evaluation technique to migrate the necessary state when it is necessary. The mechanism schedules migration of parts of the state at different times, depending on when the operators need them. We introduce two migration modes within Lazy Migration. The first mode, minimizing latency, schedules the migration of state fragments at distinct times based on operator requirements, ensuring the necessary state is migrated exactly when it's needed. The second mode, maximizing utility, incorporates a utility-based approach for prioritizing state migration. Each partial state is assigned a utility value based on its anticipated future demand, with the initial migration order determined accordingly. For instance, a tuple expected to join with numerous others is assigned a higher utility. The migration mechanism is evaluated in a simulation environment in many different contexts.

Relevance for the Thesis: This paper is important for this Thesis on multiple levels. It extends DCEP-Sim with a framework for implementing and using several state-of-the-art migration mechanisms, creating DCEP-Sim 2.0. This paper relies on the previous works in terms of experimental framework (Paper III), operator migration concepts (Paper IV), and the initial operator migration implementation and other extensions to DCEP-Sim (Paper V). This paper applies the contributions from these papers and shows their relevance in a broader perspective.

Chapter 4

DCEP-Sim 2.0

This chapter describes the extensions of the DCEP-Sim [66] simulator. To distinguish between the original published version DCEP-Sim 1.0 and the extended version, we call the extended version DCEP-Sim 2.0. As such, it is an important contribution in this Thesis, and we have made it publicly available[1].

We intended to publish DCEP-Sim 2.0, but this was not possible due to the limited time frame of this Ph.D. work. The extensions to DCEP-Sim 1.0 are essential for providing a platform for the implementation and evaluation of operator migration mechanisms, which are relevant for Paper V [78] and Paper VI [76]. Without DCEP-Sim 2.0, the operator migration mechanisms would have to be implemented in a real-world system, which would substantially increase the workload, in addition to making it difficult to compare our work and the results with the state-of-the-art solutions.

4.1 Limitations of DCEP-Sim

One of the major challenges of DSPSs and distributed CEP systems in a fog computing environment is the heterogeneity of the systems. The physical resources vary in terms of capacity, reliability, and connectivity with the other nodes in the network. DCEP-Sim simulates the connectivity aspects fairly well because it is built on top of the network simulator ns-3. The originally published version of DCEP-Sim 1.0 by Starks et al. [66] laid the architectural foundation, but some important aspects are missing. In particular, the processing capacity of nodes is ignored entirely, which results in simulations that cannot distinguish between a weak sensor mote and a powerful cloud-based device. Worse yet, ns-3 is a discrete event simulator where processing tasks do not take any time at all. Any time aspect must be explicitly added during the simulation. This aspect is crucial for a simulator because one of its motivations is to simulate the amount of resources necessary for an application and to assess the costs. Other limiting aspects are the query processing operators, only offering the AND and OR operators of CEP. A real DSPS system usually offers many more operators, including filter, group by, join, pattern-matching, aggregation, and select.

DCEP-Sim has been changed significantly throughout this Thesis, each extension opening up new opportunities for research. Table 4.1 summarizes these improvements and which papers were relevant for which changes.

[1] https://github.com/espv/dcep-sim

Parameter	Research paper	Before	After
Processing delay simulation	I [73], II [77]	None	Processing tuples in an operator takes time and occupies limited hardware resources
Experimental framework	V [78], VI [76]	None	Expose
Adaptations	V [78], VI [76]	None	All-at-once, Rhino, Megaphone, Meces, and Lazy Migration
Query functionality	II [77], V [78], VI [76]	AND/OR	Join (equijoin and non-equijoin), filter operator, select operator, time-based and tuple-based aggregation, pattern-matching, and group by

Table 4.1: Improvements of DCEP-Sim 1.0 during this Ph.D. work

DCEP-Sim has been used in Paper I [73], Paper II [77], Paper V [78] and Paper VI [76]. Expose was not added to DCEP-Sim for the Expose paper [75] because we explored real-world scenarios. Later on, we realized that it was a very good framework to use with DCEP-Sim, in Papers V [78] and VI [76].

4.2 Query Processing

In order to add query processing to DCEP-Sim, a new module was created in ns-3 called *stream-processing*. The module contains code for the DSPS operators and the resource manager of the simulation nodes. A StreamQuery object consists of one or more objects that are subclasses of the Operator class. The Operator subclasses that we have implemented are shown in Figure 4.1. The abstract Operator class defines two primary methods for query processing: $process(Ptr < TupleWrapper > input\_tuple)$, which each subclass of Operator must implement, and $emit(Ptr < TupleWrapper > output\_tuple)$, which is implemented by the abstract Operator class. The $process$ method is called to process $input\_tuple$ and $emit$ is called when the operator produces a tuple that will be emitted. If an Operator is connected to a parallelized subsequent operator, it may have multiple output Operator objects. With the availability of multiple CPU cores in the node, performance

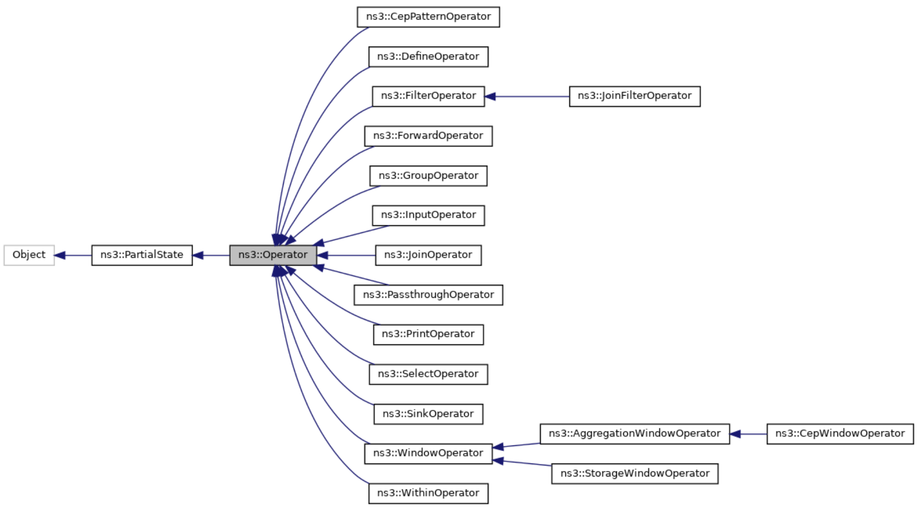


Figure 4.1: Operator class diagram

can be enhanced, allowing for more efficient concurrent processing.

4.2.1 Data Stream Processing Operators

At their core, distributed stream processing operators transform incoming data streams through specific, predefined actions. Ideally, given their defined behaviors, these operators should be straightforward to implement. Despite their seemingly straightforward nature, the execution environment presents complexities. In a typical DSPS, numerous operators are often active simultaneously, competing for limited hardware resources. Allocating these resources efficiently, especially in terms of scheduling available threads, is a significant challenge that goes beyond basic implementation. We focus on implementing the most common DSPS operators:

- Filter operator: Screens incoming tuples based on predefined criteria. Only tuples that meet the criteria are emitted; all others are discarded.
- Group by operator: Categorizes incoming tuples based on specified key attributes, forming distinct groups. Each group consists of tuples that share the same key values. Depending on the subsequent operations, results derived from each group (e.g., aggregates) are emitted.
- Select operator: Forms a new tuple based on specific attributes from the incoming tuples and emits the newly formed tuple.
- Join operator: Takes tuples from two data streams and joins them based on a specific predicate. Tuples from Stream $S1$ are compared with those within the window of Stream $S2$, and any matches are emitted.

- Aggregation operator: Aggregates values of incoming tuples based on a defined key and attributes within a specified window. The aggregated results are emitted according to the emission policy of the operator, which might be time-based or event-based (e.g., upon the arrival of new tuples).
- Pattern-matching operator: Looks for patterns in sequences of tuples, identifying sequences that match a predefined pattern, similar to the function of regular expressions in textual data. Whether an incoming tuple completes, breaks or continues a sequence depends on how the tuple aligns with the tuples in the sequence.

For DSPS operators, there are two main aspects that we must model: (1) processing delay and (2) operator logic.

Processing delay

There are many ways of modeling the processing delay of data stream processing operators, with different levels of complexity. Integrating processing delay into the query processing of DCEP-Sim is not just about realism; it is functionally essential. Given that DCEP-Sim operates in the networked environment of ns-3, processing delay helps prevent overflowing queues and buffers. For instance, if an experiment produces thousands of tuples adding up to 10 megabytes, sending this data all at once via TCP without any processing time can lead to queues overflowing and data being lost. This highlights the need for simulating processing delay, especially when conducting complex DSPS and CEP experiments.

Therefore, it was decided to model the processing delay in three ways: (1) uniform, (2) state-dependent and (3) realistic processing delay models. With uniform processing delay models, each operator takes the same amount of time to process a tuple, regardless of how much state is built up. With state-dependent models, an operator may take longer time to process a tuple when more state has been built up. With realistic processing delay models, the processing delay comes from simulation of the system, and making these models requires tracing real-world systems.

In Paper I and II, our ambition was to create realistic processing delay models. However, as the focus of the Ph.D. work shifted towards operator migration, it became less necessary to have realistic processing delay. In Paper V and VI, it was more important that the different migration mechanisms that were compared experienced the same processing delay, than for the processing delay to be accurate. Therefore, we opted for the uniform processing delay models in Paper V and VI.

However, we aspired to create processing delay models that strike a balance between realism and scalability, and this is where the state-dependent processing delay models come in. These models are inspired by complexity theory, in that they mainly describe how the run-time of the operators scale with the state. In order for the models to yield a specific processing delay, each operator has a weight associated with it. For a stateless operator, i.e., that does not have any state, the weight represents the full processing delay for each incoming tuple. For a stateful operator, the processing delay may increase as the operator state builds up. In this chapter, we explore how these models can be realized.

Operator logic

In the following section, we describe each DSPS operator that we introduced earlier. We provide a pseudocode algorithm that represents from a high level the $process$ method that the operators implement. In addition, we provide a state-dependent processing delay function that shows how the processing delay might increase as the state within the operator accumulates.

Select

The select operator is a stateless operator that forms a new tuple based on specific tuple attributes of the input tuple. The pseudocode for the $process$ method of the select operator is shown in Algorithm 1.

Algorithm 1 Pseudocode of the $process$ method of the select operator

```
procedure Select(tuple)
    selectedTuple ← ExtractAttributes(tuple, attributes)
    emit(selectedTuple)
end procedure
```

The processing delay function can be modeled as a constant, using the weight $wt_{selection}$ where each tuple takes the same amount of time to be processed:

$$D(S(t_{s1})) = wt_{selection} \tag{4.1}$$

Filter

The filter operator is a stateless operator that screens incoming tuples based on predefined criteria. If a tuple meets the criteria, it is emitted; otherwise, it is discarded. Algorithm 2 shows the pseudocode for the $process$ method of the filter operator.

Algorithm 2 Pseudocode of the $process$ method of the filter operator

```
procedure Filter(tuple)
    if condition(tuple) then
        emit(tuple)
    end if
end procedure
```

The processing delay function can be modeled as a constant, expressed by the weight wt_{filter}, each tuple taking the same time to be processed:

$$D(F(t_{s1})) = wt_{filter} \tag{4.2}$$

Group by

The group by operator is a stateful operator that categorizes tuples into distinct groups and subsequently emits them. The purpose of these groups is to enable the query to produce results for distinct groups of tuples, given a set of key attributes. For instance, a query that aggregates a certain value might group on the *city* attribute of the data stream, and thus, produce aggregate values for each city. Algorithm 3 shows the pseudocode for the $process$ method of the group by operator.

Algorithm 3 Pseudocode of the $process$ method of the group by operator

```
procedure GroupByOperator(tuple)
    key ← ExtractKey(tuple, groupFields)
    if key not in groupMap then
        groupMap[key] ← CreateEmptyGroup()
    end if
    AddToGroup(groupMap[key], tuple)
end procedure
```

Incoming tuples can be mapped to a key using a hash-based data structure. This means that the processing delay does not increase with the number of groups. While creating a new group may momentarily elevate the processing delay, such instances are anticipated to be less frequent than processing tuples that belong to existing groups. Therefore, the processing delay function can be represented by the weight $wt_{groupby}$ as:

$$D(G(t_{keys})) = wt_{groupby} \tag{4.3}$$

Join

The join operator is a stateful operator that pairs tuples from a stream, $S1$, with those in a designated window of another stream, $S2$, based on a predicate θ as represented by:

$$S1 \bowtie_{\theta} S2 \tag{4.4}$$

The algorithmic complexity of a join operator can vary significantly depending on the implementation. In the specific scenario of an equijoin operator, the aim is to pair tuples with identical attributes. This can be efficiently modeled using a hash table. The join predicate attribute serves as the key for this hash table. In this case, the pseudocode for the $process$ method of the equijoin operator can be modeled by Algorithm 7. It is worth noting that non-equijoins can introduce greater complexity due to varied join

Algorithm 4 Pseudocode of the *process* method of the equijoin operator

```
procedure Join(tupleA)
    window ← getCorrespondingWindow(tupleA)
    matchingTuples ← window.getTuples(tupleA.joinPredicateAttribute)
    for each tupleB in matchingTuples do
        emit(Combine(tupleA, tupleB))
    end for
end procedure
```

predicates. To keep the presentation clear and concise, we have chosen to present the more straightforward equijoin operator.

For an equijoin operator that employs a hash table, access to the matches is on average constant. As such, the processing delay scales with the number of matches, as shown in the following processing delay function:

$$D(J(t_{s1})) = \sum_{t_{s2}}^{window_{s2}} wt_{join} * match(t_{s1}, t_{s2}) \tag{4.5}$$

Where wt_{join} is the weight used for the operator, $window_{s2}$ is the window that the incoming tuple t_{s1} can join with, and $match(t_{s1}, t_{s2})$ returns 1 if there is a match, and 0 if not.

Aggregation

The algorithmic complexity of the aggregation operator is influenced by its implementation, similar to the join operator. Specifically, for window-based aggregation, there are two primary approaches for maintaining the operator state: retain tuples and compute aggregations on-demand as output tuples are produced, or maintain partial aggregates that are updated when tuples arrive. In this Thesis, we adopt the second method of keeping partial aggregates that are updated by incoming tuples.

If the data stream is grouped with a group by operator, the operator produces an aggregate per key. The exact moment when output tuples are formed and emitted depends on the emission policy of the operator, e.g. time-based or event-based. The pseudocode for the *process* method of this aggregation operator is shown in Algorithm 5.

Algorithm 5 Pseudocode of the *process* method of the aggregation operator

```
procedure Aggregate(tuple)
    key ← tuple.groupAttribute
    for each partialAggregate in aggregates[key] do
        partialAggregate.value ← aggregationFunction(partialAggregate.value, tuple.value)
        if shouldEmit(partialAggregate) then
            emit(partialAggregate)
        end if
    end for
end procedure
```

Given the partial aggregate-based state implementation, the processing delay function can be modeled as being linear with regard to the number of partial aggregate results to update for each incoming tuple:

$$D(A(t)) = wt_{aggregation} * (size_{window}/jump_{window}) \tag{4.6}$$

Where $wt_{aggregation}$ is the weight used, $size_{window}$ is the size of the window, $jump_{window}$ is the window jump, and $size_{window}/jump_{window}$ is the number of partial aggregate results that each incoming tuple updates.

Pattern-matching
The pattern-matching operator is a stateful operator that identifies tuple sequences that match a specified pattern. Although various implementations are possible, our focus is on patterns that resemble regular expressions. The pseudocode for the process function of this operator is shown by Algorithm 6.

Algorithm 6 Pseudocode of the *process* method of the pattern-matching operator

```
procedure PatternMatch(tuple)
    for each eventSequence in sequences do
        eventSequence.append(tuple)
        if isMatch(eventSequence) then
            emit(constructResult(eventSequence))
            sequences.remove(eventSequence)
        else if breaksPattern(eventSequence) then
            sequences.remove(eventSequence)
        end if
    end for
    if startsNewPattern(tuple) then
        newSequence ← createNewSequence(tuple)
        sequences.append(newSequence)
    end if
end procedure
```

The processing delay of this operator scales with the number of tuple sequences that the tuple may contribute to, and the complexity of the pattern, as shown in the following processing delay function:

$$D(P(t)) = \sum_{sequence_i} wt_{pattern} * C(sequence_i) \tag{4.7}$$

Where the summation iterates over all sequences relevant to tuple t, $C(sequence_i)$ gives the complexity of the pattern of $sequence_i$ as a floating point number within a range $[1, n]$, and $wt_{pattern}$ is the weight for the operator.

4.3 Operator Migration

DSPSs and CEP systems have a variable workload, like any distributed system. Depending on the complexity of the operators, quantity of state and input load, the need for hardware resources varies. Moreover, the network may at times become saturated or nodes may fail. Therefore, it is impossible to configure such a system in a way where it will never benefit from changes in hardware resources or geographical location. As such, an integral mechanism of DCEP-Sim is to handle adaptations, and this topic is one of the main topic of this Ph.D. Thesis, with three of the six publications revolving around this topic.

The originally published work by Starks et al. [66] had laid the foundation in terms of architecture, but had no adaptation mechanism implemented and tested. Before it is meaningful to implement adaptation mechanisms, actual query processing must take place. The complexity starts when attempting to migrate stateful operators, and various adaptation mechanisms exploit assumptions that can be made when migrating various stateful operators. Rhino [19], for instance, migrates the full state of operators with no downtime. While this state is being migrated, the system continues running and state changes that happen during the migration are migrated as an incremental checkpoint afterward. The amount of new state depends on the operator type and the amount of tuples that have been processed. Rhino's method is versatile, but for certain operators, like aggregations, its performance may suffer if the entire state is updated during the initial migration phase.

On the other hand, Megaphone [30] and Meces [26] employ a more specialized approach. They segment the state into micro-batches based on keys, with Megaphone operating on both the old and new hosts, transitioning in stages to minimize downtime. Meces reroutes all tuples to the new host, and enables the new host to fetch the state that incoming tuples need.

Both Megaphone and Meces excel with operations that have a direct relationship between incoming tuples and state, like equijoin. An equijoin operator joins data streams with a specified set of attribute values, whereas non-equijoin operators can look at a range of values. Challenges arise with non-equijoin operations, as they might require Megaphone to keep tuples on both the old and new hosts, or cause Meces to issue numerous fetch requests due to slight value variations.

Paper VI uses all the previous topics for the implementation and evaluation of the evolved DCEP-Sim implementation. Expose helps this by being able to define benchmarks and executions. Finally, adaptations are implemented, with either full or variable consistency, depending on the requirements and context.

It was very important to extend DCEP-Sim with the ability to allow for different migration mechanisms to be implemented, considering the focus of adaptations in distributed CEP in this Ph.D. Thesis. Figure 4.2 shows the classes that were made to represent the migration mechanisms that were implemented during the Ph.D. Thesis. In the original DCEP-Sim, there were no functionality in place for this, only the idea of having different policies that made it possible to define how adaptations should be performed. It was essential that the most important migration mechanisms could be expressed

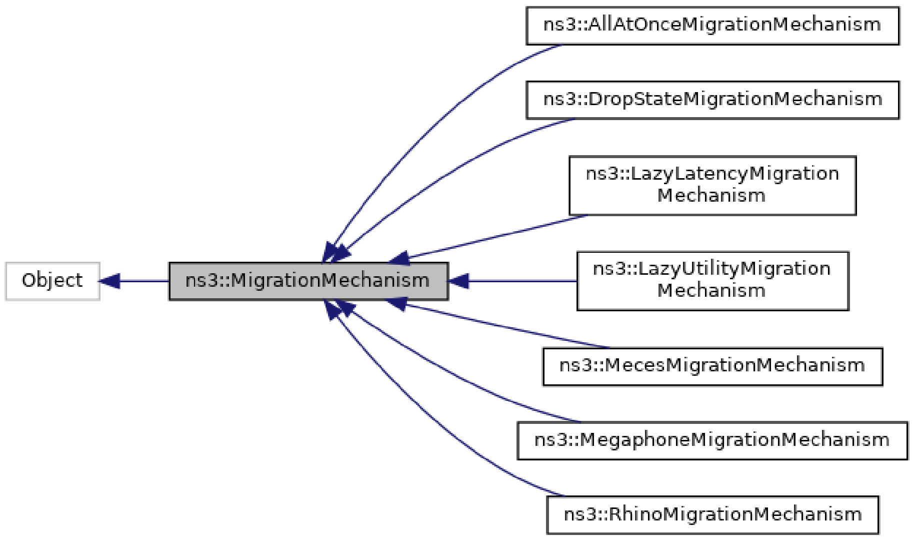

ns3::AllAtOnceMigrationMechanism
ns3::DropStateMigrationMechanism
ns3::LazyLatencyMigration Mechanism
Object
ns3::MigrationMechanism
ns3::LazyUtilityMigration Mechanism
ns3::MecesMigrationMechanism
ns3::MegaphoneMigrationMechanism
ns3::RhinoMigrationMechanism

Figure 4.2: Migration mechanism class diagram

in DCEP-Sim, with most of the supported operators, as illustrated in Figure 4.1. The goal then became to enable users to conduct adaptations with a specified migration mechanism, and extend the simulator with new migration mechanisms.

A crucial concept of the state-of-the-art migration mechanisms is the idea of *partial state*. Partial state is the concept of splitting the full state into smaller parts that can be migrated independently. In order to model this, an abstract class called $PartialState$ was made, that represents a component that can be migrated. Figure 4.3 shows all subclasses of this class. Each class counts as a state that can be migrated independently. Different migration mechanisms use different ways of prioritizing state and migrating it.

The implementation of distributed stream processing operators is a prerequisite for the implementation of migration mechanisms. The reason why is that migration mechanisms are all about exploiting opportunities for more efficient operator migration. If the operator state implementation is merely hypothetical, there is not sufficient detail to fully understand or express how migration mechanisms can be performed.

Therefore, with the operator execution implemented, two methods opened up for the migration of any distributed stream processing operator: ExtractState and ImportState. ExtractState copies the state of an operator on the old host to a container that is sent to the new host, and ImportState is executed on the new host to copy this state into the new operator. These two methods serve as the fundamental methods for implementing migration mechanisms.

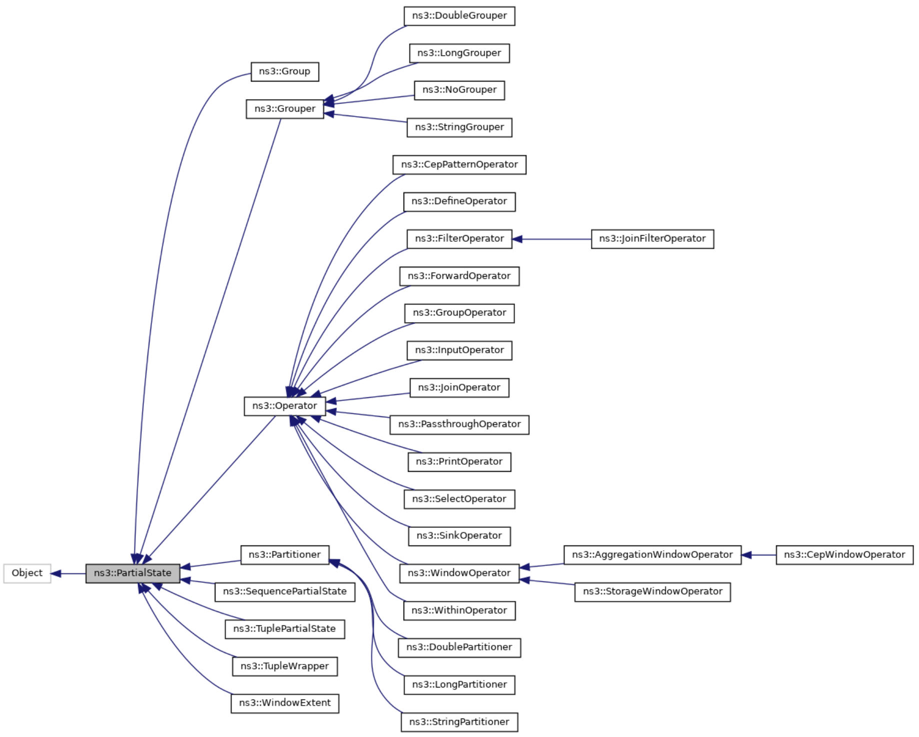


Figure 4.3: Partial state class diagram

The migration controller is the node that is performing the migration from one node to one or more other nodes. The controller invokes the ExtractState method on all the operators, and places the states to move in an order that complies with the migration mechanism that is applied. It also disables the operator execution on the old host according to when the migration mechanism specifies it. This can vary, e.g., some state might be sent before the operator is shut down.

In Lazy Migration [76], five state queues were defined, each playing a crucial role in the migration mechanism: preamble state, critical state, fetch state, deadline state, and normal state. For a detailed explanation of these state queues, as well as the fundamental ExtractState and ImportState methods, readers are referred to [76]. Within this work, it is demonstrated how these queues were sufficient to express migration mechanisms such as Lazy Migration's minimize latency mode, maximize utility mode, Rhino [19], Megaphone [30], Meces [26], and the standard All-at-once migration mechanism.

4.4 Communication

Communication in DCEP-Sim is done using the models provided by ns-3. Originally, DCEP-Sim only supported UDP. To enable adaptation that might require the migration of gigabytes of state, we had to enable TCP support.

While TCP is a standard transport protocol, it is not trivial to make it work in a robust way in ns-3. Despite the large amount of models in ns-3, it falls short in areas crucial for transmitting large data volumes. For instance, the TCP library lacks helper functions for sending large payloads, making the simultaneous receipt of payloads from multiple nodes more complex. A partially received packet header requires merging with the subsequent packet for complete reading. Debugging such problems, even within the controlled environment of ns-3, proves challenging. Ns-3 also has limitations when it comes to monitoring packet activities. If a packet goes missing, using the available tools to trace it is impossible. Therefore, if the system is not optimized for the intended load, operators might produce fewer tuples than anticipated, making it difficult to discern between query processing errors and communication issues.

4.5 Expose

Expose was originally designed as a framework for performing distributed stream processing experiments in real world scenarios. However, it was clear that there were features in Expose that could be applied to other scenarios. The API that defines high-level tasks could be used in production environments for making changes to the environment. The tasks can be used to communicate between live systems, directing each other. Furthermore, it can be used to manage simulations in DCEP-Sim, and this was a good way to improve the usability of DCEP-Sim. By extending the Expose configuration with network characteristics of the nodes in the simulation and detailed query definition features, Expose was expressive enough to define fully-fledged DCEP-Sim simulations.

Figure 4.4 shows the relationship between Expose and DCEP-Sim. Expose takes care of setting up the simulation environment, e.g., the node topology and communication links. Each node connects with a router, and this way, each node can reach the other nodes in two hops. Expose orchestrates the overarching simulation events such as query deployment and data transmission, while DCEP-Sim performs the discrete event simulation, and thus simulates the dynamics of packet transmission and reception as well as performing the query processing.

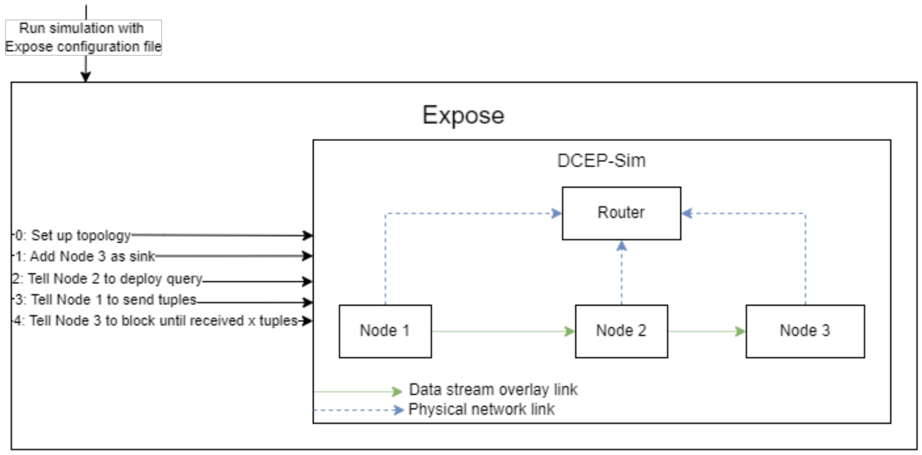

Run simulation with Expose configuration file
Expose
DCEP-Sim
Router
0: Set up topology
1: Add Node 3 as sink
2: Tell Node 2 to deploy query
3: Tell Node 1 to send tuples
4: Tell Node 3 to block until received x tuples
Node 1
Node 2
Node 3
Data stream overlay link
Physical network link

Figure 4.4: Integration of DCEP-Sim with the Expose experimental framework

4.5.1 Query definition

After adding detailed query definition features to DCEP-Sim, the challenge was to make an intuitive interface for the user to define such queries. There are generally four ways of doing this:

- programming language API,
- SQL-like language,
- natural language, or
- graph-based with a Graphical User Interface (GUI) editor.

In order to make the best choice, the query definition should fulfill the following criteria:

- Cr1: Should be easy for a non-expert to use.
- Cr2: Must not add steps to the experiment execution.
- Cr3: Must make it straightforward to configure the processing pipeline.

Cr1 is desirable because the intention with DCEP-Sim is to make it easier to run distributed experiments. If the configuration of the simulations is too challenging, it will make it less viable for most users. Cr2 is necessary because the experiment configuration should be possible to change without recompiling the code. Cr3 is essential because we want to be able to configure how the data pipeline in the DSPS query works, without arbitrary rules or query optimization causing changes in it.

Programming language API

A programming language API for defining queries in DSPSs is quite common, and is similar to how databases are accessed and queried from programs using an object-relational mapping framework such as SQLAcademy for querying MySQL in Python.

DSPSs like Apache Flink [10] offer a programming language interface to write queries from code. For instance, a timed aggregation query can be written as:

```
// specify table program

Table auctions = tableEnv.from("Auction");
Table bids = tableEnv.from("Bid");

// Perform join, tumbling window, and aggregation
Table result = bids
        .join(auctions)
        .where($("Auction.id").isEqual($("Bid.auction")))
        .window(Tumble.over(lit(7).days()).on($("Bid.proctime")).as("weekWindow"))
        .groupBy($("Bid.auction"), $("weekWindow"))
        .select($("Bid.auction"), $("Bid.price").avg().as("avg_price"));
```

The problem with this approach is that it requires compilation of the code, which violates criterion Cr2. Programming also often requires an expert-user, because different systems are written in different languages. Therefore, this approach violates criterion Cr1.

SQL-like language

The second option is to define an SQL-like language for defining SQL queries, similar to what is offered by most of the DSPS systems that are discussed in this dissertation. A timed aggregation query can then be defined as:

```
select A.id, max(B.price)
from Auction A
join Bid b on B.auction = A.id
group by A.id, tumble(1 week)
```

The benefit of this approach is that it is a fully declarative way of defining DSPS queries. The downside is that while SQL queries explain what data the user wants, the user cannot decide how it is done. Meaning, the resulting operator graph might vary depending on the query optimization rules. Therefore, this violates criterion Cr3. For a normal user, this is helpful because it can help improve the performance of the query. In our system, however, it is important that we can decide the data pipeline explicitly.

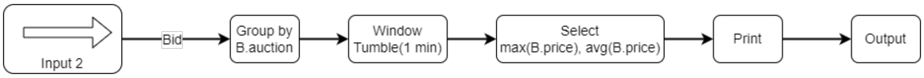


Figure 4.5: Grouped aggregation query in GUI

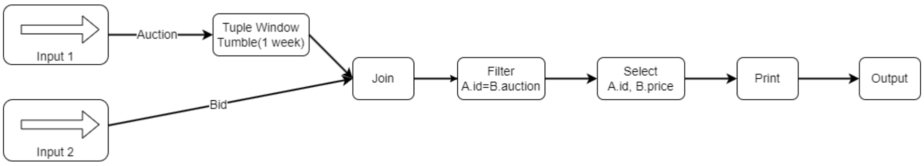


Figure 4.6: Join query in GUI

Natural language

Natural language processing (NLP) has been studied with regard to query processing [7, 24, 36, 37, 40, 52–54, 71], but with a focus on converting natural language text to SQL queries. Meaning, it has the same problem as using an SQL-like language has: that it is not easy to define exactly how the distributed stream processing operators should be connected. Meaning, NLP can be used to define what data you want, but not exactly how it is done. Therefore, it violates criterion Cr3.

Graph-based with GUI editor

With a GUI, the user can define operator graphs visually using a drag-and-drop system. Such a system is easy for a non-expert to use (Cr1), does not add steps to the experiment execution (Cr2), and the processing pipeline is straightforward and easy to change (Cr3). This means that all the criteria are fulfilled, and therefore, we use this approach.

Figures 4.5, 4.6 and 4.7 represent queries as visualized in the GUI, but where minor aesthetic enhancements have been made for clarity. The text in the vertices is just for visually representing its contents. The user can click on the vertices to view and edit parameters of the operators. Figure 4.5 shows a query where a stream is grouped and aggregated. Figure 4.6 shows a query that joins two streams. Figure 4.7 shows a query where two streams are first joined, before they are grouped and aggregated. In all queries, the results are printed out, which is usually done on the sink nodes. Appendix A shows what these queries look like stored in the YAML format that Expose has in its configuration. Listing A.4 shows the query from Figure 4.5, Listing A.3 shows the query from Figure 4.6, and Listing A.4 shows the query from Figure 4.7.

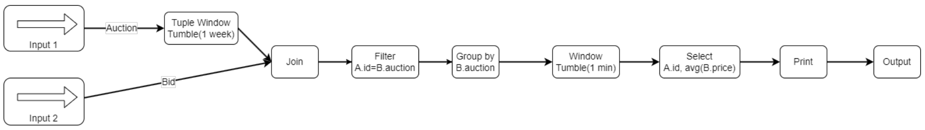


Figure 4.7: Join followed by grouped aggregation query in GUI

4.5.2 Simulation setup

To integrate DCEP-Sim with Expose, DCEP-Sim needs to set up the simulations entirely from the YAML configuration file provided by the user. An example configuration file is shown in Listing A.1. Here, we see definition of stream schemas (denoted by stream-definitions), experiments, datasets, queries (denoted by spequeries), results creation (denoted by plots), and network configuration.

DCEP-Sim must interpret each of these components to correctly set up the simulations. The schema definitions are necessary in order to understand the contents of the datasets and the input and output of queries. Therefore, the 'stream-id' is utilized when transmitting datasets as data streams in experiments and when defining the input and output of stream processing queries.

Schema definition

A schema definition specifies the attributes present in a tuple, their types, and their order. The supported types are:

- int,
- long,
- timestamp,
- string,
- double, and
- float.

Internally, there are only three core types: *long*, *string*, and *double*. *int* and *timestamp* are interpreted as *long*, and *float* is interpreted as *double*. Each node maps the schemas upon start of the experiments, to ensure that they can interpret incoming tuples of all schemas.

Experiment

An experiment is structured as a sequence of tasks set to be executed on specific nodes. Each task is directed to a designated node and is then managed by that node's Expose wrapper. In real-world scenarios using Expose, every participating DSPS must have its respective Expose wrapper. For our integration of DCEP-Sim and Expose, we have created a compatible Expose wrapper. The Expose wrapper implements all tasks that should be possible to perform in a DCEP-Sim experiment. This includes manual tasks such as sending a dataset as a data stream, deploying queries and performing adaptations. Furthermore, it includes automatic tasks that are issued by the DSPS during simulation, e.g., adding stream schemas, ending experiment, and tasks used for communication between nodes during adaptation.

The experiment configuration is also used to determine which nodes take part in the experiment. This way of defining the node composition comes from reverse-engineering the original Expose's way of waiting to execute the experiment until all nodes have registered. Instead of waiting to execute the experiment, the coordinator uses the list of nodes to find out which nodes to initialize.

Dataset

The dataset tag describes the file location of a dataset that can be parsed and sent as a data stream. The supported types are currently YAML and CSV, where CSV offers superior performance.

Query

As discussed previously, we decided to develop a GUI-based solution, where the user defines operator graphs explicitly, connecting the operators with arrows, to indicate the data pipeline flow. Each query has a set of input operators, indicating the incoming data streams, and an output operator for indicating the end of the query. This information needs to be converted to the YAML format, which can then be interpreted by DCEP-Sim's C++ code. There are two key pieces of information in an operator graph: the vertices (operators), and edges (data flow). The vertices define the operator type and parameters, and the edges define the connections between the operators, i.e., where to send the output of an operator.

Take the very simple example SQL query:

```sql
select B.price, B.auction
from Bid
```

This can be represented in YAML using three operators and two edges. Each operator has a name, type and a set of parameters. The type of operator can be input, select, filter, join, output, etc. The set of parameters is operator-dependent, e.g., a list of fields in the select operator, the filter predicate for the filter operator, etc. Each edge has a stream name, an input operator with the "from" tag, and an output operator with the "to" tag.

Listing 4.1: Vertices and edges to represent simple distributed stream processing query

```
operators:
— name: Output 0
  type: output
  parameters: {stream—id: —1}
— name: Select 0
  type: select
  parameters:
    fields: [B.price, B.auction]
— name: input 0
  type: input
  parameters: {stream—id: 3, alias: B}

edges:
— stream: Bid
  from: {name: input 0}
  to: {name: Select 0}
— stream: OutputQuery
  from: {name: Select 0}
  to: {name: Output 0}
```

Plots

The plots tag can be used to define what kind of metrics the users want to collect from the simulations. Specifically, it configures DCEP-Sim to trace a certain type of measurements, which later on can be used to calculate the specified metrics. Given that distributed stream processing is a highly complex application with many possible things to measure and trace, it is infeasible to collect all results. Moreover, we want the Expose configuration to be all that a user needs to configure in order to configure experiments, and collecting measurements is a critical aspect of conducting experiments. Therefore, these plot tags are a good way of reducing the complexity of running the simulations. Examples of metrics include input tuple latency, window emission delay, window aggregation accuracy, and more.

Network configuration

The network configuration tag in the YAML file defines the bandwidth and latency between the nodes in the experiment. These are crucial pieces of information that affect the behavior of the experiments. A higher bandwidth increases the maximum rate at which tuples or operator state can be transmitted over the network. If the bandwidth is too high, the nodes may get saturated with tuples or state, causing overload on nodes. If it is too low, it could cause the network to get saturated, leading to loss of data.

Topology configuration is done in DCEP-Sim in a simplified manner. When the nodes have been discovered, a star network is created, which means that each node is connected to a router. This ends up with a star pattern where the router is in the middle. This way, each node is connected to each other through the router.

4.6 Conclusions & Future Work

One of the ambitions in this dissertation was to publish an extended version of the DCEP-Sim paper that consists of accurate processing delay models that are based on real-world DSPSs such as Esper[2], Flink [10], Siddhi [67], or T-Rex [17]. To achieve this, we would need to methodically measure the processing time performance in a real-world system that could be applied to the DSPS operators in DCEP-Sim. The problem is that the more complex a system becomes, the harder it is to model.

Therefore, the later works Paper V and VI of the Thesis applied uniform processing delay models, where an operator takes the same amount of time to process any tuple, regardless of how much state an operator has built up. We also introduced the concept of state-dependent models, where processing times may vary based on the accumulated state of the operator. These models hold potential for a realistic approach that is promising for future research.

For DCEP-Sim to truly mirror real-world DSPSs, its execution model must also be precise. Achieving this is no simple task, particularly with more advanced DSPSs. Whereas some systems such as Siddhi can run in a simple library mode with a single thread for the execution, advanced DSPSs such as Flink have a complex execution framework that manages many threads. In Flink, a Flink program is deployed to the JobManager, which is managed by resource frameworks like YARN, and then deployed to worker nodes called TaskManager's. Each job is managed by a JobMaster, and multiple jobs can execute in one Flink cluster, each managed by a JobMaster. To which extent these entities affect the performance of the system is not trivial to understand. Therefore, extensive experimental research needs to be conducted to learn more. However, we have laid a foundation with the papers provided in this dissertation that can be used to explore this matter further.

[2]https://www.espertech.com/esper

Chapter 5

Conclusion

This dissertation has focused on enabling configurable, replicable and scalable DSPS experiments, and to expand the state-of-the-art to perform adaptations in DSPSs with new migration mechanisms that can deal with geo-distribution and unstable scenarios with lower bandwidth than centralized cloud scenarios.

5.1 Answering the Research Questions

In this section, we delve into a discussion on how the research questions are comprehensively addressed within the scope of this dissertation.

RQ1: Is it feasible to model accurately the behavior of DSPSs in a discrete event simulator like DCEP-Sim?

We found that it is feasible to model some aspects of DSPSs behavior, notably concerning forwarding devices and query processing, within a discrete event simulator like DCEP-Sim. However, given the intricate nature of DSPSs, it became clear that a comprehensive experimental framework is essential for both creating and critically evaluating these models. This realization naturally paved the way for the formulation of RQ2.

As we extended the query processing functionality with more query processing operators, we settled for uniform processing delay models that add a specific amount of delay each time a tuple is processed, which only depends on the operator type. Without any such delay, the network links overflow almost immediately. Scaling up the experiments with larger workloads and higher speed networks would lead to network saturation, if operators received or produced tuples at too high frequency, compared to the processing delay. Therefore, throughout the rest of the dissertation, the uniform processing delay models were used.

RQ2: Can common tasks and adaptation mechanisms be identified across different DSPSs and CEP systems, and be used to fairly compare and evaluate these systems through realistic experiments?

Answering RQ2 required a deep understanding of the literature, in addition to existing systems and frameworks.

The practical problem of executing distributed experiments in a simple way was answered with Expose [75]. We identified a common set of tasks that are generally executed during a DSPS experiment. This includes tasks for deploying DSPS queries, setting up the stream topology, sending datasets as data stream, and more.

One of the core features of DSPSs is the adaptation mechanisms, and we explore this in Paper IV, that is a tutorial and survey of operator migration [74]. This paper defines a conceptual model of operator migration that can be used to understand the different types of migration mechanisms that exist, and also the motivations for performing operator migration.

By answering RQ2, it was possible to identify gaps in the literature with respect to migration mechanisms in geo-distributed environments.

RQ3: How can an adaptation mechanism for DSPSs work well in geo-distributed environments while minimizing disruptions during migration?

With RQ3, we aimed to investigate how operator migration mechanisms can be made to work more efficiently in geo-distributed environments. We saw an opportunity for new migration mechanisms that can work even when the migration cannot be fully completed. In geo-distributed environments, the network might degrade or nodes may fail. When nodes are geo-distributed, it introduces challenges that can affect the reliability of the network and complicate node management. One common way that DSPSs deal with overload scenarios is that they apply load shedding and drop incoming tuples to reduce the load. By combining load shedding and operator migration, we developed Travel Light [78], a migration mechanism that sends the operator state in descending order of usefulness. The most important state is sent first, which ensures that even if the operator migration must end prematurely, the system can continue on the new host with the most useful parts of the operator state.

Another improvement to operator migration mechanisms was developed after realizing that operators do not necessarily need all the state to be able to process tuples. We then recognized the potential to migrate the operator state based on the order required by the incoming tuples during migration.

The positive results from the evaluation of Travel Light in Paper V [78] and Lazy Migration in Paper VI [76] are a clear reflection of their well-constructed design and functionality. These outcomes emphasize the robustness and versatility of Expose and DCEP-Sim, demonstrating their viability and accuracy in the realm of distributed stream processing.

5.2 Summary of Contributions

This Thesis has made several key contributions in the domain of DSPSs. In this section, we highlight these core advancements:

- C1: We made a tracing framework for resource constrained WSN devices that can trace the systems in low workload settings with high timestamp accuracy, and high workload settings with reduced timestamp accuracy. The low workload settings mode was used to create accurate processing delay models, based on the modeling methodology by Kristiansen et al. [42], and the high workload setting was used to evaluate the accuracy of the models in terms of packet loss simulation.
- C2: We created detailed processing delay models of the Siddhi and T-Rex DSPS systems, based on the modeling methodology by Kristiansen et al. [42]. In this process, we discovered limitations to DCEP-Sim, and the need for more advanced experimentation frameworks, that can be used to obtain data for the creation of such processing delay models.
- C3: We identified a set of common tasks that are executed by DSPSs in experiments and general execution. These include setting up stream topology, setting nodes as sink nodes, deploying DSPS queries, sending datasets as data streams, block until a specified number of tuples have been received, and more.
- C4: A framework for performing distributed stream processing experiments was created. Experiments are defined in a human readable and declarative way, by using YAML. A coordinator node starts up with an experiment definition, and each system that is included in the experiment must implement the API described by C3. The coordinator parses the tasks from the experiment and issues them to the correct node.
- C5: We created a conceptual model of operator migration to consolidate understanding of the state-of-the-art, using a mix of new and established terminology. This conceptual model was used to survey the literature on operator migration, both with respect to how the migration is done (migration mechanisms), and what triggers it (migration decisions). We did a quantitative evaluation of the conceptual model through a migration decision use case and operator migration experiments, that showed that advanced migration mechanisms can migrate 62 times bigger state, and 19 times faster.
- C6: We introduced state shedding as a strategy to manage operator migrations in event processing systems when full state transfers might

degrade QoS. This technique partitions the operator state and assigns utilities to prioritize the migration of the most valuable parts within a given time frame. Our simulations with pattern-matching queries show that state shedding sustains higher QoS and has less impact on query results compared to traditional migration approaches.

- C7: We introduced Lazy Migration, an operator semantic aware migration mechanism for DSPSs. This innovative mechanism optimizes operator state management based on its semantics, like window extents for aggregation or tuple lists for joins. Lazy Migration has two distinct modes: the latency mode, which harnesses window semantics to strategically migrate state partitions reducing output latency, and the utility mode that schedules partial state migrations prioritizing the most crucial states. Through extensive experimentation and comparison with state-of-the-art solutions, using various join and aggregation use-cases, we have shown the distinct advantages of Lazy Migration over other prevalent migration strategies.
- C8: Through the successful implementation and evaluation of Lazy Migration, the capabilities and extended functionalities of both Expose and the DCEP-Sim are demonstrated. This highlights their practical applicability and efficiency in distributed stream processing applications.

Additionally, this Thesis made an effort to provide the research community with open source code of each project that we developed.

5.3 Critical Assessment

This work attempts to solve the described problems in the best way, but we acknowledge that there are limitations, like in all research. The topic is very broad and complex, and modeling such systems is a non-trivial task. Our approach has been systematic and deliberate, but there are ways that would have improved the investigation and results. First, developing use cases that are based on actual queries and workloads that are observed in real-world situations where data stream processing is done. If we had, e.g., three such use cases that are created in collaboration with companies, we could create benchmarks that are more realistic. However, the difficulty of developing such use cases and doing collaboration with companies is high. Companies cannot collect any data from their systems that relates to customers without analyzing the ethical issue in detail. Moreover, the general difficulties of collecting, understanding and cleaning real data is a significant challenge in experimental research.

The decision to utilize DCEP-Sim [66] as the evaluation platform for Lazy Migration [76] was a carefully considered one. DCEP-Sim offers a controlled environment to conduct experiments with replicable results (with ease). However, its representation of real-world systems is somewhat constrained due to its simplistic processing delay models and restricted scalability. The scalability concern primarily stems from the inherent limitations of ns-3 [58], the underlying foundation for DCEP-Sim. ns-3 runs its detailed simulation models, simulating every aspect of packet transmission, in a single-threaded manner. This approach contrasts with real-world distributed systems where nodes can not only forward packets and process tuples concurrently but also operate several processes in parallel, leveraging the capabilities of each CPU core. In DCEP-Sim, this single-threaded restriction is a significant limitation. To navigate around this during our experiments, we executed multiple simulations simultaneously, thus exploiting the multi-core capability of our testbed. However, this strategy introduced a new challenge: RAM became a significant bottleneck. Despite having a substantial amount of RAM (377 GB) at our disposal, we had to oversee memory consumption, ensuring that concurrently running simulations, even of a relatively modest scale, did not overwhelm the memory capacity of the system.

5.4 Future Work

This dissertation opens up for several possible future work topics, and we explore two important topics in this section: extensions to DCEP-Sim and the integration of DCEP-Sim in a real-world DSPS to function as a digital twin.

5.4.1 DCEP-Sim extensions

DCEP-Sim can be extended in multiple ways to reflect real-world conditions more realistically. In mobile settings, energy consumption is an important consideration. If the simulated setting includes mobile devices, DCEP-Sim should use energy consumption models that drain the battery of devices when data is received and transmitted, as well as when data is processed by the DSPS.

Ns-3, the network simulator that DCEP-Sim is built on, supports energy models for simulating the energy usage of radio devices. In order to use these models to simulate the energy consumption of data transmission, DCEP-Sim would have to use wireless communication.

Modeling the energy usage of a DSPS processing tuples is harder, since it requires new energy models. The energy models would have to consider the energy consumption when the device is inactive, and when it is processing tuples. There are multiple ways of doing this. The easiest way is to consume a specified amount of energy for each millisecond that the device processes a tuple. This type of model can then be evaluated by comparing it with real-world measurements. A more advanced way would be to model the hardware of the device, such that the time that it takes to process tuples is based on clock cycles, and each clock cycle consumes a certain amount of energy. This type of cycle-accurate energy model has been explored by Chang et al. in [12].

5.4.2 Digital twin

A digital twin is a simulated system that reflects a real-world constituent in a way to help perform critical decisions.

DCEP-Sim 2.0 is meant to be a simulator that simulates the most important aspects of DSPSs, including making adaptation decisions when it benefits the DSPS the most. One way to apply DCEP-Sim could be to embed it in a real-world DSPS, feed predicted data traffic into DCEP-Sim, run simulations, and extract the adaptation decisions that were predicted to improve the performance. These adaptation decisions can then be scheduled in the real-world system before the adversary conditions are faced that demand a change.

This requires a few significant changes and enhancements to DCEP-Sim. First, it requires realistic data traffic models for producing data. The data produced must reflect the real-world scenario in order to have value. However, it is unrealistic to aim for fully accurate traffic generation models that can predict the future. Instead, the query processing of DCEP-Sim can be modified to work with approximated data instead of actual data tuples. In this case, data tuples would have a certain size and data stream ID, but not have actual attribute values. Predicting unique attribute values is impossible, as they can vary significantly. The problem is that operators like filter and join in DSPSs require specific attribute values in order to determine whether to produce output tuples or not.

The solution to this problem is to modify the query processing of DCEP-Sim to a mode where query operators such as filter and join emit tuples based on statistical selectivity. The real-world system collects statistics on how many tuples pass the filter, and are joined with other tuples, and pass this to DCEP-Sim. DCEP-Sim then simulates with these statistics to determine how many tuples to produce in the conditional operators. This way, the traffic generator models only have to know how many tuples are expected, and not detailed attribute values. Several considerations would need to be addressed in order to apply statistics to aid DCEP-Sim. How frequently should the real-world system update the statistics to keep the simulation relevant? What level of variance in the data would render the statistics outdated? How would potential anomalies or outliers be handled?

In conclusion, this Thesis pushed forward the state-of-the-art in tools for performance evaluation of DSPS and operator migration, and opened up for further research challenges.

Bibliography

[1] Abadi, D. et al. “Aurora: a data stream management system”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 666–666.

[2] Agrawal, J. et al. “Efficient pattern matching over event streams”. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, pp. 147–160.

[3] Amarasinghe, G. et al. “ECSNeT++: A simulator for distributed stream processing on edge and cloud environments”. In: *Future Generation Computer Systems* vol. 111 (2020), pp. 401–418.

[4] Arasu, A. et al. “Linear road: a stream data management benchmark”. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment. 2004, pp. 480–491.

[5] Babcock, B., Datar, M., and Motwani, R. “Load shedding for aggregation queries over data streams”. In: *Proceedings. 20th international conference on data engineering*. IEEE. 2004, pp. 350–361.

[6] Begoli, E. et al. “One SQL to Rule Them All-an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1757–1772.

[7] Bhadgale, A. M., Gavas, S. R., and Goyal, P. “Natural language to SQL conversion system”. In: *International Journal of Computer Science Engineering and Information Technology Research* vol. 3, no. 2 (2013), pp. 161–166.

[8] Boden, C. et al. “PEEL: A framework for benchmarking distributed systems and algorithms”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2017, pp. 9–24.

[9] Bonomi, F. et al. “Fog computing and its role in the internet of things”. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. 2012, pp. 13–16.

[10] Carbone, P. et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* vol. 36, no. 4 (2015).

[11] Castro Fernandez, R. et al. “Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 725–736.

[12] Chang, N., Kim, K., and Lee, H. G. “Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI”. In: *Proceedings of the 2000 international symposium on Low power electronics and design*. 2000, pp. 185–190.

[13] Chang, X. “Network simulations with OPNET”. In: *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1*. 1999, pp. 307–314.

[14] Chapnik, K., Kolchinsky, I., and Schuster, A. “DARLING: data-aware load shedding in complex event processing systems”. In: *Proceedings of the VLDB Endowment* vol. 15, no. 3 (2021), pp. 541–554.

[15] Chi, Y. et al. “Loadstar: A load shedding scheme for classifying data streams”. In: *Proceedings of the 2005 siam international conference on data mining*. SIAM. 2005, pp. 346–357.

[16] Chintapalli, S. et al. “Benchmarking streaming computation engines: Storm, flink and spark streaming”. In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2016, pp. 1789–1792.

[17] Cugola, G. and Margara, A. “Complex event processing with T-REX”. In: *Journal of Systems and Software* vol. 85, no. 8 (2012), pp. 1709–1728.

[18] Cugola, G. and Margara, A. “Processing Flows of Information: From Data Stream to Complex Event Processing”. In: *ACM Comput. Surv.* vol. 44, no. 3 (June 2012), 15:1–15:62.

[19] Del Monte, B. et al. “Rhino: Efficient management of very large distributed state for stream processing engines”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2471–2486.

[20] Fülöp, L. J. et al. “Predictive complex event processing: a conceptual framework for combining complex event processing and predictive analytics”. In: *Proceedings of the fifth Balkan conference in informatics*. 2012, pp. 26–31.

[21] Gedik, B. et al. “Adaptive load shedding for windowed stream joins”. In: *Proceedings of the 14th ACM international conference on Information and knowledge management*. 2005, pp. 171–178.

[22] Gedik, B., Wu, K.-L., and Philip, S. Y. “Efficient construction of compact shedding filters for data stream processing”. In: *2008 IEEE 24th International Conference on Data Engineering*. IEEE. 2008, pp. 396–405.

[23] Gedik, B. et al. “A load shedding framework and optimizations for m-way windowed stream joins”. In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 536–545.

[24] Ghosh, P. K., Dey, S., and Sengupta, S. “Automatic sql query formation from natural language query”. In: *International Journal of Computer Applications* vol. 975 (2014), p. 8887.

[25] Goyal, T., Singh, A., and Agrawal, A. “Cloudsim: simulator for cloud computing infrastructure and modeling”. In: *Procedia Engineering* vol. 38 (2012), pp. 3566–3572.

[26] Gu, R. et al. “Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 539–556.

[27] Gupta, H. et al. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. In: *Software: Practice and Experience* vol. 47, no. 9 (2017), pp. 1275–1296.

[28] Hanif, M., Yoon, H., and Lee, C. “Benchmarking tool for modern distributed stream processing engines”. In: *2019 International Conference on Information Networking (ICOIN)*. IEEE. 2019, pp. 393–395.

[29] Hesse, G. et al. “Senska–Towards an Enterprise Streaming Benchmark”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2017, pp. 25–40.

[30] Hoffmann, M. et al. “Megaphone: Latency-conscious state migration for distributed streaming dataflows”. In: *Proceedings of the VLDB Endowment* vol. 12, no. 9 (2019), pp. 1002–1015.

[31] Holz, H. J. et al. “Research Methods in Computing: What are they, and how should we teach them?” In: *Working group reports on ITiCSE on Innovation and technology in computer science education*. 2006, pp. 96–114.

[32] Isah, H. et al. “A survey of distributed data stream processing frameworks”. In: *IEEE Access* vol. 7 (2019), pp. 154300–154316.

[33] Jepsen, T. et al. “Life in the fast lane: A line-rate linear road”. In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–7.

[34] Joyce, J. et al. “Monitoring distributed systems”. In: *ACM Transactions on Computer Systems (TOCS)* vol. 5, no. 2 (1987), pp. 121–150.

[35] Karimov, J. et al. “Benchmarking distributed stream data processing systems”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1507–1518.

[36] Kate, A. et al. “Conversion of natural language query to SQL query”. In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE. 2018, pp. 488–491.

[37] Kaur, S. and Bali, R. S. “SQL generation and execution from natural language processing”. In: *Int. J. Comput. Bus. Res* (2012), pp. 2229–6166.

[38] Kiatipis, A. et al. “A Survey of Benchmarks to Evaluate Data Analytics for Smart-* Applications”. In: *arXiv preprint arXiv:1910.02004* (2019).

[39] Kleiminger, W., Kalyvianaki, E., and Pietzuch, P. “Balancing load in stream processing with the cloud”. In: *2011 IEEE 27th International Conference on Data Engineering Workshops*. IEEE. 2011, pp. 16–21.

[40] Kombade, C. et al. “Natural language processing with some abbreviation to SQL”. In: *International journal for research in applied science and engineering technology* vol. 8, no. 5 (2020), pp. 1046–1048.

[41] Korableva, O. N., Kalimullina, O. V., and Kurbanova, E. “Building the Monitoring Systems for Complex Distributed Systems: Problems and Solutions.” In: *ICEIS (2)*. 2017, pp. 221–228.

[42] Kristiansen, S., Plagemann, T., and Goebel, V. “A methodology to model the execution of communication software for accurate network simulation”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* vol. 26, no. 1 (July 2015), pp. 1–31.

[43] Kufel, Ł. “Tools for distributed systems monitoring”. In: *Foundations of Computing and Decision Sciences* vol. 41, no. 4 (2016), pp. 237–260.

[44] Lu, R. et al. “Stream bench: Towards benchmarking modern distributed stream computing frameworks”. In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE. 2014, pp. 69–78.

[45] Luckham, D. “A Brief Overview of the Concepts of CEP”. In: *Carbon* vol. 45 (2007), p. 15.

[46] Mahmud, R. et al. “iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments”. In: *Journal of Systems and Software* vol. 190 (2022), p. 111351.

[47] Mansouri-Samani, M. and Sloman, M. “Monitoring distributed systems”. In: *IEEE network* vol. 7, no. 6 (1993), pp. 20–30.

[48] Mansouri-Samani, M. and Sloman, M. *Monitoring distributed systems: A survey*. Citeseer, 1992.

[49] Mayer, R. et al. “Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures”. In: *2017 IEEE Fog World Congress (FWC)*. IEEE. 2017, pp. 1–6.

[50] Mendes, M. R., Bizarro, P., and Marques, P. “A framework for performance evaluation of complex event processing systems”. In: *Proceedings of the second international conference on Distributed event-based systems*. 2008, pp. 313–316.

[51] Mendes, M. R., Bizarro, P., and Marques, P. “FINCoS: benchmark tools for event processing systems”. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 2013, pp. 431–432.

[52] Naik, B. B. et al. “An SQL query generator for cross-domain human language based questions based on NLP model”. In: *Multimedia Tools and Applications* (2023), pp. 1–24.

[53] Narhe, A. et al. “SQL Query Formation for Database System using NLP”. In: *International Journal of Engineering Research and* vol. 8 (2019).

[54] Norouzifard, M., Davarpanah, S., Shenassa, M., et al. “Using natural language processing in order to create SQL queries”. In: *2008 International Conference on Computer and Communication Engineering*. IEEE. 2008, pp. 600–604.

[55] Ottenwälder, B. et al. “MCEP: A mobility-aware complex event processing system”. In: *ACM Transactions on internet technology (TOIT)* vol. 14, no. 1 (2014), pp. 1–24.

[56] Qayyum, T. et al. “FogNetSim++: A toolkit for modeling and simulation of distributed fog environment”. In: *IEEE Access* vol. 6 (2018), pp. 63570–63583.

[57] Rabl, T. et al. “The vision of BigBench 2.0”. In: *Proceedings of the Fourth Workshop on Data analytics in the Cloud*. 2015, pp. 1–4.

[58] Riley, G. F. and Henderson, T. R. “The ns-3 network simulator”. In: *Modeling and tools for network simulation*. Ed. by Wehrle, K., Güneş, M., and Gross, J. Berlin, Heidelberg: Springer, 2010, pp. 15–34.

[59] Rivetti, N., Busnel, Y., and Querzoni, L. “Load-aware shedding in stream processing systems”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 2016, pp. 61–68.

[60] Schwiderski, S. *Monitoring the behaviour of distributed systems*. Tech. rep. University of Cambridge, Computer Laboratory, 1996.

[61] Shukla, A., Chaturvedi, S., and Simmhan, Y. "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* vol. 29, no. 21 (2017), e4257.

[62] Singh, S. P. et al. "Simulation and emulation tools for fog computing". In: *Recent Advances in Computer Science and Communications (Formerly: Recent Patents on Computer Science)* vol. 15, no. 3 (2022), pp. 315–322.

[63] Slo, A., Bhowmik, S., and Rothermel, K. "hSPICE: state-aware event shedding in complex event processing". In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 2020, pp. 109–120.

[64] Slo, A., Bhowmik, S., and Rothermel, K. "State-Aware Load Shedding from Input Event Streams in Complex Event Processing". In: *IEEE Transactions on Big Data* (2020).

[65] Sonmez, C., Ozgovde, A., and Ersoy, C. "Edgecloudsim: An environment for performance evaluation of edge computing systems". In: *Transactions on Emerging Telecommunications Technologies* vol. 29, no. 11 (2018), e3493.

[66] Starks, F., Plagemann, T. P., and Kristiansen, S. "DCEP-Sim: An Open Simulation Framework for Distributed CEP". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 180–190.

[67] Suhothayan, S. et al. "Siddhi: A second look at complex event processing architectures". In: *Proceedings of the 2011 ACM workshop on Gateway computing environments*. 2011, pp. 43–50.

[68] Tatbul, N. and Zdonik, S. "Window-aware load shedding for aggregation queries over data streams". In: *VLDB*. Vol. 6. 2006, pp. 799–810.

[69] Tatbul, N. et al. "Load shedding in a data stream manager". In: *Proceedings 2003 vldb conference*. Elsevier. 2003, pp. 309–320.

[70] Tucker, P. et al. *NEXMark—A Benchmark for Queries over Data Streams DRAFT*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[71] Uma, M. et al. "Formation of SQL from natural language query using NLP". In: *2019 International Conference on Computational Intelligence in Data Science (ICCIDS)*. IEEE. 2019, pp. 1–5.

[72] Varga, A. “OMNeT++”. In: *Modeling and tools for network simulation*. Springer, 2010, pp. 35–59.

[73] Volnes, E., Kristiansen, S., and Plagemann, T. “Improving the accuracy of timing in scalable WSN simulations with communication software execution models”. In: *Computer Networks* vol. 188 (2021), p. 107855.

[74] Volnes, E., Plagemann, T., and Goebel, V. “To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing”. In: *IEEE Communications Surveys & Tutorials (in revision)* (2023).

[75] Volnes, E. et al. “EXPOSE: Experimental Performance Evaluation of Stream Processing Engines Made Easy”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2020, pp. 18–34.

[76] Volnes, E. et al. “Lazy Migration: Just-In-Time Fragmented State Migration For Distributed Stream Processing”. In: 2023.

[77] Volnes, E. et al. “Modeling the Software Execution of CEP in DCEP-Sim”. In: *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*. 2019, pp. 244–247.

[78] Volnes, E. et al. “Travel light: state shedding for efficient operator migration”. In: *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. 2022, pp. 79–84.

[79] Zhang, X., Freschl, J. L., and Schopf, J. M. “A performance study of monitoring and information services for distributed systems”. In: *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*. IEEE. 2003, pp. 270–281.

[80] Zhao, B. “Complex event processing under constrained resources by state-based load shedding”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1699–1703.

[81] Zhao, B., Hung, N. Q. V., and Weidlich, M. “Load shedding for complex event processing: Input-based and state-based techniques”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1093–1104.

[82] Zhu, Y., Rundensteiner, E. A., and Heineman, G. T. “Dynamic plan migration for continuous queries over data streams”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. GSCC: 0000194. 2004, pp. 431–442.

Papers

Paper I

Improving the accuracy of timing in scalable WSN simulations with communication software execution models

Espen Volnes, Stein Kristiansen, Thomas Plagemann

Published in *Elsevier Computer Networks*, April 2021, volume 188, DOI: 10.1016/j.comnet.2021.107855.

Abstract

Emerging infrastructure-less network architectures such as WSNs consist of devices that perform packet processing in software. General-purpose network simulators do currently not possess models to simulate the intra-node delay of such devices. For example, a TelosB mote with TinyOS spends seven ms on processing packets with a size of 36 bytes and fifteen ms on packets of 124 bytes. The core problem addressed in this work is that simulation does not include such delays, and therefore, the results are inaccurate. To overcome this problem, we create a communication software execution model of TelosB that accounts for its temporal behavior to enable more accurate WSN simulations in the ns-3 simulator. A challenge is to create a tracing framework for TinyOS that can be used to accurately and reliably trace the behavior of a very resource-constrained system. By analyzing the software execution of TelosB running TinyOS in the emulator Cooja/MSPSim and on a real device, we discover discrepancies in the temporal behavior. The evaluation of our model shows that it is scalable and accurate; the simulated intra-OS delay deviates at most 5% from the intra-OS delay in the real mote. When we include the model in simulations, the forwarding capacity of a mote is decreased by 36%. The WSN community can use this model for more realistic simulations, and future WSN mote models will be easier to make with it as a foundation.

I.1 Introduction

Network simulators are typically used to test and evaluate networks and communication protocols. Users expect that these simulators produce accurate results concerning the functional protocol behavior and temporal behavior of executing the protocols. The temporal behavior can be divided into two groups: (1) transmission delay when packets are sent in between nodes and (2) intra-node delay when intermediate nodes forward packets. This paper explores intra-OS delay, the part of intra-node delay that is caused by the communication software execution (CSE). General-purpose network simulators differ from emulators and real testbeds in that they use high-level models that facilitate scalability, extensibility, and the analysis of models. These models accurately simulate transmission delay and queuing delay, but ignore software execution times on the assumption that they are comparably insignificant, which is the case for high-speed Internet routers that consist of efficient and specialized hardware. These models become significantly inaccurate, however, when applied in scenarios where that assumption does not hold, e.g., with highly resource-constrained forwarding devices in wireless sensor networks (WSNs).

A WSN is a network that consists of wireless sensor nodes (motes) that collect useful information for various applications [27, 46]. Examples include monitoring of the environment for hazardous events [21], health monitoring of patients [31], military applications [12], automated traffic control systems [23], underwater applications [26], agricultural applications [41], medical applications [38], motion tracking of people [50], and more. Many useful application areas for WSNs are yet to be deployed in real scenarios. Therefore, high-level simulation is an excellent first step in the direction of deploying a WSN application.

We show in this paper that using general-purpose network simulators to simulate forwarding nodes in WSNs should be accompanied by CSE models for simulating the delay caused by intermediate devices, because they add a significant delay to the total end-to-end delay. For example, Table I.1 describes the transmission delay (Tx delay) and intra-OS delay of an intermediate mote that forwards packets of various sizes. Intra-OS delay is the amount of time that the OS spends processing the packets. The table shows intra-OS delay readings from three different intermediate motes: a TelosB mote emulated with Cooja/MSPSim [14, 39, 51], a real TelosB mote and a simulated mote in the discrete-event network simulator ns-3 [42]. The intra-OS delay caused by a real TelosB mote that runs TinyOS (called $^{TinyOS}/_{TelosB}$) is substantial for two reasons. First, it is much higher than the transmission delay. Second, the intra-OS delay varies greatly with the packet size; the delay for the largest packet is more than twice as much as for the smallest packet. Since ns-3 does not simulate packet processing in the node, the intra-node delay is 0 ms for all packet sizes. Consequently, ignoring intra-OS delay results in inaccurate simulation of end-to-end delay in ns-3.

		Intra-OS delay		
Packet size	**Tx delay**	**Cooja/MSPSim**	**Real $^{TinyOS}/_{TelosB}$**	**ns-3**
36 bytes	1.3 ms	5.65 ms	7.1 ms	0 ms
57 bytes	2 ms	6.9 ms	9.2 ms	0 ms
76 bytes	2.5 ms	7.9 ms	10.7 ms	0 ms
120 bytes	4 ms	10 ms	14.8 ms	0 ms

Table I.1: Comparison of transmission (Tx) and intra-OS delays when sending packets of variable sizes.

The discrepancy in end-to-end delay between the end-to-end delay from a real $^{TinyOS}/_{TelosB}$ system versus an emulated mote in Cooja/MSPSim in Table I.1 demonstrates that accurate software execution simulation of a system is not trivial. MSPSim fails to simulate this crucial aspect even though it executes the same code as the real system does. We describe later that the reason for the inaccuracy is that transferring data between the radio transceiver and the main memory is inaccurate in MSPSim. Cooja is a network simulator designed for Contiki OS, but with the help of an emulator such as MSPSim [15] also runs the code of other OSs such as TinyOS [14, 39, 51]. Another issue with emulators such as MSPSim is that they are several orders of magnitude less scalable than general-purpose network simulators such as ns-3, as we show in Section I.5. Therefore, they might be more practical for simulating WSNs that might contain several thousand motes.

ns-3, along with all popular high-level network simulators such as OMNET++ and OPNET, do not simulate the intra-OS delay, which leads to a lower accuracy. Therefore, we seek to improve the accuracy of WSN simulations in ns-3. We do this by creating and integrating a CSE model of the WSN system $^{TinyOS}/_{TelosB}$ into ns-3. This model adds processing delay to the simulation by delaying the execution of existing models by the appropriate amount. Additionally, the model includes packet and service queues which can have limited capacity. An overflowed queue may result in packet loss. Depending on the load on the system and size of the packets, the delay can vary. As a result, the model may increase the accuracy of the simulated end-to-end delay, jitter, and packet loss. Our goal is that the model reflects the behavior of a real mote reasonably well. In heterogeneous simulations where nodes have different capacity, such a model might have a significant impact on applications, because the slower nodes might not keep up with the network traffic. Moreover, since the processing delay (shown in Table I.1) is so significant, a reasonably accurate CSE model of a WSN system will vastly increase the accuracy of the simulations when the alternative is no model at all.

Although there exist cycle accurate simulators for many hardware platforms that can accurately simulate the timing of software execution, they are generally too computationally complex for large-scale network simulation. In addition, they lack the high-level abstractions necessary to efficiently set up, run, and analyse large scale network simulations. In other words, there is a lack of simulation tools to close the gap between very accurate simulation of individual computers, and very scalable network simulation. An increasing amount of new and emerging network technologies, like sensor networks and Internet of Things (IoT), require such tools for accurate network simulation. Closing this gap requires a trade-off between accuracy on one hand, and scalability and high-level abstractions on the other. A central modeling challenge is the large heterogeneity of software and hardware employed in such networks, and the fact that network simulators fundamentally are not designed to be extended with such models.

We use a modeling methodology defined by Kristiansen et al. [29] that describes how to use traces from real systems to create reusable software execution models. They only have to be created once, and can afterward easily be used by others for simulations. This methodology has previously been used to model the CSE of hand-held mobile devices, i.e., Galaxy Nexus, Google Nexus One, and Nokia N900 [9, 29]. In this work, we model a much more resource-constrained device, i.e., a TelosB mote that runs the TinyOS operating system. $^{TinyOS}/_{TelosB}$ is a relevant system to model for several reasons. First, since it is a low-power system and has a CPU frequency of only 4MHz, the software execution delay is much more significant than in many other systems. Second, the straightforward single-threaded and single-core nature of the system indicates that the system can be accurately simulated using simpler models. Finally, WSN is a relevant type of network that is becoming ubiquitous with the advent of the IoT. The model is used to validate the methodology and may be used by others to improve the accuracy of their own ns-3 simulations. We hope the software execution model can be of value to IoT researchers by facilitating more realistic simulations.

The contributions of this work comprise:

1. an analysis of the temporal behavior of the communication software in a TelosB mote running TinyOS,

2. a comparison of the execution in a real system, an emulated mote, and ns-3,

3. a tracing framework that overcomes the challenges caused by the very low amount of memory on the mote,

4. the analysis and instrumentation of $^{TinyOS}/_{TelosB}$ to trace its temporal behavior,

5. a CSE model based on traces and its integration in ns-3, and

6. an evaluation of the CSE model.

This paper is an extended version of the conference paper in [48]. It includes a new experiment for evaluating the accuracy of the model. We extend Experiment 1 to additionally compare the execution times of the packet processing in an emulated mote using Cooja/MSPSim. Furthermore, we include an additional experiment for evaluating the accuracy of the model. Finally, everything is explained more in detail, including the motivations for the model and contributions such as the tracing framework, analysis of the communication software, model creation, and evaluation.

The remainder of the paper is structured as follows. In Section I.2, the related work is discussed. In Section I.3, the modeling methodology is presented. In Section I.4, the tracing framework is explained, we analyze the communication software of $^{TinyOS}/_{TelosB}$, and show how we derive the model from the traces. In Section I.5, we evaluate the model, and in Section I.6, we conclude the paper.

I.2 Related Work

Tracing the temporal behavior of TinyOS is discussed several times in the literature.

In [20], a lightweight tracing framework is presented that enables tracing of behavioral and timing events in TOSSIM. In [19], the same authors present an improved tracing framework called LIMOW for tracing real devices. With LIMOW, trace tuples are transmitted over the radio. We use serial communication because the radio is used as part of the application that we trace, and it is important for us that the tracing affects the application behavior minimally. Moreover, instrumentation with LIMOW is semi-automatic to minimize the changes that need to be done to a traced application, where we use a selective and coarse-grained instrumentation approach. Thus, tracing any location in the TinyOS code with our framework is easy to do. In [43], a generic and efficient logging framework called TinyLTS is presented as an alternative to ad-hoc tracing frameworks. The primary evaluation criteria used in the paper is flexibility and ease-of-use, whereas we need low tracing delay and storage overhead. Since our tracing framework is made only for the purpose of model creation and evaluation and must, therefore, incur minimal overhead, we instead use a specialized solution.

Several software execution delay modeling methodologies exist in the literature aside from the one we use, with varying degrees of accuracy, scalability, and modeling effort.

In [3], a methodology to create high-level models of network devices is presented, which requires little knowledge about the system's internals. The modeling methodology we use is different because it requires a deep understanding of the modeled system and thus has a higher modeling effort, but also results in more realistic models. In [35], an approach to model the intra-node delay of resource-constrained network nodes is presented and used in [4] to create models to simulate delay caused by networking software in the NAPI and NIC drivers in Linux. Their framework does not appear to support the simulation of branching in intra-node behavior based on the value of state variables, which ours does. Moreover, their framework appears to involve a manual creation of the model followed by calibration based on real measurements, whereas our methodology describes how to generate models automatically from traces. A simulation tool called RTNS is presented in [40] that can simulate communication and intra-node delays of WSN devices. It consists of an integration of the general-purpose network simulator ns-2 with the real-time OS simulator RTSim, both of which are discrete-event simulators. RTNS models can not be parameterized to reflect any given real device, while our methodology explains how to derive models from traces captured on real systems.

I.3 Modeling Methodology

We use the methodology defined by Kristiansen et al. [29], which enables modelers to make the above mentioned trade-off in a flexible manner. It is based on a set of general, high-level abstractions and events definitions that facilitate the modeling of a wide range of device-types, which can be extended when necessary for new types of software and hardware. While the modeling of an entire OS stack requires quite a lot of modeling effort, the models are inherently modular and each sub-model (called Software Execution Models, SEM) need only to be created once, after which they can be re-used and re-combined in arbitrary ways for subsequent simulations. The models are defined using high-level statements similar to those used in a programming language, facilitating low-effort studies of the impact of modifications, where re-parametrization and alternative compositions of the network stack impact results. The methodology defines a highly flexible mechanism to map SEM onto protocol models in existing network simulators to endow these protocol models with the timing behaviour of real software implementations. Crucially, this mechanism is network simulator agnostic, i.e., it works with any discrete event simulator provided with a well-defined extension.

The methodology defines a step-by-step approach to derive from a real device a trace-based CSE model that can be executed in a discrete-event simulator such as ns-3. Including a CSE model in an ns-3 simulation improves the accuracy of packet delay and packet loss. Packet delay gets simulated more accurately because we add intra-OS delays. As packets queue up inside the node due to intra-OS delay, packet queues may become full and cause packet loss.

We perform five steps to create the model. First, the software is instrumented to capture all relevant temporal behaviors, which requires a tracing framework for TinyOS and an analysis of the communication software of TinyOS. Second, the software is traced with different packet sizes to capture how the processing times change with the packet size. Third, we verify the correctness of the traces. Fourth, the traces are converted into services that together represent the temporal behavior of the modeled system, and are placed in a device file. Fifth, the accuracy of the model is assessed. The model's ability to simulate latency and packet loss is evaluated by comparing its latency and packet loss to a real mote at low and high packet rates, respectively. Therefore, models created with this methodology are designed to simulate both latency and packet loss accurately, even though we only collect traces at a low packet rate to create the model.

The device file is parsed to create Service Execution Models (SEMs). SEMs model individual portions of software execution. These can be executed inside a model of their execution environment in a discrete-event simulator. When we run the SEMs in the simulator, they introduce the delay based on the observed temporal behavior of the service during the tracing. It can still be the case that an SEM is inaccurate, which is why one should compare the behavior of the SEM with a real device to see if they have similar behavior when run. An SEM is either invoked by another SEM or from existing protocol models, which are called Functional Service Models (FSMs). FSMs first invoke SEMs, for instance, when a transceiver model receives a packet, and the SEM invokes another FSM through triggers defined in the SEMs.

Figure I.1 describes a step-by-step process on how the CSE model is executed in discrete-event simulators. It starts with the simulator setting up the execution environment that executes the CSE model (1). The execution environment then parses the device file and sets up all events that represent the CSE (2). Afterward, the simulator invokes FSMs (3) that invoke SEMs set up by the execution environment (4). As the simulation goes on, triggers in the SEMs invoke FSMs (5). SEMs trigger FSMs and FSMs trigger SEMs until the packet forwarding is finished. Hence, the execution of FSMs is delayed by alternating between executing FSMs in the protocol model and SEMs in the CSE model.

I.4 Design

In this section, the design of our methods and the model is described. First, the choice of simulator and system to model are explained. Second, the

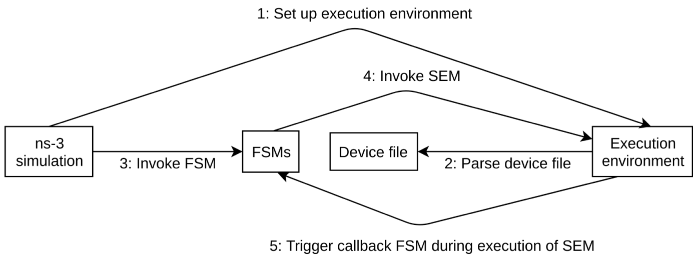


Figure I.1: ns-3 simulation with a CSE model

design of a mote tracing framework to capture the intra-OS delay and forwarding rate of a real TelosB mote is presented. Third, the analysis of the communication software of $^{TinyOS}/_{TelosB}$ is described. Finally, we explain how we create the model.

I.4.1 Choice of Simulator and Modeled System

The model we create is based on a TelosB mote that runs TinyOS. It is currently supported for the ns-3 network simulator. Below, we outline our reasons for choice of system and simulator.

I.4.1.1 OS

TinyOS is one out of many OSs that are used in WSNs. Other examples of OSs include MANTIS [5], Contiki OS [11], Nano-RK [16], MansOS [45], LiteOS [7] and RIOT-OS [2]. TinyOS is minimalistic and can be executed on resource-constrained devices. Furthermore, it is event-driven, has a RAM footprint of only 400 bytes, is single-threaded, and there is no scheduler preemption [32]. TinyOS does offer a library called TOSThreads that enables some high-level multi-threading features described in [28], but it is not used in this work. Additionally, heap memory does not exist, which means all data is stored statically with no temporary memory allocation. TinyOS 2.1.2.1 is the version of TinyOS that we model.

Although TinyOS has been around since 1999, it is still used in recent research literature [1, 37]. We argue that the specific operating system used is not as important for the temporal behavior of such motes as the choice of hardware is. That is because the components of the mote become the bottleneck, rather than the OS code itself. Examples are the speed of transferring data between the radio transceiver's queues and the main memory of the mote, and copying data from places within the main memory. These operations are primarily limited by the hardware components themselves. Therefore, most of the delay caused by processing is expected to be seen in other WSN OSs.

nesC is the programming language in which TinyOS and its applications are written [18]. It is a subset of the C programming language with some extra features, such as three new types of functions: events, commands, and tasks [17]. Events can be viewed as software/hardware interrupts or callback functions, commands as regular functions and tasks as deferred procedure calls.

I.4.1.2 Mote

The TelosB mote, also called Tmote Sky, is an open-source platform that can be used to measure light, humidity and temperature [34]. TelosB is often the mote used to run WSN OSs such as TinyOS, and is therefore a natural pick for TinyOS. TelosB is also a popular mote that can run resource-constrained WSN OSs such as MansOS, Mantis, Contiki OS, Nano-RK and TinyOS [13, 36]. RIOT-OS cannot run TelosB because it has a too high memory footprint [6].

I.4.1.3 Simulator

Since WSNs may contain thousands of nodes, we choose the scalable discrete-even network simulator ns-3. Other alternatives include OMNET++ [47] and Opnet [49]. As we shall see later, emulators such as MSPSim are several orders of magnitude less scalable than these network simulators, and real-world experiments are several orders of magnitude less scalable than emulators. One might ask why we do not choose TOSSIM (TinyOS SIMulator) [33], which is a well-known discrete-event simulator for simulating TinyOS applications. The reason is that our goal is not specifically to simulate TinyOS, but rather to enable WSN nodes to be simulated accurately that may coexist with other, more powerful, devices. TOSSIM cannot easily be used in conjunction with models of other systems, and so we use a general-purpose simulator.

I.4.2 Tracing Framework

Minimizing the memory usage and the time it takes to trace events (tracing delay) is essential, especially since TelosB only has 10kB of RAM and 4MHz CPU as compared to 256MB—1GB of RAM and 256MHz—1.2GHz CPU in the previously modeled devices (i.e., Google Nexus One, Nokia N900, and the Galaxy Nexus).

Accurate timestamps are required to measure the processing delays on a real device. When collecting traces to create the model, the packet rate is kept low so that the mote can process one packet at a time to avoid non-deterministic behavior affecting the delay measurements. If the number of tracepoints is relatively small, storing the trace tuples in RAM is possible.

When measuring the forwarding rate, on the other hand, the packet rate must be high, which makes it impossible to store tuples in RAM. However, we do not need to store accurate timestamps when measuring the forwarding rate because we only need to know the percentage of packets that are dropped. Since tracing the mote at high packet rates results in many trace tuples in a short period, we cannot store the tuples in RAM. Therefore, we developed two tracing methods: (1) batch tracing that is used to measure intra-OS delay where we store trace tuples with accurate timestamps in RAM until it fills up, and (2) continual tracing that is used to capture the forwarding rate at high packet rates without interruption.

The tracing framework is used in three steps. The first step is to capture the execution delays of the OS services in the communication software of $^{TinyOS}/_{TelosB}$. These traces are used to create the CSE model. The second step also involves capturing such execution delays, but at a less detailed level to evaluate the accuracy of the model for simulating intra-OS delay. The final step is to capture the forwarding rate when the mote processes packets at high packet rates to evaluate the accuracy of the model for simulating packet loss.

We minimize memory consumption of trace tuples by compressing them on the mote and later decompressing them to CSE events. These CSE events describe the communication software execution behavior of the traced system, based on concepts that are defined in the modeling methodology. After the CSE events are decompressed, they are used to create the CSE model. Significantly reducing the size of the 32-byte CSE events is not trivial because it requires knowledge about what data in CSE events is inferable. For instance, we can infer the Process ID (PID) of TinyOS because it is single-threaded. We have found that the only data needed for each trace tuple in TinyOS is a 1-byte tracepoint ID (0–255) and a 4-byte timestamp, as illustrated by Figure I.2a. Therefore, the CSE events in Figure I.2b can be compressed and traced as a single trace tuple, as displayed in Figure I.2c. This compression reduces the memory consumption of each trace tuple from 32 bytes to 5 bytes and reduces the number of tuples to trace since they can represent several CSE events. Note that if the memory consumption needs to be kept low in multithreaded and multicore systems, the tracepoint ID would need to be partitioned such that some bits can identify which thread and CPU core recorded the trace tuple. For instance, in a dual-core system where two threads run on each core to process packets, one bit in the tracepoint ID identifies the CPU core, one bit identifies the thread running, and the remaining six bits (0–63) identifies which exact tracepoint is executed.

Batch and continual tracing differ in memory consumption and the way trace data is transmitted to the connected PC. Conceptually, both methods perform tracing as illustrated in Figure I.3. Batch tracing involves storing tuples in RAM until the buffer is filled up with 700 trace tuples. When the

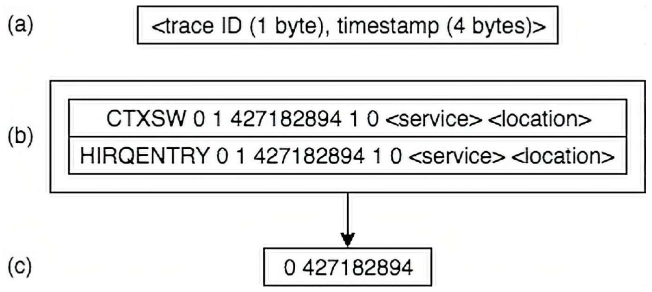

(a)
<trace ID (1 byte), timestamp (4 bytes)>
(b)
CTXSW 0 1 427182894 1 0 <service> <location>
HIRQENTRY 0 1 427182894 1 0 <service> <location>
(c)
0 427182894

Figure I.2: Trace tuple format (a) used to compress two CSE events (b) to a single trace tuple (c).

buffer is full, the tracing is paused, and the trace tuples are transmitted over serial communication to the PC connected by USB. Continual tracing involves transmitting tracepoint IDs immediately using serial communication to the listening PC, which listens for 1-byte tracepoint IDs and adds a timestamp to each tracepoint ID to complete the trace tuple. Both methods are efficient, but batch tracing has accurate timestamps and limited buffer size, whereas continual tracing has less accurate timestamps and can be used to trace TelosB for arbitrarily long periods.

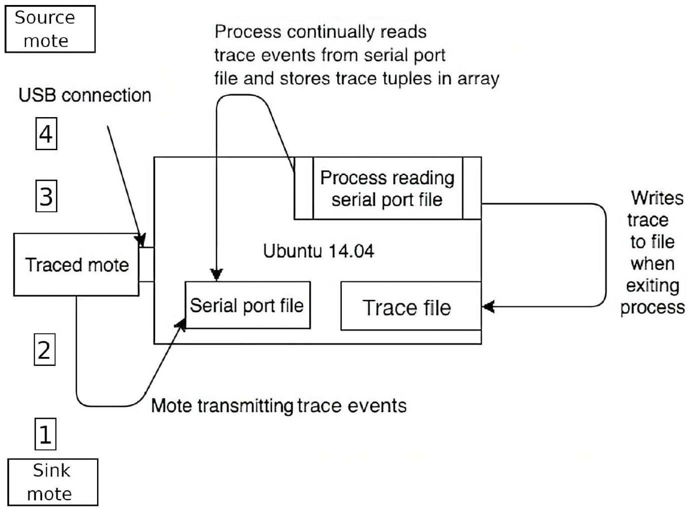

Source mote
USB connection
4
3
Traced mote
2
1
Sink mote
Process continually reads trace events from serial port file and stores trace tuples in array
Process reading serial port file
Ubuntu 14.04
Serial port file
Trace file
Writes trace to file when exiting process
Mote transmitting trace events

Figure I.3: Process of tracing a real mote.

The metrics used to evaluate the tracing framework are tracing delay and memory consumption. The tracing framework must be efficient and enable us to capture all the necessary information to be able to use the modeling methodology to create a sufficiently accurate CSE model.

Table I.2 sums up the difference between batch and continual tracing. With batch tracing, timestamp accuracy is high, the memory consumption is five bytes, and it takes 20 µs to trace an event. Furthermore, the number of tuples the mote can keep in memory is 700. With continual tracing, timestamp accuracy is low, the memory consumption is one byte, and it takes 40 µs to trace an event. Since the trace tuples are transmitted immediately using serial communication, there is no limit to how many events can be traced without interruption.

Tracing type	**Delay**	**# events**	**Timestamp accuracy**
Batch tracing	20 µs	700	High
Continual tracing	40 µs	No limit	Low

Table I.2: Difference between batch and continual tracing

I.4.3 Analysis and Instrumentation of TinyOS

Our analysis of the communication software execution in TinyOS shows that the functional part of the packet processing can be described as a receiving and sending activity with two queues, as illustrated in Figure I.4. In the illustrated instance, eight packets (shown in gray) are processed. Data received by the transceiver is placed in the 128-byte CC2420 receive (Rx) queue. The receiving activity writes the packet from the Rx queue to RAM, performs a routing table lookup, and places the packet into the IP layer packet queue (IPAQ) which has a capacity of three packets. The sending activity takes a packet from the IPAQ and forwards it. These processing activities can only handle one packet each at a time, most likely to keep the memory consumption low and the OS as simple as possible.

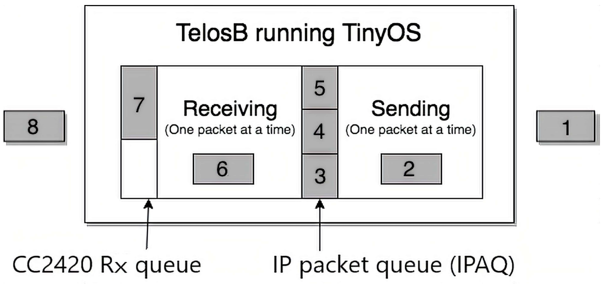


Figure I.4: Summary of the TinyOS packet forwarding process with packets being processed in gray.

I.4.3.1 Forwarding application

Figure I.5 contains an overview of the network stack used by the application. Mote A sends a packet using UDP for the transport layer, IPv6 and 6LoWPAN for its network-layer protocols, and IEEE 802.15.4 for MAC sublayer and PHY. The CSE model does not simulate 6LoWPAN fragmentation of packets, but it can be considered in future work. Combined, the headers in the packets sent by Mote A have a size of 36 bytes, and 38 bytes when adding the two CRC bytes. Further, when results are presented, figures distinguish between UDP payload size and packet size. A UDP payload size of zero bytes means a packet size of 36 bytes.

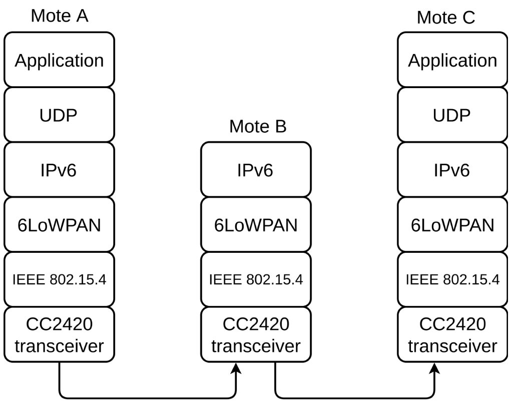


Figure I.5: Forwarding app network stack

The application uses IPv6 which by default requires devices to send ICMPv6 packets and the CC2420 driver requires acknowledgment of packets. Both ICMPv6 packets and acknowledgment packets are disabled because only a simple packet forwarding scenario is modeled. Clear Channel Assessment (CCA) is implemented in two ways on the mote. The first is a backoff timer in TinyOS that makes the mote wait a random amount of time before sending packets to avoid collisions. The second is a feature that the CC2420 radio chip implements to prevent packets from being sent if it senses that the medium is not clear [10]. The radio chip's CCA functionality is kept enabled, but the backoff timer is disabled because the random backoff time causes unwanted variation in intra-OS delay. These two modifications are made to simplify the forwarding process and do not affect our results.

I.4.3.2 Packet forwarding flow

When a packet is received, a hardware interrupt event is executed in the CC2420 driver. Only one packet can be written into RAM at a time, and awaiting packets get processed once the current one is finished with the receiving part. If the driver is ready to process a new packet, it starts writing the packet into memory. When the entire packet is written into RAM, the driver checks the last byte of it to see if the CRC check succeeded (a check performed by the CC2420 radio chip). If it fails, the mote drops the packet and starts reading the next one. If it succeeds, task `receiveDone_task` is posted to run later, which sends the packet to the upper layer protocols and hands it over so that the next packet can be processed. First, a duplication check drops previously received packets. Next, the packet is sent to the layer handling 6LoWPAN.

The 6LoWPAN adaptation layer decompresses the packet header before it sends the packet to the code handling packet forwarding. As the IP layer finds out that the packet is destined for another mote, it finds the route from Mote B to C. If less than three packets are waiting to be sent, the packet is placed in IPAQ. Otherwise, the packet is dropped. At this point, the receiving part ends, and now the next packet can be written to memory.

`sendTask` starts the sending part, and its job is to prepare a packet enqueued into the IPAQ to be sent to its destination, which is Mote C in this case. If no packet is currently being sent or awaiting acknowledgment, the packet will be processed and is sent to the lower layers. The sending part of the CC2420 driver writes the packet to the transmit (Tx) queue of CC2420. When the packet is written to the Tx queue, a hardware interrupt is raised, and the transceiver sends the packet. Once the transceiver has finished sending the packet, the packet is removed from IPAQ, and the next packet can be transmitted or received.

I.4.3.3 Instrumentation

We instrument the communication software of $^{TinyOS}/_{TelosB}$ to capture the temporal behavior of the software execution. When creating and evaluating the model, we use two different instrumentation configurations called model- and evaluation-centered instrumentation. In the model-centered instrumentation, the trace captures which services are called, how long they execute, and the events within services and their time of occurrence. In TinyOS, the latter events occur when packets are enqueued into the IPAQ and when the radio chip is ready to attempt to transmit a packet. The tracepoints described in Table I.3 are used to collect the traces for model creation. In the evaluation-centered instrumentation, the focus is on measuring the intra-OS delay and packet forwarding rate. By instrumenting the five places in Table I.4, we capture when packets are received, sent, and dropped.

According to Cooja/MSPSim, only 98 microseconds of the packet processing is not caused by software processing on the MCU. Those 98 microseconds occur between the radio transceiver receiving the packet, and the OS being notified that a packet has been received. In that time, the

Function	What is captured
task-scheduler	Before running task
task-scheduler	After running task
All six hardware interrupts	Start of interrupt
All six hardware interrupts	End of interrupt
`receiveDone_task`	Enqueuing packet into IPAQ
`attemptSend`	Transceiver attempting to send packet

Table I.3: Description of the model-centered tracepoints

Function	What is captured
`readDone #1` (HIRQ-2)	Receiving new packet
`readDone #3` (HIRQ-4)	Drop packet due to failed CRC check
`receiveDone_task`	Drop packet due to full IPAQ
`writeDone` (HIRQ-5)	Transceiver attempting to send packet

Table I.4: Description of the evaluation-centered tracepoints

transceiver does some pre-processing like adding RSSI, CRC, and FCF data to the received packet [25].

I.4.4 Model Creation

In this section, the creation of the CSE model is described. Traces gathered from running Mote B at a low packet rate are used to create the model. We develop a protocol model for TelosB in ns-3 and place the CSE model in a device file that is parsed by an execution environment to add the temporal behavior of the CSE of $^{TinyOS}/_{TelosB}$ to the simulation. After having collected the traces, the remaining steps of the modeling methodology to create the CSE model are:

- Analyze the traces to determine if they are accurate.
- Decompress the traces to CSE events.
- Convert the CSE events into signatures.
- Create a TelosB model in ns-3 that uses the CSE model to simulate the packet forwarding.
- Create a simulation program in ns-3 that uses the TelosB model to forward packets.

A trace might be inaccurate and contain inconsistencies that must be found through analyzing the trace. The trace used to create the CSE model is collected at a low packet rate, and so the same events are expected in the forwarding application every time a packet is processed. Even if the packet rate is high, the code in TinyOS spends about the same amount of time each time it is executed. As such, the processing times should be similar when tracing. If they are not, it might be because of an error. Occasionally, a timestamp in a tuple is incorrect, e.g., lower than that of trace tuples preceding it. To identify this kind of error and related tracing errors, we use a tool that outputs the maximum, average, median and minimum time differences between two tracepoint IDs, and all the various time differences sorted by the number of occurrences.

Figure I.6 illustrates the relationship between trace tuples and the software execution model. Each trace tuple represents one or more CSE events. The trace generated by Mote B is decompressed to CSE events by a script that maps tracepoint IDs to CSE events. The script parses the trace file tuple by tuple and injects the timestamp for each CSE event. An example of this is in Listing I.1. The resulting list of CSE events is written to an output file, which is used as input to the automatic analysis script.

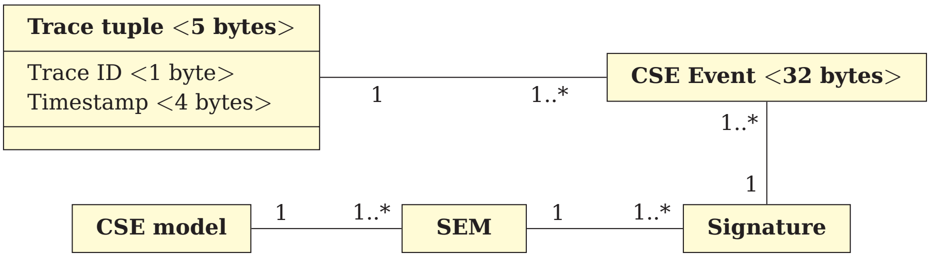


Figure I.6: Relationship between trace tuples and the software execution model

```
                    321451 5
                    321455 2
                    321462 1
                       |
                 Decompresses to
                       ↓
SRVENTRY 0 0 321451 0 0 0 service 0
QUEUECOND 0 0 321455 0 0 0 service notempty
PKTQUEUE 0 0 321455 0 0 0 service 0
SRVEXIT 0 0 321462 0 0 0 service 0
```

Listing I.1: Example of decompressing trace to CSE events

The analysis script takes the CSE events as input and generates the signatures for the CSE model as output, as displayed in Listing I.2 in the appendix. The signatures themselves are similar to function definitions and can invoke each other as long as the invoked signature is defined above the caller, as in the C programming language. One file is created for each signature, and each SEM consists of one or more signatures, depending on the presence of queue or state conditions. A queue condition such as the one in Listing I.2 causes two different signatures to be defined. The execution environment parses the signature to create a single SEM with branching points in the places where the conditionals are found. The upper signature is called if `packet_queue` is not empty and the lower one if it is. When all signatures are generated, the most significant part of the CSE model is created.

I.4.4.1 Device File

Our $^{TinyOS}/_{TelosB}$ device file includes settings for, e.g., a byte queue, a packet queue, one thread, one service queue, a CPU and several callback triggers. The byte queue represents the Rx queue of the radio chip, which has a limit of 128 bytes. The packet queue is analogous to the IPAQ, has a capacity of three packets, and follows a tail-dropping policy. As mentioned above, the CPU runs at 4MHz. The thread models the scheduler in TinyOS and executes services from the service queue. The Tx byte queue of the radio chip does not need to be modeled because it never overflows in our forwarding application. All these settings are manually configured and are based on the analysis of the CSE of TinyOS in Section I.4.3.

I.4.4.2 Packet Processing Flow

The final task in creating the CSE model is to write the TelosB FSMs and integrate them with the other models. In our case, the CC2420 transceiver model is connected to the CSE model. When a packet to be forwarded is received by the transceiver, the CSE model delays the forwarding and passes it to the transceiver model when it has finished processing it.

Packets can be dropped in the CSE model for three reasons. First, due to Rx queue overflow, which causes the radio chip to stop receiving incoming packets until the queue has been flushed. Second, after having written the entire packet into memory and the packet has a bad CRC checksum. Third, when the IPAQ is full. The first two are unlikely to occur in our case when the CC2420 CCA feature is enabled. Usually, packet loss occurs due to IPAQ overflow.

I.5 Evaluation

Five experiments are conducted to evaluate the model. They are used to assess the accuracy, scalability, and the significance of including the execution times in simulations. Conceptually, the testbed for the experiments comprises three motes in which packets are generated by Mote A and sent to

Mote C via Mote B. Practically, we only need Mote A and Mote B to perform the experiments since acknowledgments are disabled, which means that Mote C never sends any information to Mote B.

Figure I.7 shows the models and metrics used to evaluate the accuracy and scalability of the CSE model in the context of the packet forwarding process. In an ns-3 simulation where transmission of packets is only performed with the CC2420 transceiver model, the full end-to-end delay only consists of the transmission delay and around 98 µs of preprocessing delay caused by the transceiver.

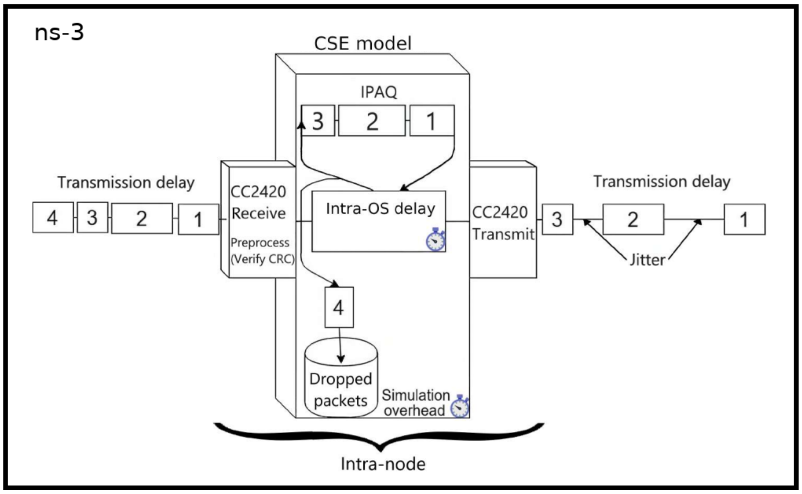


Figure I.7: Metrics and models used to evaluate the CSE model and their place in the packet forwarding process.

Table I.5 summarizes the parameterization of experiments with real motes (R) and simulation experiments (S). Experiments 1–3 regard the accuracy of the model, and therefore, include a real-world experiment and its simulation with the CSE model. Experiments 4–5 are simulated only where Experiment 4 deals with the scalability and Experiment 5 with the impact of including or excluding the CSE model in a simulation. The packet size we use for Experiments 1–3 varies between the minimum and maximum packet sizes, and for Experiments 4–5, we only use the largest packet size. A slow data rate means that each packet is processed one at a time with no contention. In Experiment 3, we vary between 40 and 150 pps for different packet sizes. In Experiment 5, we vary between 63 and 101 kbps.

For all simulations, we use the network simulator ns-3 [22]. While ns-3 offers many models for running the experiments, we make use of a small subset of these. Aside from Experiment 4, all experiments have one instance of the CSE model for the forwarding node. The source and destination nodes do not run the CSE model. The topology for these experiments can be seen

Exp	Goal	Type	Pkt size	Data-rate	Metric	Varies
1	Accuracy	R+S	36–124 b	Slow	Intra-OS delay	Packet size
2	Accuracy	R+S	36–124 b	Slow	Intra-OS delay	IPAQ fill-level
3	Accuracy	R+S	36–124 b	40–150 pps	Throughput	Packet rate, packet size
4	Scalability	S	124 b	Slow	Execution time, RAM usage	# packets, simulation time, # nodes
5	Impact	S	124 b	63–101 kbps	Forwarding capacity difference	Data-rate

Table I.5: Experiment and simulation parameters (R+S includes both a real-world experiment and its simulation, and S only includes simulation).

in Figure I.8. The CSE model is connected to the ns-3 Node model through ns-3's object aggregation feature. Packets that are sent are instances of the ns-3 Packet model. For transmission of packets, Experiments 1–4 use a simplified CC2420 transceiver model we developed that offers the same temporal behavior as a real CC2420 radio, but without the lower-level details of communication. In Experiment 5, we use the CC2420 transceiver model from [24].

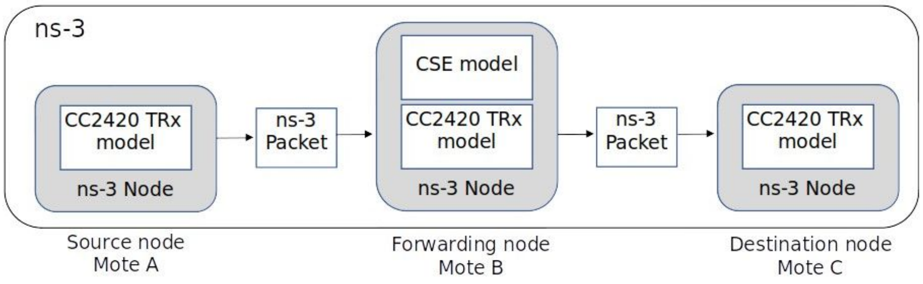


Figure I.8: Models used in the simulations.

I.5.1 Accuracy

In Experiments 1–3, we run real-world experiments to capture the behavior of a real mote performing IP forwarding. Thereafter, we run simulations with the CSE model to observe how accurate it is. In the real-world experiments, Mote A runs a forwarding application TinyOS. It sends packets with IP destination address of Mote C and frame (link-layer) address destination of Mote B. Mote B receives them and has a manually inserted route to Mote C.

In Experiment 1, we assess the accuracy of the model in simulating intra-OS delay by comparing the intra-OS delay of the simulation model and a real mote at low packet rate. We perform batch tracing of the temporal behavior of Mote B while it performs packet forwarding, which means it keeps trace tuples in main memory. In the real-world experiment, Mote B forwards 256 packets with 12 different sizes. Afterward, the same packet sequence is forwarded by the simulation model.

Figure I.9, Figure I.10a, and Figure I.10b contain the results from Experiment 1 from different perspectives. Figure I.9 illustrates the results of Experiment 1, where the intra-OS delays of a real mote and the CSE model are compared for varying packet sizes. Since TinyOS timers are "binary" with respect to one second [8] and the intra-OS delays in TinyOS are measured with a microsecond timer, there are 1048 milliseconds per second in the figure. The lines overlap nearly perfectly, and the simulated intra-OS delay deviates at most 5% from the intra-OS delay in the real mote, which shows that the model is highly accurate. The results only show one run because the intra-OS delay is a deterministic function of the packet size.

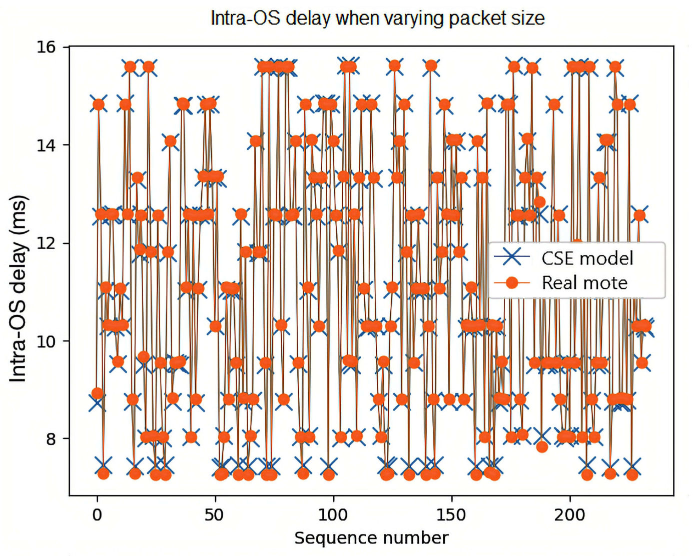
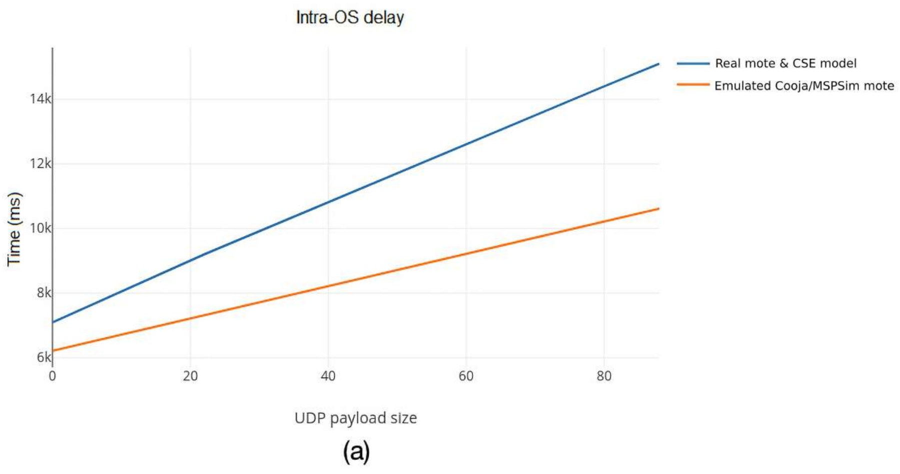


Figure I.9: Intra-OS delay comparison between real mote and ns-3 with the CSE model at 40 pps.

Figure I.10a shows that the intra-OS delay (y-axis) increases linearly with the packet size (x-axis) for a real mote, the CSE model, and an emulated Cooja/MSPSim mote. Note that the intra-OS delay is not "binary" as in Figure I.9. The same line represents the real mote and CSE model because their data is indistinguishable. Furthermore, one can see how different an emulated Cooja/MSPSim node behaves compared to a real mote and the CSE model. Cooja/MSPSim starts with 14% and ends with 30% less intra-OS delay than the real mote and CSE model. These results illustrate how inaccurate the temporal behavior of Cooja/MSPSim is compared to the CSE model and the real mote.

Figure I.10b shows more results from Experiment 1, where we measure how the end-to-end delay increases with the packet size. The packet size (x-axis) affects the total end-to-end delay (y-axis) for a real mote, the CSE model, the CC2420 transceiver model, and an emulated Cooja/MSPSim node. All lines increase linearly with the packet size. The transmission delay includes the time it takes for Mote A to send packets to Mote B, and for Mote B to send packets to Mote C. These results demonstrate how significant the processing delay is compared to the transmission delay in $^{TinyOS}/_{TelosB}$.



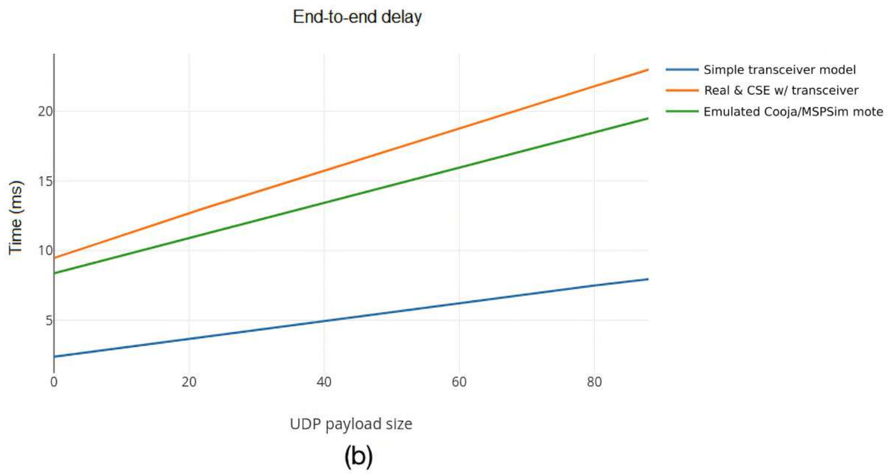


Figure I.10: Intra-OS and end-to-end delay depending on the packet size

In Experiment 2, we investigate how much the fill-level (the number of packets in a queue) of the IPAQ affects intra-OS delay. The queue has a maximum capacity of three packets, and packets are enqueued into it a bit more than halfway through the forwarding process. Variations in intra-OS delay in TinyOS are due to queuing and packet size. We enqueue the same packet three times in the IPAQ instead of once to measure the queuing time while other packets are processed before it. Additionally, we observe how long the receiving and sending parts of the CSE are. The same tracepoints as in Experiment 1 are used to measure the intra-OS delay, and it is calculated in the same way. Additionally, the ns-3 simulation reproduces the real-world experiment in the same way as in Experiment 1 by using the trace from the real mote, except that now each packet is enqueued into the IPAQ three times. The intra-OS delay of the CSE model and real mote can be compared for the different IPAQ fill-levels 0–2 to assess the accuracy of the CSE model.

Figure I.11 contains the results of Experiment 2; it compares the variation in intra-OS delay for different IPAQ fill-levels for a real mote with that of the simulated CSE model. Same as in Figure I.9, there are 1048 milliseconds per second because of the "binary" timers in TinyOS [8]. The intermediate Mote B receives 25 packets of variable sizes at a low packet rate, each with a sequence number (x-axis). One can distinguish between the delay before enqueuing the packet into the IPAQ (receiving part) and afterward (sending part). The intra-OS delay is almost identical in the real mote and the CSE model, the same as in Experiment 1.

The intra-OS delay for a packet that is placed in an empty IPAQ only consists of the processing delay caused by executing the receiving and sending parts once; the same receiving and sending parts as can be seen in Figure I.4. When a packet has to wait in the IPAQ for one packet, the intra-OS delay is the same as when the IPAQ is empty plus the processing delay of sending the enqueued packet. When the IPAQ fill-level is two, the intra-OS delay consists of the processing delay when the IPAQ is empty plus the intra-OS delay of sending two enqueued packets.

In Experiment 3, we assess the accuracy of the model in simulating packet loss by comparing the forwarding rate of the model to a real mote when forwarding packets at high packet rates. Since high packet rates can yield erratic behavior, we make the mote forward many more packets than in Experiment 1. More specifically, the mote forwards 256 packets for several combinations of packet size and packet rate, and each point on the line denotes the percentage of successfully forwarded packets among these 256. In the ns-3 simulation, we reproduce the real-world experiment by using the same packet generation logic. Unlike Experiment 1 and 2, Mote B in Experiment 3 performs continual tracing, mentioned in Section I.4.2, which means it transmits tracepoint IDs to the host PC each time a trace event occurs. The reason is that continual tracing is much faster than batch tracing since the tracing mote does not add a timestamp (see Table I.2). The timestamps added to the trace tuples by the host PC are not sufficiently

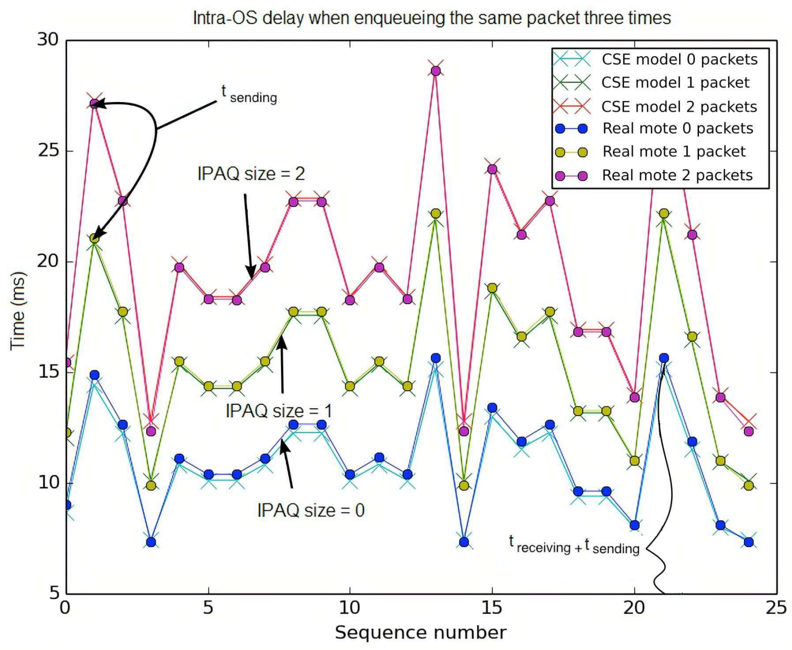


Figure I.11: Variation in the IPAQ fill-level affecting the intra-OS delay.

accurate to be used to reproduce the experiment in ns-3. Therefore, the experiment is not trace-driven, which means that the CSE model will not send packets at the exact same times in the ns-3 simulation. Consequently, the results are not expected to be identical for the model and real mote.

Figures I.12a (UDP payload size 24 bytes) and I.12b (UDP payload size 80 bytes) illustrate the results from Experiment 3. Figure I.12a shows that packets start to drop heavily around 120 packets per second and in Figure I.12b at around 80 packets per second. The reason for the big difference is that larger packets require more processing than smaller ones, and therefore fewer packets can be processed per second before packets start to drop. The figure includes only results for the case that Mote A is not in saturation mode and sends at least 90% of the target packet rate. The results that are included are adjusted for the actual packet rate. While the curves for the model and mote do not entirely overlap, the same trends are seen in both of them, which means the model can approximate the behavior of CSE reasonably well even when the device is saturated.

Experiment 1, 2, and 3 demonstrate that the model can simulate intra-OS delay and packet loss with high accuracy. One of the reasons for the high accuracy is that TelosB's MCU MSP430 has no memory cache, which means that memory access times are deterministic. Another reason for the

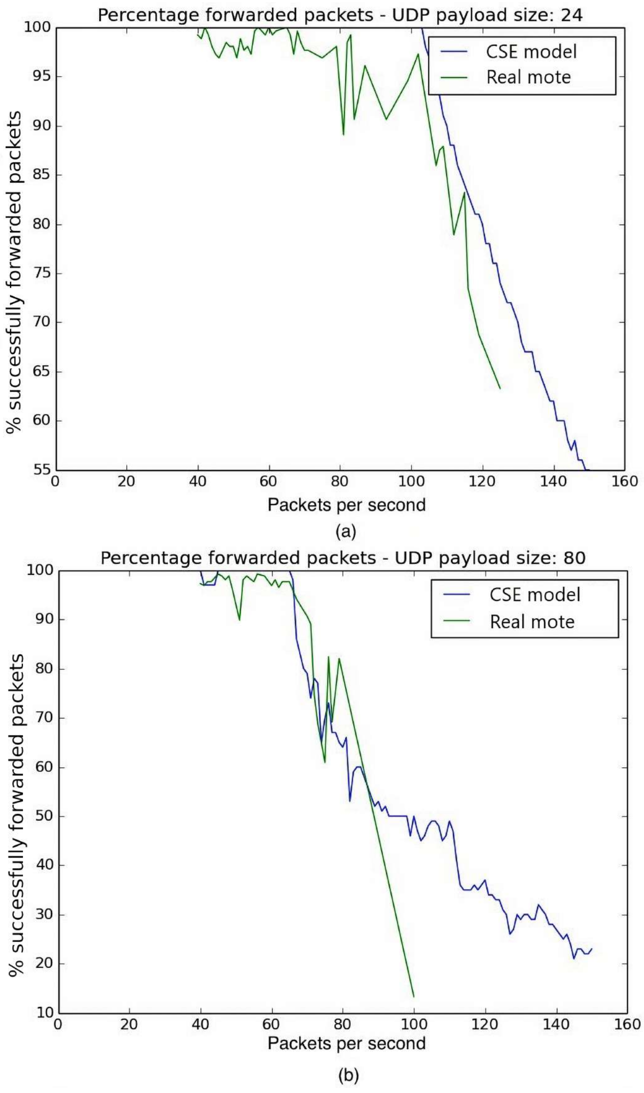


Figure I.12: The forwarding rate when sending packets at various packet rates.

high accuracy is that TinyOS is a much simpler operating system than the previously modeled mobile variations of Linux. Please note, these results clearly demonstrate that the model which is based on low packet rate traces simulates rather precise when packet loss starts under high packet rates.

Furthermore, Experiment 3 confirms that if Mote A sends packets at a high rate to C via B, Mote B eventually drops packets because of a full IPAQ. Note that only Mote A sends packets to B, which means that a mote that needs to send many packets must deliberately restrict transmission rate to avoid packet loss in the intermediate mote. That does not happen when using unmodified TinyOS because it includes an initial backoff feature that causes a random waiting time before sending each packet.

I.5.2 Scalability of Model

In Experiment 4, the scalability of the model is assessed by measuring the time required to complete a simulation run with various parameter settings in four runs[1]. First, the number of packets that a single node forwards is varied. This parameter is the most important and likely to have the greatest effect on the simulation execution time. Second, the number of simulated seconds a single node is idle is varied. Varying this parameter should not result in a significant increase in simulation execution time. The final parameter is the number of nodes that are included in the simulation. For this parameter, we perform two runs: one where the nodes process a single packet each, and another where the nodes spend ten million simulated seconds being idle. That way, we can uncover any added simulation time that is caused merely by increasing the number of nodes. The experiment is conducted on a PC with 4.2GHz quad-core CPU (Intel i7-7700k) running Ubuntu 16.04 LTS.

Table I.6 lists the results from Experiment 4 regarding the simulation execution time. The scalability experiment shows that one node can forward 60,000 packets in five seconds, one node can be idle for 5 billion simulated seconds in less than half a second, 100,000 nodes can each be idle for ten million simulated seconds in 134 seconds, and 10,000 nodes can forward a packet each in 45.5 seconds. Additionally, we measured the memory consumption of installing 100,000 nodes and found that they consume 11.6 GB RAM. Moreover, Table I.7 compares the simulation execution times of Cooja/MSPSim and ns-3, where the data from the Cooja/MSPSim mote is extrapolated from a 263-second long simulation. It shows that ns-3 is up to six orders of magnitude faster than Cooja/MSPSim when idle nodes are simulated.

WSNs can contain up to thousands of nodes, and since we can simulate thousands of nodes processing thousands of packets in a matter of seconds

[1]In all runs, ns-3 is compiled with the g++/gcc optimization flag "-O3", which reduces the simulation execution time substantially.

# nodes	# packets	Simulated seconds	Execution time
1	60,000	Until completion	5.00 sec
1	0	$5 * 10^9$	0.28 sec
100,000	0	10^7	134.00 sec
10,000	1 per node	Until completion	45.50 sec

Table I.6: Results from the simulations in Experiment 4.

Real mote	Cooja/MSPSim	CSE model
1+e6 sec	263 sec	22.5 ms
1+e7 sec	2630 sec	24.5 ms
1+e8 sec	26300 sec	39.8 ms
1+e9 sec	263000 sec	178 ms
6+e9 sec	1578000 sec	1 sec

Table I.7: Comparison of execution time of a real mote, an emulated Cooja/MSPSim mote (extrapolated data) and the CSE model in ns-3.

using commodity hardware with 16GB RAM, we can conclude that our models are sufficiently scalable for WSN simulations. Compared with the evaluation of the previous models in [9, 29], the scalability remains similar.

I.5.3 Impact of Model

In Experiment 5, the impact of the model is assessed by replicating an experiment conducted by Igel et al. in [24] and comparing the results with and without the CSE model. The topology is the same as in Experiment 1 and 2, and the goodput is measured. In this case, goodput means the rate at which packets are received by Mote C, where the bit-rate includes packet headers. Four runs are executed with 124-byte packets with or without the CSE model. The run without the CSE model is the same as the original experiment. In Run 1, Mote A sends packets at the highest packet rate without packet loss when the CSE model is excluded. In Run 2, the data rate is the same as in Run 1, but the CSE model is included. In Run 3, Mote A sends packets at the lowest packet rate with packet loss when the CSE model is included, and Run 4 has the highest packet rate without packet loss with the CSE model [2].

Table I.8 contains the results of Experiment 5. By adding the CSE model, the data rate must be decreased from 100kbps (Run 1) to 64kbps (Run 4) to prevent any data from being lost. That means our model starts to drop

[2]These thresholds are determined by measuring the forwarding rate in ns-3 with and without the CSE model at many different packet rates

packets at 65kbps (packet size 124 bytes), while just the CC2420 transceiver model starts dropping packets at 101 kbps. This 36% decrease in forwarding capacity demonstrates that our model enables significantly more accurate simulations.

Run	CSE model included	Data-rate	% forwarded
1	No	100kbps	100%
2	Yes	100kbps	56%
3	Yes	65kbps	87%
4	Yes	64kbps	100%

Table I.8: Goodput experiment with and without CSE model.

I.5.4 Discussion

We argue that CSE models are needed to simulate the temporal behavior of WSN devices accurately in ns-3, with Table I.1 as motivation. The results in Figure I.10b support this claim, i.e., the end-to-end delay with the real mote is significantly larger than indicated by the CC2420 transceiver model alone. Furthermore, Experiment 5 replicates a goodput experiment initially conducted in [24], and the goodput reduces by 36% when including the CSE model. The significant reduction shows that the intra-OS delay is non-negligible, demonstrating the need for CSE models.

This CSE model is more impactful than the previous models because the modeled device is more computationally constrained. TelosB is a single-core device and TinyOS a single-threaded OS. Furthermore, TelosB does not use a cache for faster memory access, the CPU speed of TelosB is constant at 4MHz, and TinyOS does not use optimization techniques in the instrumented drivers that can cause variable temporal behavior either. Previously modeled devices [9, 30] are much more complex.

The accuracy of the CSE model is high. There is almost no difference between the CSE model and real mote in the tested scenarios. Several aspects of the device are modeled and assessed in experiments: (1) intra-OS delay, (2) variation in delay due to packet size, (3) variation in delay due to IPAQ fill-level, and (4) packet loss.

A discovery we made is that the processing stages during the IP-forwarding that take the most time to execute are three stages: (1) when transferring packets from the radio transceiver to main memory, (2) when copying the packet from the driver of the radio transceiver to the IPAQ, and (3) when copying the packet from the IPAQ to the radio transceiver. The delay is mostly bound by hardware limitations. For Points 1 and 3, the delay is mostly bound by the time it takes to copy data back and forth between the radio transceiver. For Point 2, the delay is mostly bound by the read and write speeds of the main memory. What we can deduce from this analysis is that TinyOS is not responsible for the high intra-OS delay or the variance in it. If TelosB runs Contiki OS, it will most likely also exhibit the same behavior. Therefore, the model can most likely be used to represent other WSN OSs than TinyOS.

During emulation with Cooja/MSPSim, we discovered some issues: two bugs and inaccurate execution times of two processing stages. The first bug is that the microsecond clock on TelosB displays approximately four times larger value than it should be when compared to both the millisecond clock and the Cooja simulation clock, seen in Figure I.13. The microsecond time does neither correspond with the millisecond time nor the Cooja/MSPSim simulation time. The second bug is that the CC2420 CCA feature does not work. When the channel is not clear, packets are still sent, which causes collisions. The inaccurate execution times occur in the processing stages when transferring packets between the radio transceiver and main memory. As mentioned above, these processing stages are the ones that takes the longest time to execute on a real mote. These processing stages are too short compared to a real mote, which is the main reason why the Cooja/MSPSim intra-OS delays in Figure I.10 are different from the real mote and CSE model. Only two bytes are transferred in each transaction, and so a slightly inaccurate processing time estimation will result in a large gap as the packet size increases.

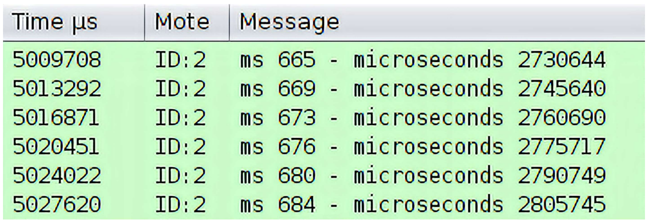

```
Time µs    Mote   Message
5009708    ID:2   ms 665 - microseconds 2730644
5013292    ID:2   ms 669 - microseconds 2745640
5016871    ID:2   ms 673 - microseconds 2760690
5020451    ID:2   ms 676 - microseconds 2775717
5024022    ID:2   ms 680 - microseconds 2790749
5027620    ID:2   ms 684 - microseconds 2805745
```

Figure I.13: Timer issue when emulating in Cooja/MSPSim.

The mote does not get saturated with the initial CCA backoff enabled because the motes have sufficient time to process incoming packets while waiting to transmit packets. Figure I.14 shows the intra-OS delay for 124 bytes packets that are sent from Mote A to C as fast as possible with initial backoff enabled. As a result of the backoff, no packets are dropped, and the intra-OS delay is much higher than it would if the initial backoff were disabled. If the backoff were disabled, this exact scenario would result in packet loss. The need for this backoff to prevent packet loss demonstrates how resource-constrained TelosB is, and also motivates the need for a software execution model.

The tracing framework described in Section I.4.2 is used to capture the CSE behavior of $^{TinyOS}/_{TelosB}$ and proves to be flexible, efficient with low

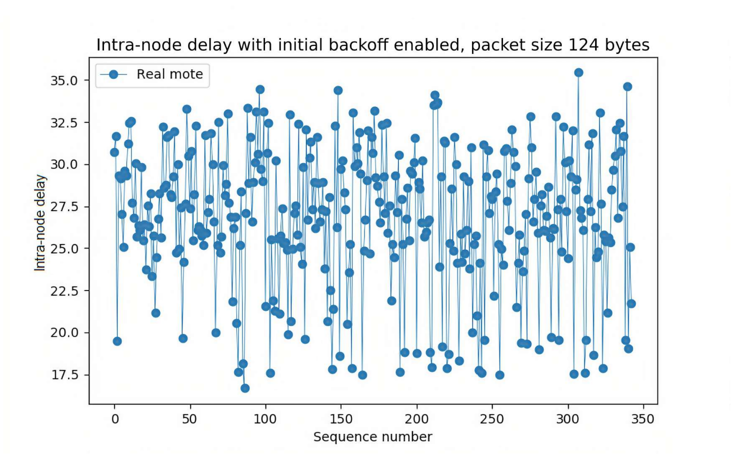


Figure I.14: 124-byte packets sent as fast as possible with initial backoff in TinyOS enabled, and no packets are dropped.

tracing delay and memory consumption for each trace tuple. It is flexible because of the two methods of tracing, namely, batch and continual tracing. The former enables saving trace tuples with accurate timestamps in RAM until it is full and the latter enables tracing for arbitrarily long periods, albeit with less accurate timestamps. If someone attempts to model another WSN device in the future, the same design can be reused.

As opposed to previously created CSE models, the TelosB mote requires compression of traces because it only has 10kB RAM in total with approximately 3.5kB available when the OS and forwarding application are installed. After the traces are collected, they are decompressed to CSE events. Two benefits of this way of tracing are that (1) we can focus entirely on tracing efficiently without worrying about the meaning of the traces and (2) a single trace tuple can be converted to several CSE events, which means even less tracing delay and memory consumption. Additionally, the compressed trace can be analyzed more easily since it is simpler, which can reveal errors that occur during execution.

I.6 Conclusion

With this paper, we create a CSE model of $^{TinyOS}/_{TelosB}$ that adds realistic temporal behavior to simulations of packet forwarding in ns-3. First, a tracing framework is created and used to capture the temporal behavior of the CSE of $^{TinyOS}/_{TelosB}$ running on the real device. The framework is used both for creating and evaluating the model. Our results show that our model is accurate; the simulated intra-OS delay deviates at most 5% from the intra-OS delay in the real mote. The model is also scalable; we can simulate

IP-forwarding of 60,000 packets in five seconds, and 100,000 nodes can be simulated with 11.6 GB RAM. Finally, the model has a significant impact on the simulation results; including the $^{TinyOS}/_{TelosB}$ CSE model requires a reduction of 36% in data rate to prevent packet loss. The code developed and used for this work is available in [44].

An important insight we gained with this work is that most of the delay in the system and variance in delay stems from hardware limitations. As a result, we expect to observe a similar temporal behavior in other OSs than TinyOS. This work indirectly functions as an evaluation of the modeling methodology's application on WSN systems. A benefit of creating models of different types of systems is that these models can be used as a template for other systems in the same domain. For instance, the $^{TinyOS}/_{TelosB}$ CSE model can be used to create a future $^{Contiki}/_{TelosB}$ CSE model and $^{TinyOS}/_{MIKAz}$ CSE model.

For future work, we consider modeling other operating systems such as Contiki OS that runs on TelosB, to see how Contiki OS differs in execution. Moreover, we are interested in modeling different hardware such as MIKAz that runs TinyOS, to see how TelosB and MIKAz differ. With these new models, we can also use the model from this paper to assess the accuracy of retargeting models to represent different hardware or operating systems. If the accuracy of the retargeted models is sufficiently high, further models might be created without going through the entire modeling process again. We also have plans to automate the instrumentation step of the methodology, which would simplify the modeling process significantly. Additionally, we are looking into using the same modeling methodology to model the software execution of other types of applications, such as complex event processing.

Acknowledgments

We thank Dr. Anuscha Igel and Prof. Dr. Reinhard Gotzhein for kindly providing us with code for the CC2420 transceiver model from [24].

I.7 Appendix

References

[1] Amjad, M. et al. "TinyOS-new trends, comparative views, and supported sensing applications: A review". In: *IEEE Sensors Journal* vol. 16, no. 9 (2016), pp. 2865–2889.

[2] Baccelli, E. et al. "RIOT OS: Towards an OS for the Internet of Things". In: *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Apr. 2013, pp. 79–80.

```
SRVENTRY 0 0 100000 0 0 0 service x
QUEUECOND 0 0 100004 packet_queue packet_queue 0 service notempty
PKTQUEUE 0 0 100004 packet_queue packet_queue 0 service x
SRVEXIT 0 0 100011 0 0 0 service x

SRVENTRY 0 0 200000 0 0 0 service x
QUEUECOND 0 0 200004 packet_queue packet_queue 0 service empty
SRVEXIT 0 0 200011 0 0 0 service x
                 |
             Converts to
                 ↓
SIGSTART
NAME service
PEU cpu
RESOURCES cycles normal
FRACTION 100%   1940 1940

0 START
x PROCESS        4 0
x QUEUECOND packet_queue packet_queue notempty
x DEQUEUE PKTQUEUE 0 packet_queue
x PROCESS 7 0
0 STOP

SIGEND

SIGSTART
NAME service
PEU cpu
RESOURCES cycles normal
FRACTION 100%   1940 1940

0 START
x PROCESS        4 0
x QUEUECOND packet_queue packet_queue empty
x PROCESS 7 0
0 STOP

SIGEND
```

Listing I.2: CSE events to signatures conversion

[3] Begin, T. et al. “High-level approach to modeling of observed system behavior”. In: *Performance Evaluation* vol. 67, no. 5 (2010), pp. 386–405.

[4] Beifuß, A. et al. “A study of networking software induced latency”. In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. Mar. 2015, pp. 1–8.

[5] Bhatti, S. et al. “MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms”. In: *Mob. Netw. Appl.* vol. 10, no. 4 (Aug. 2005), pp. 563–579.

[6] Bloessl, B. et al. “Low-cost interferer detection and classification using TelosB sensor motes”. In: *ACM SIGMOBILE Mobile Computing and Communications Review* vol. 16, no. 4 (2013), pp. 34–37.

[7] Cao, Q. et al. “The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks”. In: *2008 International*

Conference on Information Processing in Sensor Networks (ipsn 2008). Apr. 2008, pp. 233–244.

[8] Cory Sharp, Martin Turon, David Gay. “TEP 102: Timers”. In: ().

[9] Dale, Ø. “Modeling, analysis, and simulation of communication software execution on multicore devices”. MA thesis. 2016.

[10] David Moss, Jonathan Hui, Philip Levis and Jung Il Choi. “TEP 126: CC2420 Radio Stack”. In: (2007).

[11] Dunkels, A., Gronvall, B., and Voigt, T. “Contiki - a lightweight and flexible operating system for tiny networked sensors”. In: *29th Annual IEEE International Conference on Local Computer Networks*. Nov. 2004, pp. 455–462.

[12] Đurišić, M. P. et al. “A survey of military applications of wireless sensor networks”. In: *2012 Mediterranean conference on embedded computing (MECO)*. IEEE. 2012, pp. 196–199.

[13] Elsts, A. and Selavo, L. “Improving the Usability of Wireless Sensor Network Operating Systems.” In: *FedCSIS Position Papers*. 2013, pp. 89–94.

[14] Eriksson, J. et al. “COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks”. In: *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*. Simutools ’09. Rome, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, 27:1–27:7.

[15] Eriksson, J. et al. “Mspsim - an extensible simulator for msp430-equipped sensor boards.” In: *:* 2007.

[16] Eswaran, A., Rowe, A., and Rajkumar, R. “Nano-RK: an energy-aware resource-centric RTOS for sensor networks”. In: *26th IEEE International Real-Time Systems Symposium (RTSS’05)*. Dec. 2005, 10 pp.–265.

[17] Gay, D. et al. *nesC 1.1 language reference manual*. 2003.

[18] Gay, D. et al. “The nesC Language: A Holistic Approach to Networked Embedded Systems”. In: *SIGPLAN Not.* vol. 38, no. 5 (May 2003), pp. 1–11.

[19] Hammad, M. and Cook, J. “Lightweight Deployable Software Monitoring for Sensor Networks”. In: *2009 Proceedings of 18th International Conference on Computer Communications and Networks*. Aug. 2009, pp. 1–6.

[20] Hammad, M. and Cook, J. “Lightweight Monitoring of Sensor Software”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC ’09. Honolulu, Hawaii: ACM, Jan. 2009, pp. 2180–2185.

[21] Hart, J. K. and Martinez, K. “Environmental sensor networks: A revolution in the earth system science?” In: *Earth-Science Reviews* vol. 78, no. 3-4 (2006), pp. 177–191.

[22] Henderson, T. R. et al. “Network simulations with the ns-3 simulator”. In: *SIGCOMM demonstration* vol. 14, no. 14 (2008), p. 527.

[23] Hussian, R. et al. “WSN applications: Automated intelligent traffic control system using sensors”. In: *Int. J. Soft Comput. Eng* vol. 3, no. 3 (2013), pp. 77–81.

[24] Igel, A. and Gotzhein, R. “A CC2420 Transceiver Simulation Module for ns-3 and its Integration into the FERAL Simulator Framework”. In: ().

[25] Instruments, T. *2.4 GHz IEEE 802.15.4 / ZigBee-ready RF transceiver*. 2006.

[26] Jouhari, M. et al. “Underwater wireless sensor networks: A survey on enabling technologies, localization protocols, and internet of underwater things”. In: *IEEE Access* vol. 7 (2019), pp. 96879–96899.

[27] Kandris, D. et al. “Applications of wireless sensor networks: an up-to-date survey”. In: *Applied System Innovation* vol. 3, no. 1 (2020), p. 14.

[28] Klues, K. et al. “TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS”. In: ACM, Nov. 2009.

[29] Kristiansen, S., Plagemann, T., and Goebel, V. “A methodology to model the execution of communication software for accurate network simulation”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* vol. 26, no. 1 (July 2015), pp. 1–31.

[30] Kristiansen, S., Plagemann, T., and Goebel, V. “Extending network simulators with communication software execution models”. In: *2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)*. IEEE. 2013, pp. 1–10.

[31] Kristiansen, S. et al. “Event modeling and processing to simplify real-time analysis of physiological signals”. In: (2017).

[32] Levis, P. et al. “TinyOS: An operating system for sensor networks”. In: *Ambient intelligence* (2005). Ed. by Weber, W., Rabaey, J. M., and Aarts, E., pp. 115–148.

[33] Levis, P. et al. “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications”. In: *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*. SenSys ’03. Los Angeles, California, USA: ACM, 2003, pp. 126–137.

[34] Memsic Inc. “TelosB Datasheet”. In: ().

[35] Meyer, T. et al. “Extensible and realistic modeling of resource contention in resource-constrained nodes”. In: *2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. July 2013, pp. 1–9.

[36] Modium, D. K. and Kolla, K. P. “ENHANCING REAL TIME CAPABILITIES OF NANO-RK FOR TELOSB PLATFORM”. In: ().

[37] Musaddiq, A. et al. “A survey on resource management in IoT operating systems”. In: *IEEE Access* vol. 6 (2018), pp. 8459–8482.

[38] Neves, P., Fonsec, J., and Rodrigue, J. “Simulation tools for wireless sensor networks in medicine: a comparative study”. In: *Int. Jt. Conf.* Vol. 2. Jan. 2007, pp. 111–114.

[39] Osterlind, F. et al. “Cross-Level Sensor Network Simulation with COOJA”. In: *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*. Nov. 2006, pp. 641–648.

[40] Pagano, P. et al. “Simulating Real-Time Aspects of Wireless Sensor Networks”. In: *EURASIP Journal on Wireless Communications and Networking* vol. 2010, no. 1 (Dec. 2009), p. 107946.

[41] Pierce, F. and Elliott, T. “Regional and on-farm wireless sensor networks for agricultural systems in Eastern Washington”. In: *Computers and electronics in agriculture* vol. 61, no. 1 (2008), pp. 32–43.

[42] Riley, G. F. and Henderson, T. R. “The ns-3 network simulator”. In: *Modeling and tools for network simulation*. Ed. by Wehrle, K., Güneş, M., and Gross, J. Berlin, Heidelberg: Springer, 2010, pp. 15–34.

[43] Sauter, R. et al. “TinyLTS: Efficient network-wide Logging and Tracing System for TinyOS”. In: *2011 Proceedings IEEE INFOCOM*. Apr. 2011, pp. 2033–2041.

[44] Stein Kristiansen, Espen Volnes. *CSE Modeling Framework*. 2018.

[45] Strazdins, G., Elsts, A., and Selavo, L. “MansOS: easy to use, portable and resource efficient operating system for networked embedded devices”. In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. 2010, pp. 427–428.

[46] Thakur, D. et al. “Applicability of wireless sensor networks in precision agriculture: A review”. In: *Wireless Personal Communications* vol. 107, no. 1 (2019), pp. 471–512.

[47] Varga, A. “OMNeT++”. In: *Modeling and tools for network simulation*. Springer, 2010, pp. 35–59.

[48] Volnes, E., Kristiansen, S., and Plagemann, T. P. “Communication Software Execution Model of a WSN Device for More Accurate Simulation in Ns-3”. In: *Proceedings of the 11th International Conference on Computer Modeling and Simulation*. ICCMS 2019. North Rockhampton, QLD, Australia: ACM, 2019, pp. 184–189.

[49] Xinjie Chang. “Network simulations with OPNET”. In: *WSC’99. 1999 Winter Simulation Conference Proceedings. ’Simulation - A Bridge to the Future’ (Cat. No.99CH37038)*. Vol. 1. Dec. 1999, 307–314 vol.1.

[50] Zhou, H. and Hu, H. “Human motion tracking for rehabilitation—A survey”. In: *Biomedical signal processing and control* vol. 3, no. 1 (2008), pp. 1–18.

[51] Österlind, F. *A Sensor Network Simulator for the Contiki OS*. Tech. rep. 2006:05. SICS, 2006, p. 40.

Paper II

Modeling the Software Execution of CEP in DCEP-Sim

Espen Volnes, Stein Kristiansen, Thomas Plagemann, Vera Goebel, Morten Lindeberg

Published in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, June 2019, DOI: 10.1145/3328905.3332508.

Abstract

DCEP-Sim facilitates simulation of distributed CEP where the latency and bandwidth limitations in the network are well reflected, but it currently lacks models to simulate the temporal behavior of event processing. In this demonstration, we use a modeling methodology to model the software execution of a CEP system called T-Rex. We instrument and trace T-Rex to parameterize a software execution model that is integrated into DCEP-Sim. Furthermore, we use this instance instance of DCEP-Sim to run simulations and see how significant the processing delay introduced by the model is compared to the transmission delay.

II.1 Introduction

Simulation of distributed complex event processing (DCEP) is an efficient way of testing different topologies, queries, and workloads. It can be used to estimate the requirements for the DCEP infrastructure and is a much cheaper alternative to real-world experiments. Moreover, reproducing results is much simpler with simulation. For these reasons, DCEP-Sim is created [7]. DCEP-Sim extends the network simulator ns-3, which ensures that it simulates the network communication using validated models.

An essential aspect of DCEP that DCEP-Sim has been unable to simulate is the delay caused by software execution. The time that a node spends processing an event differs depending on the type of event, the number of queries that are deployed onto a node, query parameters, and complexity. Without this feature, the simulation will assume that all nodes in the network have the same processing capacity. Moreover, DCEP-Sim will overestimate

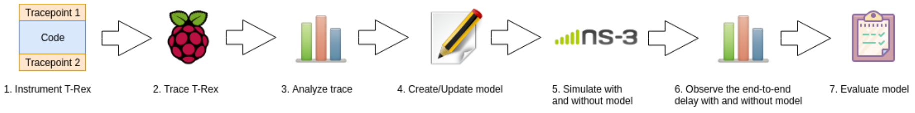


Figure II.1: Demonstration overview.

the throughput and underestimate the total end-to-end delay caused by network and event processing.

The inability to distinguish between weak and powerful nodes may limit the realism of the operator placement mechanism. Some operator placement algorithms will place partial queries near the data source to reduce energy consumption [6]. However, a class of networks called fog networks typically places central processing in powerful data centers and weak data source nodes at the edge, e.g., Raspberry Pi or sensor motes [8]. In this case, complex queries might need to be placed further from the source where the processing capacity is sufficient. With a software execution model, the operator placement algorithm will be able to take into account both the processing capacity of nodes and proximity to data sources.

Developing simulation models for the execution time of DCEP is a challenging task, and in this demonstration, we show a case that it is possible to create such models, how this is done, and how simulation results can be improved. In particular, we demonstrate the creation and simulation of a software execution model of the T-Rex CEP system [1] running on a Raspberry Pi 3 B. To perform the demonstration steps in Figure II.1, we apply the software execution modeling methodology from [5] on CEP. This methodology has previously only been used to model processing by IP-forwarding. Therefore, the first obstacle is to determine the most significant factors that affect the temporal behavior of CEP systems.

II.2 Modeling CEP

In this section, we briefly explain the modeling methodology to enable the reader to understand better what we show in the demonstration. Additionally, we describe the considerations to take when applying the methodology to the T-Rex CEP system.

II.2.1 Modeling Methodology

The modeling methodology in [5] has previously been used to model IP-forwarding of four different mobile systems. These models have been made for the network simulator ns-3, which DCEP-Sim extends. Moreover, the models are trace-based, which means that we execute the software in multiple contexts while tracing it, and use that data to generate the models.

Technically, the models are simulator agnostic, but they require an execution environment specific to a simulator that is based on the principles of the methodology.

Multiple factors affect the execution time of software:

- type of hardware,
- speed of CPU,
- operating system used,
- programming language of the software, and
- execution behavior.

All these factors motivate the need for a software execution model that is tailored to combinations of hardware, operating system, and software. On the other hand, if two different operating systems or sets of hardware have been demonstrated to have similar run-time behavior, a model can be reconfigured slightly to represent a new system, thus avoiding the full modeling process.

These models introduce processing delay to discrete event simulators by delaying (in simulated time) the execution of specific functions in the models. This delay is introduced by replacing a regular function call with a callback that is executed after we have introduced processing delay.

The methodology describes the following steps to create a model:

- instrument the software to model,
- trace the temporal behavior of the system in multiple contexts to get an accurate representation of the system,
- investigate the traces to make sure that the system has been appropriately instrumented,
- create the model based on the traces, and
- evaluate the model.

II.2.2 Use Case: T-Rex

We model the software execution of T-Rex, a prototype pub/sub CEP server system. T-Rex is chosen for two reasons. First, Cugola and Margara made T-Rex [1] and defined CEP concepts [2] on which DCEP-Sim is based. Second, the system is more straightforward than modern stream-processing engines in terms of programming language, system architecture and properties like reliability, fault tolerance and quality of service. In addition to CEP, these systems handle everything from batch operations to machine learning [4].

Our model supports queries that expect sequences of two events, in addition to different types of constraints on these events. Value constraints are used to filter events based on attribute values. A window constraint in T-Rex means that an event that follows another must arrive no later than a certain number of milliseconds. These operators are chosen because they represent the most common use of CEP: to detect patterns in event sequences.

The performance of CEP systems is significantly affected by the temporal behavior of software during its execution, including the factors listed in Section II.2.1. Based on our experiments and the evaluation of T-Rex in [1], the factors that may affect the execution behavior of T-Rex notably include:

- size of the event packet,
- size of the input queue of T-Rex,
- number and type of constraints in queries,
- number of queries,
- number of expected events in a query,
- type of operators used in the queries,
- number of utilized CPU cores, and
- number of subscribers of a given event.

These factors must be considered separately by instrumenting and tracing to capture the processing delay of T-Rex. To which extent they affect the temporal behavior can only be found through run-time analysis.

II.3 Event processing in T-Rex

T-Rex utilizes all CPU cores on the server to maximize CPU utilization during query processing. By default, the program spawns two threads for each CPU core: we name them the event and query threads. The event thread handles incoming events and the query thread performs query processing according to a set of queries for which it is responsible. This means if the number of events processed at the same time equals or exceeds the number of utilized cores, T-Rex utilizes all the event threads. If the number of deployed queries is higher or equal to the number of utilized cores, T-Rex utilizes all the query threads.

Figure II.2 illustrates how these threads work together to process incoming events. First, an incoming packet is handled by the event thread in the first available CPU core. The thread checks if the event matches any constraint in any of the standing queries. If a constraint is fulfilled, the

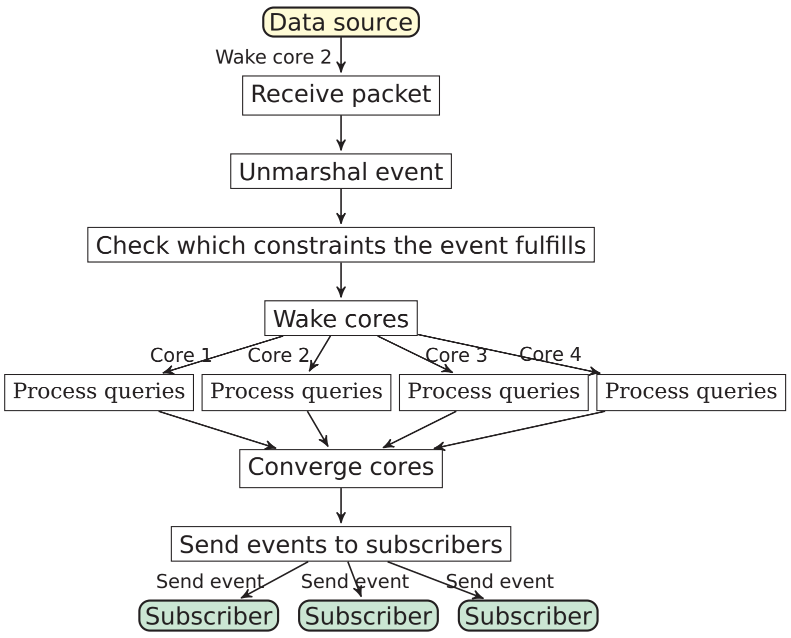


Figure II.2: Flow diagram of event processing in T-Rex.

thread writes it into the shared memory of the thread that handles the said query. After constraint checking, the incoming event is handled by the query threads in all the CPU cores. The query threads loop through their queries and check if the event proceeds on any existing event sequence. If an event sequence matches a query, the complex event is produced, and potential cleanup is done to free memory. Finally, the event thread sends the incoming atomic event and any potential complex events that were generated to the subscribers of these events.

The time that T-Rex spends processing these tasks depends on factors like those mentioned in Section II.2.2. For instance, each constraint takes a certain amount of time to process. Also, the processing time depends on whether the type of the constraint is a string, integer or float. If more than one core is utilized, multiple packets can be received and processed at the same time, and queries can be processed in parallel. The simulation of two models with different configurations yields different temporal behaviors. Consequently, the effect of the variables on the temporal behavior of T-Rex must be understood before they can be modeled.

II.4 Demonstration

We demonstrate in this section the steps of creating and simulating a CEP model of a Raspberry Pi 3 B that executes T-Rex. We lock the Pi's CPU frequency to 600 MHz (by default varies between 600 and 1200 MHz) because our model can currently not vary its CPU frequency during run-time realistically. The demonstration steps are illustrated in Figure II.1. In a test scenario, the Pi is traced while executing an instrumented version of T-Rex. These traces are then analyzed to verify that T-Rex is adequately instrumented. Afterward, the model is created or updated based on these traces. Finally, we simulate DCEP-Sim with the software execution model to observe how the transmission delay compares to the processing delay that is introduced by the model.

II.4.1 Model Creation

We trace an instrumented version of T-Rex with tracepoints at the entry and exit points of the incoming events. A test scenario in T-Rex that utilizes one CPU core is executed locally instead of distributively to avoid network overhead and increase replicability. Listing II.1 contains Query 1, a two-state fire detection query that is written in the Tesla event specification language [3]. We deploy it to T-Rex, and send the two expected events in order one at a time. The query expects a humidity event with humidity less than 25%, followed by a temperature event with a temperature above $45\,^{\circ}\mathrm{C}$. The temperature event must arrive not more than 5 seconds after the humidity event. The 'Consuming' clause dictates that these events cannot trigger more than one complex event. Afterward, all other event sequences that include these events are deleted.

The histogram in Figure II.3 shows a bimodal processing delay distribution (two peaks) after tracing T-Rex with only two tracepoints in the code placed before and after the processing of individual events. Events #1 and #2 are sent in order to trigger the complex event. We can see how the processing delay for an event differs depending on whether the event is number one (Event #1) in the query or the event triggering the complex event (Event #2). Keep in mind that Event #1 and #2 might as well be identical; the reason why Event #2 takes longer time to process than Event #1 is that the final event prompts T-Rex to create a complex event, which causes more processing to occur. According to the modeling methodology, this instrumentation is incomplete until we reach a unimodal distribution (one peak) [5].

Unimodal processing delay distributions can be reached by instrumenting all the conditional branching statements and loops that we know have a differing temporal behavior depending on workload and queries. When we analyze and plot the processing delay distributions for each processing

Listing II.1: Query to detect fire (Query 1).

```
Assign 10 => Temp, 11 => Humidity, 12 => Fire
Define Fire(area:string, temp: float)
From Temp(value > 45) and
last Humidity([string] area = Temp.area, perc < 25)
within 5000 from Temp
Where area := Temp.area, temp := Temp.value
Consuming Temp, Humidity
```

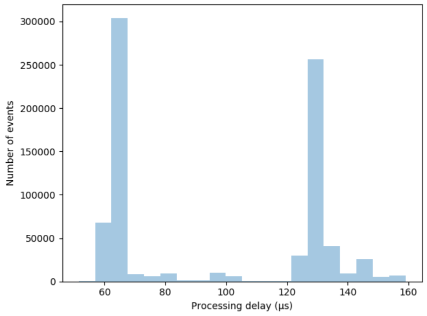


Figure II.3: Distribution of processing delays in T-Rex when processing two types of events with a two-state query deployed.

stage with proper instrumentation, each plot should have one peak. Even though the distributions are unimodal, a system can still be improperly instrumented if the set of workloads in which the system is traced is too low. Modeling a system is, therefore, an iterative process that requires us to revisit instrumentation, tracing, updating model and simulation multiple times. For the rest of the demonstration, we use an already sufficiently instrumented T-Rex.

When T-Rex is traced, the model is created or updated. Creating the model includes specifying the number of threads, CPU cores, and the model's integration with the simulator. Listing II.2 illustrates the signature of a partial model within the software execution model that represents a traced service in T-Rex. It has a processing stage that takes 5000 CPU cycles to execute. When the simulator executes this processing stage, it delays the simulation by `5000/f` seconds, where `f` signifies the CPU frequency of the model. This delay can be updated based on traces.

II.4.2 DCEP-Sim

We run DCEP-Sim with the software execution model to see its impact on temporal behavior. The topology for the simulation consists of four nodes, as illustrated by the screenshot in Figure II.4 from the simulation animation tool NetAnim from ns-3. Node A_{source} and Node B_{source} generate events every 10

Listing II.2: Signature of partial model.

```
SIGSTART
NAME process_received_packet
PEU cpu
RESOURCES cycles normal
FRACTION 100% 1 1

0 START
x PROCESS 5000 0
x CALL check–constraints
0 STOP

SIGEND
```

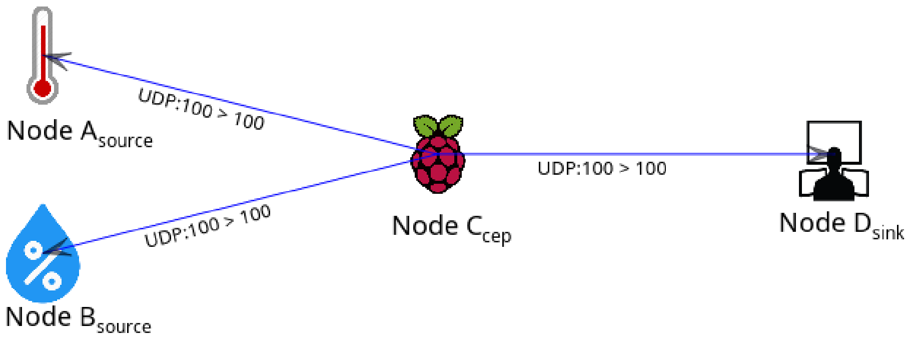


Figure II.4: Netanim screenshot from the simulation playback.

ms and send them to the third node C$_{cep}$, which acts as the T-Rex processing node. All complex events that are generated by C$_{cep}$ are transmitted to the fourth node D$_{sink}$. We deploy the DCEP-Sim equivalent of Query 1 to Node C$_{cep}$. Both data sources generate sensor data: Node A$_{source}$ generates data from a thermometer and Node B$_{source}$ from a humidity sensor. The events from Node A$_{source}$ and Node B$_{source}$ correspond to Event #2 and #1 in Query 1, respectively. The simulation is executed five times with the same events being generated but with one to five queries deployed to Node C$_{cep}$.

We measure and compare two types of delays: the transmission from the data source nodes to Node C$_{cep}$ and processing delays from Node C$_{cep}$. Ns-3 already provides transmission delay through its network models. Our model provides processing delay to the simulation. The measurements from DCEP-Sim are illustrated in Figure II.5. The x-axis shows the number of deployed queries and the y-axis denotes the delay in µs. The figure has three lines. The first lines illustrates the time taken to transmit both Event #1 and #2 being from one node to another. The second and third lines show the amount of time Node C$_{cep}$ spends processing Event #1 and #2 with the model, respectively. We can see how the delays for the second and third lines correspond to the first and second peaks of Figure II.3 when x equals 1, which means the model simulates the processing delay of Event #1, Event

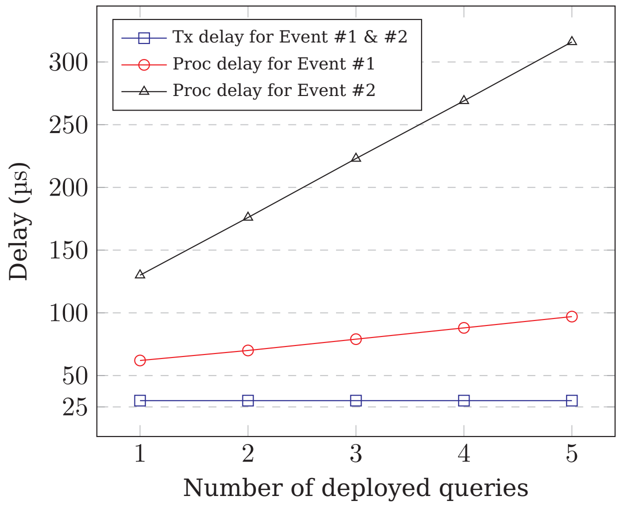


Figure II.5: Transmission and processing delay in DCEP-Sim (54 Mbit/s data-rate)

#2, and a single instance of Query 1 accurately. These results highlight how significant the processing delay can be compared to the transmission delay.

The processing delay introduced by the model is several times higher than the transmission delay from ns-3. Also, the processing delay varies significantly depending on the number of queries deployed to Node C_{cep} and whether the event is the first in a sequence or the one triggering a complex event. This results in a variable and higher end-to-end delay and may result in a decrease of the maximum throughput. If the model utilized more than one CPU core, the difference in delay when increasing the number of queries would be smaller. The processing delay becomes particularly important to simulate if a simulated network is a heterogeneous mix of devices with different processing capabilities. Then we can include multiple different software execution models depending on the processing capability of the nodes. Consequently, weak nodes can get overloaded and drop packets because of the added processing delay, which is an effect that would otherwise be lost without the model.

References

[1] Cugola, G. and Margara, A. “Complex event processing with T-REX”. In: *Journal of Systems and Software* vol. 85, no. 8 (2012), pp. 1709–1728.

[2] Cugola, G. and Margara, A. “Processing Flows of Information: From Data Stream to Complex Event Processing”. In: *ACM Comput. Surv.* vol. 44, no. 3 (June 2012), 15:1–15:62.

[3] Cugola, G. and Margara, A. “TESLA: A Formally Defined Event Specification Language”. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS ’10. Cambridge, United Kingdom: ACM, 2010, pp. 50–61.

[4] Dayarathna, M. and Perera, S. “Recent Advancements in Event Processing”. In: *ACM Comput. Surv.* vol. 51, no. 2 (Feb. 2018), 33:1–33:36.

[5] Kristiansen, S., Plagemann, T., and Goebel, V. “A methodology to model the execution of communication software for accurate network simulation”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* vol. 26, no. 1 (July 2015), pp. 1–31.

[6] Starks, F. and Plagemann, T. P. “Operator placement for efficient distributed complex event processing in MANETs”. In: *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. Oct. 2015, pp. 83–90.

[7] Starks, F., Plagemann, T. P., and Kristiansen, S. “DCEP-Sim: An Open Simulation Framework for Distributed CEP”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS ’17. Barcelona, Spain: ACM, 2017, pp. 180–190.

[8] Yi, S., Li, C., and Li, Q. “A Survey of Fog Computing: Concepts, Applications and Issues”. In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata ’15. Hangzhou, China: ACM, 2015, pp. 37–42.

Paper III

EXPOSE: Experimental Performance Evaluation of Stream Processing Engines Made Easy

Espen Volnes, Thomas Plagemann, Vera Goebel, Stein Kristiansen

Published in *Technology Conference on Performance Evaluation and Benchmarking*, August 2020, DOI: 10.1007/978-3-030-84924-5_2.

Abstract

Experimental performance evaluation of stream processing engines (SPE) can be a great challenge. Aiming to make fair comparisons of different SPEs raises this bar even higher. One important reason for this challenge is the fact that these systems often use concepts that require expert knowledge for each SPE. To address this issue, we present Expose, a distributed performance evaluation framework for SPEs that enables a user through a declarative approach to specify experiments and conduct them on multiple SPEs in a fair way and with low effort. Experimenters with few technical skills can define and execute distributed experiments that can easily be replicated. We demonstrate Expose by defining a set of experiments based on the existing NEXMark benchmark and conduct a performance evaluation of Flink, Beam with the Flink runner, Siddhi, T-Rex, and Esper, on powerful and resource-constrained hardware.

III.1 Introduction

Distributed Stream Processing Engines (SPE) process tuples at potentially high rates, perform filter operations, aggregation operations, and derive higher-level events. These are performed without the need to store the tuples persistently. Such systems are becoming more and more relevant for an increasing number of applications, ranging from classical financial services to sensor-based smart-* systems. As such, stream processing becomes highly relevant for fog networks where Big Data processing shifts from only occurring at resourceful data centers to access points closer to data sources. That means data is processed on resource-constrained systems close to the client and in resourceful data centers. This diversity in terms of applications and processing environments implies that the age of ”one size fits all” has ended for SPEs [23]; and naturally leads to a large number of different SPEs, all with their particular strength and weaknesses. These include SPEs aimed towards data centers for high concurrency, throughput, and integration possibilities, like Apache Flink, Esper Enterprise Edition, and Apache Storm, as well as SPEs suitable for relatively resource-constrained systems, like the library versions of Siddhi and Esper. Furthermore, there are different types of SPEs, including Data Stream Management Systems, Complex Event Processing (CEP) systems, and Big Data processing systems.

With many SPEs available, each with its own qualities, choosing the correct one for a given application is a challenge. Benchmarks might help, but there are limitations in the existing SPE benchmarks, and a given benchmark might not reflect the needs for a particular application and its processing environment. Comparative experimental performance evaluation would be the best foundation, but these require expert knowledge for all involved SPEs to achieve a fair comparison. Furthermore, experiments require a lot of effort, including configuration management, workload generation, and monitoring to gather performance numbers. Experimenting with distributed SPEs is even more complex. To overcome these challenges, we present in this paper a framework to simplify experimental performance evaluation of distributed SPEs, called Expose. Before we describe the fairness aspect in more detail, we briefly identify other use cases for Expose: (1) researchers that develop new SPE (mechanisms) and want to compare them with state-of-the-art solutions, (2) developers that want to identify bottlenecks in SPEs, (3) users that want to understand the impact of different processing environments on the performance of an SPE, and (4) committees that want to define benchmarks.

While heterogeneity in SPEs is an important reason to perform comparative experiments, it is also the main reason why it is difficult to do. Different SPEs use different abstractions, concepts, and lack a definite standard. An expert for one system might not integrate another system as well in an experiment. This can lead to unfairness and bias, and a system might perform best because the developer knows it the best, not because it is best for the application. Ideally, we would like to run experiments by using for all SPEs a common set of concepts like ”Deploy queries,” ”Define data stream,” ”Add sink for a stream,” which are all implemented in the different SPEs.

To the best of our knowledge, no existing work provides such a generic interface for executing complex distributed SPE experiments. Apache Beam and the standardization initiative in [3] aim for a unified interface for SPEs and other data management systems, but lack support for experimental performance evaluation of SPEs. The PEEL experiment framework [4] provides users with the ability to define experiments with less effort than normal, but each SPE still needs to be treated in a different way when defining experiments. Moreover, PEEL does not enable the user to configure the distribution of the SPEs. Multiple microbenchmarks exist that address individual stream processing operators, but they do not test SPEs on application level and lack distribution.

We aim to reduce the workload when defining and executing distributed SPE experiments through a declarative approach. Our proposal is a framework for defining distributed SPE experiments and automating the execution of the experiments. The same experiment definition can be executed with all supported SPEs. A user can utilize any dataset for transmitting tuples and define schemas to be used in experiments. Moreover, the user can choose to trace new metrics and events of interest, which can, for example, be used to create new benchmarks. Stream topologies can be set up in the same way for all supported SPEs, regardless of whether the SPE system internally uses the publish/subscribe or data source and data sink abstractions.

The core of Expose is an API based on a set of SPE tasks. These are used to define the experiments with commands like "Add Query 5 to Node 2," "Wait until the stream has ended on Node 2," and "Send Dataset 2 as a stream on Node 1." To support a new SPE in Expose, we expect experts to implement a "wrapper" for the SPE that supports these commands. This limited one-time effort should allow for fairness since an expert should be able to implement it in the best possible way. Through this declarative approach, non-experts with a basic understanding of SPE concepts are able to define complex distributed experiments, including the performance metrics of interest. The experiments are automatically executed, and the results are prepared for the experimenter. A summary of our contributions is as follows:

- framework for evaluating and comparing distributed SPEs in a fair and replicable way,
- declarative API for the execution of SPEs,
- implementation of the API in five SPE systems (Flink [5], Beam [1], Siddhi [24], Esper [8], and T-Rex [7]),
- open-source repository that contains the code, available at GitHub[1],
- implementation of a well-known SPE benchmark in Expose, and
- execution of the benchmark on the supported SPEs.

[1]GitHub repository available at https://github.com/espv/expose

The paper is structured as follows. Section III.2 presents the framework. In Section III.3, we demonstrate the use of the framework. Section III.4 discusses related works, and Section III.5 concludes the paper.

III.2 Design

The main innovation introduced by Expose is the ability to define simple distributed SPE experiments with multiple SPEs. The output of the systems is homogeneous and thus directly comparable. The experimenter does not need to know the internal workings of these systems, but must rather define experiments as a list of tasks to be executed on the specified node in the experiment, each of which represents one SPE instance or cluster in the experiment topology. Examples of tasks are adding a node as next hop for a given stream, deploying SQL queries, and streaming a dataset to the next hops. How these tasks are executed on each specific SPE is up to a "wrapper" that is implemented by an expert in that SPE.

The target user for Expose is anyone with an interest in conducting a performance evaluation of SPEs. Benchmarks are great for performance evaluation, but new ones are always needed because they often have a narrow focus [14]. The downside of benchmarks is that they are rigid and require all stakeholders to agree that the benchmark provides meaningful results [9]. Sometimes, custom benchmarks or performance evaluations are preferable, which requires a way of defining, changing, and executing them. Our goal is to give the user the ability to easily define and execute their own performance evaluation such that they can gain with a low effort the type of results and insights they are interested in. They can select custom datasets, measurements, and parameters to use for the evaluation, and set up the stream topology with an arbitrary number of nodes.

III.2.1 Experiments

An SPE experiment is a clearly defined execution of SPEs that uses an experiment definition as input and gives performance results as output. The experiment definition includes all information needed to execute the experiment. Therefore, the experiments are repeatable. In order to use different SPEs in the experiments, Expose works in a declarative way. The experimenter decides what tasks should be executed, but not how. We remove the need for the experimenter to understand the specifics of the SPE. To make the experiment definition human-readable, we use the YAML configuration format.

If the output from SPEs is system-independent, analysis of the results and comparison among the systems becomes much more feasible. To achieve this, it is necessary to implement or reuse a minimalistic tracing module for each SPE that can be used to trace arbitrary events in the code. The reason to do this instead of using the SPE’s internal runtime logs is to ensure that the output trace is in the same format for every SPE. The SPE expert adds tracepoints, each with its own ID, and then the experimenter can activate or deactivate tracepoints in experiments. Therefore, the experiment definition should specify not only the tasks to be executed, but also what events are traced during the experiment.

The lack of a standardized query language for SPEs is well known and resulted, for example, in the popular Apache Beam and "One SQL" [3] standardization initiative. Apache Beam can imply performance penalties ranging from factor 4 to 58, depending on the SPE and operators being used [11]. This work is focused on performance evaluation that results in reliable performance numbers. As such, it is important that the solution is lightweight with minimal impact on the performance evaluation and support tasks that are necessary to perform experiments and to deliver quite reliable performance numbers. Therefore, we do not build upon Beam, but instead create an independent solution that can evaluate all kinds of SPEs, including unifying systems such as Beam.

For any experiment, the SQL queries, stream schemas, and datasets need to be given in the experiment definition. Each element of the experiment definition is SPE-agnostic except for the SQL queries. Even though the SPEs support an SQL-like language, their syntax may be completely different from each other, and the same features might not be supported in all SPEs. The SPE wrappers can translate small variations in syntax, but translating complex SQL queries without any performance penalty is outside the scope of this paper.

On the other hand, we can define SPE tasks, which are common across all SPEs. Examples are "send dataset as a stream," "add next hop for stream ID," and "deploy SQL query." These tasks are materialized as an API that each SPE wrapper implements. The SPE tasks are defined based on an analysis of the SPEs that we study in this paper. As such, the set of tasks might not be comprehensive, but it is sufficient to describe a wide variety of distributed stream processing experiments. To create an SPE wrapper, the SPE expert maps the tasks to the SPE by using the special functionality offered by the SPE. That way, we preserve the unique capabilities of the SPEs while enabling different SPEs to run the same experiments.

In Table III.1, we list all SPE tasks that are needed for the experiments. We distinguish between tasks that are manual and explicitly added to the experiment definition by the experimenter and tasks that are automatically issued by the SPE wrapper. The SPE tasks describe how to set up or expand stream topologies with `addNextHop`. Stream schemas can be added with `addSchemas`. SQL queries can be deployed on the specified node with `deployQueries` and removed with `clearQueries`. The runtime environment can be started with `startRuntimeEnv` and stopped with `stopRuntimeEnv`.

SPE tasks	Manual	Description
addNextHop	Yes	Add next hop to a stream
deployQueries	Yes	Deploy specified query a given number of times
startRuntimeEnv	Yes	Start runtime environment
stopRuntimeEnv	Yes	Stop runtime environment
setParallelism	Yes	Set desired level of parallelism for processing tuples
sendDsAsStream	Yes	Convert dataset to stream, and send it to all next hops
clearQueries	Yes	Remove all existing queries
writeStreamToCsv	Yes	Write tuples from stream to file
addSchemas	No	Add stream schemas
startExperiment	No	First task executed by SPEs in the experiment
endExperiment	No	Final task executed by SPEs in the experiment

Table III.1: SPE tasks

The level of parallelism can be set with `setParallelism`. Streaming datasets to the next hops can be done with `sendDsAsStream`, realistically using timestamps from the tuples, as fast as possible, or with an arbitrary rate. Tuples from a stream can be written to file using `writeStreamToCsv`.

In addition to the SPE tasks, experiments require another class of tasks called experiment tasks, listed in Table III.2. These tasks are necessary for the execution of experiments, most of which are automatically issued to the SPE nodes by the coordinator. Examples of experiment tasks include looping through a sequence of tasks a given number of times and waiting until no tuple has been received in a number of milliseconds.

Distribution is required in most SPE applications, but it makes experimentation more complex. It is not trivial to ensure that the tasks in the experiments are executed in the correct order, at the correct time, and by the correct node. Moreover, we need to decide what entity is responsible for deciding which node executes which task and when. To support distributed experiments and give the experimenter full control of the end-to-end data stream pipeline, we introduce a `coordinator`, as illustrated in Figure III.1. `m` SPE nodes participate in this experiment, each of which is an instance of an SPE or an SPE cluster. Arbitrarily many SPE nodes can run on a single machine. The coordinator has an overview of these nodes and is the entity that issues tasks to them. The tasks in the experiment definition must denote which node should execute the task. We introduce the node ID to make the experiment definitions fully reusable and avoid specifying IP addresses and port numbers in the experiment definition. During the start phase of an

Experiment task	Manual	Description
traceTuple	Yes	Custom-defined tracepoint
retEndOfStream	Yes	Task that returns from the SPE when no tuples have been received for a specified number of milliseconds
loopTasks	Yes	Coordinator task to loop through a sequence of tasks a given number of iterations
setNidToAddress	No	Coordinator broadcasts mapping from node ID to address when new SPE instance/cluster registers
addTpIds	No	Setup task to let SPEs know which tracepoints should be active
startExperiment	No	First task executed by SPEs in the experiment
endExperiment	No	Final task executed by SPEs in the experiment

Table III.2: Experiment tasks

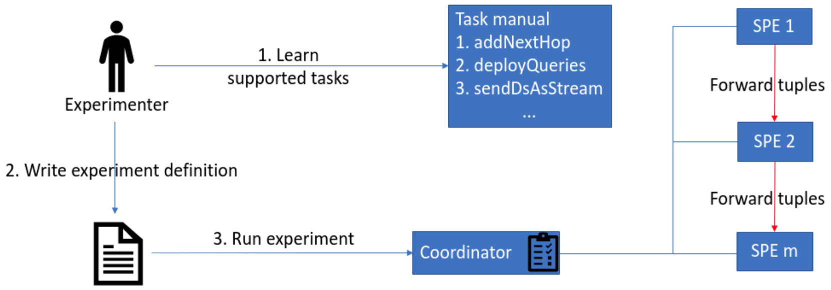


Figure III.1: Workflow of performing an experiment with Expose

experiment, each SPE node registers with the coordinator and provides their node ID and port on which the other nodes can reach them. The coordinator then broadcasts the mapping from node ID to the IP address and port to the rest of the SPE nodes. This way, all SPE nodes know how to transmit tuples to the next hop, and the coordinator knows to which address to transmit tasks. As such, the end-to-end pipeline can be specified in the experiment definition without low-level details such as IP addresses and port numbers that are dependent on the particular execution environment of an SPE.

The experimenter starts the coordinator and SPE nodes, either manually or through a reusable script. Each SPE node is started with the coordinator's address and uses it to connect to the coordinator's TCP server. Therefore, the SPE nodes can be on the same machine as the coordinator, on a different machine in the same local area network, or anywhere else, as long as the coordinator is accessible. The coordinator and the SPE instances/clusters can, for instance, be started remotely with software such as Ansible [13], in which commands can execute remotely on another server. The experimenter only needs to install the SPEs and provide the Ansible node names in the script. These tasks can be done by someone with limited technical skills.

III.2.2 Measurements and Analysis

The overall goal of Expose is to get performance numbers from executing SPE experiments. This section explains how Expose supports the analysis and comparison of results. The analysis is based on the output execution traces from the SPEs. Two types of trace events are captured, listed in Table III.3: processing and state events. Processing events are used to calculate throughput and execution time. Examples of these include tracing when a tuple is received and when it is finished processing. With state events, we can calculate the execution time and throughput with respect to, e.g., the number of deployed queries or the size of the windows. Examples of state events include when a query is deployed, when all queries are cleared and when a new data sink or data source is added.

Tracepoint names	Type
Start experiment	State
Receive Tuple	Processing
Finished Processing Tuple	Processing
Deploy Query	State
End of stream	State
Increase number of sources	State
Increase number of sinks	State

Table III.3: Tracepoints

The processing and state trace events can be used to visualize and represent the results in various ways. The processing events are used to calculate the performance metric, and the state events are used to calculate the control parameter. In a 2D graph, one would typically see the value of the performance metric on the y-axis and the control parameter on the x-axis.

III.2.3 SPE Wrapper

In this section, we address three challenges to overcome when attempting to create an SPE wrapper to include in Expose. They are (1) to implement the task API, (2) implementing a communication module between the wrapper and the coordinator, and (3) implementing a tracing module for the wrapper. The task API consists of the methods listed in Tables III.1 and III.2. The main challenge involves mapping the SPE tasks to the functionality offered by the

SPE. For instance, setting the next hop for streams in Esper, Siddhi, and T-Rex involves transmitting the produced tuples to the recipients within a method call, whereas in Beam and Flink, it requires setting up the data pipeline before the runtime environment has started. When tuples are produced, they are automatically transmitted to the next hops. How to deploy SQL queries depends completely on the abstractions and data structures used in the SPE. Sending a dataset as a stream is challenging to implement because most SPEs have their own way of ensuring that the tuples conform to the schemas. Moreover, how tuples are sent to the next-hop nodes depends on which connector is used by the SPE, e.g., TCP or Kafka. When the API is implemented, the SPE wrapper can execute experiments, but only locally.

To enable the SPE to participate in distributed experiments, we need the communication module between the wrapper and the coordinator. At startup, the coordinator waits for SPE nodes to register by contacting its TCP server. The wrapper is provided the address information to the coordinator, and the communication module establishes a connection with the coordinator. This module starts an infinite loop where it waits for tasks to execute from the coordinator. When it receives one, it calls the corresponding method in the wrapper with the provided arguments. After the task is finished executing, it replies to the coordinator with the return value and waits for another task to execute. The communication pattern between the SPE wrappers and the coordinator is the same, regardless of the SPE. Therefore, Expose can execute experiments with different SPEs. This module can also be entirely reused for a new SPE if it is written in the same programming language as a previous SPE.

The final component to implement in an SPE wrapper is the tracing module. This module is required to be able to record and retrieve results from the experiments. The most important requirement for this module is that all the SPEs that participate in the experiment have a similar module that causes minimal overhead to the experiments. As with the communication module, this module can be reused if the SPE is written in the same language as another SPE with a ready wrapper.

We have created SPE wrappers for Siddhi 5.0.0, Esper 8.3.0, T-Rex, Flink 1.9.1, and Beam 2.21.0. Siddhi, Esper, Flink, and Beam are Java-based SPEs, whereas T-Rex is a C++-based SPE. Beam is by itself not an SPE, but a system that gives a unified SPE interface and can be used to execute a variety of SPE engines, such as Flink and several other SPEs. We choose in the experiments later to run it with the Flink runner, which we call Beam Flink, to investigate the performance difference between it and Flink.

III.3 Use-Case: NEXMark Benchmark

The goal of this section is to demonstrate that (1) Expose can be used to evaluate and compare various SPEs, (2) how easy it is to perform such experiments, and (3) that we can define benchmarks with Expose with low effort. To achieve this, we have on, the one hand, implemented wrappers for Flink, Beam, Esper, Siddhi, and T-Rex, and on the other hand, implemented an experiment definition for a well-known and accepted benchmark called NEXMark [25]. The benchmark describes a set of eight queries that process data from three schemas: Person, Auction, and Bid. We implement all the queries in addition to a passthrough query that only selects and forwards Bid tuples, which comprise 92% of the dataset. Each query is used in one experiment. Below, we describe the queries:

0. Passthrough: forwards all Bid tuples.
1. Currency Conversion: uses a user-defined function to convert the Bid prices from dollar to euro.
2. Selection: filters Bid tuples based on auction ID.
3. Local Item Suggestion: a join between Auction and Person tuples with predicates on the Person.state and Auction.category.
4. Average Price for a Category: the average price of auction categories.
5. Hot Items: the auction with the most Bid tuples is selected.
6. Average Selling Price By Seller: the average price of the auction items of each seller is selected.
7. Highest Bid: the Bid tuple with the highest price is selected.
8. Monitor New Users: Person tuples are joined with Auction tuples that were received within 12 hours of each other.

We execute the benchmark on two different servers: a powerful Intel Xeon server with 48GB RAM and two Intel Xeon Gold 5215 SP CPUs with ten cores, each running at 2.50GHz, and a weak Raspberry Pi 4 B+ (RPI) with a Broadcom BCM2711 CPU that has four cores running at 1.5GHz, with 4GB RAM. The experiment topology consists of three SPE instances: the data driver that produces the tuples, the system under test (SUT) that processes the tuples according to the deployed query, and the sink node that receives all the produced tuples. The data driver and sink run on the same hardware as the coordinator, and the SUT runs on either the Intel Xeon server or the

RPI. This way, the SUT is completely isolated and is not affected by Expose or the other SPEs. NEXMark leaves the size of the dataset open, but it provides a dataset generator that enables us to generate datasets of different sizes. We use a dataset with size 1,000,000 tuples for the Intel Xeon server, and the RPI uses 40,000 tuples. In some of the queries, the memory consumption increases for each incoming tuple. Thus, the small amount of RAM available on the RPI makes it impossible for the RPI to process a larger dataset.

The experiment instructions used for all the queries are given in Listing III.1, where only `query_id` and `output_stream_id` differ for each query. Each SPE runs the experiment for all its supported queries and requires no changes to the experiment instructions. The experiment is started by setting up the data stream topology with `addNextHop`, i.e., set Node 2 as a recipient of the streams in the dataset, and Node 3 as the recipient of the output stream from the query. After the topology is set up, a loop with ten iterations starts where the dataset is sent as a stream in each iteration. From the traces, we calculate the throughput of each SPE and query as the average number of tuples per second (TPS) and the relative standard deviation (RSD). Only the final five iterations in the loop are used for these calculations; the first five iterations serve as a warmup to enable any runtime optimizations to activate, which can have a significant effect on performance in Java.

Listing III.1: NEXmark experiment instructions

```
# Set Node 1 to send Auction, Bid and Person to Node 2
— {task: addNextHop, arguments: [1, 2], node: 1}  # Person stream
— {task: addNextHop, arguments: [2, 2], node: 1}  # Auction stream
— {task: addNextHop, arguments: [3, 2], node: 1}  #  Bid stream
# Set Node 2 to send output stream to Node 3
— {task: addNextHop, arguments: [output_stream_id, 3], node: 2}
# Deploy query to Node 2
— {task: deployQueries, arguments: [query_id, 1], node: 2}
# Stream the dataset ten times
— {task: loopTasks, node: coordinator, arguments: [10, [
  {task: startRuntimeEnv, node: 2},
  {task: startRuntimeEnv, node: 3},
  {task: sendDsAsStream, arguments: [8], node: 1},
  {task: retEndOfStream, node: 3, arguments: [2000]},
  {task: traceTuple, node: 2, arguments: [200, []]},
  {task: stopRuntimeEnv, node: 2},
  {task: stopRuntimeEnv, node: 3}]]}
```

Not all the SPEs support the necessary query processing functionality for these specific queries. Therefore, even though all the SPE wrappers support the same tasks, some SPEs do not run all the queries. For instance, T-Rex is a CEP system that only supports Query 0. Beam does not support joining streams without special window constructs, and so, it does not run Queries 3, 4, 6, and 8. Moreover, Query 6 requires the ability to group output by a key and then limit the number of tuples in each group to ten tuples, which is not supported by the SQL language of any of the SPEs. Therefore, we modify Query 6 to look at all the tuples instead, for all the SPEs. Queries 4–6 define sliding windows over multiple aggregations, which is only supported in Flink. The problem is that performing aggregations over aggregations, such as calculating the average of several maximum values, requires the ability to invalidate outdated tuples. A newly received tuple might replace an old maximum, and so, the old tuple should be removed from the average. As far as we know, only Flink supports this feature among the SPEs. Therefore, Siddhi and Esper perform a variation of the queries using tumbling windows instead of sliding windows, which does not require this feature. A tumbling window version of Query 4–6 for Flink and Beam is not possible either because of their limited support for tumbling windows using an external timestamp and so cannot execute them. Therefore, we have a sliding and tumbling window version of Query 4–6.

Listings III.2, III.3, III.4 and III.5 show the implementations of Query 4 for Siddhi, Flink and Esper, in addition to the template query. Notice how the implementations are completely different, as the SPEs do not (1) support the same query processing functionality, and (2) use different syntax among each other. In particular, Siddhi implements this query as three separate queries, where the output from the first query is used as input to the second, and the output from the second query is used for the third. Esper implements it as two queries where the output from the first query is used as input to the second, and Flink implements it as a single query with a subquery. Siddhi requires three queries because it must separate the (1) join between Auction and Bid from (2) the maximum aggregation that uses the tumbling window, and (3) the average aggregation. With Esper, we can combine the first two queries and only have to separate the last average aggregation. Flink supports subqueries, and therefore performs the maximum aggregation in the subquery and the average aggregation in the top-level of the query. Beam also supports the syntax of the query that Flink uses, but it has limited support with regard to joining streams, and therefore, we do not execute this query with Beam. T-Rex also has no support for this query. This issue of different SQL languages and query processing features illustrates the challenge with comparing SPEs, and thus, why Expose requires each SPE explicitly to have its version of a given query in the experiment definition.

Listing III.2: NEXMark Query 4 with sliding window in Siddhi

```
from Bid#window.time(999 years) as B
join Auction#window.time(999 years) as A on A.id == B.auction
select B.dateTime, B.price, A.category, B.auction, A.expires
insert into MQ4_1;

from MQ4_1#window.externalTimeBatch(dateTime, 1 min)[dateTime < expires]
select max(price) as final, category group by auction, category
insert into MQ4_2;

from MQ4_2#window.time(999 years)
select avg(final) as price, category group by category
insert into OutQuery4;
```

Listing III.3: NEXMark Query 4 with tumbling window in Flink

```
select avg(final), category
from (select MAX(B.price) AS final, A.category from Auction A, Bid B
      where A.id=B.auction and B.dateTime2 < A.expires2
      group by A.id, A.category) Q
group by category
```

Listing III.4: NEXMark Query 4 with sliding window in Esper

```
insert into MQ4_2
select max(B.price) as final, A.category as category
from Auction#time(999 min) A, Bid#ext_timed_batch(dateTime, 1 min) B
where A.id = B.auction and B.dateTime < A.expires
group by B.auction, A.category;

insert into OutQuery4
select avg(final) as price, category
from MQ4_2
group by category;
```

Listing III.5: NEXMark Query 4 template

```
select Istream(avg(Q.final))
from (select Rstream(max(B.price) as final, A.category)
      from Auction A [rows unbounded], Bid B [rows unbounded]
      where A.id=B.auction and B.dateTime < A.expires and
            A.expires < current_time
      group by A.id, A.category) Q
group by Q.category;
```

III.3.0.1 Results

Tables III.4 and III.5 contain the results from running NEXMark on Intel Xeon and RPI, respectively. In Queries 4–6, we distinguish between two versions: T stands for tumbling window, and S stands for sliding window. The SPEs run the benchmark on a single CPU core because the SPEs have a varying degree of concurrency support. An SPE like Flink is made for scalability, whereas the library versions of Esper and Siddhi are not. Kafka is used for communication between nodes in Flink and Beam, and therefore, also runs on a single core. The queries that cannot execute on the SPEs have empty table cells. Beam Flink means that Beam is executed with the Flink runner.

Flink performs the best among the SPEs in almost all queries, by a significant factor. One explanation might be that Flink has the most advanced processing environment out of all the SPEs. Flink's aggregation queries produce many fewer output tuples compared to the other SPEs, which positively affects the throughput. In contrast, Beam Flink has the worst performance in all cases. Beam Flink has a throughput between 6 and 73 times lower on the RPI, and between 2 and 11 times lower on the Intel Xeon server. These results seem to correspond with previous results from [11], in which Beam has a slowdown factor of between 4 and 58 times compared to Flink. The reason for the lower throughput might be a combination of overhead caused by Beam's attempt to be compatible with many SPEs and

Query	Beam Flink		Flink		Siddhi		T-Rex		Esper	
	TPS	RSD	TPS	RSD	TPS	RSD	TPS	RSD	TPS	RSD
0	26.3k/s	0.69%	68.7k/s	8%	34k/s	0.51%	20.9k/s	0.15%	29.4k/s	0.61%
1	25.7k/s	0.61%	68.6k/s	7.7%	34.5k/s	0.45%	—	—	32.4k/s	0.84%
2	81.3k/s	0.49%	197.6k/s	6.2%	58.5k/s	1.4%	—	—	46.8k/s	1.5%
3	—	—	184k/s	10.4%	56k/s	1.4%	—	—	47.1k/s	2.2%
4 (T)	—	—	—	—	585.7/s	0.62%	—	—	26.4k/s	15.9%
4 (S)	—	—	103.6k/s	6.8%	—	—	—	—	—	—
5 (T)	—	—	—	—	59.6k/s	1.48%	—	—	30.2k/s	16.8%
5 (S)	15.4k/s	0.25%	107.8k/s	11.6%	—	—	—	—	—	—
6 (T)	—	—	—	—	549.2/s	0.89%	—	—	30.9k/s	20%
6 (S)	—	—	107.1k/s	7.2%	—	—	—	—	—	—
7	12.7k/s	0.2%	134.1k/s	16.1%	59.3k/s	1.2%	—	—	34.7k/s	9.2%
8	—	—	184.4k/s	16.5%	25.9k/s	0.67%	—	—	47k/s	3.6%

Table III.4: TPS and RSD when NEXMark runs on Intel Xeon five times.

Query	Beam Flink		Flink		Siddhi		T-Rex		Esper	
	TPS	RSD	TPS	RSD	TPS	RSD	TPS	RSD	TPS	RSD
0	679.9/s	1.6%	5k/s	8.8%	3.9k/s	0.44%	10.6k/s	0.56%	2.8k/s	0.8%
1	678.3/s	2.8%	4.1k/s	8.1%	3.9k/s	0.21%	—	—	2.9k/s	0.47%
2	2.9k/s	3.3%	28.3k/s	5.4%	8k/s	0.43%	—	—	7.6k/s	0.77%
3	—	—	37.1k/s	27.3%	7.7k/s	0.88%	—	—	7.4k/s	1.9%
4 (T)	—	—	—	—	2.1k/s	0.98%	—	—	6.4k/s	1.9%
4 (S)	—	—	10.8k/s	13.5%	—	—	—	—	—	—
5 (T)	—	—	—	—	7.8k/s	0.34%	—	—	6.5k/s	6.1%
5 (S)	235.4/s	1.1%	9.8k/s	16.6%	—	—	—	—	—	—
6 (T)	—	—	—	—	2.1k/s	1.7%	—	—	5.7k/s	6.7%
6 (S)	—	—	11.5k/s	13%	—	—	—	—	—	—
7	236.2/s	1.2%	17.3k/s	11.3%	7.5k/s	0.65%	—	—	6.4k/s	4.9%
8	—	—	34.6k/s	22.5%	7.1k/s	0.5%	—	—	7.1k/s	1.8%

Table III.5: TPS and RSD when NEXMark runs on an RPI five times.

different policies regarding emission rate. However, more investigation is required to find out the reason why.

Noticeably, Queries 4 and 6 on Siddhi have a low throughput that is even higher on the RPI than the Intel Xeon server. The reason for the poor performance of those queries is that they perform joins on many tuples, with which Siddhi seems to perform poorly. Moreover, the Intel Xeon server uses a much larger dataset than the RPI. The first tuples require only around 2 microseconds processing time, but as the number of tuples in the windows increases, the final tuples require around 2.3 milliseconds of processing time each on the Intel Xeon server.

In Table III.6, we scale the number of CPU cores that Flink and Kafka utilize and the number of queries deployed to Flink. The query is the same as Query 4 from the above tables. As we can see, the performance degrades much faster when running only one CPU core versus ten, as the number of queries increases. Even with one query deployed, running on multiple

CPU cores increases the throughput from 110,900 tuples per second to 185,800 tuples per second. As the number of queries increases, the number of produced tuples also increases, which are forwarded to the sink node. Therefore, the throughput decreases for two reasons: the processing or networking capacity is overloaded.

# CPU cores	1 query	2 queries	5 queries	10 queries	15 queries	20 queries
1	110.9k/s	73.3k/s	36.3k/s	18.8k/s	13.2k/s	9.3k/s
10	185.8k/s	177.3k/s	134.2k/s	63.5k/s	47.7k/s	28.3k/s

Table III.6: TPS when Flink has deployed Query 4 from NEXMark and runs on Intel Xeon five times with a different number of CPU cores and queries.

III.4 Related Work

Few experiment frameworks for SPEs exist that aim at providing a user-friendly experience. The PEEL experiment framework is one of them [4]. It enables users to define experiments, execute them, and repeat them. Runtime logs from the running systems are collected, and so the experiments can be used to benchmark systems. However, they do not have a homogeneous input and output as we do. Their experiment definitions need special treatment for each SPE. Although the SPE-specific code for all SPEs is placed in the same experiment definition, it is not obvious that the SPEs will run the same tasks, as it is in our case. Moreover, their experiment definitions are programs written in Scala, which require significantly more effort than writing experiment definitions with Expose. Our result analysis is flexible with regard to the traced and varied parameters because of the tracing module that the SPEs must implement. In contrast, PEEL relies on the logs from the SPEs, making it harder for the SPEs to trace the same data.

FINCoS [20] is another experiment framework, which is an extended version of the benchmarking framework in [19]. They enable users to use their own datasets and can communicate with different SPE engines. However, it does not support automation of performance evaluations, which is a key feature of Expose. Moreover, since only Esper is mentioned to be supported, it is hard to determine the effort required to create a new "wrapper," whereas we have focused our efforts on simplifying this process. FINCoS does not appear to support arbitrary stream topologies; each node is either a data driver (data source), the SUT, or data sinks. Expose enables the experimenter to control the stream topology freely in the experiment. Nodes can be set to forward or redirect streams to an arbitrary number of nodes.

Multiple benchmarks and benchmark tools for stream processing exist in the literature [12, 17]. One of the earliest works introduced is the Linear

Road benchmark [2], which can be used to simulate traffic in motor highways. Systems may then achieve an L-rating that is a measure of their supported query load. Although Linear Road is relatively old, it is still implemented for new systems like Apache Flink [10], and for SPEs written in P4 for ASICS [15]. Other benchmarks include [6, 12, 16, 18, 21, 22]. These benchmarks have in common that they are mainly meant for heavyweight SPEs such as Apache Storm, Apache Samza, Apache Spark and Apache Flink. In contrast, we consider the more lightweight SPEs that are relevant in fog networks and which might be sufficient for, e.g., Internet of Things applications.

The unification of the use of SPE systems is an ongoing effort. A Stream SQL standard is recently proposed in [3] and is in the process of being implemented for existing SPEs. Apache Beam is a framework that attempts to unify SPEs by providing a unified interface for writing SPE applications. It performs a task similar to Expose in that Expose unifies the definition and execution of distributed experiments, and Beam unifies the execution model of the SPEs.

III.5 Conclusion

We present in this work a framework that simplifies the definition and execution of distributed SPE experiments. A set of experiments can be defined in an experiment definition file, and the same experiment definition can be used to execute different SPEs, which makes comparisons of multiple SPEs easier. Combined with our design choice that experts implement the SPE wrappers and the fact that we add only a very thin software layer on top of the SPEs, Expose achieves fair treatment of SPEs. This prevents bias from experimenters that are experts in one class of SPEs and novices in others.

To demonstrate the ease with which experiments can be defined and executed, we create an experiment definition that describes a well-known benchmark called NEXMark using Expose. Then we run it with Flink, Beam Flink, Siddhi, T-Rex, and Esper on a powerful server and a resource-constrained Raspberry Pi 4. The experiment definition is concise, reusable, and can be changed by a user to suit their particular needs.

For future work, we aim at adding the ability to decentralize the coordination of nodes, which means that nodes can issue tasks to other nodes. That way, we can test out time-critical algorithms and variations between them. An example of this is operator migration algorithms in distributed CEP. For that to be possible, the SPE wrappers must be extended with more tasks, such as the ability to move query state between nodes and stopping and buffering streams.

References

[1] *Apache Beam*. https://beam.apache.org. [Online; accessed 6-August-2020].

[2] Arasu, A. et al. “Linear road: a stream data management benchmark”. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment. 2004, pp. 480–491.

[3] Begoli, E. et al. “One SQL to Rule Them All-an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1757–1772.

[4] Boden, C. et al. “PEEL: A framework for benchmarking distributed systems and algorithms”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2017, pp. 9–24.

[5] Carbone, P. et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* vol. 36, no. 4 (2015).

[6] Chintapalli, S. et al. “Benchmarking streaming computation engines: Storm, flink and spark streaming”. In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2016, pp. 1789–1792.

[7] Cugola, G. and Margara, A. “Complex event processing with T-REX”. In: *Journal of Systems and Software* vol. 85, no. 8 (2012), pp. 1709–1728.

[8] *Esper*. http://www.espertech.com/esper. [Online; accessed 6-August-2020].

[9] Folkerts, E. et al. “Benchmarking in the cloud: What it should, can, and cannot be”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2012, pp. 173–188.

[10] Hanif, M., Yoon, H., and Lee, C. “Benchmarking tool for modern distributed stream processing engines”. In: *2019 International Conference on Information Networking (ICOIN)*. IEEE. 2019, pp. 393–395.

[11] Hesse, G. et al. “Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems”. In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 1381–1392.

[12] Hesse, G. et al. “Senska–Towards an Enterprise Streaming Benchmark”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2017, pp. 25–40.

[13] Hochstein, L. and Moser, R. *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. " O'Reilly Media, Inc.", 2017.

[14] Huppler, K. "The art of building a good benchmark". In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2009, pp. 18–30.

[15] Jepsen, T. et al. "Life in the fast lane: A line-rate linear road". In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–7.

[16] Karimov, J. et al. "Benchmarking distributed stream data processing systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1507–1518.

[17] Kiatipis, A. et al. "A Survey of Benchmarks to Evaluate Data Analytics for Smart-* Applications". In: *arXiv preprint arXiv:1910.02004* (2019).

[18] Lu, R. et al. "Stream bench: Towards benchmarking modern distributed stream computing frameworks". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE. 2014, pp. 69–78.

[19] Mendes, M. R., Bizarro, P., and Marques, P. "A framework for performance evaluation of complex event processing systems". In: *Proceedings of the second international conference on Distributed event-based systems*. 2008, pp. 313–316.

[20] Mendes, M. R., Bizarro, P., and Marques, P. "FINCoS: benchmark tools for event processing systems". In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 2013, pp. 431–432.

[21] Rabl, T. et al. "The vision of BigBench 2.0". In: *Proceedings of the Fourth Workshop on Data analytics in the Cloud*. 2015, pp. 1–4.

[22] Shukla, A., Chaturvedi, S., and Simmhan, Y. "RIoTBench: An IoT benchmark for distributed stream processing systems". In: *Concurrency and Computation: Practice and Experience* vol. 29, no. 21 (2017), e4257.

[23] Stonebraker, M. and Çetintemel, U. "" One size fits all" an idea whose time has come and gone". In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018, pp. 441–462.

[24] Suhothayan, S. et al. "Siddhi: A second look at complex event processing architectures". In: *Proceedings of the 2011 ACM workshop on Gateway computing environments*. 2011, pp. 43–50.

[25] Tucker, P. et al. *NEXMark—A Benchmark for Queries over Data Streams DRAFT*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

Paper IV

To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing

Espen Volnes, Thomas Plagemann, Vera Goebel

Under minor revision in *IEEE Communications Surveys & Tutorials*.

Abstract

One of the most important issues in distributed data stream processing systems is using operator migration to handle highly variable workloads cost-efficiently and adapt to the needs at any given time on demand. Operator migration is a complex process involving changes in the state and stream management of a running query, typically without any data loss, and with as little disruption to the execution as possible. This tutorial aims to introduce operator migration, explain the core elements of operator migration, and provide the reader with a good understanding of the design alternatives used in existing solutions. We developed a conceptual model to explain the fundamentals of operator migration and introduce a unified terminology, leading to a taxonomy of existing solutions. The conceptual model separates mechanisms, i.e., how to migrate, and policy, i.e., when to migrate. This separation is further applied to structure the description of existing solutions, offering the reader an algorithmic perspective on various design alternatives. To enhance our understanding of the impact of various design alternatives on migration mechanisms, we also conducted an empirical study that provides quantitative insights. The operator downtime for the naïve migration approach is almost 20 times longer than when applying an incremental checkpoint-based approach.

IV.1 Introduction

Distributed stream processing (DSP) has been researched for more than 20 years, and is becoming ubiquitous in application domains where real-time decision-making is essential [96], like the Internet of Things (IoT), fraud, and anomaly detection, smart cities [39], and autonomic systems. DSP is a useful technology whenever there is too much data to store all of it, and when the data is only valuable shortly after it is generated. Deep learning can be applied to facilitate analytics on streaming data in IoT [98]. Industry 4.0 is a term that means the fourth generation of the industrial revolution and applies stream processing to do data collection, analysis, storing and querying [121].

DSP is currently used by companies that need to process and analyze billions of events every day. For example, the popular stream processing engine Apache Flink [16] is used by Alibaba, AWS, Comcast, Ebay, Huawei, Lyft, Uber, Zalando, and many more companies, to perform real-time processing [1]. Another indicator of the wide use and importance of stream processing is that most cloud vendors offer support for deploying managed stream processing pipelines [34], and a sign of its future relevance is the estimated economic impact of the IoT industry, estimated to be between \$3.9 trillion and \$11.1 trillion a year by 2025, around 11% of the global economy [92].

Stream processing engines (SPE) come in several flavors, are deployed in different environments (i.e., cloud, fog, edge, in-network), and perform data stream management, real-time stream analytics, event stream processing, and complex event processing (CEP). The common denominator in all these systems is that data arrive continuously (generally as tuples) from multiple sources, and need to be processed as soon as they arrive (in memory) to enable immediate decision-making. Thus, the response time must be short, even in case of large loads.

SPEs take queries as input and compile them into operator graphs. In the simple example in Figure IV.1 the query at the top of the figure is compiled to an operator graph with two data producing operators (Auction and Bid stream), a join operator, and a data consuming operator (Section IV.2.3 builds on this simple example and gives further details). Operator graphs are directed acyclic graphs (DAGs), as illustrated in Figure IV.1, that represent the logical execution of a query, which includes the operators (i.e., state management of subqueries) and the dependencies between them (i.e., stream management) represented as vertices. If these operators are mapped to several physical hosts and form an overlay network, a DSP system is established. Incoming data tuples to an operator are processed, e.g., by filtering and joining as in Figure IV.1, or transforming, aggregating, or running a user-defined function.

A key requirement for DSP is the ability to handle system dynamics, like changes in workload, resource availability, and mobility. Operator migration is the key mechanism for handling such changes. The four primary goals that motivate different operator migration solutions are: (1) to re-balance uneven distribution of computational tasks across nodes (load balancing), (2) adapting the amount of allocated resources to increasing or decreasing

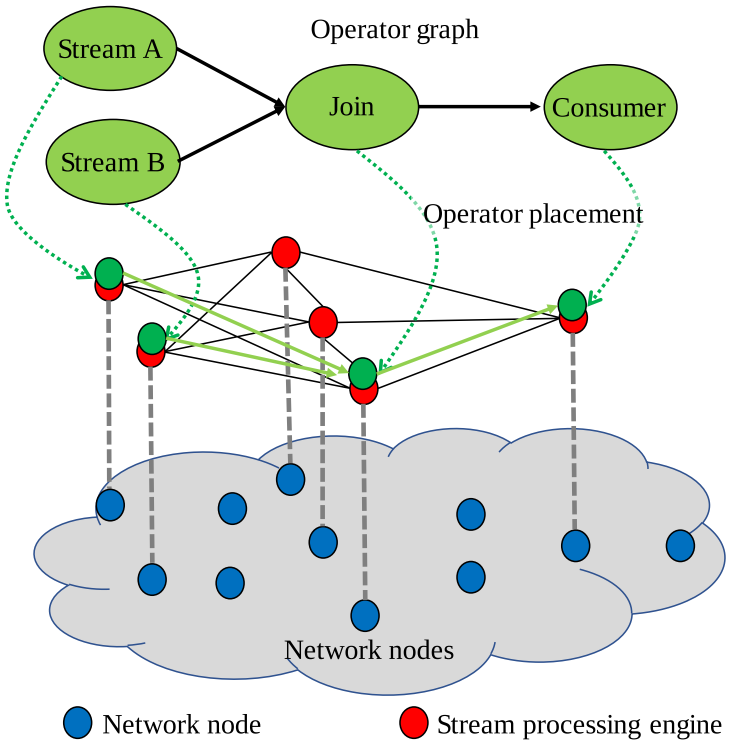


Figure IV.1: Overview of operator placement

workload (elasticity), (3) maintaining system operability even in the presence of hardware failure and other faults (fault tolerance), and (4) maintaining or optimizing the Quality of Service (QoS). Operator migration entails (1) state management to move the state of the operator from an old host to a new host, and (2) stream management to change data stream routing in the overlay network. Decisions on when to migrate the data and where to migrate them to are key aspects of operator migration. The potential approaches to state management, stream management, and decision-making as well as their combinations result in a large design space for operator migration algorithms.

This tutorial aims to give the reader a good understanding of (1) the need for operator migration, (2) the core elements of operator migration, (3) the design of existing solutions, and (4) how design decisions can impact the performance of operator migration solutions. To this end, we develop a conceptual model that captures the fundamental components of operator migration, i.e., the components on which all solutions are based and their relationships. This model provides a unified terminology and is used to establish a taxonomy of existing solutions. Based on this, we describe the main existing operator migration solutions.

Operator migration introduces some form of cost, like freeze time during migration or increased resource consumption to move the state of the operator. Keeping these costs low is a core requirement in the design of operator migration algorithms. Furthermore, during decision-making, it is important to balance the costs of migration against its benefits. There is a general awareness of this trade-off, but surprisingly, few studies have explicitly described how costs and benefits are considered in the migration algorithm and decision-making. Therefore, we place particular emphasis on costs and benefits in our analysis of work in this area. We structure the description of existing solutions into two parts:

- Mechanisms: How does the operator migration work, and which mechanisms are used?
- Policies: When should operator migration be performed, and how is the migration decision executed?

In addition to this functional view of operator migration, we perform an empirical study to gain and mediate quantitative insights into different operator migration and decision models. The aim is to illustrate the quantitative effect of different design decisions. This empirical quantification demonstrates the advantage of a comprehensive migration model beyond the contribution of the literature. We use Apache Flink [16] and Siddhi [129], two operator migration algorithms, and apply part of the NEXMark benchmark [134] as workload to measure the run-time performance.

IV.1.1 Tutorial Novelty and Contributions

To the best of our knowledge, this tutorial represents the first comprehensive effort to explain operator migration mechanisms and related decision-making in data stream processing systems. There exists a short description of a tutorial given in 2014 [46] by Heinze et al. However, due to space limitations the published version of the tutorial cannot be comprehensive and it can not capture developments after 2014. Therefore, this tutorial is unique in its scope and contribution to the current state of knowledge on the topic.

There is a range of surveys that cover operator migration [12, 13, 21, 51, 55, 65, 74, 111, 120, 133, 137] to a certain extent. However, all these surveys have a broader scope than this tutorial and do therefore not explore operator migration in corresponding depth and detail. For example, none of these surveys presents a type of framework for operator migration like a taxonomy or a conceptual model, and none of the surveys provides a unified terminology for operator migration. This tutorial distinguishes itself by presenting different types of operator migration algorithms in detail and the relationship between the cost and benefit of migration, the decision to migrate, and the migration algorithm. Furthermore, the surveys do not give the reader an insight into the quantitative impact of certain design decisions for operator migration.

Table IV.1 characterizes related surveys with respect to their focus area as well as the questions of whether:

1. any kind of framework, like a taxonomy or a conceptual model for operator migration is provided;
2. details of the decision-making process are given, such as the goal of migration, the performance of migration, and the cost of migration;
3. the survey methodology;
4. the deployment environment is considered in the discussion of existing solutions;
5. the paper uses some experimental investigations to demonstrate and quantify the effect of different design decisions.

Papers	Focus area	Framework for operator migration	Details of migration decision	Methodology	Deployment	Experiment
[65]	Placement for Internet-Scale stream processing	No	Partially	Explanatory	Yes	No
[55]	Elastic stream processing in the cloud	No	No	Enumerative & explanatory	No	No
[51]	Stream processing optimizations	No	No	Enumerative & explanatory	No	Yes
[133]	State management in big-data processing systems	No	No	Enumerative & explanatory	No	No
[111]	Adaptation of stream processing	No	No	Enumerative	No	No
[120]	Parallelization and elasticity in stream processing	No	Yes	Enumerative & explanatory	Yes	No
[74]	Resource management and scheduling in stream processing	No	Yes	Enumerative & explanatory	Yes	No
[13]	Geo-distributed big-data analytics	No	No	Enumerative & explanatory	Yes	No
[21]	Runtime adaptation of stream processing	No	Yes	Enumerative & explanatory	Yes	No
[137]	Self-adaptation on parallel stream processing	No	No	Enumerative	Yes	No
T	Operator Migration in stream processing	Yes	Yes	Enumerative & explanatory	Yes	Yes

Table IV.1: Summary of related surveys and comparison with this tutorial (T)

Lakshmanan et al. [65] focused on operator placement and reconfigurations for Internet-scale data stream systems. They distinguished between reconfiguration solutions based on where the change is made: either in the network, data, or flow graph. Moreover, different triggers for migrations were studied, such as thresholds, constraint violations, and periodic re-evaluations. However, they did not investigate the different varieties of operator migration in any detail. Hummer et al. [55] focused on elasticity in the cloud, which can be achieved through event reordering and prioritization, load shedding, deferred processing, and operator migration. They also investigated the state in different types of windows and how these have to be migrated in case of a scaling operation. Moreover, the cost of migration is also problematized, i.e., that performing a scaling operation costs time that might adversely affect the performance of the system. Similarly, Röger et al. [120] investigated elasticity and parallelization in stream processing. Operator migration is in this context one method to achieve elasticity, but details about operator migration mechanisms and migration decision-making are not given.

Hirzel et al. [51] cataloged different types of stream processing optimizations, including operator graph optimizations, operator placement, load balancing, state sharing, batching, load shedding, and several more. Operator migration is relevant for load balancing and operator placement, but the paper's aim is too broad to describe migration in the detail targeted in this tutorial. Microbenchmarks with InfoSphere Streams are used to demonstrate the profitability of the optimization, but operator migration is not experimentally investigated.

To et al. [133] studied state management in stream processing systems, with a focus on big data cloud-based systems. They investigated existing ways of representing state in the system, optimizing performance, and provide insights into various state management techniques. Operator migration, elasticity and load balancing are three of the 18 concepts of state management that are presented.

Similarly, Assunção et al. [12] explored migration in relation to stream processing and edge computing. They provided informative summaries of multiple generations of DSP systems and analyzed existing work on elasticity to adapt resource allocation to handle the workload of stream processing services. Operator migration is one of many means for elasticity and the inner workings of operator migration are not analyzed and described in detail.

Liu et al. [74] presented a taxonomy of resource management and scheduling in DSP. Operator migration and state management are not explored in the paper, as it is assumed that the mechanisms have been studied and are provided by the state-of-the-art systems. On the other hand, the decision-making process is investigated in depth.

Qin et al. [111] defined a taxonomy for different live reconfigurations in SPEs. This includes 17 types of adaptations, including operator migration, load balancing, and scaling. However, this tutorial investigates these three issues as fundamentally being similar types of adaptations. Furthermore, the survey [111] is of pure enumerative nature and does not aim to explain how operator migration works. Similarly, Cardellini et al. [21] presented a survey on run-time adaptations. They studied the methodological and architectural approaches for adaptation control and differentiate 14 adaptation mechanisms. Their presentation of adaptation goals includes a popularity analysis of metrics used for adaptation. In Section IV.5 of this tutorial, the metrics used by papers is discussed in depth. Bergui et al. [13] surveyed geo-distributed frameworks, some of which are described in this tutorial. Moreover, they discussed several challenges pertaining to geo-distributed data analytics, where operator migration plays only a minor role in some of the solutions.

Vogel et al. [137] presented a systematic literature survey of self-adaptation mechanisms of parallel stream processing. Operator migration is not explored in this paper, but a conceptual framework is proposed that includes adaptation goals and decision-making. The scope is much broader than this tutorial, and can therefore not go into the same depth in the related topics.

The main contributions of this tutorial are as follows:

- We propose a conceptual model of operator migration that provides a unified terminology and leads to a taxonomy of operator migration. Moreover, this model facilitates the development of new operator migration solutions.
- We describe the main works on operator migration and analyze not only current stream management and state management solutions (i.e., mechanisms), but also emphasize a cost-benefit analysis of the migration decision (i.e., policies).
- We perform an experimental study involving two migration algorithms on Apache Flink and Siddhi to gain insight into the quantitative aspects of operator migration.

IV.1.2 Literature search methodology

The conceptual model of operator migration has been created in an iterative manner using the existing works in the literature. The focus has been to select the works that describe their migration mechanism in detail, or how the migration decisions are made.

We have searched for existing literature in the most popular search engines, e.g., Google Scholar and Web of Science. The searches have included DSP and many keywords that relate to operator migration, elasticity, load balancing and fault tolerance. We have included works in the tutorial that have a substantial contribution to migration mechanisms or migration decision-making. This search is not straightforward, since sometimes, operator migration might be applied in a way where it is not the main contribution. Moreover, while some works categorize operator migration as a specific subset of big data adaptation techniques [111], our tutorial takes a broader perspective. It presents operator migration not as a specific adaptation, but as a crucial mechanism that enables other key features of big data adaptation, including load balancing, elasticity, QoS and fault tolerance.

Many works exist on migration of services in Multi-access Edge Computing (MEC) [24, 30, 35, 54, 82, 83, 85, 91, 102, 105, 117, 136, 141, 143, 144, 155, 156]. One of the core features of MEC is the ability to offload heavy tasks to a host with more resources or better conditions for completing the task [84]. The entities to migrate may be virtual machines (VM) [102], containers [82, 83] or Virtual Network Functions [155]. The methods proposed in these papers may also be applicable to operator migration when it comes to deciding when and where to migrate, which is discussed in Section IV.5. However, the migration mechanisms that have been developed specifically for DSP and described in Section IV.4, are different from the ones used when migrating services, since operators in DSP are more fine-grained. The migration entity is not an entire application, but rather an internal state. Whereas the entire service must be at the new host until the service can be restarted in the case of MEC, certain optimizations can be made in DSP, depending on what type of stateful operator is migrated.

Table IV.2 lists the studies considered in this work that form the foundation of the conceptual model. It classifies them according to the environment of their deployment and the goal of migration, which are important factors for the migration decision and placement. The most common deployment environments for DSP are cloud, fog, and edge networks. *Cloud* has been used to classify data center applications that might handle very high throughputs, and can scale the systems both horizontally and vertically to handle variable traffic loads. The pay-as-you-go business model makes the hardware provisioning easier [122]. The concepts of fog and edge are relatively new terms that seem similar, but have some significant differences. *Edge* computing often focuses on offloading heavy tasks from local resource-constrained devices to either a close base station or a data center [79]. With edge computing, heavy tasks, like deep learning computation and videogames, can be executed using edge devices such as smartphones and laptops [146]. *Fog* is an extension of the cloud in which the computing tasks of an application are distributed on multiple devices, including end devices, edge resources, and the cloud itself [99, 153]. As such, clients may send most information to a server close to them instead of a centralized data center to reduce energy consumption, congestion on the Internet, and response times for clients.

Table IV.3 lists the goals of migration and the overlap between studies in the area in terms of percentage. For instance, 40% of the studied papers on elasticity also consider load balancing. This is a common combination, because load balancing can be used after performing a scaling operation to redistribute the load. Few fault tolerance-based solutions describe migration mechanisms, but it is natural that fault tolerance overlaps with load balancing

Category	Sub-category	Papers
Deployment env.	Cloud	[15, 17, 19, 22, 28, 32, 33, 36, 38, 41, 45, 48, 52, 53, 56, 57, 66, 69, 73, 75, 76, 78, 86–88, 90, 93, 94, 118, 124, 131, 140, 142, 150, 154, 157, 158]
	Fog	[9, 18, 50, 60, 106, 113–115, 119, 145]
	Edge	[14, 23, 61, 72, 80, 101, 103, 104, 151, 159]
Migration goal	Load balancing	[12, 15, 25–28, 32, 33, 36, 42, 43, 56, 66, 68, 69, 76, 77, 81, 86, 88, 93, 101, 113, 114, 124, 131, 135, 140, 142, 145, 150, 154, 159]
	Elasticity	[17, 18, 25, 26, 28, 38, 41–43, 47, 48, 52, 56, 60, 68, 69, 72, 77, 78, 87, 93, 100, 119, 130, 149, 152, 154]
	Fault tolerance	[14, 28, 57, 72, 94, 142, 149]
	QoS	[19, 26, 28, 50, 60, 61, 71, 73, 77, 78, 80, 87, 90, 103, 104, 106, 108, 114, 115, 135, 145, 158]

Table IV.2: Overview of studies on the categories of operator migration

Migration goal	Load balancing	Elasticity	QoS	Fault tolerance
Load balancing	100% (33/33)	33% (11/33)	18% (6/33)	6% (2/33)
Elasticity	40% (11/27)	100% (27/27)	22% (6/27)	11% (3/27)
Fault tolerance	28% (2/7)	42% (3/7)	14% (1/7)	100% (7/7)
QoS	27% (6/22)	27% (6/22)	100% (22/22)	4% (1/22)

Table IV.3: Goals of migration and the overlap in studies in the area

or elasticity as they are often cloud-based solutions, and steps to restore the number of states of a node are similar to those of a scale-in operation. Approaches that use QoS constraints on operators to determine when to migrate, often also use load balancing. This is because a clear sign that workload rebalancing is necessary is when the QoS guarantees of an operator have been violated.

IV.1.3 Tutorial structure

Figure IV.2 sketches the structure of this tutorial, i.e., the sections and some of their content, and identifies the three core parts of the tutorial. Section IV.2 describes some basic concepts of distributed data stream processing. The first core part introduces in Section IV.3 a conceptual framework for operator migration. The two main concerns of operator migration, i.e., to move the operator state from one host to another host and to decide whether to migrate, structure the conceptual model as well as the other two core parts of the tutorial. Sections IV.4 and IV.5 form the second part of the tutorial. The aim of this part is to explain to the reader how the concepts are applied in existing research works to design operator migration algorithms (Section IV.4) and to perform migration decisions (Section IV.5). The third part, i.e., Section IV.6, follows a "hands-on" approach and aims to give the reader quantitative insights gained through empirical investigations. On the one hand, decision models for operator migrations are developed and analyzed through a use-case study, and on the other hand, two different operator migration algorithms are implemented, and their performance is analyzed through experimentation.

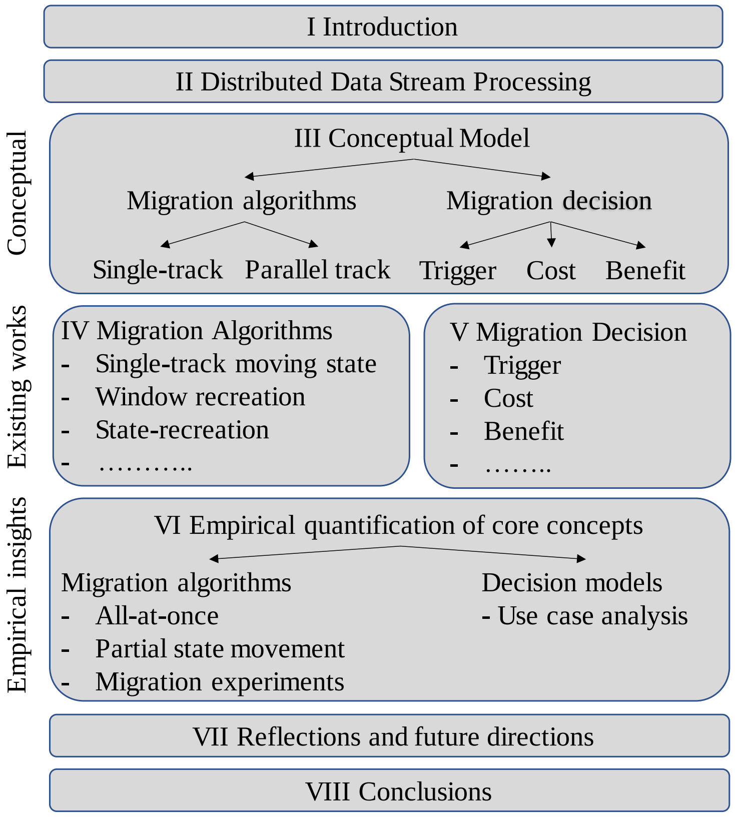


Figure IV.2: Paper structure

To ensure that readers from various backgrounds have a clear and unified understanding of the key terms used in this tutorial, we provide a glossary of critical terms in Table IV.4. These terms are integral to the discussion and understanding of migration algorithms, state management, and other aspects covered in this paper. The glossary covers terminology that is prevalent in the context of DSP systems, offering definitions aimed at beginners in the field. Readers are encouraged to familiarize themselves with these terms for a comprehensive understanding of the subsequent sections.

Table IV.4: Glossary of Terms

Term	**Definition**
Data stream	A continuous flow of data, typically consisting of a sequence of data tuples that can be processed in a streaming fashion, without the need to store everything in memory.
Downstream node	A node that receives data tuples from other nodes in a stream processing topology.
Elasticity	The ability to adaptively scale computational resources up, down, out or in, according to real-time needs.
Load balancing	Distributing computational tasks evenly across available nodes to avoid bottlenecks and maximize resource usage.
Migration trigger	An event or set of conditions that triggers operator migration.
Operator state	The temporary storage of historical data and intermediate results by an operator, essential for producing accurate processing outcomes.
Proactive migration	Triggering operator migration in anticipation of potential issues or changes in system requirements, rather than in response to them.
QoS (Quality of Service)	A set of performance metrics, such as latency, bandwidth, and availability, that the system aims to optimize.
Reactive migration	Triggering operator migration in response to current system conditions, such as failures or performance degradation.
SPE operator	A computational element that processes data streams, e.g., transforming, filtering, aggregating or joining the data.
Tuple latency	The time it takes for a data tuple to be processed or moved through the system.
Upstream node	A node that sends data tuples to other nodes in a stream processing topology.

In summary, the tutorial first introduces a conceptual model, presents and discusses afterwards design choices of existing works, and demonstrates in the last part the impact of particular design choices through empirical studies. The tutorial is completed through some reflections and discussion of future directions in Section IV.7 and the conclusions in Section IV.8.

IV.2 Distributed Data Stream Processing

IV.2.1 Data Stream Processing

Data stream processing deals with continuous processing of data tuples. The system model applied in this tutorial is based on [22].

A data stream is an unbounded sequence of tuples that are continuously generated over time. It is denoted as $S = t_1, t_2, t_3, ...$, where t_i represents the i_{th} tuple in the stream. For example, a data stream that represents stock market prices could be denoted as $S = t_1, t_2, t_3, ...$, where $t_i =$ (symbol, AAPL), (price, 150.23), (time, 2022-02-14 10:30:00), representing the stock symbol, price, and timestamp of the i_{th} price update in the stream.

A tuple is an ordered list of attribute-value pairs that represents a single unit of data. It is denoted as a set of key-value pairs, where the keys represent the attribute names and the values represent the corresponding attribute values. For example, a tuple that represents a person's information could be denoted as (name, John), (age, 30), (gender, male). Instead of including the attribute names in the tuples, a data stream expects the incoming tuples to follow a schema, and thus, the attributes are inferred.

A query is a function that processes a data stream and returns a result based on some criteria. It is denoted as Q(S), where S is the input data stream and Q is the function that defines the query. For example, a query that computes the average of a stream of numbers could be denoted as $Q(S) = (1/n) * \sum n_i = 1x_i$, where n is the number of elements in the stream, x_i is the i_{th} element in the stream, and $\sum$ is the summation operator.

Nodes in an operator network can be static or mobile, and have one or more of the following roles:

- *Data producer*: Examples of this include sensors that convert analog signals into data tuples, often with a fixed sampling rate, and software monitors that might create data tuples at a dynamic rate. Crucially, the operator network must be able to process all tuples produced by these sources for further processing.

- *Data consumer*: These are nodes that request a service, and typically have some QoS requirement, such as less than a given tuple latency.

- *Operator host*: These nodes execute at least one operator and contribute to event forwarding in the operator network, i.e., map the input events (from upstream nodes) of the operators they execute to output events, and forward them to downstream nodes in the operator network.

Data stream processing queries may be *stateful* or *stateless*. The focus of this paper is on the stateful queries. For instance, when joining two data streams, one tuple arrives before the other, and is placed within a data window, where it will remain until it expires and is deleted. When aggregating state, such as counting words, we are typically interested in creating an aggregate per group/key. In concurrent systems, each key produces output separate from other keys, and as such, these aggregates can be produced by different threads/processes. Therefore, it is common to parallelize such queries, and execute some keys on one host and other keys on another host, in a cluster.

Numerous data stream processing systems exist, including but not limited to Storm [2], Flink [16], Esper [31], and Siddhi [129]. For a more comprehensive list, please refer to the survey by Isah et al. [58]. In Section IV.6, we run real-world experiments with Siddhi and Flink. As these systems are difficult to execute, there have been efforts to simplify the interface for running such systems. Apache Beam [3] is a system that provides a unified interface to existing stream processing systems, where each supported system needs a runner that represents the integration between Beam and a given system. Expose [138] is a stream processing evaluation framework that provides an easy interface for running distributed experiments with any distributed stream processing system that has a wrapper, that implements an API that represents the core functionality of the system. There is also an interest in simulating these systems, and efforts have been made in DCEP-Sim [128] and ECSNeT++ [11].

When it comes to simulation of fog and edge computing that model more generic services, with mobility and service migration, there has been a range of simulators, including, CloudSim [40], iFogSim [44], iFogSim2 [89], EdgeCloudSim [125], FogNetSim++ [110], IoTSim-Edge [59], MobFogSim [109], YAFS [67], PureEdgeSim [95], IoTNetSim [123], SatEdgeSim [147], and IoTSim-Osmosis [10]. These simulators, however, do not focus on data stream processing, and are often more focused on the interactions between edge, fog, and cloud nodes, and using generic services, instead of specific operators, which are necessary in data stream processing.

IV.2.2 Initial placement

A DSP can be considered to be a set of collaborating SPEs that form an overlay network to process queries over data streams. Consider Figure IV.1 as an example of such an application. SPEs run on network nodes that provide the computational and networking resources for the DSP overlay. The objective of the initial operator placement is to distribute the processing of a query over network nodes such that the goals of the system can be met as adequately as possible [20]. The first step is to transform a query into an operator graph. An operator graph can be modeled as a DAG in which the operators derived from the query are represented as vertices. The placement of these operators in a network, i.e., finding appropriate network nodes to host the operators, is typically driven by an objective function. Such an objective function typically includes (contradicting) criteria of optimization, like low latency of event delivery, low resource consumption (e.g., bandwidth and energy), reliability, and fault tolerance. Typically, a placement function is used to calculate a placement score based on the criteria of optimization. To find the optimal placement is usually an NP-hard problem [20], and heuristics are often used to find close to optimal solutions.Both centralized and decentralized versions of operator placement can be used to establish an operator network, and are generally implemented as an overlay for the DSP.

DSP is performed in a dynamic context involving variable workload, resource availability, and possibly mobility. As such, initial placement might, after some time, become sub-optimal and the operator network should be adapted by migrating one or several operators to a new host.

IV.2.3 Naïve Migration Example

To illustrate the challenges of state migration, we go through the simple example introduced in Figure IV.1. Consider a data stream processing application that involves three entities: the data producers, the operator hosts, and the data consumers. The data producers may include sensor nodes that produce data. This data is sent to the operator hosts to process the tuples, i.e., to transform, filter, aggregate, and join the data. Data with a specific schema is called a data stream. Different data streams have different attributes in them.

In this simple naïve migration example, a node gets overloaded and has to migrate stateful operators to another node. We use a join query that joins all Bid events with their corresponding Auction event. This query is based on the NEXMark benchmark [134], and is also applied in Section IV.6 for the empirical quantification of the conceptual model. As Auction tuples arrive, they are stored as part of the state in the join operator until expiration. In this way, incoming Bid tuples can be matched against Auction tuples.

```
select A.id, B.amount
from Auction A
join Bid B on B.auction = A.id
```

In Table IV.5, four Auction tuples are described that are sent before the migration starts. Table IV.6 lists the Auction and Bid tuples that are sent after the migration. If no state is migrated, the output will only be one tuple,

Table IV.5: Input before migration

Stream	Attribute 1	Attribute 2
Auction	id: 1	name: Mona Lisa
Auction	id: 2	name: The Scream
Auction	id: 3	name: Salvator Mundi
Auction	id: 4	name: Orange Marilyn

Table IV.6: Input after migration

Stream	Attribute 1	Attribute 2	Attribute 3
Bid	Auction: 2	amount: 105M	bidder: 7
Auction	id: 5	name: Spring	
Bid	Auction: 3	amount: 400M	bidder: 4
Bid	Auction: 1	amount: 800M	bidder: 12
Bid	Auction: 4	amount: 190M	bidder: 1
Bid	Auction: 5	amount: 70M	bidder: 4

Table IV.7: Output with stateless migration

Stream	Attribute 1	Attribute 2
Output	Auction: 6	bid: 70M

Table IV.8: Output with stateful migration

Stream	Attribute 1	Attribute 2
Output	Auction: 2	Bid: 105M
Output	Auction: 3	Bid: 400M
Output	Auction: 1	Bid: 800M
Output	Auction: 4	Bid: 190M
Output	Auction: 5	Bid: 70M

shown in Table IV.7. If the Auction tuples from before the migration are migrated, the output will be IV.8. The reason why so many more tuples are produced after the migration when Auction tuples are migrated is that the incoming Bid tuples find a matching Auction tuple to join with. Without them being migrated, only the new Auction tuple with id 5 can be joined with. This is a simple example that shows the necessity of state migration. The migration mechanisms that are introduced in Section IV.3.1, and described more extensively in Section IV.4, face the same problem of making the operator state available on the new host, such that the operators produce the correct output.

IV.3 A Conceptual Model of Operator Migration

We establish a conceptual model of operator migration to capture the basic concepts and elements on which consensus has been achieved in the literature, and form a unified terminology for operator migration. For the understanding and the design of operator placement, it is important to separate between mechanism and policy. There are two major concerns for operator migration mechanisms: (1) stream management to stop, buffer, redirect, and start streams; and (2) state management to establish the current state of the operator at the new operator host, which may require moving the state from the old host to the new one, and starting a replica for the operator on the new host before the state transfer is finished. All algorithms require some stream management functions, such as stop, start, buffer, and redirect. State management stands apart due to the multitude of design alternatives it presents. For example, one could choose to move the entire state at once or incrementally. Furthermore, it is possible to execute the operator exclusively at one host during migration, or to do so in parallel on multiple hosts. The policy is implemented in the operator migration decision component that needs to determine whether and when to migrate an operator. This involves several steps. Migration decisions first require a trigger for when to make a migration decision and then a placement mechanism to determine the placement that yields the best performance. The cost of the migration must be weighed against its benefit. The policy must determine the degree to which the migration decision should be proactive (e.g., before a host becomes overloaded) or reactive (e.g., when a host is overloaded). The more proactive a migration decision is, the higher uncertainty it has. The more reactive a decision is, the higher cost of migration it has.

Figure IV.3 illustrates the concepts and building blocks that make up migration algorithms. It also highlights the relevant decision-making processes, outlines the properties of state management mechanisms in migration, and presents the associated costs of migration. Migration cost is important for migration mechanisms and migration decisions, because every migration mechanism introduces some form of costs and the migration decision needs to take the costs into account to determine whether it is worthwhile to migrate. This relationship between the migration mechanism and the migration decision makes the migration cost a core element of the conceptual model. The state management node describes the dimensions of migration mechanisms that are explored in this tutorial.

The remaining structure of this section directly reflects the structure of Figure IV.3, i.e., a detailed discussion of the components and design alternatives for migration mechanisms is given in Section IV.3.1, followed by an overview of the common cost parameters in Section IV.3.2, and, in Section IV.3.3, the migration decision.

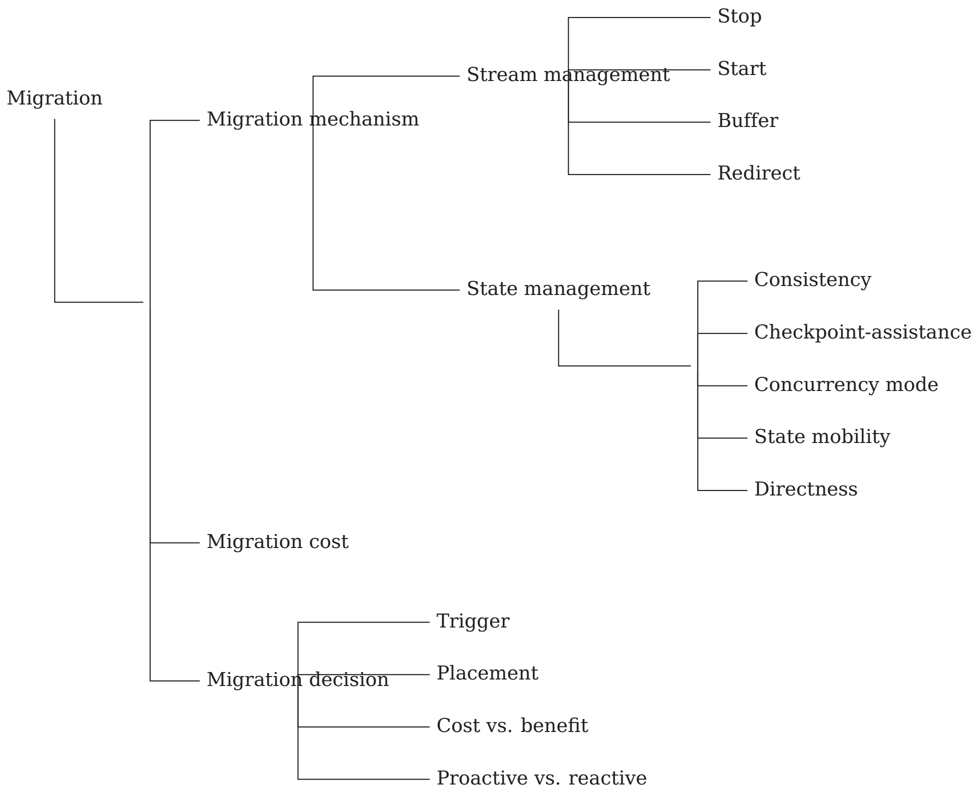


Figure IV.3: Concepts of migration

IV.3.1 Migration mechanism

The two major concerns of migration mechanisms are state management and stream management. State management is relevant for operators that derive their output based on multiple tuples, e.g., looking for a sequence of tuples using CEP, joining streams, or aggregating tuples over windows [133]. The *state* can be thought of as tuples. The internal state of the operator is, in practice, typically optimized to include only the necessary information for the given operator, such as the given aggregate value for the extent of a window, or as a finite state machine in CEP. In addition to such stateful operators, there exist also stateless operators, like filter and map. These operators do not require state because they process each input tuple independently. Therefore, operator migration distinguishes itself markedly from VM migration, where the entire VM must be transferred to the new host. While some solutions to operator migration, such as the MCEP [103], do include VM migration, VM migration is not addressed in this tutorial. The simplest method of operator migration for a stateful operator is to move it to the new host and replay all necessary historical tuples from the upstream nodes [64]. This technique is used in current publish-subscribe systems, such as Kafka [4], to achieve fault tolerance in stream processing systems like Flink [16]. Using this technique also makes it possible to migrate the data to a different stream processing system, which is usually not possible when extracting the state from the system, because the internal state is system specific. However, as the state can become very large, it is often undesirable to replay all tuples. Therefore, this tutorial focuses on operator migration techniques that extract the state from the stream processing system and move it to the new host.

The purpose of state management in operator migration is to establish, at the new host, an operator with the state of the operator at the old host when switching the processing from the old host to the new host. In a *moving state* algorithm, the old host extracts the state of the operator and sends it to the new host. Some algorithms do not need to perform this task, either because they manage stateless operators, e.g., filter operators, or because the old and the new host can schedule a seamless handover of the operator. In a *parallel-track* algorithm, originally a term used by Zhu et al. [160], both the old and new hosts receive the same tuples for some time during migration. The handover from the old to the new host is carried out gradually such that the downtime of the operator is minimized. The cost of this approach is that upstream nodes must send twice as many tuples during some part of the migration. A parallel-track algorithm with moving state is called *state-recreation*, and one without moving state is called *window-recreation*. These terms are inspired by StreamCloud [42]. In a *single-track* algorithm, the upstream nodes send tuples either to the old host or to the new host.

Stream management deals with notifying upstream and downstream nodes of changes made to the DSP overlay. Typically, nodes have to update their routing table to reflect the new topology at the upstream node, and this results in a redirection of the outgoing stream to the new operator host. To prevent tuples from getting lost when the operator is down, streams might be stopped and tuples need to be buffered. There are three locations at which tuples can be buffered: upstream nodes, the old host, and the new host. The tasks of redirecting streams, stopping streams, buffering streams, and restarting streams are coordinated among the hosts involved through control messages. Both centralized and decentralized coordination is possible. As such, there are several design options that can be implemented for a particular operator migration solution.

For presentation purposes, the taxonomy is divided into two parts: single-track algorithms (Figure IV.4) and parallel-track algorithms (Figure IV.5). In single-track algorithms, each tuple is processed either on the old host or the new host, but not both, at any given time. This means that even if an operator is active on both hosts during the migration, each individual tuple is processed exclusively on one host or the other. In contrast, parallel-track algorithms involve tuples being processed on both the old and new hosts simultaneously during the migration period, meaning the same tuple is processed on both hosts.The small text in brackets under some of the categories denotes a term for the given type of algorithm. For instance, a *pause-drain-resume* algorithm is a single-track algorithm without moving state, and a parallel-track algorithm with moving state is a state-recreation algorithm. The most basic operator migration algorithm is a pause-drain-resume algorithm, and works only with stateless operators or in cases where some state inconsistency is permitted. The operator to migrate is first started on the new host while the old host is also running it. Then upstream nodes redirect their output streams from the old to the new host. After this, the old host can stop the execution of the operator. Since no state needs to be moved, migration occurs without any downtime. A few control messages must be sent (1) from the controller to the old host, (2) from the old host to the new host, and (3) from the old host to the upstream nodes. As there is no downtime for the operator, any delay caused by these messages is negligible.

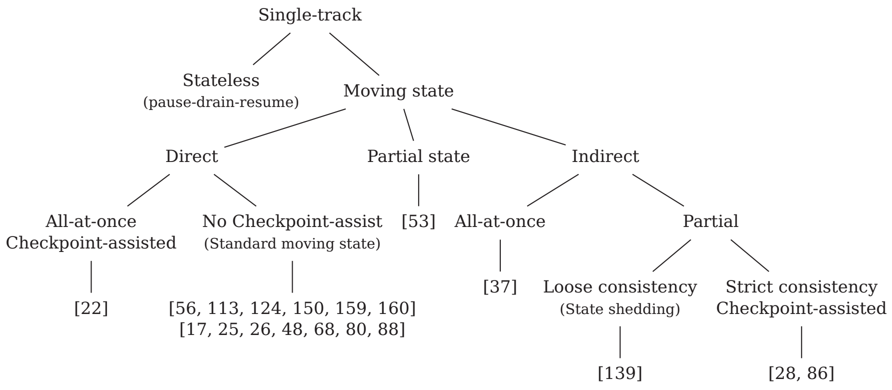


Figure IV.4: Single-track migration algorithms

When state must be moved and a single-track is used, the operator has some downtime. Specifically, tuples can neither be processed on the old nor the new host while the state is in transmission. In this process, tuples from the upstream nodes must be buffered before the new host can process them. The buffering can be carried out on the upstream nodes, the old host, or the new host. In many cases, tuples can be received after query processing has been stopped. These tuples need to be forwarded from the old host to the new host.

Partial state movement involves splitting the state to be migrated into several parts and moving these parts to the new host while the operator is still processing on the old host. This approach avoids having to stop operator processing for the entire state transfer. If the state is periodically

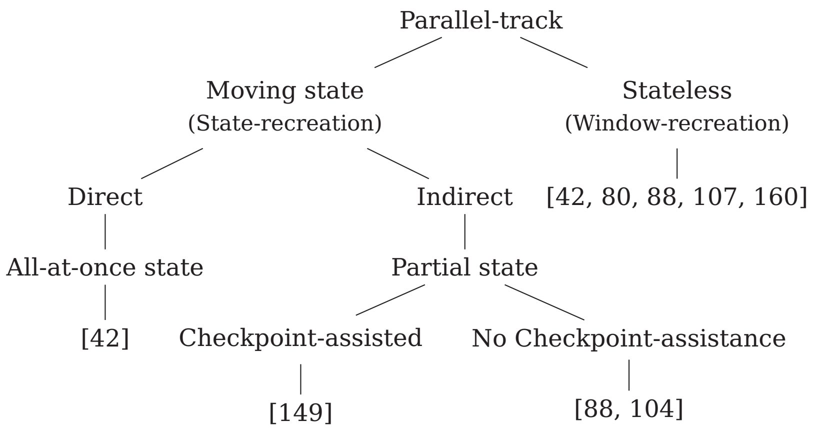


Figure IV.5: Parallel-track migration algorithms

checkpointed and distributed on different nodes, this is called *checkpoint-assisted migration*, and can substantially reduce or eliminate the downtime of the operator. Either the entire state already exists on the new host or an incremental checkpoint is extracted before the operator shuts down, and then sent to the new host. While the last checkpoint is sent to the new host, the operator stops for a much shorter time compared to sending the entire checkpoint at once. A single-track moving state solution can never avoid downtime. This is the reason why parallel-track solutions have been developed.

Parallel-track algorithms differ from single-track algorithms in a fundamental way. They can achieve zero downtime, but at the cost of running the old and new hosts with duplicate input streams and, sometimes, duplicate output streams [149]. A moving-state parallel-track algorithm performs state-recreation, which means that the new host receives the state from the old host while also receiving the same tuples as it does from upstream. A parallel-track algorithm without state migration performs window-recreation, which means that the new host receives the same tuples as the old host until the old tuples expire and they both have the same tuples in their windows. At this point, the upstream nodes redirect their streams to the new host, and it takes over without the tuples being buffered or any waiting time.

Most of the existing works assume fully consistent state for the operator migration. This means that before and after the migration, the internal state of the operators looks like it would if there was no migration, except that some state might arrive in different order. No state is lost. This is an important principle for adaptive stream processing systems; that adaptations occur transparently to the data producers and consumers. In a recent work [139], however, state shedding is presented as the idea of performing a migration where the most important partial states are migrated, and some less important partial states are dropped if the total state is too big to migrate. This is meant to be used in volatile cases where the system fails unless an adaptation is done quickly.

An important motivation for establishing the terminology and building blocks in Figure IV.4 and IV.5 is that existing work has described the same concepts by different names. For instance, what Zhu et al. [160] called *parallel-track* is described as *window-recreation* in StreamCloud [42], *smooth migration* in Enorm [88], and the *seamless minimal state* in TCEP [80]. In StreamCloud, a different algorithm called *state-recreation* is also parallel-track, but also involves moving state. In contrast to parallel-track, single-track with moving state is called *disruptive migration* in Enorm [88] and *Pause & Resume* in [46] because it leads to downtime, as opposed to smooth migration that eliminates downtime. Instant migration [88] is single-track migration without moving state. Checkpoint-assisted algorithms have been described in [28, 86, 149]. A characteristic of these algorithms is that a minimal state needs to be sent during migration.It should be noted that what is considered state differs among different systems. Therefore, what is considered partial state or all-at-once state might differ. TCEP has a fine-grained migration algorithm that moves during operator migration the entire tuple state all at once, but since the entire operator consists of multiple elements where the tuple state is just part of it, it is not considered a partial state movement algorithm.

IV.3.2 Migration cost

Operations performed as part of state management and stream management lead to two classes of the cost of migration, related to resource consumption and temporal aspects. The temporal costs are caused by the fact that the operator is not operational during state extraction, state serialization, state movement, state deserialization, and runtime initialization. The bandwidth required to move the state from the old to the new host is the most commonly considered resource in resource-aware geo-distributed cases [13]. The computational requirements of extracting the state from the old host and the messages needed to coordinate stream management are more commonly considered in centralized data centers. Stream management messages may also have an impact on the operator downtime, e.g., streams from upstream nodes need to be stopped and no events should arrive at the operator until they have been redirected and started again. The operator is further suspended during state extraction at the old host, moving the state from the old to the new host, and installing it at the new host.

Two metrics are used to assess temporal cost: *freeze time* and *latency spikes*. Freeze time quantifies the duration for which an operator cannot work, i.e., freeze time = t_{start} - t_{stop}, where t_{stop} is the point in time when the old host stops the operator and t_{start} is when the new host resumes it.

Latency spikes quantify the increased latency of event delivery caused by a non-working operator. It is often approximated by the time needed for state movement, which is the duration for which the state is in transit between the old and the new host, i.e., state movement time = $t_{receive}$ - t_{send}, where t_{send} is the time at which the old host starts sending the state, and $t_{receive}$ is the time at which the new host has received the entire state. The state movement time depends on the size of the state and the available bandwidth between old and new host. Thus, state size can be seen as related to the costs of both resources and time. Existing research is largely concerned with the tuple delay of a placement [159], but tuple latency caused by migration has not been given the same priority.

It should be noted that latency spikes reflect the cost much better than freeze time since it is possible for the operator not to produce any event during the freeze period. Examples of such a case are incoming events during this period that do not match the pattern that can trigger the operator to produce an event, or if a tumbling window implemented by the operator is much larger than the migration time such that all delayed incoming events can be processed on the new host before the window expires.

From the descriptions of the different types of algorithms above, it is easy to see that they differ in the cost of migration. Operator downtime or the latency of the output tuple can be considered a reasonable definition of the cost of migration for single-track state movement algorithms. However, for parallel-track algorithms, this definition of cost can result in excessively frequent migrations, as it typically results in a value close to zero. Therefore, it is necessary to define the cost of migration in such a way that the migrations do not become too frequent.

With parallel-track, operator replicas need to be executed during migration, and upstream nodes must send duplicate streams to the old and new hosts. They may also take a significant amount of time to execute when using window-recreation [42], which, in addition to using operator replicas during this time, might result in a significant increase in monetary costs. Therefore, it makes sense to consider the monetary cost when using parallel-track algorithms.

IV.3.3 Migration decision

To perform the migration decision several steps are necessary (see Figure IV.6). First, the decision process needs to be triggered. Then, a better placement has to be selected. The cost and benefits of a migration to the host must be estimated and compared in order to determine whether it is worthwhile to migrate.

Most studies have handled migration as part of an adaptation mechanism, where the goal is to improve execution or recover it in case of node failure. Regularly collected metrics can be used to indicate the need for an adaptation.

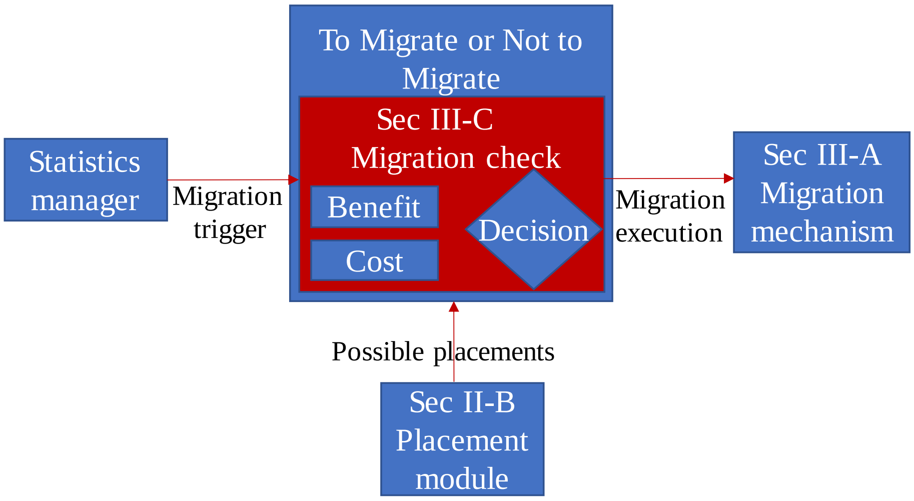

To Migrate or Not to Migrate
Sec III-C Migration check
Benefit
Cost
Decision
Statistics manager
Migration trigger
Migration execution
Sec III-A Migration mechanism
Possible placements
Sec II-B Placement module

Figure IV.6: Migration decision-making

Recent surveys have focused on adaptation mechanisms [34, 111], while this tutorial focuses on operator migration. Migration is usually the most costly aspect of an adaptation, and this perspective can be useful for better understanding adaptations. Even though adaptations differ in some aspects, they share major parts in terms of cost.

IV.3.3.1 Migration goal

This section describes four of the most common goals of migration: *load balancing* to distribute the load evenly on the available nodes, *elasticity* to efficiently leverage computational resources, *fault tolerance* to ensure that the DSP system can continue processing in the event of failures, and improving the QoS. This is not a comprehensive set of migration goals but it constitutes the main categories. For instance, security can be another reason for making adaptation changes. However, the general migration goal is to improve performance of the system. For instance, a DDOS attack might cause a migration, but this would also be addressed using QoS as the migration goal.

While all goals of migration can be applied to any deployment environment, the solutions with load balancing and elasticity are mainly aimed at cloud-based DSP systems and executed within a single data center, whereas QoS optimization is normally carried out when an operator undergoes back-pressure and needs an adaptation to improve the QoS, which can happen in any deployment environment. While migration is relevant for fault tolerance, few solutions describe it as a mechanism to facilitate reliable execution. Instead, solutions often use an upstream rollback approach [64] that replays to the new host tuples that are part of the failed operator. Below, we analyze all the goals of migration except fault tolerance.

QoS-driven migration is employed to enhance the system's QoS. Crucial QoS metrics in operator migration include bandwidth, availability, latency, throughput, and more. For instance, to maintain the goal of low latency in mobile settings, the system works to position the operators close to the data producer. This can be seen as a broad optimization problem, with the objective of maximizing the selected QoS metrics, as depicted in Equation IV.1. Later, in Section IV.5, we delve into which of these metrics are frequently taken into account.

$$\max \quad \sum_{i=1}^{n} QoS(i) \tag{IV.1}$$

Another important parameter in mobile settings is energy preservation for resource-constrained nodes [23, 127]. In a cluster setting, the goal is often to ensure that the nodes are not overloaded and that the latency of the tuple is not too high. If the latency of an operator increases significantly, it might be migrated to a node that can provide lower latency.

The basic goal of migration is to improve the QoS. Most solutions are more specific about the goal of migration because finding the optimal solution is usually an NP-hard problem [20], which is unfeasible to solve for networks of most sizes. A simpler approach is to add constraints to the operator. If the operator cannot fulfill these constraints, it must be relocated. This is typically a much more scalable solution that looks for a placement that is good enough, instead of looking for the optimal solution. It is a push-based manner of letting the coordinator know when the operator needs to be relocated. It should be noted that constraints or thresholds are also often used to achieve the other goals of migration. One characteristic of QoS-based migration is that it is mainly related to the migration of individual operators.

Load balancing is a necessity in distributed streaming systems, because the workload might vary significantly over time, leading to unbalanced distributions of state over the processing nodes. The coordinator should monitor the resource usage on the nodes to ensure that neither the network nor the CPU resources become bottlenecks for the performance of the operators. If resource usage on the nodes is unbalanced, the coordinator moves some of the tasks among the nodes. If these are stateful processes, the tasks to be moved must be paused, moved, and restarted on the new node. Load balancing-driven migration differs from QoS-based migration in the sense that the data consumers do not necessarily benefit much from the balancing, and in that multiple operators are usually migrated through load balancing. However, the decision on when to perform load balancing and where to migrate operators must still take into account the same concerns as for constraint violation, i.e., whether the cost of migration is worth the benefit of the new placement.

Load balancing is typically modeled as a resource scheduling problem, where the goal is to distribute the load as evenly as possible. For instance, minmax the latency on the nodes [131], as shown in Equation IV.2.

$$\min \quad (\max \quad l(G_i)) \tag{IV.2}$$

Elasticity refers to adding or removing operator replicas that facilitate parallel processing (also called operator scaling). For instance, a query with stateful windows that are grouped by a key can be run in parallel in multiple threads, where each thread is responsible for a subset of the keys. Four scaling operations are commonly used:

- *Scale up*: Create a new process and migrate some partitions of existing threads to it.
- *Scale down*: Migrate all partitions of a thread to the other threads and shut it down.
- *Scale out*: Create a new worker to which some threads can be moved.
- *Scale in*: Remove a worker and move its threads to existing workers.

In a cloud setting, the scaling out of a streaming system means adding more servers to a cluster. The streaming system then automatically decides which operators to move to it and potentially scale up. Scaling in means the opposite: A server is removed from the cluster. First, all the server's operators are moved to other servers and some scale-down operations might be performed. Scaling in and out can be modeled as special cases of load balancing. When scaling out, a new container or VM is started on a new machine, which is then added to a load balancing pool. The load balancer can then use this new machine for load balancing. When scaling in, a machine is eliminated from the load balancing pool, and at least its own state must be migrated to the other nodes.

IV.3.3.2 Triggers

To determine whether migration should be performed, it is necessary to compare the current placement with an alternative placement to estimate the benefits of migration. If these benefits are significantly greater than the costs, migration is beneficial. However, the calculation of a new placement, its benefits, and the related costs might require a non-negligible amount of resources. As such, the naïve approach to scheduling a migration decision with a fixed frequency might be too costly. Instead, some form of context awareness needs to be supported to detect changes in the system (e.g., related to workload, resource availability, or mobility) that indicate that there might be a good chance of determining a better placement. The relevance of such changes is generally implied by the goal of migration. Monitoring the runtime system is an important task to detect such changes. The DSP system can also perform some book-keeping, like the number of operators a node hosts, and trigger a migration decision if a threshold is reached.

A simple trigger is a constraint or threshold. For instance, load balancing systems may make balancing decisions when the load imbalance of the systems is above a certain threshold. For elasticity-based solutions, checks on whether to scale in or out are similarly performed using thresholds. If the system has a balanced load and its use is still above a given threshold, the system might decide to scale out. If the utilization is below a threshold, the system can scale in. If an operator has latency constraints that are not fulfilled, the coordinator can be notified that migration must occur. The coordinator can either be the node hosting the operator in a decentralized solution or a centralized controller in a data center. In all scenarios, a unit collects metrics from the runtime system in order to make a decision.

IV.3.3.3 Timing decision

Migration decisions can be made reactively or proactively. In the former case, a system migrates when the given situation calls for a change to be made, such as when QoS guarantees for an operator are not fulfilled. Proactive migration decisions rely on predictions about future changes that require migrations.

In several cases, the need for migration scales with its cost. For instance, if the migration is triggered when the tuple rate exceeds a limit and causes QoS violations, more tuples are affected by operator downtime when the need for migration is more pressing. In other words, the more pressing the need to migrate is, the higher the cost of migration is. If a node is over-provisioned, and cannot handle a higher input rate for a given operator, the operator benefits from being migrated to another node. If this situation is detected when the input rate is already too high, a potential migration results in latency spikes for the affected tuples. However, if it is possible to predict that the tuple rate will increase, one can reduce the cost of migration by proactively migrating before the tuple rate becomes too high.

The cost-benefit analysis for making migration decisions is not trivial as the cost of migration is a one-time investment and the benefit from better performance is accumulated over time. When confronted with dynamic surroundings in stream processing scenarios, it makes sense to consider a given placement only for a given amount of time. This time can be regarded as the horizon for which predictions are made. Migration decisions are then made in such a way that the new placement amortizes cost during that time. As such, this time horizon is called amortization time. The notion of working with a limited future horizon for making optimization decisions is also used in model predictive control (MPC), and has been applied by De Matteis et al. [26] to make proactive scaling decisions.

The higher the number of tuples that are impacted, the more the migration option is penalized. However, the number of tuples impacted is an estimate that depends on the accuracy of the prediction. It is possible to assume that tuples are sent evenly across the time window of the horizon, in a single burst, as fast as possible, or a mix between the two. To make such predictions, it is necessary to collect metrics from upstream nodes to determine the density of distribution of the transmitted tuples.

IV.3.3.4 Cost versus benefit

Once the decision process has been initiated, it is necessary to determine a better placement and relate its benefits to the costs of the migration to determine whether to migrate [55]. One clear approach to calculating a new placement is to re-run the original placement algorithm with the same objective function. Some of the data needed for calculating a new placement might be available from the monitoring component that triggers the migration decision. In most cases, additional live data must be collected, where this represents a substantial part of the overhead of making the migration decision.

The gain in performance owing to a new placement is generally reflected in the output of the objective function of the old placement versus that of the new placement [20, 23, 126]. By optimizing the objective function during placement, a new placement that delivers the best performance is identified. The problem with simply migrating to the host with the best performance is that the cost of migration might be so high that it is not worth migrating. It might be that a sub-optimal placement is preferred in terms of the objective function owing to a lower migration cost, or maybe that no migration is worth it at all. What makes the comparison of cost and placement performance challenging is that they are not directly comparable. On the one hand, multiple, possibly contradicting, metrics can be used to determine cost and performance. On the other hand, the cost of migration is a one-time investment while the performance of a placement represents how a placement performs over a certain amount of time. The placement performance continuously increases the overall benefit as long as there are no changes in the system. As such, there is a need to distinguish between the benefits of placement and migration. The *benefit of placement* simply expresses the difference in placement scores between a new host and the given host, while the *benefit of migration* is calculated based on (1) the cost of migration, (2) the placement performance, and (3) the amortization time.

Three common ways to avoid excessively frequent migrations have been discussed by Lakshmanan et al. [65]: (1) A threshold to ensure that the score of the new placement is significantly better than that of the current placement. (2) If the QoS guarantees of an operator are violated, it triggers a migration, which means that migrations are performed only when necessary. (3) Periodic re-evaluation of the objective function where the interval is set to be reasonably high. In a more recent example, Buddhika et al. [15] regularly calculated interference scores of operators that describe the need for migration, and migrated them to a node where they were subjected to less interference. However, neither Lakshmanan et al. [65] nor Buddhika et al. [15] performed an explicit cost-benefit analysis. This is of interest to us not only to avoid excessively frequent migrations, but to understand why migration is worth it in some cases and not in others based on its costs and benefits. Suppose a placement is an improvement over the given placement. In that case, we want to be able to state exactly why the migration is worth performing (or not) in a meaningful and understandable way. The amortization of the cost of migration is a simple goal to understand as long as one weighs the one-time cost of migration against the benefit of the continuous performance of the new placement, but this deliberation is often not presented explicitly in existing works.

IV.4 Migration Mechanisms

In this section and Section IV.5, we present an overview of existing literature on operator migration. Specifically, we examine the design of current migration strategies, with a focus on those that assume full consistency of the operator state.

As the volume and velocity of data have increased with the emergence of big data [7], the simple single-track moving state algorithm has become inadequate. Specialized and innovative solutions that provide no downtime and solutions that leverage fault tolerance mechanisms, such as periodically performed back-ups, have been designed. We explore the state-of-the-art migration algorithms and provide a historical perspective on innovations proposed.

Migration mechanisms are characterized by their state and stream management. This involves executing certain tasks, such as redirecting, buffering, pausing streams, and moving states between nodes. Moreover, it is important to specify whether these tasks can be executed in parallel and where it is most beneficial to execute them. The most important properties identified in Section IV.3.1 are whether the algorithms require state migration and how this is performed, and whether they are single-track or parallel-track. Most of the investigated migration mechanisms can be derived from these properties. For instance, some mechanisms are centralized, and rely on a coordinator, such as [28, 42, 124], whereas others are decentralized and initiate migration on the operator host, e.g., [104, 113]. In some cases, multiple dependent migrations are planned and performed in sequence, but the details of managing multiple migrations are not presented in this tutorial. Examples of such algorithms include load balancing, where many keys of an operator may be moved to a new location, and when an operator graph is distributed geographically and several operators are migrated, e.g., in TCEP [80].

The most fundamental mechanisms are single-track without state migration, single-track with state migration, and parallel-track without state migration, i.e., window-recreation. These mechanisms were introduced together by Zhu et al. [160] and were later applied to the SPE CAPE [118]. The authors discussed the steps of migration and cost models of the different mechanisms. They called them moving state, parallel-track, and pause-drain-resume migration mechanisms. Using the terminology established in Section IV.3, the moving state mechanism is single-track moving state, the parallel-track mechanism is parallel-track without state migration, and the pause-drain-resume mechanism is single-track without state migration. The paper by Shah et al. [124] forms the basis for load balancing, and presented a means of repartitioning keys in a key-value-partitioned operator state, which is relevant for cluster-based systems. We characterize this mechanism as a single-track moving state algorithm, but in which the operators are already running on the destination node. In contrast to some studies, for instance by Qin et al. [111], we do not consider state movements in load balancing and operator migration to be fundamentally different, and posit that only the entities being migrated are different, i.e., keys are moved instead of operators. In load balancing, the entities being migrated are often a set of keys and their associated states, whereas in operator migration, the entities are usually an operator and its associated state.

IV.4.1 Mechanism descriptions

This section describes the relevant mechanisms in a concise and systematic manner. Since details of what happens in migration mechanisms are typically omitted from research papers, our descriptions may deviate to some extent from the original implementations of the migration mechanisms considered. For the most significant variations of these mechanisms, we show how migration is performed using a figure that illustrates the topology of stream processing and the communication between nodes. Please refer to the legend in Figure IV.7 for the description of symbols that are used in this section to describe the migration mechanisms. The following types of nodes are used: old host (OH), new host (NH), upstream nodes (US), and downstream nodes (DS). The upstream node and downstream node can both represent one or more nodes, but for the sake of simplicity, only one of them is shown in the figures. Each figure is accompanied by an enumerated description of the steps of the relevant algorithm on the right-hand side, and each step is provided in the figure to facilitate the understanding of the algorithm. Furthermore, we show in subsequent listings the contents of the control messages sent to provide the reader with a kind of computational viewpoint of the involved nodes. Since the control messages comprise the tasks that must be executed by the nodes, there is a natural correspondence to the steps listed in the figure.

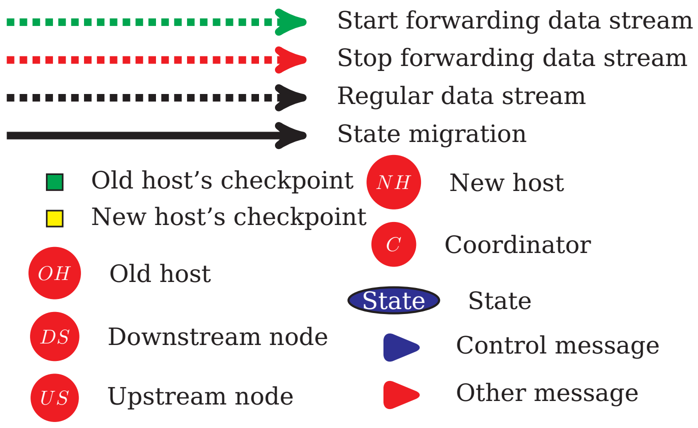
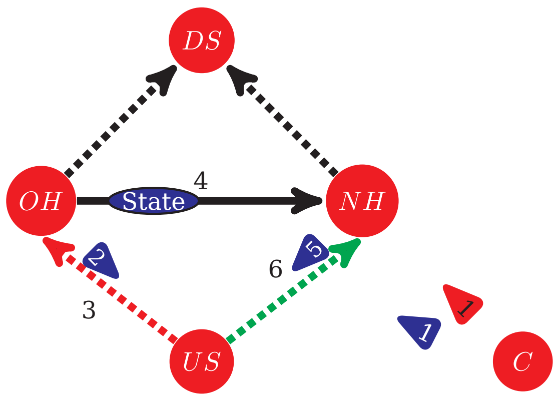


Figure IV.7: Legend for the migration mechanism illustrations in Figure IV.8–IV.12

Control messages used for migration are typically embedded into the data streams. These tell the nodes that a migration will be performed, and might be used for other coordination tasks. In some solutions, this message is sent only to the old or the new host; in other solutions, it is sent to the old and the new host, or to upstream nodes, old and new hosts, and downstream nodes. There may be many reasons for notifying different nodes about migration, such as updating the view of where key partitions are maintained, and routing streams. We describe only a subset of control messages for each mechanism, and they describe the essential tasks that should be executed to perform stream and state management tasks. In the illustrations, the control messages are shown in blue whereas the other messages are shown in red. In addition to the visual representation of the topology and messages sent among nodes, the essential tasks to execute during migration are shown in a listing. The first blue control message from the coordinator, which features in step one of each algorithm, is shown in this listing. All other control messages represent subsequent steps in the algorithm that are described in the first blue control message.

IV.4.2 Standard moving state

The standard moving state mechanism (see Figure IV.4 and IV.5) uses direct state movement between the old and the new hosts, and the entire state is sent all at once. Aside from moving the state, migration requires changing the stream routing. Figure IV.8 shows the steps involved in the standard moving state algorithm developed by Shah et al. [124]. They proposed an operator called Flux that can adapt the state partitioning of the pipelines of dataflow using a state movement algorithm. Other state movement mechanisms largely follow the same steps, but they might vary in their approach to stream management or in the roles assigned to specific nodes.



1. Coordinator forwards control message to the old and new host
2. Old host pauses upstream
3. Upstream stops sending to old host
4. Old host migrates state to new host
5. New host resumes the upstream
6. Upstream starts sending to new host

Figure IV.8: Moving state (according to [124])

Our interpretation of the moving state mechanism's blue migration control message from Step 1 in Figure IV.8 is described in Listing IV.1. The upstream nodes buffer, stop, and redirect streams from the old host to the new host. Following this, the task of migrating the state from the old host to the new host is issued to the old host, after which the streams are resumed. Instead of stopping the upstream nodes, other solutions [28, 42] redirect streams from the upstream nodes to the new host. The new host buffers the streams and starts to process them when the state from the old host has been received and installed. Other solutions send the control message to the old host instead of the upstream nodes [113], or even to the new host [56]. The benefit of this class of migration mechanisms is that it is straightforward and simple, but the downside is that it may cause significant downtime.

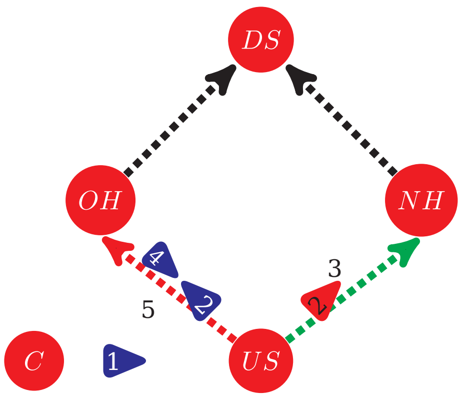
```
ControlMessage(OH
  ControlMessage(Upstream
    BufferStreams(Streams(query))
    StopStreams(Streams(query))
    Redirect(Streams(query), OH, NH)
    ControlMessage(OH, MoveState(query, NH))
    Resume(Streams(query))))
```

Listing IV.1: Single-track moving state

IV.4.3 Parallel-track

There are two types of parallel-track mechanisms: state-recreation and window-recreation mechanisms. The difference between them is that state-recreation involves moving state and window-recreation does not. The completion time for a state-recreation algorithm is proportional to the state size, whereas for a window-recreation algorithm, it is proportional to the window size [42]. Zhu et al. [160] introduced the window-recreation parallel-track migration mechanism. Gulisano et al. [42] presented both a window-recreation and a state-recreation mechanism, and Ottenwalder et al. [104] performed state-recreation migrations based on changes in mobility. Madsen et al. proposed a direct window-recreation mechanism in Enorm [88] and a checkpoint-assisted state-recreation mechanism in [86]. ChronoStream [149] performs a checkpoint-assisted state-recreation migration of state slices to provide horizontal elasticity. UniMiCo [107] (uninterruptable migration of continuous queries) is a direct window-recreation algorithm that can handle both time-based and tuple-based window semantics.

StreamCloud's [42] state-recreation and window-recreation mechanisms are shown in Figure IV.9 and Figure IV.10. In both mechanisms, a handover between the old and new hosts is scheduled using a timestamp. In window-recreation, the handover is performed in a way such that the old host empties its windows and the new host fills them in parallel, resulting in a smooth handover. For this purpose, the upstream nodes send tuples to both the old and new hosts. In state-recreation, the old host sets the handover timestamp immediately before serializing and transmitting the state to the new host. Any subsequent tuples with a timestamp lower than the handover timestamp are processed by the old host, and the other tuples are processed by the new host. Operator downtime can be avoided here if the handover timestamp is set to a time after the new host is expected to have received the state and started its execution. When the state is received by the new host, it processes all tuples it receives from the upstream nodes in parallel with the old host, but produces only tuples caused by input tuples with a timestamp higher than the handover timestamp.



1. Coordinator injects control message
2. Upstream forwards control message
3. Upstream starts sending to NH
4. OH sends EndOfReconfiguration to US
5. US stops forwarding to OH

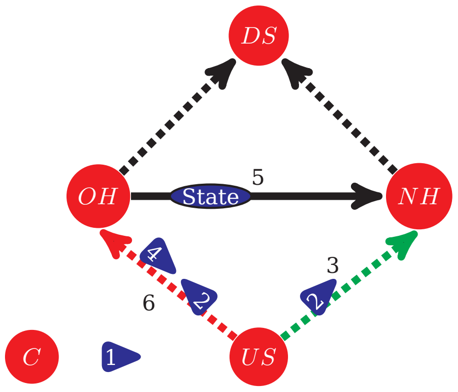
Figure IV.9: Parallel-track window-recreation algorithm (according to [42])

Our interpretation of the window-recreation mechanism for the blue migration control message from Step 1 in Figure IV.9 is described in Listing IV.2. The control message is sent by the coordinator to the upstream nodes. From there, it is forwarded to the old host, which schedules the takeover time for the new host and sends it to the upstream nodes. From then on, the upstream nodes send tuples to both the old and the new hosts. The new host processes the same tuples as the old host, but does not produce any tuple until the old host has stopped processing.

Our interpretation of the state-recreation mechanism for the blue migration control message from Step 1 in Figure IV.10 is described in Listing IV.3. The control message is sent by the coordinator to the upstream nodes. This algorithm requires slightly greater coordination between the old and the new host than in case of window-recreation, because the old host must move its state to the new host, and the latter needs to know the takeover time. These classes of migration mechanisms have as benefit that they may result in zero downtime for the operators, but at the expense of overhead when duplicating the data streams and maintaining two copies of the stream

```
ControlMessage(Upstream
  ControlMessage(NH,
    StartQuery(query))
  ControlMessage(OH
    ControlMessage(Upstream,
      Schedule(RemoveNextHop(Streams(query), OH)
                TakeoverTime(query))
      AddNextHop(Streams(Upstream), NH))))
```

Listing IV.2: Window-recreation



1. Coordinator injects control message
2. Upstream forwards control message
3. Upstream starts sending to NH
4. OH sends EndOfReconfiguration to US
5. OH migrates state to NH
6. Upstream stops sending to OH

Figure IV.10: Parallel-track state-recreation algorithm (according to [42])

processing system. Window-recreation requires no state transfer, but at the expense of increasing the total migration time, which in a pay-as-you-go scenario increases the monetary cost of running the system.

```
ControlMessage(Upstream
  ControlMessage(OH
    ControlMessage(NH,
      StopStreams(OutputStreams(query))
      StartQuery(query)
      Schedule(TakeoverTime(query)
               StartStreams(Streams(query))))
    ControlMessage(Upstream,
      Schedule(RemoveNextHop(Streams(query), OH),
               TakeoverTime(query))
      AddNextHop(Streams(Upstream), NH)))
  MoveState(query, NH))
```

Listing IV.3: State-recreation

IV.4.4 Indirect state movement

Gedik et al. [37] described an indirect state migration mechanism for load balancing that has been used as the basis in several studies [17, 26, 69]. They proposed an operator that outputs to multiple replicas partitioned by keys, called a splitter, that can decide to change the distribution of the keys, which requires state migration between replicas. Moreover, they introduced a two-phase approach to migration: donate and collect. In the donate phase, the state to be migrated is moved from the old host's in-memory store to a backing store. In the collect phase, the new host retrieves the state from the backing store. This method was subsequently used by Cardellini et al. [17] and Li et al. [69] to implement features of elasticity in migration in Apache

Storm. The drawback of this method is that streams from the upstream nodes are paused during execution. De Matteis et al. [25, 26] defined a similar state migration mechanism. However, their implementation contains a number of improvements, e.g., the splitter can send new tuples during state movement instead of blocking until migration is complete.

In the donate phase of the mechanism proposed by Gedik et al. [37], replicas place the state to be moved into packages, one for each replica that takes over the state. The data are moved away from the in-memory store of the replicas to a backing store. A vertical barrier is used across the replicas to ensure that they do not progress to the next phase until all packages have been donated. In the collect phase, the replicas check the backing store for any packages that contain the state that they take over and restore it. Following this, a horizontal barrier is used to prevent the splitter from sending any tuples until the migration process has been completed.

The benefit of the two-phase approach is that it involves an API where an operator simply requires implementing methods to extract the state, and sends it to a backing store instead of requiring intricate communication among operators. Moreover, it can use existing fault tolerance mechanisms that periodically create checkpoints of states for the backing store.

IV.4.5 Partial state movement

With partial state movement, the state is partitioned and each partition is moved individually, to minimize operator downtime. MigCEP [104] is an algorithm designed for frequent migrations to minimize downtime. The state is split up into two parts: immutable and mutable. An immutable or static state includes the operator and, possibly, databases whose data have not changed during migration. A mutable state consists of tuples that are being processed in the operator.

A further improvement involves sending the last incremental checkpoint of the state to the new host before the operator goes down. This is the case in ChronoStream [149] and Rhino [28], where the state is split before the operator is migrated, and an incremental checkpoint that includes the new state after the first part has been extracted. This can be seen as analogous to the immutable and mutable states described in MigCEP [104].

Megaphone [53] is a state migration technique for migrating many keys in an efficient way to minimize latency spikes. In this case, the state is split into many equal-sized parts. Each causes some downtime for the system. However, while the total migration time increases, the spikes due to tuple latency are substantially reduced compared to sending the entire state all at once. However, the Megaphone mechanism introduces some additional overhead to operators during non-rescaling periods. Another state migration technique called Meces [41] may improve upon Megaphone by being more lightweight and prioritizing the migration of partial states that are needed by incoming tuples. A newly received tuple that requires a given partial state fetches it from the old host, instead of waiting for it in a larger batch, or relying on complex synchronization mechanisms. The commonality between Megaphone and Meces is that they are only beneficial with operators where the state is split up into many keys, e.g, word count with words as keys, equijoin operators, or other aggregation operators with keys.

Fragkoulis et al. [34] distinguished between all-at-once and continuous state movements, which are classified in this tutorial as all-at-once and partial state movements, respectively. Megaphone, Rhino, and ChronoStream are characterized in this tutorial as exemplars of partial state movement, while Fragkoulis et al. categorized Megaphone as using continuous state movement, and Rhino and ChronoStream as using all-at-once state movement. The reason for this difference is that ChronoStream and Rhino rely on *distributed checkpoint replication*, and need only to send the state that has been built up since the last checkpoint. In this tutorial, migration is further divided by distinguishing between solutions that use distributed checkpoint replication and those that do not. Megaphone and Meces do not use it, and send the entire state directly from the old host to the new host, whereas ChronoStream and Rhino depend on distributed checkpoint replication. If Rhino and ChronoStream do not use distributed checkpoint replication, this means that the initial checkpoint is sent from the old host to the new host instead of existing on the new host already. Therefore, using partial state movement is not necessarily an indication that multiple states are sent during migration.

State shedding combines load shedding and operator migration in a way that demands fine-grained migration [139]. Each partial state is assigned a utility based on how important it is, similar to how it is done in load shedding. The most important partial states are then migrated, whereas the least useful partial states may be dropped. The primary advantage of state shedding is the ability to prioritize the migration of critical states. However, it does require calculating the utility of partial states, which can be challenging to predict.

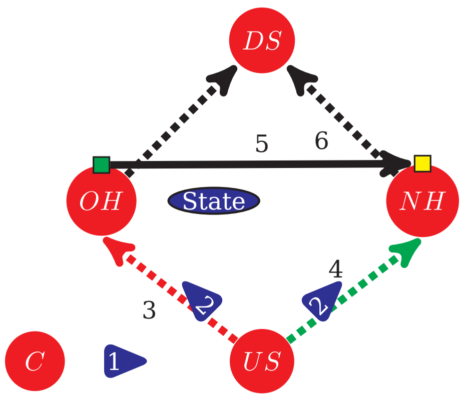
IV.4.6 Distributed checkpoint replication

Some solutions leverage fault tolerance mechanisms to improve the scalability and performance of migration using periodically updated, and distributed and replicated checkpoints of the state of stream processing. Since these algorithms use checkpoint solutions that may already exist, they are called checkpoint-assisted algorithms. If the target of migration is a host that already contains the state, a migration algorithm can be as simple as one that loads the checkpoint in memory and replays the upstream tuples to the new host. This requires exactly-once guarantees, as provided by pub-sub systems such as Kafka [4]. A parallel-track algorithm can work similarly, but, instead of stopping the old host before replaying tuples on the new host, both the old host and the new host run until the latter takes over. In this process, output tuples need to be filtered to remove duplicate tuples. ChronoStream [149] uses distributed checkpoint replication to implement a parallel-track algorithm, Rhino [28] to realize a single-track algorithm, and the proposal of Madsen et al. [86] to carry out both. The algorithms often also use partial state movement when updating checkpoint replicas to send as little state as possible.

Del Monte et al. [28] introduced a checkpoint-assisted single-track migration mechanism called Rhino that can migrate state sizes of up to terabytes 15 times faster than the state-of-the-art solutions (as of 2020) by using incremental checkpointing. Their algorithm is shown in Figure IV.11. Most of the state is sent before the old host is stopped. Afterward, it sends an incremental checkpoint that represents a change in the original state. In this way, only tuples that arrive after migration has started need to be migrated in the incremental checkpoint. This algorithm is a cluster-based migration mechanism that is executed by a handover manager (HM). The HM informs all workers about the migration and about what will happen, by injecting a control message into the source streams (inspired by Chi [90]). Afterward, the source nodes are redirected. When the old host has received a control message on all of its incoming streams, it sends the state to the new host. The green box indicates the state repository for the old host, while the yellow box indicates the state repository for the new host. The intermediary hosts, including the old and the new host, send control messages to their next hop nodes. When the nodes have completed their tasks, including the redirection of streams and the migration of state, they acknowledge the HM. The migration is complete when all nodes have acknowledged the HM.

Our interpretation of the checkpoint-assisted moving state algorithm's blue migration control message from Step 1 in Figure IV.11 is described in Listing IV.4. The control message is issued to the upstream nodes, which forward a control message to all downstream nodes. We describe tasks that the old host might be assigned. The main difference between this algorithm and the standard moving state algorithm is that most of the state is assumed to be on the new host before the migration starts. As such, when the state is moved, it is moved using the partial state movement task `MoveIncrementalState` instead of `MoveState`.

Wu et al. proposed ChronoStream [149], a checkpoint-assisted state-recreation migration algorithm that provides horizontal elasticity, as illustrated in Figure IV.12. The states of all tasks on a node are periodically backed up and sent to the other nodes. As a result, migration only involves updating a subset of the backed-up state, which significantly reduces the number of states to be moved. This process is split into four phases: migration preparation, state rebuilding, dataflow rerouting, and resource



1. Coordinator injects control message
2. Upstream forwards control message
3. Upstream stops forwarding to the OH
4. Upstream starts forwarding to the NH
5. OH stops processing and migrates incremental checkpoint to NH
6. NH loads existing checkpoint and incremental checkpoint, and starts processing

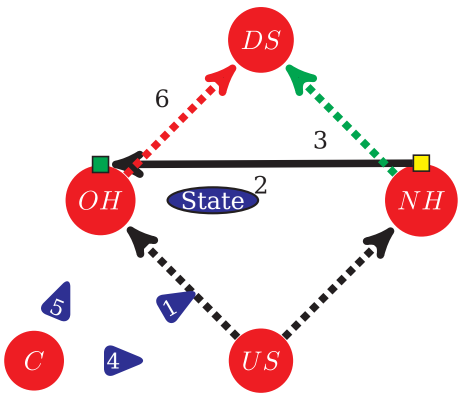
Figure IV.11: Checkpoint-assisted single-track mechanism (according to [28])

```
# Bootstrapping
ControlMessage(OH, ReplicateCheckpoint(NH))

# Migration
ControlMessage(Upstream
  ControlMessage(NH,
    BufferStreams(NH, Streams(query))
    StopStreams(NH, Streams(query)))
  ControlMessage(OH,
    Redirect(Streams(query), OH, NH)
    MoveIncrementalState(query, NH)
    ControlMessage(NH, StartStreams(Streams(query)))
))
```

Listing IV.4: Checkpoint-assisted single-track

release. The first phase sets up a container for the operator on the destination node if this has not been done already. In the second phase, the new host fetches the operator's state locally or remotely and rebuilds it, and notifies the master node when finished. The green box indicates the state repository for the old host, while the yellow box indicates the state repository for the new host. The third phase involves the master telling the data sources to send tuples to the new host as well, including any tuple that is not included in the state that the new host received. At this point, the new host participates in the processing and produces the same tuples as the old host, and duplicate output tuples are filtered out by downstream operators based on the sequence numbers of the tuples. Finally, the controller tells the old host to release the resources such that the new host is the only node running the operator.



1. Coordinator tells NH to upgrade slice
2. NH fetches state
3. New host start processing
4. Controller tells upstream nodes to re-forward tuples
5. Controller tells OH to release resources
6. Old host stops processing

Figure IV.12: Checkpoint-assisted parallel-track algorithm (according to [149])

Our interpretation of the checkpoint-assisted parallel-track mechanism for the blue migration control message from Step 1 in Figure IV.12 is described in Listing IV.5. The main difference between the parallel-track checkpoint-assisted mechanism and a non-checkpoint-assisted mechanism is that the immutable state is sent or made available on the new host before any downtime occurs.

Checkpoint-assisted migration mechanisms can greatly reduce operator downtime, providing a significant performance boost in cases where fault tolerance features are already established. However, if the system does not already have checkpointing functionality, it must be added.

```
# Bootstrapping
ControlMessage(OH, ReplicateCheckpoint(NH))

# Migration
ControlMessage(US, AddNextHop(Streams(query), NH)
  RemoveNextHop(Streams(query), OH))
ControlMessage(NH,
  ControlMessage(OH, MoveImmutableState(query, NH)))
ControlMessage(OH, StopQuery(query))
```

Listing IV.5: Checkpoint-assisted parallel-track

IV.5 Migration Decision

We review the elements of migration decision-making including the calculation of the costs and benefits of migration. There are existing surveys that go into depth of what methods are used for decision-making [74], and that is not the purpose of this tutorial. We introduce some migration decision methods and describe which metrics are used to base migration decisions on, and what measurements are done in evaluations. This gives a picture of the struggle of making decisions on incomplete data, because the consequences of a migration choice are not known until it is done.

IV.5.1 How to make migration decisions

To start out, we give a small introduction to the topic of how to make migration decisions. We can split this problem in two phases: the problem definition phase and the problem solution phase. The problem definition defines mathematically what is the goal of the system, e.g., to minimize load imbalance, latency and maximize throughput. Thereafter, the problem solution involves some way to achieve this. Re-placement or re-scheduling of operators on different nodes or CPU cores is an NP-hard problem. Some works [20, 60] solve the problem using an Integer Linear Programming (ILP) solver such as gurobi [5] or CPLEX [6]. However, they do not scale well since they require a global view of the network. Operator scaling, on the other hand, is a problem about minimizing the amount of instances to operators while upholding the QoS guarantees [62].

In Table IV.9, we summarize a selection of the studied papers that contribute to the modeling techniques, algorithms, and decision-making strategies for operator migration. For each paper, we highlight the specific problem being addressed, the applied algorithm or technique, and the approach to decision-making.

The proposed solutions typically use heuristics, which are approximate methods that are designed to quickly find a solution that is close to optimal. These heuristics often involve using specific algorithms that are tailored

Table IV.9: Modeling techniques and algorithms for performing decision-making in operator migration

Paper Reference	**Problem**	**Algorithm/Technique**	**Decision-Making**
Cardellini et al. [20]	Re-placement or re-scheduling of operators	ILP solver (Gurobi)	Minimize load imbalance, latency; Maximize throughput
Jonathan et al. [60]	Re-placement or re-scheduling of operators	ILP solver (CPLEX)	Minimize load imbalance, latency; Maximize throughput
Pietzuch et al. [108]	Optimization of network usage	Relaxation (Heuristic)	Minimize the network usage of a query
Buddhika et al. [15]	Load balancing	Prediction rings (Heuristic)	Reduce interference that negatively impacts performance
Gedik et al. [36]	Load balancing	Scan, redist, readj (Heuristics)	Balance the mapping from key to server
Hochreiner [52]	Resource scaling	Threshold-based heuristic	Scale resources up or down based on CPU utilization

to the particular optimization problem at hand. For example, a common objective in migration is to minimize the latency of data tuples or maximize the rate at which data tuples are processed.

Pietzuch et al. [108] define a stream-based overlay network (SBON) that uses a placement and adaptation algorithm called Relaxation. Relaxation is a heuristic algorithm that minimizes the network usage of a query. The overlay network consists of a cost space with three latency dimensions (each direction), and one load dimension. The latency dimensions constitute the latency space, and physical nodes are placed in this space such that the distance between nodes represents the communication latency.

Buddhika et al. [15] present a heuristic algorithm for load balancing where the goal is to reduce interference that negatively impacts the performance of stream processing performance. A construct called prediction rings is applied that predicts the future resource usage of stream processing computations, and these are used to calculate the interference score. Thereafter, the goal is to move stream processing computations to the nodes with the least interference. If the interference score exceeds a predefined threshold, the operator is migrated to a node with less interference.

Gedik et al. [36] introduced three heuristic partitioning algorithms to perform load balancing, namely scan, redist and readj. The job of the partitioning functions is to make sure that the mapping from key to server is balanced. These have in common that they apply the same metrics for making the decisions and have the same end-goal, but have different ways of achieving it.

Hochreiner [52] proposed a platform for elastic stream processing, called PESP, which uses heuristics with predefined thresholds to make scaling decisions. Specifically, when the CPU utilization of the system exceeds a certain threshold, PESP scales up the resources to handle the increased workload. Conversely, when the CPU utilization drops below a certain threshold, PESP scales down the resources to avoid overprovisioning.

IV.5.2 Parameters

We first provide an overview of the parameters of optimization, and the cost and benefit metrics used in existing work (see Table IV.10 and the pie charts in Figure IV.13 and IV.14. We then describe (1) how cost values are modeled and measured, (2) approaches for optimization to increase benefits, and (3) reactive and proactive methods. The figure and table show similar results to Figure 8 by Cardellini et al. in [21], but here we go more in depth of how the metrics are used.

Even though there are many different definitions of the parameters of optimization, they are often related. Therefore, we group them in Table IV.10 into six categories: network performance (e.g., bandwidth, bandwidth latency product), tuple performance (e.g., tuple latency, tuple rate), load, costs of migration, monetary costs, and energy usage. Since the goal of migration is important for optimization, we differentiate between the categories of parameters of optimization in research according to the goals of migration. The most prominent goal is load balancing, and load is the most commonly used optimization parameter. While monetary cost is not commonly used as optimization parameter, migration is often used to avoid the need for over-provisioning and, thus, indirectly reduces monetary costs.

Parameter	Migration goal	Papers
Tuple performance	Load balancing	[15, 56, 76, 77, 145]
	Elasticity	[26, 38, 43, 48, 56, 72, 75, 77, 100, 119, 152, 157]
	QoS	[60, 77]
Network performance	Load balancing	[66, 145]
	Elasticity	[154]
	QoS	[19, 50, 61, 103, 104, 108, 115, 145, 158]
Load	Load balancing	[15, 32, 36, 56, 66, 76, 77, 114, 131, 142, 145, 150, 159]
	Elasticity	[18, 26, 43, 52, 56, 77, 78, 154]
	QoS	[50, 61, 77, 78, 108, 114, 158]
	Fault tolerance	[57]
Migration costs	Load balancing	[32, 36, 66, 114, 142]
	Elasticity	[18, 26, 43, 154]
	QoS	[61, 71, 114]
Monetary costs	Elasticity	[52, 75, 119, 154]
	QoS	[19, 50]
Energy usage	Load balancing	[101]

Table IV.10: Goals of optimization grouped by the goal of migration

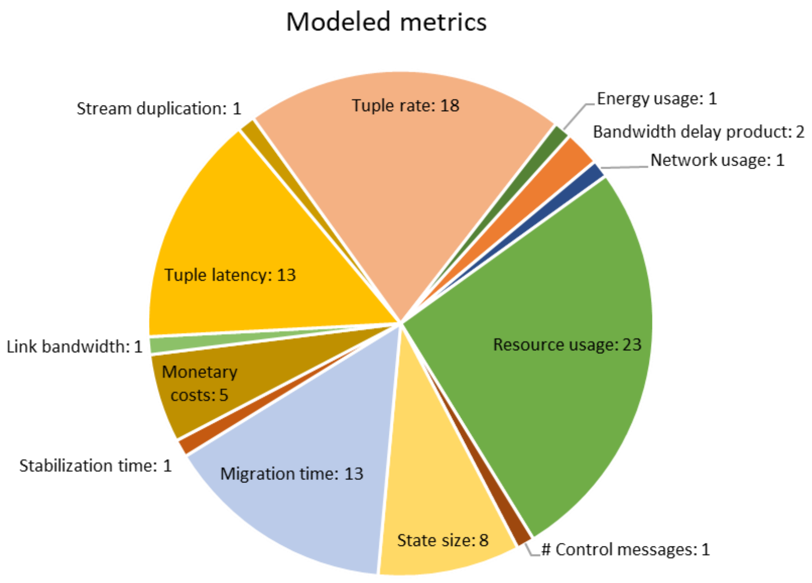


Figure IV.13: Popularity of metrics for modeling migration cost and placement benefit, shown by usage frequency

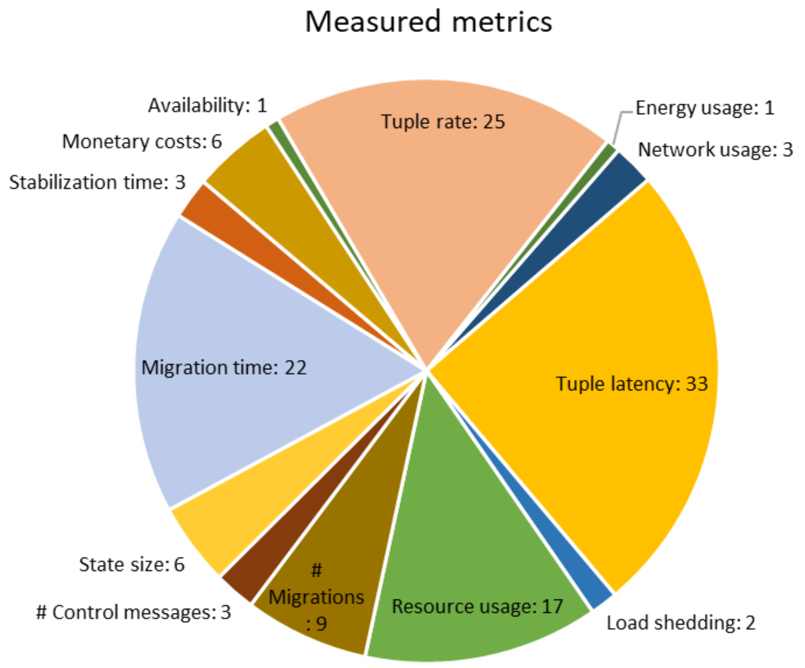


Figure IV.14: Popularity of metrics for measuring migration cost and placement benefit, shown by usage frequency

Figure IV.13, IV.14 and Table IV.11 give an overview of the metrics used to define the modeled and measured costs of migration, and the modeled and measured benefit of migration. Table IV.11 catalogs each paper by the specific environment for migration, the goal of migration, the cost models, the benefits, and actual costs and benefits measured. This table offers a quick reference to understand how various researchers have modeled and evaluated their solutions. When a metric is used for modeling the migration cost or placement benefit, it means that it is part of an equation that is typically applied to decision-making. This may include attempts at calculating the current system state, or predicting future system state doing proactive migration decisions. Ideally, the measured and modeled metrics should be identical, but they are not. One reason for this mismatch is, that it is much easier to measure values for certain metrics, than to use them for decision-making, like tuple latency and tuple rate. Kalavri et al. [62] discuss how the observed tuple rates may not be good for doing decision-making, because what is really interesting is to know the capacity of a system, or the "true" tuple rate. Values for costs and benefits need to be estimated for each migration decision, whereas values of the evaluation metrics are measured during migration. The mismatch between the modeled cost and the benefit, and the measured evaluation metrics might also help to complement future migration decisions and the assessment of migration using further metrics. The most commonly used parameters to determine the cost of migration are the migration time and state size, and few systems use more precise cost parameters, such as latency spike and performance penalty. While some approaches, such as [66], are listed in Table IV.10 for performing placement optimization based on the cost of migration, they are notably absent from Table IV.11 because these approaches do not use any specific metric to describe the cost of migration.

IV.5.3 Migration cost

Accurately defining the cost of migration is essential for making the correct migration decisions. It can be manifested as increased resource consumption and any kind of degradation of execution, such as decreased throughput or increased tuple latency. Table IV.11 indicates that it is more common to measure the cost of migration than it is to model it for making migration decisions.

The vast majority of solutions use migration-specific metrics to model the cost, as opposed to metrics that are used for measuring the benefit of the adaptation. Tuple processing performance is used in many cases to model and measure benefit, but very few approaches have used it to calculate the cost of migration. Heinze et al. [48] modeled and predicted tuple latency as part of the cost of migration, by using the predicted input rate, migration

Table IV.11: Overview of papers on migration decisions, covering deployment environment, migration goals, and metrics used for migration cost and benefit

Paper	Deployment environment	Migration goal	Modeled migration cost	Measured migration cost	Modeled placement benefit	Measured benefit of migration
[14]	Edge	Fault tolerance		Migration time, Stabilization time		
[15]	Cloud	Load balancing		# Migrations	Tuple latency, Tuple rate, Resource usage	Tuple latency, Tuple rate, Resource usage
[17]	Cloud	Elasticity		State size, Migration time		Tuple latency, Resource usage
[20]						Tuple latency, Tuple rate
[18]	Fog	Elasticity		Tuple latency	Resource usage	Tuple latency, Tuple rate
[19]	Cloud	QoS	Migration time	Tuple latency, Migration time, Monetary costs	Tuple latency, Monetary costs	Monetary costs
[26]		Load balancing, Elasticity, QoS		Tuple latency, Resource usage, # Migrations	Tuple rate	
[27]		Load balancing			Tuple rate	Tuple rate
[28]	Cloud	Load balancing, Elasticity, Fault tolerance, QoS		Tuple latency, Tuple rate, Resource usage		Tuple latency, Resource usage
[33]	Cloud	Load balancing	State size			Tuple rate
[32]	Cloud	Load balancing	State size	State size	Resource usage	Tuple latency, Tuple rate
[22]	Cloud			Migration time		
[37]						Tuple rate, Resource usage
[36]	Cloud	Load balancing	State size	State size	Resource usage	
[38]	Cloud	Elasticity	Migration time, Stabilization time	Migration time	Tuple latency, Tuple rate	Tuple latency, Tuple rate, Resource usage
[41]	Cloud	Elasticity		Network usage, Resource usage, Migration time		Tuple latency, Tuple rate
[42]		Load balancing, Elasticity		State size, Migration time		

Table IV.11: (Continued)

Paper	Deployment environment	Migration goal	Modeled migration cost	Measured migration cost	Modeled placement benefit	Measured benefit of migration
[43]		Load balancing, Elasticity		Migration time		Tuple latency, Tuple rate, Resource usage
[45]	Cloud			Tuple latency		
[48]	Cloud	Elasticity	Tuple latency	Tuple latency, Resource usage	Tuple rate	
[49]					Tuple rate	Tuple rate, Resource usage
[50]	Fog	QoS	Monetary costs	Monetary costs	Resource usage, Monetary costs	Tuple latency, Resource usage, Availability, Monetary costs
[52]	Cloud	Elasticity			Resource usage, Monetary costs	Monetary costs
[53]	Cloud			Resource usage, Migration time		Tuple latency, Resource usage
[56]	Cloud	Load balancing, Elasticity	State size	Migration time	Stream duplication, Tuple rate, Resource usage	
[57]	Cloud	Fault tolerance		Migration time	Resource usage	
[60]	Fog	Elasticity, QoS	Migration time	Migration time, Stabilization time	Link bandwidth, Tuple latency, Tuple rate	Tuple latency, Load shedding
[61]	Edge	QoS			Resource usage, Energy usage	
[66]	Cloud	Load balancing			Resource usage	
[68]		Load balancing, Elasticity				Tuple latency, Tuple rate, Resource usage
[76]	Cloud	Load balancing	State size		Tuple latency, Tuple rate, Resource usage	Tuple latency
[73]	Cloud	QoS				Tuple latency, Tuple rate, Resource usage, Load shedding
[72]	Edge	Elasticity, Fault tolerance			Tuple rate	Tuple latency

Table IV.11: (Continued)

Paper	Deployment environment	Migration goal	Modeled migration cost	Measured migration cost	Modeled placement benefit	Measured benefit of migration
[71]		QoS		# Migrations		Tuple rate
[75]	Cloud			# Migrations	Tuple rate	Monetary costs
[77]		Load balancing, Elasticity, QoS			Tuple latency, Resource usage	Tuple rate
[78]	Cloud	Elasticity, QoS	Migration time	Tuple rate	Tuple rate, Resource usage	Tuple latency, Tuple rate, Resource usage
[80]	Edge	QoS	# Control messages, Migration time	State size, Migration time		Tuple latency
[81]		Load balancing	Resource usage, Migration time			
[86]	Cloud	Load balancing	Migration time	Tuple latency, Migration time		
[88]	Cloud	Load balancing	Migration time	Tuple latency, Tuple rate, Migration time		
[87]	Cloud	Elasticity, QoS	State size			
[94]	Cloud	Fault tolerance	Migration time	Migration time		
[100]		Elasticity				Tuple rate
[101]	Edge	Load balancing				Energy usage
[104]	Edge	QoS	Bandwidth delay product			
[103]	Edge	QoS	Bandwidth delay product			
[108]		QoS		# Migrations	Resource usage	Network usage, Tuple latency
[113]	Fog	Load balancing		# Migrations, # Control messages		

Table IV.11: (Continued)

Paper	**Deployment environment**	**Migration goal**	**Modeled migration cost**	**Measured migration cost**	**Modeled placement benefit**	**Measured benefit of migration**
[114]	Fog	Load balancing, QoS		# Migrations	Tuple latency, Tuple rate, Resource usage	Tuple latency, Tuple rate
[115]	Fog	QoS		# Migrations, # Control messages		
[119]	Fog	Elasticity			Tuple latency, Monetary costs	Monetary costs
[118]	Cloud			Migration time		
[124]	Cloud	Load balancing		Tuple latency, Tuple rate		Tuple latency, Tuple rate
[131]	Cloud	Load balancing			Tuple latency, Tuple rate, Resource usage	Tuple latency
[130]		Elasticity			Tuple latency, Tuple rate, Migration time	
[135]		Load balancing, QoS		# Migrations, # Control messages		
[145]	Fog	Load balancing, QoS			Network usage, Resource usage	Network usage, Resource usage
[140]	Cloud	Load balancing	State size	Migration time		
[142]	Cloud	Load balancing, Fault tolerance	State size	Tuple latency, Tuple rate, Migration time	Resource usage	Tuple latency, Tuple rate
[149]		Elasticity, Fault tolerance		Migration time		Tuple latency, Tuple rate
[150]	Cloud	Load balancing	Migration time		Resource usage	
[152]		Elasticity		Migration time, Stabilization time	Tuple rate	
[151]	Edge				Tuple latency, Tuple rate, Resource usage	Tuple latency, Tuple rate
[154]	Cloud	Load balancing, Elasticity	Migration time, Monetary costs	Migration time	Tuple latency, Resource usage	Monetary costs
[157]	Cloud				Tuple rate	Tuple latency, Tuple rate, Resource usage
[159]	Edge	Load balancing		State size		Tuple latency
[158]	Cloud	QoS			Resource usage	
[160]			Migration time			

time, and time before queued events can be processed. The tuple latency is defined in Equation IV.3.

$$latSpike(op) = pause_{Time}(op) + delay_{proc}(op) - delay_{arrival}(op) \tag{IV.3}$$

No solution was provided to model the tuple rate, because it is much easier to measure tuple processing performance than to predict it, when it comes to the cost of migration. Operator downtime is an indicator of spikes in latency but also depends on the tuple rate, because only those tuples that are supposed to be processed during operator downtime are affected. As we discuss later, several solutions have been proposed to model and predict the future tuple rate in a DSP. These predictions can be leveraged to make migration decisions. However, none of the existing solutions take into account the cost of migration in terms of reduced tuple rate.

The migration time and the size of the state to be moved are the most common costs of migration metrics. Migration time is typically calculated as a function of state size, bandwidth, and latency. In environments where the bandwidth and latency are stable, such as within data centers, the state size is often interchangeable with migration time. In most cases, the migration time is assumed to be easy to model and no calculation for it is given. Some solutions can migrate multiple operators at a time, and thus define the migration time as the maximum time it takes to move any of the operators [19, 60]. Cardellini et al. [19] used a data center-based solution to define the operator downtime based on the type of adaptation made, size of the state to be moved, and the round-trip delay between the nodes and the computational resources. WASP [60] is a wide area network solution that defines the time it takes to move an operator based on the state size and bandwidth between links, the latter of which is significantly more limited and variable in a wide area network than a data center. Using the migration time, WASP [60] makes the decision of where to migrate by solving a *minmax* problem by minimizing the slowest migration: $minmax(\frac{|state_{s1}|}{B_{\mathrm{s1}}^{\mathrm{s2}}})$, where $B_{\mathrm{s1}}^{\mathrm{s2}}$ represents the available bandwidth from site s1 to s2 and $|state_{s1}|$ represents the size of the state on the old host (s1).

Zhu et al. [160] focused on the time needed for each step of migration, such as the time spent cleaning the accumulated tuples, state matching, moving the state, and recomputing it.

In Elysium [78], migration time is defined as: $R_{state} + R_{restart} + R_{queue}$, where R_{state} is the time it takes to send the state, $R_{restart}$ is the time it takes to restart the topologies, and R_{queue} is the time it takes to process the tuples that were received and queued up during $R_{state} + R_{restart}$.

Ma et al. [81] defined a migration cost model: $cost = (t2 - t1) \times (i2 - i1)$, where $t2-t1$ is the migration time, and $i2-i1$ is the difference in performance between the new host and the old host. The logic is that migration decisions need to make a trade off between the migration cost and benefit. If the new host has significantly better performance than the old host, it might be worth to do the migration, even though the migration takes a long time.

Using state size as the cost of migration is among the easiest ways of defining cost because it requires only looking at the size of the state to be migrated. The solutions that we analyzed that use state size as cost metric are all cloud-based, which makes sense since data centers feature a high and stable bandwidth between nodes, in contrast to geo-distributed environments. The state size is frequently used as part of the objective function when making migration decisions [36, 56] as part of a constraint to prevent costly solutions from being selected [87], and can even be the only criterion to minimize when making load balancing decisions [32, 33].

Luthra et al. [80] used the number of control messages during migration as part of the definition of the cost of migration. This parameter is significant because if nodes have to wait for acknowledgments for these messages, the total migration time then depends on the distance between nodes. When the cost of migration is defined in terms of migration time, only the time taken to move the state is generally included in the equation, and might result in an inaccurate view of the cost.

The bandwidth delay product is a measure of how much data can be sent in a given duration. As part of the cost of migration, it represents the amount of data that can be sent when a migration is underway. The more tuples that can be sent, the higher is the cost of migration, and the less desirable a migration is. MigCEP [103, 104] uses the average bandwidth delay product during migration as its cost. This represents the utilization of the network due to migration.

The monetary costs of migration have been modeled by Zacheilas et al. [154] and Hiessl et al. [50]. VISP [50] distinguishes between the enactment cost of a placement, which is the cost of running the current topology, and the migration cost, which is the cost of making a change.

Enactment cost:

$$C_{op}(x) = \sum_{i \in V_{dsp}} \sum_{u \in V^{\mathrm{i}}_{\mathrm{res}}} C_u x_{i,u} \tag{IV.4}$$

Migration cost:

$$C_{mig}(x) = \sum_{i \in V_{dsp}} \sum_{u \in V^{\mathrm{i}}_{\mathrm{res}}} \sum_{u \in V^{\mathrm{i}}_{\mathrm{res}}} C(i,u,v) x^{prev}_{\mathrm{i,u}} x_{i,v} \tag{IV.5}$$

where V_{dsp} represents the operators, V_{res} represents the compute nodes, $x_{i,v}$ represents the placement of operator $i \in V_{dsp}$ on the new host $v \in V_{res}$, and $x^{prev}_{i,u}$ represents the placement of operator $i \in V_{dsp}$ on the old host $u \in V_{res}$.

Considering both the enactment cost and the migration cost makes it possible to assess whether the long-term cost savings from scaling down to fewer instances exceed the short-term cost of migration, leading to a reduction in the frequency of migrations.

IV.5.4 Benefit

To explore the optimization goals of different migration solutions, we begin by examining the most important metrics used to assess the benefits of migration. The benefit of migration is based on performance in terms of the placement, amortization time, and the cost of migration (as explained in Section IV.3.2). This is either explicitly defined or implicit in the decision-making, where the goal is to maximize performance in terms of the placement and minimize the cost of migration.

One could argue that all goals of optimization are relevant to all goals of migration. However, some are more tightly coupled than others. For instance, load balancing involves using the load of a system to make balancing decisions. QoS solutions, on the contrary, are not bound specifically to any goal of optimization. Elasticity-based solutions aim to minimize resource usage while maintaining the QoS. In other words, they use as few resources as possible for an application, and trigger a scaling operation when the load is above or below a given threshold. Fault tolerance-based solutions involve migrations when nodes fail and the operators must be migrated to new or existing nodes.

Network performance as a goal of optimization means using the quality of the network links to determine performance in terms of placement. Important metrics in this context include the bandwidth between links in the overlay topology, the latency between nodes, and the bandwidth delay product. Tuple processing performance in query processing is the most popular indicator of the quality of an adaptation, as shown by the number of studies that have measured the benefit of migration in terms of tuple latency or rate. If a node is overloaded in a data center, the latency of the tuple might exceed acceptable levels, leading to QoS violations. A long migration time might temporarily worsen performance, but if the general gain in performance outweighs the degradation, the migration is considered worth it. The load of a system is an important goal of optimization that makes it possible to run as many operators on a node as it can handle, and to make changes when the workload is above or below a given threshold. The cost of migration is essential to consider when making migration decisions to avoid excessively frequent migrations and ensure that the benefit of the new placement outweighs the cost of migration. When the cost of migration is used to calculate its benefit, the result is the modeled benefit of migration. Monetary cost can be useful as a goal of optimization to make a tradeoff between the cost of resources and the performance of the system.

IV.5.4.1 Network

In decentralized fog and edge computing solutions, network usage as well as bandwidth and latency between links are crucial metrics. Pietzuch et al. [108] developed an overlay network that can make network-aware placement and migration decisions. Parameters, like the latency and bandwidth of overlay links and the load on nodes are used as criteria of optimization when placing and migrating operators. Rizou et al. [115] implemented a similar method that converges to the optimal placement in fewer migrations than the solution by Pietzuch et al [108].

IV.5.4.2 Tuple performance

Being able to analyze streaming data as soon as it arrives and to react immediately to certain patterns in the data is one of the core motivations for SPEs. Therefore, tuple performance is of significant importance. Furthermore, in a resource-constrained environment, tuple latency can be an indicator of energy consumption and the goal to minimize latency can implicitly lead to energy reduction. For most existing approaches, the goal of migration directly or indirectly involves improving performance. Most elasticity-based and load balancing-based solutions are cluster-based, and are more concerned with the load on the system than the bandwidth of or latency between links.

The tuple rate of a data stream can be used to detect backpressure, i.e., when the input rate is higher than the processing rate. This can be used as an indicator of the load on the nodes, and to calculate the variance in load. For instance, Buddhika et al. [15] proposed a methodology to reduce the interference between stream processing operators using migration. To achieve this, the interference score of an operator is calculated, where the higher the score is, the greater the need is for migration. This interference score is based on the prediction of future packet load. Similarly, the WASP system [60] relies on the expected input and output rates of an operator instead of merely on the observed rates. Repantis et al. [114] defined latency constraints on the operators and used tuple latency to determine when an operator must be migrated.

Tuple latency is a common constraint to have on the operator performance. If the latency constraints are not kept, it causes the system to scale out or perform load balancing. Röger et al. [119] studied the relationship between latency constraints and monetary costs. In particular, the lower the latency constraints are, the more instances a system needs to run, and thus, the more costly the operation is. As such, the optimization problem is:

$$\begin{aligned} \min \quad & \sum_{cu \in path} cost(cu, latency) \\ \forall paths \quad & \sum_{cu \in path} latency(cu) <= \text{e-to-e latency bound} \end{aligned} \tag{IV.6}$$

IV.5.4.3 Load

Unsurprisingly, all load balancing solutions use either load as a parameter when making decisions or tuple performance to model load. One method is to minimize the variance in load between nodes in a cluster [36]. In this case, a coordinator monitors the load on the system nodes and, when a balancing decision has to be made, selects the configuration with the least variation in load. This might, however, require expensive migrations of large loads among many nodes, and redistributing loads that are not the cause of the imbalance. Another method is to trigger a load balance when the imbalance has crossed over a given threshold to re-balance the load to at least below a given threshold [36]. In other words, load balancing is used as a constraint. In this case, the goal is to minimize the cost of migration by redistributing the minimum amount of load to achieve an acceptable load balance. This method achieves an acceptable load balance while moving the smallest load.

Resource usage can mean multiple metrics that relate to the system's workload. It can be the number of threads assigned to the operators, as described in the work by Xu et al. [151]. Alternatively, it can also be understood in terms of the CPU queue state of a computing node, as illustrated by Sun et al. [131]:

$$l_{cn,[t_s,t_e]} = \sum_{v_i \in V_{cn,[t_s,t_e]}} n_{v_i,cn,[t_s,t_e]}, \tag{IV.7}$$

where $n_{v_i,cn,[t_s,t_e]}$ indicates the number of tuples of operator v_i during $[t_s, t_e]$, and $v_i \in V_{cn,[t_s,t_e]}$. The load is then partially based on the tuple rate, and later used as a constraint in a minmax load balancing optimization problem.

Elasticity-based solutions increase or reduce the number of resources used by an application based on its variable workload. If a cluster is overloaded after load balancing, this is a sign that the system should scale out [77]. In decentralized fog-based solutions, the load of a system is not known beforehand, and therefore, there might be a tradeoff between latency and load. Pietzuch et al. [108] introduced a cost space model in which a topology of systems is constructed based on the latency and bandwidth between nodes as well as the load on systems. If the load of a system is large, the relevant node appears farther in the cost space when mapping an operator graph to a physical topology, and thus is less likely to be selected.

IV.5.4.4 Monetary costs

In the cloud model, followed by the fog and edge models, users mostly pay based on usage. Users can allocate a certain amount of resources and scale out or in whenever more or less resources are needed, to keep the resource usage cost low. A complicated issue in this case is balancing the monetary costs with the benefits of improved placement. None of the load balancing solutions use monetary cost as an optimization criterion. This makes sense as the load balancing problem involves evenly distributing the load over a fixed amount of resources, whereas elasticity can increase or reduce the amount of resources. In terms of hardware resources, there is nothing to optimize as they are already paid for. The monetary cost of moving states during migration can thus be minimized. Typically, this is implicitly done by designing the objective function to minimize the number of state that need to be moved. Elasticity-based solutions require a tradeoff between resource usage and monetary costs [18, 52, 154]. An elastic solution might use a threshold for the load to determine when to scale out. However, deciding when to scale in might be more complex, considering that it requires a certain downtime for the worker to be removed.

IV.5.4.5 Costs of migration

Any type of migration introduces some costs. However, this does not mean that a migration always affects the QoS. If no tuples arrive during the downtime of the operator no tuples will be affected and neither the QoS. Most studies aim to prevent the cost of migration from affecting the QoS by implicitly minimizing the number of migrations, their frequency, or their magnitude. Zhou et al. [158] emphasized the need to minimize the time needed for query migration but did not describe a means of implementing this in their solution. Lombardi et al. [78] defined the cost of migration in terms of the time it takes to perform different steps but did not attempt to minimize it. The cost of migration can be minimized by either using single-objective optimization [32, 36, 60, 142], or simple additive weighting (SAW) with multiple objectives [18, 19, 56, 61, 66]. If only the cost of migration is minimized, constraints have to be placed on the quality of the placement to ensure that the selected placement is acceptable. With load balancing, minimizing the cost of migration while maintaining constraints on the load imbalance is a good way to ensure a balanced load that minimally affects the performance of the system.

Minimizing the number of migrations is a similar goal to minimizing the cost of migration. Repantis et al. [114] proposed a hotspot alleviation-based solution with the goal of minimizing the number of migrations that leads to an acceptable QoS for the operators. Rizou et al. [115] implemented a similar relaxation algorithm to the one in [108], and showed that it requires fewer migrations before converging to the optimal placement and fewer control messages. The easiest way to prevent needless migrations is to use a threshold that ensures that they are beneficial. Load balancing systems commonly use thresholds of load imbalance to ensure that the load is redistributed only when the load imbalance is above a certain threshold. A different type of threshold targets the migration itself to ensure that its benefit is worth its cost. Pietzuch et al. [108] proposed a method that migrates data only when the benefit in terms of network capacity is higher than a threshold based on the cost of migration.

Using the cost of migration as a goal of optimization means penalizing a placement alternative based on it. Even if a placement is better than the given placement, it might not be preferred because the cost of the reconfiguration is too high. In load balancing-based approaches, the cost of migration is commonly minimized but most often as an implicit goal rather than as part of the objective function. The goal is generally to achieve an acceptable load distribution as quickly as possible, and the redistribution itself constitutes the highest cost. The cost of migration can be minimized while maintaining a balanced load [32]. The number of migrations can be minimized while fulfilling QoS requirements [114]. Another way is to maximize the improvement in a query plan and divide the improvement in performance by the cost of migration [66]. Load balancing decisions can be made with cost of migration in mind in multiple ways [36]: minimizing the cost of migration with load balancing as a constraint, keeping the cost of migration as a constraint while minimizing the load imbalance, or combining load and cost of migration to minimize both.

In elasticity-based approaches, the cost of migration is often considered in the same way as in load balancing because scaling can be considered to be an extension of load balancing. Zacheilas et al. [154] minimized the monetary costs of computational resources, the cost of migration, and the cost of missing tuples. In this approach, a tradeoff is made between the cost of resources, the cost of missing tuples, and the migration time. A reinforcement learning-based approach was used in [18] that minimizes the cost of reconfiguration, the performance penalty due to QoS constraints, and the cost of resources for using the computational resources.

IV.5.5 Proactive migration decisions

Current migration solutions generally use reactive approaches to make migration decisions. That means migration is a reaction to a certain trigger-event that happened. For instance, a migration might be triggered if a node is overloaded and QoS guarantees are violated, such as when the tuple latency increases excessively. In contrast, proactive migration is performed before a trigger-event happens. This means that the trigger-event needs to be predicted, typically based on historical monitoring data. Most proactive solutions predict whether the node can sustain the workload.

In Table IV.12, we summarize the studied papers that contribute with proactive decision-making strategies for operator migration. For each paper, we highlight the specific prediction approach and its key idea.

Paper Reference	**Prediction Approach**	**Key Idea**
Repantis et al. [114]	Linear Regression	Uses incoming tuple rate to predict QoS violations
Lohrmann et al. [77]	Queuing Models	Develops a predictive latency model for scaling decisions using queuing models and Kingman's formula
Zacheilas et al. [154]	Gaussian Processes	Estimates load and expected latency for scaling decisions
Liu et al. [75]	Extended Gaussian Processes	Uses the upper confidence bound algorithm to search for optimal configuration for bottleneck operators
Wang et al. [140]	Incremental Learning	Predicts resource usage in real-time to choose a configuration that minimizes CPU and memory resources
De Matteis et al. [26]	Model Predictive Control (MPC)	Predicts optimal scaling decisions
Buddhika et al. [15]	Prediction Rings	Forecasts the interference score to predict system overload
Lombardi et al. [78]	Tuple Rate	A proactive scaling mode that predicts the input load over a specific horizon to make scaling decisions.
Cardellini et al. [18]	Reinforcement Learning	Applies reinforcement learning to decide when to perform scaling operations
Liu et al. [73]	Load Prediction	Predicts the load of operators as the number of tuples they need to process during a prediction horizon
Jonathan et al. [60]	Operator Input and Output Rate Estimation	Uses expected input and output rates for load estimation
Lindeberg et al. [70]	Window State Monitoring	Uses knowledge of the window state for migration scheduling
Geldenhuys et al. [38]	Multiple Regression and Time Series Forecasting	Predicts future workloads for near-optimal scaling decisions

Table IV.12: Proactive migration prediction techniques

There are many ways to model or estimate the metrics described above.

The classical way is to collect some measurements from possible migration hosts and formulate an optimization problem using, e.g., ILP, and then attempt to solve it. This is done by Cardellini et al. in [19, 20].

Some solutions predict the adaptability of QoS violations [77, 114]. Repantis et al. [114] used linear regression and the incoming tuple rate to predict QoS violations of the end-to-end execution time. They predicted QoS violations to prevent them. Lohrmann et al. [77] built a predictive latency model using queuing models and Kingman's formula [63] to make scaling decisions.

Zacheilas et al. [154] estimated the load and expected latency of Esper to make scaling decisions by using Gaussian processes [112] because they can help to estimate the uncertainty in predictions. However, this method has a cubic computational complexity due to the use of matrix inversion. Liu et al. [75] used the extended Gaussian Processes upper confidence bound algorithm to search for the optimal configuration for bottleneck operators for modeling the service capacity. Wang et al. [140] predicted resource usage in real time to choose the configuration that can minimize CPU and memory resources while fulfilling QoS guarantees. This is done using incremental learning techniques based on Weka [148] and MOA [97]. De Matteis et al. [26] used MPC to predict optimal scaling decisions, called the future horizon. Buddhika et al. [15] used prediction rings to forecast the interference score that expresses the degree to which a system is expected to be overloaded. Lombardi et al. [78] used a reactive and a proactive mode for making scaling decisions in their Elysium system. In the reactive mode, the tuple rate is used as the basis for decisions, and in the proactive mode, the input load is predicted over a certain time, called the prediction horizon.

In [18], a reinforcement learning approach is applied to decide when to perform scaling operations. Liu et al. [73] predicted the load of operators as the number of tuples that operators need to process during a prediction horizon. In WASP [60], the expected input and output rates of the operators are estimated as an alternative to backpressure monitoring for estimating load. Backpressure is weaker as it is based on the observed load instead of the actual workload, and this may lead to less accurate adaptation decisions [62]. A composition of reactive, proactive, and delayed migrations was presented in [70]. The results of this empirical study indicated that knowledge of the window state can be used to schedule a migration when the state is minimal (i.e., after completing a tumbling window, as in [80]), or when no output tuple is affected by the migration.

The Phoebe system [38] predicts future workloads as a way to make near-optimal scaling decisions. This is done using models for predicting the end-to-end latency of tuples and recovery time of the system. The end-to-end latency model uses multiple regression and clustering for estimating latencies of scaleouts and workload rates. The recovery time model bases its predictions on multistep-ahead time series forecasting [132] of the expected workload rate over time and a regression model for predicting the maximum processing capacity of the system.

IV.6 Empirical quantification of core concepts of migration

The previous two sections have given the reader an understanding of how operator migration works and which design choices for migration mechanisms and decisions have been investigated in existing works. The aim of this section is to complement the understanding of the functional aspects of operator migration with some insights into the impact of design decisions on the performance of operator migration. Therefore, we first define two direct moving state migration algorithms: (1) one that uses partial state movement, and (2) another that sends the entire state at once. They are defined in an abstract way such that they can be implemented in different SPEs, and we have decided to implement them in the two popular SPEs Apache Flink and Siddhi. We define decision models to determine when and where to migrate the data. We conducted a real migration experiment to analyze the migration algorithms based on the NEXMark benchmark [134]. We show a use case of the decision models for migration to illustrate their effect on decision-making.

IV.6.1 Migration algorithms

The difference between the partial state movement algorithm and the all-at-once state movement algorithm is that the former splits the state into a large static state and a small dynamic state. The static state is transmitted while the operator is still running and processing tuples, followed by the extraction of the dynamic state. As such, static state transmission involves little or no overhead in query processing, and constitutes only one additional step in the algorithm. Note that a partial state movement algorithm might split the state into more than two parts, such as in Megaphone [53] and Meces [41].

We use the algorithms described in Section IV.4 as basis. In particular, we divide the algorithms into functions that are executed by different nodes participating in the network according to their roles. When moving the state, the old host provides the next hops for the query. Thus, there is no need to add them explicitly in these tasks. These tasks follow a similar format to that used in Expose [138], which is a framework and toolset for efficiently defining and executing DSP experiments. Wrappers for different SPEs are provided such that all SPEs support a common set of tasks. Expose has been extended with additional tasks to enable operator migration.

Listings IV.6 and IV.7 describe the tasks we use to define the all-at-once state movement algorithm and the partial state movement algorithm, respectively. They differ slightly from similar algorithms in Section IV.4 in some respects, such as the ways in which streams are managed. It is possible to send a batch of tasks to upstream nodes, as in Flux [124], to the new host as in [56], and to the old host as in [113]. The difference between the all-at-once state movement and partial state movement algorithms is that the latter involves sending the current state of the query before the operator is paused, achieving the same effect as in checkpoint-assisted solutions.

```
ControlMessage(OH
    ControlMessage(NH,
        RequestMigration(query),
        BufferStreams(Streams(query))
        StopStreams(Streams(query)))
    ControlMessage(Upstream,
        Redirect(Streams(query), OH, NH))
    MoveState(query, NH)
    AddNextHop(Streams(query), NH))
```

Listing IV.6: All-at-once state movement

```
ControlMessage(OH
    ControlMessage(NH,
        RequestMigration(query),
        BufferStreams(Streams(query))
        StopStreams(Streams(query)))
    MoveImmutableState(query, NH)
    ControlMessage(Upstream,
        Redirect(Streams(query), OH, NH))
    MoveIncrementalState(query, NH)
    AddNextHop(Streams(query), NH))
```

Listing IV.7: Partial state movement

Implementation To facilitate the migration of any moving state operator, an SPE needs to be able to extract the runtime state and the load state. This feature is supported in different ways by Siddhi and Flink. In Siddhi, the state is loaded from the runtime system into a byte array, and requires that the entire state is available in memory. As such, there are limitations on how large the state can be. On the contrary, Flink writes the state as a set of checkpoint files, each of which does not exceed a configurable size. Therefore, the state to migrate with Flink can be larger than in Siddhi. The implementation of the other tasks, including `BufferStreams`, `StopStreams`, `ControlMessage`, `AddNextHop` and the rest, is supported through simple tasks defined in the SPE wrapper in Expose [138].

The standard moving state algorithm is implemented in Flink and Siddhi, but only Flink supports partial state movement since this requires the ability to split a given state into a large, immutable state and smaller incremental checkpoints. This feature is supported by one of the state backends in Flink called RocksDB [116]. Flink with RocksDB is also used for the checkpoint-assisted algorithm in Rhino [28], which uses partial state movement. Another benefit of RocksDB is that it does not store the entire state in memory while the system is running, but instead writes it to file and minimizes its size based on multiple criteria.

IV.6.2 Decision models

As discussed in Section IV.5, there are many different ways of making the migration decision. Our solution is to make the decision process as transparent and meaningful as possible by optimizing the QoS. The goal is to maximize the performance of a placement while penalizing it based on the cost of migration, which varies for different nodes and is zero for the current host. In this way, it is clear why a new placement is selected over the old one, for reasons other than simply that the old host is over-provisioned or the new placement delivers better performance.

The amortization time (at) varies depending on the reliability of a placement score for the operator on a given host. If the placement score is stable over time, the amortization time increases since it is less likely that the placement becomes suboptimal shortly after the migration. For instance, a mobile node might have less consistent placement score than a server located in a data center, and as such, it is even more important that the migration is worth the cost of it.

$$at(h, op) = min_{at} + (max_{at} - min_{at})/100 * (100 - rsd_p(h, op)) \tag{IV.8}$$

where $rsd_p(h, op)$ expresses the relative standard deviation (RSD) of the historical placement scores of operator op on host h.

When defining the cost of migration, operator downtime alone is not sufficient, because it does not reveal how many tuples, if any, are affected by the downtime. Therefore, we use the tuple rate during the migration as a foundation for the cost of migration. Since the data sink waits for tuples from the operator, we consider the number of expected output tuples $PT_{out}(at, op)$ that are affected by the migration to calculate its cost. Buddhika et al. [15], Phoebe [38] and Liu et al. [75] describe tuple prediction methods that can be applied here.

$$PT_{out}(at, op) = PT_{in}(at, op) * Sel(op) \tag{IV.9}$$

where $PT_{in}(at, op)$ is the predicted number of input tuples for operator op during amortization time at and $Sel(op)$ is the selectivity of operator op, which could for instance be a join, pattern-matching operator or an aggregation operator.

The cost of migration can be calculated as the operator downtime divided by the amortization time. Since we focus on output tuples from the query, the cost of migration $C(op, oh, nh)$ is defined as the ratio of the predicted output tuples ($PT_{out}(mt(oh, nh, op)$) from a query during migration to the output tuples predicted from it during the amortization time ($PT_{out}(at(nh, op))$).

$$C(op, oh, nh) = w_c * \frac{PT_{out}(mt(oh, nh, op))}{PT_{out}(at(nh, op))} \tag{IV.10}$$

The cost has a weight associated with it, meaning that the system can dynamically change how much the cost of migration matters based on the selected policy. If w_c is set to one, this suggests that the performance of a placement should be reduced in proportion to the number of tuples that are received during operator downtime. If w_c is set to 1.5, the placement is penalized further. This makes sense as buffered tuples may take some time to process, during which time no new tuples may be processed.

Given the amortization time, the benefit of the migration $B_m(op, oh, nh)$ of a placement is its finite performance penalized by the cost of migration, instead of it being a general placement score. Of two placements with the same migration cost, the one with the higher placement score is selected. The only difference arises when two placements have different costs of migration, for instance, when comparing the given placement with zero cost of migration with another placement that requires a migration. The benefit of migration can be calculated as:

$$B_m(op, oh, nh) = P(nh, op) * (1 - C(op, oh, nh)) \tag{IV.11}$$

where $P(nh, op)$ is the estimated placement score for the new host nh running operator op.

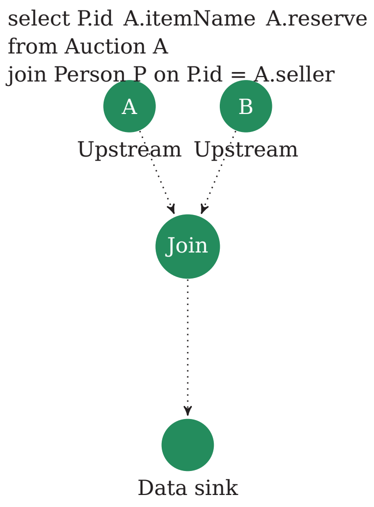
The above functions show how the migration decisions are made. Migration checks are periodically performed by calculating the placement score. Following this, the benefit of the migration of placements is calculated by penalizing the placement score based on the cost of migration. We define $M(oh, phs, op)$ as the potential host with the maximum benefit for the given operator. This host is selected as the future host for the operator, and triggers migration if it is not the given placement.

$$M(oh, phs, op) = \max_{ph \in phs} B(oh, ph, op) \tag{IV.12}$$

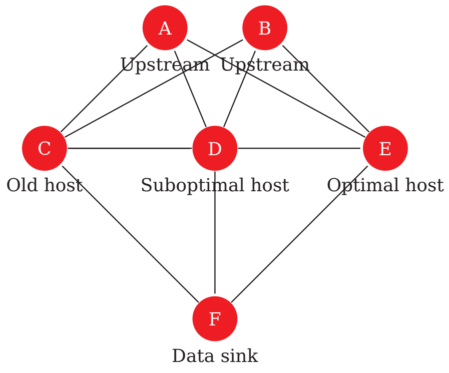
IV.6.3 Empirical evaluation

We quantitatively analyzed the proposed decision models for migration through a use case and our migration algorithm through experiments. The goal was to show the usefulness of incorporating the cost of migration into the process. We considered a use case for the decision models, because it makes the analysis and discussion of the results easier. On the other hand, implementing and running the migration algorithms on SPEs is necessary to understand the impact of migration.

Figure IV.15 illustrates our evaluation scenario: Figure IV.15a shows the operator graph used for both the use case and the migration experiment, and Figure IV.15b shows the DSP overlay topology. The mapping from the operator graph to the physical topology is demonstrated using the decision models in Section IV.6.3.1, and an experiment involving the migration of state from the join operator on one node to another is described in Section IV.6.3.2.


select P.id A.itemName A.reserve
from Auction A
join Person P on P.id = A.seller
A
B
Upstream
Upstream
Join
Data sink

(a) Operator graph



(b) DSP overlay topology

Figure IV.15: Evaluation scenario

IV.6.3.1 Decision model use case

The decision models for migration were assessed in this use case. They were applied using a prediction model oracle with 100% accuracy to make migration decisions. We expect that the migration time can be predicted based on periodically updated topological information and network statistics. By using knowledge of the number of tuples sent in the time window and the migration time, we can predict the total end-to-end latency of the tuples during a given time window. The parameters of the use case are provided in Table IV.13. We considered two source nodes A and B, three potential hosts C, D, and E, and a sink node F.

Parameter name	Parameter value
Amortization time	5 s
Bandwidth C$<->$D	200 mbit/s
Bandwidth C$<->$E	100 mbit/s
Bandwidth D$<->$E	100 mbit/s
Bandwidth Leader$<->$Hosts	200 mbit/s
Latency between all links	1 ms
Control message size	168 bytes
Migration cost (C)	0
Migration cost (D)	0.1
Migration cost (E)	0.5

Table IV.13: Parameters of the use case

Results Table IV.15 shows the results of the use case for the configurations given in Table IV.13 and the run-specific parameters are provided in Table IV.14, including amortization time and placement scores for the potential hosts. These scores would in a real-world system be calculated based on the expected performance of a node that runs the system. Node E has the best P score in all runs, as illustrated in Figure IV.15, which means it is the best candidate for running the operator. However, the cost of migration, as indicated in Table IV.13, shows that Node E has five times higher migration cost than migration to Node D, which leads to Node E only being preferred when the cost is ignored (M (NCM)), and Node C and D are preferred when the cost is considered (M (CM)). The previously described equations are used along with the at and P scores to define the benefits of the potential placements, and the preferred host is selected based on M.

Run	at	P (C)	P (D)	P (E)
1	1000	1.5	1	2.7
2	2000	1.6	1.6	2.5
3	3000	1.4	2.5	3
4	4000	1.7	2.8	2.9

Table IV.14: Placement score and amortization time parameters for each specific run in the use case

Since the P score was stable for Node E, and was significantly better than that for Node C, a decision policy may decide to dynamically increase the

Run	B_m (C)	B_m (D)	B_m (E)	M (CM)	M (NCM)
1	1.5	0.85	1.35	C	E
2	1.6	1.36	1.25	C	E
3	1.4	2.125	1.5	D	E
4	1.7	2.38	1.45	D	E

Table IV.15: Results of the use case

amortization time for nodes that had demonstrated their stability in terms of the predicted P score. On the contrary, Node D has a significantly variable P score, which increased above that of Node C with a value of 1.6 in the second row, but yielded a lower benefit of migration of 1.36, and was not selected as the new host. With a score of 2.5 that was reduced to 2.125 given the migration cost, it beat the given host, and was selected as the new host.

IV.6.3.2 Migration experiment

In this experiment, we demonstrated two migration algorithms by analyzing and comparing their execution results in two different SPEs, with one SPE limited to a single algorithm. The all-at-once state movement runs were used to send 100,000, 1,000,000, and 5,000,000 tuples. The partial state movement runs were used to send between 1,100,000 and 300,100,000 tuples. The noticeable differences in the number of migrated tuples between the all-at-once and partial state movement runs were a result of the limitations in the SPE's state backends, as explained in further detail below. The additional 100,000 tuples with partial state movement were sent during migration, and were part of the dynamic state to be sent. The experiment used a simplified version of the topology in Figure IV.15b in two ways. First, there was only one upstream node. Second, there were only two hosts: the old and the new host.

The experiment tested the cost of migration by varying the size of the state to be moved. For runs of the partial state movement, the number of tuples that were migrated during static state migration and dynamic state migration were varied. The dataset of the NEXMark stream processing benchmark [134] was used in the experiment. NEXMark is based on an auction scenario, where three streams are used: a Person, a Bid, and an Auction item stream. For this experiment, only one of the queries was used, one that joined the Person and Bid streams. We used this query because a join query makes it easier to test the migration algorithm and adjust the size of the state to migrate. One can simply send a given number of tuples of the first stream, migrate it to the new host, and send a single tuple of the second stream to the new host. If this triggered the correct number of output tuples to be produced, the migration was considered to have been successful.

Four processes with different roles were used in the experiment: a data producer node, an operator host running the operator to be migrated, a new host that contained the operator after migration, and the data sink that consumed the output tuples of the operator. We used two machines for the experiment, one for the old host, and the other to run the data producer, data consumer, and new host. The machines were connected via Ethernet cable in a local area network. The specifications of the machines are shown in Table IV.16. In the experiment, the data producer generated a certain amount of Auction tuples that were sent to the old host. The state was then migrated to the new host, and the data producer sent a single Person tuple that joined with all the Auction tuples to trigger the same number of output tuples to be sent to the data sink as Auction tuples that were sent prior to the migration. The query we used was a modification of NEXMark's [134] Query 8. Originally, this query does not select the itemName of the auction, but chooses the person's name. Each Auction tuple was augmented with 1 kB of a randomized string to increase the size of the state to be migrated.

OS	CPU	RAM	Description
Ubuntu 20.04.2	Intel Xeon Gold 5215 2.50 GHz	125 GB	Old host
Ubuntu 18.04.4	Intel Core i7-7800X 3.50 GHz	377 GB	New host, data producer, data sink, Expose coordinator

Table IV.16: Server specification

In all runs, we counted the number of tuples that were migrated, the state size, the state extraction time, the state transfer time, and the state loading time. For the partial state movement algorithm, the same parameters were used for the static and dynamic states. The state to be migrated ranged from 1 to 300 GB. However, Siddhi has a limit of 1 GB because it extracts the entire state into a single byte array, whereas Flink's state backend RocksDB splits the state into multiple files. RocksDB is used in both the all-at-once approach and partial state migration approach, but where all-at-once disables incremental checkpointing and partial state migration enables it. When migrating all-at-once with Flink, the maximum state that could be migrated is 5 GB, because the checkpoints fail at larger states. It is unknown why this issue occurs. It would be possible to run Flink with all-at-once migration with incremental checkpoints, but then it would be the same as the partial state migration, except where the state transfer of the static state is added to the freeze time.

Results Tables IV.17 and IV.18 show the experimental results of the all-at-once state movement algorithm and the partial state movement algorithm, respectively. Siddhi and Flink migrated operator states of different sizes depending on the query and the number of tuples that were processed.

The state transfer times of Siddhi and Flink were similar because they used similar implementations of the TCP socket. Siddhi performed slightly

SPE	# Tuples	State size	State extraction	State loading	State transfer	Freeze time
Siddhi	100,000	100 MB	2.8 s	1.76 s	0.94 s	3.6 s
Siddhi	1,000,000	1 GB	21.3 s	13.6 s	9 s	43.9 s
Flink	100,000	100 MB	270 ms	2.3 s	1.1 s	3.6 s
Flink	1,000,000	1 GB	3.3 s	20.3 s	11.6 s	35.2 s
Flink	5,000,000	5 GB	19.8 s	77.4 s	52.8 s	150 s

Table IV.17: Results of all-at-once moving state experiment

SPE	Tuple count (S)	Tuple count (D)	Size (S)	Size (D)	State extraction (S)	State extraction (D)	State loading	Transfer (S)	Transfer (D)	Freeze time
Flink	1,000,000	100,000	1 GB	113 MB	76 ms	149 ms	1.77 s	11.6 s	111 ms	2 s
Flink	5,000,000	100,000	5.2 GB	376 MB	363 ms	528 ms	3 s	57.2 s	4.3 s	7.8 s
Flink	25,000,000	100,000	26.1 GB	3 GB	23.3 s	1 s	8.6 s	4.35 min	30 s	39.6 s
Flink	50,000,000	100,000	52.2 GB	2.74 GB	15.4 s	1.2 s	16.8 s	8.67 min	27 s	45 s
Flink	100,000,000	100,000	104.5 GB	239 MB	618 ms	911 ms	63.8 s	17.47 min	2.5 s	67.2 s
Flink	200,000,000	100,000	208.8 GB	279 MB	61.2 s	18 s	7.7 min	35.1 min	2.6 s	8 min
Flink	300,000,000	100,000	313.3 GB	4.9 GB	31.7 s	16 s	13.1 min	52.5 min	53.3 s	14.3 min

Table IV.18: Partial moving state experiment results

better, because Flink had to read the checkpoint from multiple files, and state transfer was executed in parallel with reading the files. State extraction appeared to scale relatively poorly for both Siddhi and Flink with the all-at-once state movement algorithm, but with the partial moving state, Flink had a significantly lower state extraction overhead. Moreover, state loading using partial state movement was much faster than without it. Note that these results do not represent the general performance of the SPEs, but the outcomes for a specific join query that was used for a specific system. Another query might have yielded different results. For instance, this query was very write heavy and the Auction tuples were made to be larger in size than the benchmark normally defines. In this case, the partial state movement algorithm performed better in all respects.

One might think that the all-at-once state movement algorithm would have had faster state loading as it has a monolithic checkpoint, but this was not the case. We think this result is obtained because the incremental checkpointing uses RocksDBs native checkpoint files whereas Flink's full snapshot approach iterates through the RocksDB state and creates its own files. RocksDB is designed to be efficient, and performs indexing to increase its efficiency. This benefit was lost in the full snapshot approach.

If we assume that the number of tuples that were received during the freeze time arrived at a fixed rate, the average additional tuple latency as a result of the migration would be equal to half the freeze time. The maximum additional tuple latency would be approximately equal to the freeze time and the minimum was close to zero. The number of affected tuples could vary significantly, ranging from zero to hundreds of thousands per second.

The partial state movement algorithm performed much better than the all-at-once algorithm in terms of freeze time, almost 20 times less freeze time for the partial state movement algorithm versus the all-at-once movement algorithm when the state to migrate was around 5 GB. There are two reasons for the performance gain. First, using the incremental checkpointing led to lower state loading times. Second, most of the state was moved before the operator was shut down. This difference in performance was especially significant when considering how similar the algorithms were in terms of how they were described in Listings IV.6 and IV.7. Only one task was added to Listing IV.7, which was to migrate the immutable state before the streams were redirected by the upstream nodes. This leads to the important conclusion that the literature can benefit from a common language when defining or using a migration algorithm. Exactly what tasks are executed during the migration, in particular, those that increase the freeze time, can be described using, e.g., the concepts of the migration model in Section IV.3.

Table IV.18 shows that the size of the dynamic state in the partial state movement runs was unpredictable. For instance, when 25 million tuples were migrated, the size of the dynamic checkpoint containing 100k tuples was 3 GB, whereas it was only 279 MB for 200 million tuples. However, the actual size of the 100k tuples remained around 100 MB across all runs. This could be attributed to the fact that we disabled RocksDB compaction after extracting the static state. Had we kept the compaction enabled, the final incremental checkpoint would have potentially merged with the static state, resulting in a new set of state files that would be incompatible with the ones sent to the new host.

IV.7 Reflections and Future Directions

The historical development in operator migration, from the early single-track moving all-at-once state migration solutions to checkpoint-assisted partial state movement and parallel-track solutions without state movement, has been driven by the deployment of SPEs to the cloud environment, and improvements to them to achieve fault tolerance and dynamic scalability. The core ideas to achieve this are related to resource availability and state management. Cloud environments provide large amounts of computational resources (even though at different scales), and their servers are interconnected with low-latency high-bandwidth networks. This allows to execute operators in parallel to improve migration performance at the cost of higher resource utilization. Early single-track migration solutions treat the state as a single large binary object as such there is no other way to migrate the entire state all-at-once.

More advanced state management solutions allow to partition the state which in turn leads to more design options for migration solutions. Checkpointing, distinguishing between immutable and mutable state, and prioritization of state partitions are examples for partition mechanisms. The more advanced solutions, e.g., based on prioritization, consider the semantics of the state to determine which piece of the state should be migrated first to improve the migration performance. Very recent approaches follow the idea of prioritization to perform state shedding, which reduces the size of the state to be migrated at the cost of inconsistency between the state at the old and new host. Another way to reduce the state size is to schedule the migration. Based on these insights on how to improve operator migration for cloud-based environments it is reasonable to expect that such solutions might also work well in fog environments.

However, in fog environments that are geo-distributed, the connections between hosts have substantially lower available bandwidth and higher latencies that can impact the cost and benefit of operator migration, and require adapted migration mechanisms. The periodic checkpointing and replication of checkpoints are used in some cluster-based SPEs to facilitate fault tolerance and fast migrations, but it is not always feasible to replicate and distribute checkpoints, especially in resource constrained IoT devices. For future in-network processing solutions with mobile platforms, e.g., advanced crowd-sensing.

It is clear that the smaller the size of the data to be migrated is, the less energy is consumed. Therefore, scheduling operator migration at a point in time when the state is small or even zero is important. This can be achieved, for example, by delayed migration by waiting until a tumbling window is emptied [103], and through proactive migration. Another alternative is to allow for some inconsistent state, i.e., not the entire state is migrated to the new host. In some cases, aggregation operators can be moved without the state, resulting in zero freeze time. Alternatively, load shedding techniques can be applied to send some of the state, or components of it can be assigned a priority such that only the most important state is migrated, while the less significant part of it is omitted. However, a thorough investigation of the pros and cons of reactive, delayed, and proactive migrations in different environments with different workloads and guarantees of consistency is still elusive.

Another gap in research is an analysis and comparison of stream management techniques. Several aspects are important for such an investigation: (1) the sequence of tasks like the stopping, buffering, redirecting, and starting of streams, (2) the locations where streams are buffered, (3) the delivery semantics, i.e., at least once, at most once, exactly once as well as ordered or out-of-order delivery, and (4) tasks related to buffer management and transport protocols.

The quality of decision-making on migration depends on the data available to calculate its cost and benefit, as well as the freshness of the data. The continuous collection and dissemination of monitoring data in DSP can be expensive. Efficient monitoring solutions, and leveraging other sources of monitoring data that are, for example, used for network and system management have the potential to reduce the overall cost of DSP and ensure good decision-making.

Leveraging historical data to perform predictions with advanced statistics or modern machine learning solutions, as is done for traffic prediction in network management [8] and data prediction in wireless sensor networks [29], is another subject that deserves more attention in research. Some studies have already explored proactive migration techniques, as discussed earlier in the paper, but further investigation into this area remains necessary. Both proactive migration and the use of amortization time in the cost model require some form of prediction. The oxymoron of operator migration, i.e., that the need for migration occurs when the cost of migration is high, can be avoided with proactive migration. Furthermore, proactive migration can be used to schedule a migration when the state is still small in size. However, both traffic and data patterns might be changing during the deployment of DSP systems, and appropriate and efficient online learning solutions need to be investigated for operator migration.

One promising research direction is to further explore the application of DSP in MEC scenarios. While many previous works have focused on the migration of services, most of them have assumed an all-at-once migration approach. It is worth investigating how partial state migration, state shedding, parallel-track, and distributed checkpoint replication algorithms can be adapted to the more challenging and geo-distributed MEC settings. Specifically, how can these migration mechanisms affect decision-making and change the frequency of migration, compared to the all-at-once approach?

A key challenge in MEC scenarios is the significant increase in migration time due to limited bandwidth and variable hardware resources among potential operator hosts. Therefore, we need to investigate how different migration mechanisms can be applied in such settings, especially in low-bandwidth scenarios where these mechanisms could have the largest benefits. For example, a parallel-track migration mechanism might be a good fit for MEC, as it splits the data streams into one for migration and one for processing. This allows for normal processing to occur while the migration is ongoing, resulting in a smooth handover without discernible downtime. Alternatively, we could consider using the partial state migration approach Megaphone, which could be particularly useful in low-bandwidth scenarios. In contrast, within data centers, these mechanisms may not be necessary, because migration times are significantly lower due to high bandwidth.

IV.8 Conclusions

DSP is becoming increasingly important for handling data with high velocity and large variety. The variety is caused by data from different sources and over time as well as other system dynamics, e.g., resource availability, require adapting DSP accordingly. Operator migration is the mechanism for keeping the DSP in an "optimal" configuration over its lifetime. However, operator migration is a complex task that can be solved in many different ways, i.e., there are many design alternatives for operator migration. Which of those alternatives are a good choice depends on factors like the deployment environment, the system goal, workload, etc.

To enable the reader to gain a good understanding of how operator migration works and the design space for it, we introduced a conceptual model of operator migration based on the largest common denominator in the literature to establish a common and unified terminology and taxonomy. In the model, we separated clearly mechanism and policy, i.e., migration mechanism and migration decision. For the latter we placed emphasis on its costs and benefits. The description of existing solutions shall provide the reader with an overview on existing solutions and further foster the understanding of the design alternatives from an algorithmic viewpoint. We complemented this with an empirical study to give the reader some quantitative insights into the impact of different design alternatives for migration mechanisms (i.e., all-at-once and partial state movements), and the impact of the choice of data stream processing system (i.e., Siddhi and Apache Flink). We demonstrate how the freeze time for the naïve all-at-once migration approach is almost 20 times longer than when applying an incremental checkpoint-based partial state migration approach that is based on Rhino [28].

IV.9 Acknowledgments

This work was supported by the Parrot Project (Research Council of Norway, IKTPluss, 311197). The authors thank Fabrice Starks and Stein Kristiansen for insightful discussions on mechanisms of operator migration, and Boris Koldehofe for reviewing and commenting on an earlier version of the manuscript.

References

[1] https://flink.apache.org/powered-by. [Online; accessed 27-February-2023].

[2] https://storm.apache.org. [Online; accessed 26-February-2023].

[3] https://beam.apache.org. [Online; accessed 26-February-2023].

[4] https://kafka.apache.org. [Online; accessed 5-July-2021].

[5] https://www.gurobi.com. [Online; accessed 23-January-2023].

[6] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer. [Online; accessed 23-January-2023].

[7] Abadi, D. et al. “The Beckman report on database research”. In: *Communications of the ACM* vol. 59, no. 2 (2016), pp. 92–99.

[8] Abbasi, M., Shahraki, A., and Taherkordi, A. “Deep learning for network traffic monitoring and analysis (ntma): A survey”. In: *Computer Communications* (2021).

[9] Ahmad, Y. et al. “Network Awareness in Internet-Scale Stream Processing.” In: *IEEE Data Eng. Bull.* vol. 28, no. 1 (2005), pp. 63–69.

[10] Alwasel, K. et al. “IoTSim-Osmosis: A framework for modeling and simulating IoT applications over an edge-cloud continuum”. In: *Journal of Systems Architecture* vol. 116 (2021), p. 101956.

[11] Amarasinghe, G. et al. “ECSNeT++: A simulator for distributed stream processing on edge and cloud environments”. In: *Future Generation Computer Systems* vol. 111 (2020), pp. 401–418.

[12] Assuncao, M. D. de, Silva Veith, A. da, and Buyya, R. “Distributed data stream processing and edge computing: A survey on resource elasticity and future directions”. In: *Journal of Network and Computer Applications* vol. 103 (2018), pp. 1–17.

[13] Bergui, M., Najah, S., and Nikolov, N. S. “A survey on bandwidth-aware geo-distributed frameworks for big-data analytics”. In: *Journal of Big Data* vol. 8, no. 1 (2021), pp. 1–26.

[14] Brettlecker, G. and Schuldt, H. “Reliable distributed data stream management in mobile environments”. In: *Information Systems* vol. 36, no. 3 (2011), pp. 618–643.

[15] Buddhika, T. et al. “Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams”. In: *IEEE Transactions on Parallel and Distributed Systems* vol. 28, no. 12 (2017), pp. 3553–3569.

[16] Carbone, P. et al. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* vol. 36, no. 4 (2015).

[17] Cardellini, V., Nardelli, M., and Luzi, D. “Elastic stateful stream processing in storm”. In: *2016 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2016, pp. 583–590.

[18] Cardellini, V. et al. “Decentralized self-adaptation for elastic data stream processing”. In: *Future Generation Computer Systems* vol. 87 (2018), pp. 171–185.

[19] Cardellini, V. et al. “Optimal operator deployment and replication for elastic distributed data stream processing”. In: *Concurrency and Computation: Practice and Experience* vol. 30, no. 9 (2018), e4334.

[20] Cardellini, V. et al. “Optimal operator replication and placement for distributed stream processing systems”. In: *ACM SIGMETRICS Performance Evaluation Review* vol. 44, no. 4 (2017), pp. 11–22.

[21] Cardellini, V. et al. “Run-time Adaptation of Data Stream Processing Systems: The State of the Art”. In: *ACM Computing Surveys (CSUR)* (2022).

[22] Castro Fernandez, R. et al. “Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 725–736.

[23] Chatzimilioudis, G. et al. “A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement”. In: *Journal of Computer and System Sciences* vol. 79, no. 3 (2013), pp. 349–368.

[24] Chen, M. et al. “A Dynamic Service Migration Mechanism in Edge Cognitive Computing”. In: *ACM Trans. Internet Technol.* vol. 19, no. 2 (Apr. 2019). GSCC: 0000161 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[25] De Matteis, T. and Mencagli, G. “Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing”. In: *ACM SIGPLAN Notices* vol. 51, no. 8 (2016), pp. 1–12.

[26] De Matteis, T. and Mencagli, G. “Proactive elasticity and energy awareness in data stream processing”. In: *Journal of Systems and Software* vol. 127 (2017), pp. 302–319.

[27] Dedousis, D., Zacheilas, N., and Kalogeraki, V. “On the fly load balancing to address hot topics in topic-based pub/sub systems”. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 76–86.

[28] Del Monte, B. et al. “Rhino: Efficient management of very large distributed state for stream processing engines”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2471–2486.

[29] Dias, G. M., Bellalta, B., and Oechsner, S. “A survey about prediction-based data reduction in wireless sensor networks”. In: *ACM Computing Surveys (CSUR)* vol. 49, no. 3 (2016), pp. 1–35.

[30] Dupont, C., Giaffreda, R., and Capra, L. “Edge computing in IoT context: Horizontal and vertical Linux container migration”. In: *2017 Global Internet of Things Summit (GIoTS)*. GSCC: 0000093. IEEE. IEEE, 2017, pp. 1–4.

[31] *Espertech. Esper – Complex Event Processing*. https://www.espertech.com/esper. [Online; accessed 26-February-2023]. Sept. 2022.

[32] Fang, J. et al. “Distributed stream rebalance for stateful operator under workload variance”. In: *IEEE Transactions on Parallel and Distributed Systems* vol. 29, no. 10 (2018), pp. 2223–2240.

[33] Fang, J. et al. “Parallel stream processing against workload skewness and variance”. In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 2017, pp. 15–26.

[34] Fragkoulis, M. et al. *A Survey on the Evolution of Stream Processing Systems*. 2020. arXiv: 2008.00842 [cs.DC].

[35] Gao, Z. et al. “Deep Reinforcement Learning Based Service Migration Strategy for Edge Computing”. In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. GSCC: 0000116. 2019, pp. 116–1165.

[36] Gedik, B. “Partitioning functions for stateful data parallelism in stream processing”. In: *The VLDB Journal* vol. 23, no. 4 (2014), pp. 517–539.

[37] Gedik, B. et al. “Elastic scaling for data stream processing”. In: *IEEE Transactions on Parallel and Distributed Systems* vol. 25, no. 6 (2013), pp. 1447–1463.

[38] Geldenhuys, M. K. et al. “Phoebe: Qos-aware distributed stream processing through anticipating dynamic workloads”. In: *2022 IEEE International Conference on Web Services (ICWS)*. IEEE. 2022, pp. 198–207.

[39] Gharaibeh, A. et al. “Smart cities: A survey on data management, security, and enabling technologies”. In: *IEEE Communications Surveys & Tutorials* vol. 19, no. 4 (2017), pp. 2456–2501.

[40] Goyal, T., Singh, A., and Agrawal, A. “Cloudsim: simulator for cloud computing infrastructure and modeling”. In: *Procedia Engineering* vol. 38 (2012), pp. 3566–3572.

[41] Gu, R. et al. “Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 539–556.

[42] Gulisano, V. et al. “Streamcloud: An elastic and scalable data streaming system”. In: *IEEE Transactions on Parallel and Distributed Systems* vol. 23, no. 12 (2012), pp. 2351–2365.

[43] Gulisano, V. et al. “STRETCH: Virtual shared-nothing parallelism for scalable and elastic stream processing”. In: *IEEE Transactions on Parallel and Distributed Systems* vol. 33, no. 12 (2022), pp. 4221–4238.

[44] Gupta, H. et al. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments”. In: *Software: Practice and Experience* vol. 47, no. 9 (2017), pp. 1275–1296.

[45] Heinze, T. et al. “Auto-scaling techniques for elastic data stream processing”. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE. 2014, pp. 296–302.

[46] Heinze, T. et al. “Cloud-based data stream processing”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 2014, pp. 238–245.

[47] Heinze, T. et al. “FUGU: Elastic Data Stream Processing with Latency Constraints.” In: *IEEE Data Eng. Bull.* vol. 38, no. 4 (2015), pp. 73–81.

[48] Heinze, T. et al. “Latency-aware elastic scaling for distributed data stream processing systems”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 2014, pp. 13–22.

[49] Hidalgo, N., Wladdimiro, D., and Rosas, E. “Self-adaptive processing graph with operator fission for elastic stream processing”. In: *Journal of Systems and Software* vol. 127 (2017), pp. 205–216.

[50] Hiessl, T. et al. “Optimal placement of stream processing operators in the fog”. In: *2019 IEEE 3rd International Conference on Fog and Edge Computing (ICFEC)*. IEEE. 2019, pp. 1–10.

[51] Hirzel, M. et al. “A catalog of stream processing optimizations”. In: *ACM Computing Surveys (CSUR)* vol. 46, no. 4 (2014), pp. 1–34.

[52] Hochreiner, C. et al. “Elastic stream processing for the internet of things”. In: *2016 IEEE 9th international conference on cloud computing (CLOUD)*. IEEE. 2016, pp. 100–107.

[53] Hoffmann, M. et al. “Megaphone: Latency-conscious state migration for distributed streaming dataflows”. In: *Proceedings of the VLDB Endowment* vol. 12, no. 9 (2019), pp. 1002–1015.

[54] Hu, J. et al. “Study on dynamic service migration strategy with energy optimization in mobile edge computing”. In: *Mobile Information Systems* vol. 2019 (2019). GSCC: 0000330.

[55] Hummer, W., Satzger, B., and Dustdar, S. “Elastic stream processing in the cloud”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* vol. 3, no. 5 (2013), pp. 333–345.

[56] Hummer, W. et al. “Dynamic migration of processing elements for optimized query execution in event-based systems”. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2011, pp. 451–468.

[57] Hwang, J.-H. et al. “A cooperative, self-configuring high-availability solution for stream processing”. In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 176–185.

[58] Isah, H. et al. “A survey of distributed data stream processing frameworks”. In: *IEEE Access* vol. 7 (2019), pp. 154300–154316.

[59] Jha, D. N. et al. “IoTSim-Edge: a simulation framework for modeling the behavior of Internet of Things and edge computing environments”. In: *Software: Practice and Experience* vol. 50, no. 6 (2020), pp. 844–867.

[60] Jonathan, A., Chandra, A., and Weissman, J. “WASP: wide-area adaptive stream processing”. In: *Proceedings of the 21st International Middleware Conference*. 2020, pp. 221–235.

[61] Kakkad, V., Santosa, A. E., and Scholz, B. “Migrating operator placement for compositional stream graphs”. In: *Proceedings of the 15th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*. GSCC: 0000359. 2012, pp. 125–134.

[62] Kalavri, V. et al. “Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 783–798.

[63] Kingman, J. “The single server queue in heavy traffic”. In: *Mathematical proceedings of the cambridge philosophical society*. Vol. 57. 4. Cambridge University Press. 1961, pp. 902–904.

[64] Koldehofe, B. et al. “Rollback-recovery without checkpoints in distributed event processing systems”. In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. 2013, pp. 27–38.

[65] Lakshmanan, G. T., Li, Y., and Strom, R. “Placement strategies for internet-scale data stream systems”. In: *IEEE Internet Computing* vol. 12, no. 6 (2008), pp. 50–60.

[66] Lei, C. and Rundensteiner, E. A. “Robust distributed query processing for streaming data”. In: *ACM Transactions on Database Systems (TODS)* vol. 39, no. 2 (2014), pp. 1–45.

[67] Lera, I., Guerrero, C., and Juiz, C. “YAFS: A simulator for IoT scenarios in fog computing”. In: *IEEE Access* vol. 7 (2019), pp. 91745–91758.

[68] Li, B. et al. “Marabunta: Continuous Distributed Processing of Skewed Streams”. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 252–261.

[69] Li, J. et al. "Enabling elastic stream processing in shared clusters". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE. 2016, pp. 108–115.

[70] Lindeberg, M. and Plagemann, T. "A Study on Migration Scheduling in Distributed Stream Processing Engines". In: *Proceedings of the 23rd International Conference on Distributed Computing and Networking*. ACM New York, NY, USA. 2022.

[71] Liu, F. et al. "DROAllocator: a dynamic resource-aware operator allocation framework in distributed streaming processing". In: *Network and Parallel Computing: 17th IFIP WG 10.3 International Conference, NPC 2020, Zhengzhou, China, September 28–30, 2020, Revised Selected Papers*. Springer. 2021, pp. 349–360.

[72] Liu, P., Da Silva, D., and Hu, L. "DART: A scalable and adaptive edge stream processing engine". In: *USENIX Annual Technical Conference*. 2021.

[73] Liu, S. et al. "An adaptive online scheme for scheduling and resource enforcement in Storm". In: *IEEE/ACM Transactions on Networking* vol. 27, no. 4 (2019), pp. 1373–1386.

[74] Liu, X. and Buyya, R. "Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions". In: *ACM Computing Surveys (CSUR)* vol. 53, no. 3 (2020), pp. 1–41.

[75] Liu, Y., Xu, H., and Lau, W. C. "Online Resource Optimization for Elastic Stream Processing with Regret Guarantee". In: *Proceedings of the 51st International Conference on Parallel Processing*. 2022, pp. 1–11.

[76] Liu, Y., Shi, X., and Jin, H. "Runtime-aware adaptive scheduling in stream processing". In: *Concurrency and Computation: Practice and Experience* vol. 28, no. 14 (2016), pp. 3830–3843.

[77] Lohrmann, B., Janacik, P., and Kao, O. "Elastic stream processing with latency guarantees". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE. 2015, pp. 399–410.

[78] Lombardi, F. et al. "Elastic symbiotic scaling of operators and resources in stream processing systems". In: *IEEE Transactions on Parallel and Distributed Systems* vol. 29, no. 3 (2017), pp. 572–585.

[79] Luo, Q. et al. "Resource scheduling in edge computing: A survey". In: *IEEE Communications Surveys & Tutorials* vol. 23, no. 4 (2021), pp. 2131–2165.

[80] Luthra, M. et al. “TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. 2018, pp. 136–147.

[81] Ma, K., Yang, B., and Yu, Z. “Optimization of stream-based live data migration strategy in the cloud”. In: *Concurrency and Computation: Practice and Experience* vol. 30, no. 12 (2018), e4293.

[82] Ma, L. et al. “Efficient Live Migration of Edge Services Leveraging Container Layered Storage”. In: *IEEE Transactions on Mobile Computing* vol. 18, no. 9 (2019). GSCC: 0000095, pp. 2020–2033.

[83] Ma, L., Yi, S., and Li, Q. “Efficient Service Handoff across Edge Servers via Docker Container Migration”. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. SEC ’17. GSCC: 0000181. San Jose, California: Association for Computing Machinery, 2017.

[84] Mach, P. and Becvar, Z. “Mobile Edge Computing: A Survey on Architecture and Computation Offloading”. In: *IEEE Communications Surveys Tutorials* vol. 19, no. 3 (2017), pp. 1628–1656.

[85] Machen, A. et al. “Live Service Migration in Mobile Edge Clouds”. In: *IEEE Wireless Communications* vol. 25, no. 1 (2018). GSCC: 0000256, pp. 140–147.

[86] Madsen, K. G. S. and Zhou, Y. “Dynamic resource management in a massively parallel stream processing engine”. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 2015, pp. 13–22.

[87] Madsen, K. G. S., Zhou, Y., and Cao, J. “Integrative dynamic reconfiguration in a parallel stream processing engine”. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE. 2017, pp. 227–230.

[88] Madsen, K. G. S., Zhou, Y., and Su, L. “Enorm: Efficient window-based computation in large-scale distributed stream processing systems”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 2016, pp. 37–48.

[89] Mahmud, R. et al. “iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments”. In: *Journal of Systems and Software* vol. 190 (2022), p. 111351.

[90] Mai, L. et al. “Chi: A scalable and programmable control plane for distributed stream processing systems”. In: *Proceedings of the VLDB Endowment* vol. 11, no. 10 (2018), pp. 1303–1316.

[91] Mandal, U. et al. “Greening the cloud using renewable-energy-aware service migration”. In: *IEEE Network* vol. 27, no. 6 (2013). GSCC: 0000089, pp. 36–43.

[92] Manyika, J. et al. “Unlocking the Potential of the Internet of Things”. In: *McKinsey Global Institute* vol. 1 (2015).

[93] Martin, A., Brito, A., and Fetzer, C. “Scalable and elastic realtime click stream analysis using streammine3g”. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. 2014, pp. 198–205.

[94] Martin, A. et al. “User-constraint and self-adaptive fault tolerance for event stream processing systems”. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2015, pp. 462–473.

[95] Mechalikh, C., Taktak, H., and Moussa, F. “PureEdgeSim: A simulation toolkit for performance evaluation of cloud, fog, and pure edge computing environments”. In: *2019 international conference on high performance computing & simulation (HPCS)*. IEEE. 2019, pp. 700–707.

[96] Mehmood, E. and Anees, T. “Challenges and Solutions for Processing Real-Time Big Data Stream: A Systematic Literature Review”. In: *IEEE Access* vol. 8 (2020), pp. 119123–119143.

[97] *MOA*. https://www.cs.waikato.ac.nz/ml/weka/. [Online; accessed 28-August-2021].

[98] Mohammadi, M. et al. “Deep learning for IoT big data and streaming analytics: A survey”. In: *IEEE Communications Surveys & Tutorials* vol. 20, no. 4 (2018), pp. 2923–2960.

[99] Mukherjee, M., Shu, L., and Wang, D. “Survey of fog computing: Fundamental, network applications, and research challenges”. In: *IEEE Communications Surveys & Tutorials* vol. 20, no. 3 (2018), pp. 1826–1857.

[100] Ni, X. et al. “Automating multi-level performance elastic components for IBM streams”. In: *Proceedings of the 20th International Middleware Conference*. 2019, pp. 163–175.

[101] Oliveira, E. et al. “Latency and energy-awareness in data stream processing for edge based IoT systems”. In: *Journal of Grid Computing* vol. 20, no. 3 (2022), p. 27.

[102] Osanaiye, O. et al. “From Cloud to Fog Computing: A Review and a Conceptual Live VM Migration Framework”. In: *IEEE Access* vol. 5 (2017). GSCC: 0000316, pp. 8284–8300.

[103] Ottenwälder, B. et al. “MCEP: A mobility-aware complex event processing system”. In: *ACM Transactions on internet technology (TOIT)* vol. 14, no. 1 (2014), pp. 1–24.

[104] Ottenwälder, B. et al. “Migcep: Operator migration for mobility driven distributed complex event processing”. In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. 2013, pp. 183–194.

[105] Pande, S. K., Panda, S. K., and Das, S. “Dynamic service migration and resource management for vehicular clouds”. In: *Journal of Ambient Intelligence and Humanized Computing* (2020), pp. 1–21.

[106] Papaemmanouil, O., Cetintemel, U., and Jannotti, J. “Supporting generic cost models for wide-area stream processing”. In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 1084–1095.

[107] Pham, T. N. et al. “Uninterruptible migration of continuous queries without operator state migration”. In: *ACM SIGMOD Record* vol. 46, no. 3 (2017), pp. 17–22.

[108] Pietzuch, P. et al. “Network-aware operator placement for stream-processing systems”. In: *22nd International Conference on Data Engineering (ICDE’06)*. IEEE. 2006, pp. 49–49.

[109] Puliafito, C. et al. “MobFogSim: Simulation of mobility and migration for fog computing”. In: *Simulation Modelling Practice and Theory* vol. 101 (2020), p. 102062.

[110] Qayyum, T. et al. “FogNetSim++: A toolkit for modeling and simulation of distributed fog environment”. In: *IEEE Access* vol. 6 (2018), pp. 63570–63583.

[111] Qin, C., Eichelberger, H., and Schmid, K. “Enactment of adaptation in data stream processing with latency implications—A systematic literature review”. In: *Information and Software Technology* vol. 111 (2019), pp. 1–21.

[112] Rasmussen, C. E. “Gaussian processes in machine learning”. In: *Summer school on machine learning*. Springer. 2003, pp. 63–71.

[113] Repantis, T. and Kalogeraki, V. “Alleviating hot-spots in peer-to-peer stream processing environments”. In: *Proceedings of the 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing, DBISP2P, Vienna, Austria*. Citeseer. 2007.

[114] Repantis, T. and Kalogeraki, V. “Hot-spot prediction and alleviation in distributed stream processing applications”. In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE. 2008, pp. 346–355.

[115] Rizou, S., Dürr, F., and Rothermel, K. “Solving the multi-operator placement problem in large-scale operator networks”. In: *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE. 2010, pp. 1–6.

[116] *RocksDB*. https://rocksdb.org/. [Online; accessed 23-January-2023]. 2021.

[117] Rodrigues, T. G. et al. “Hybrid Method for Minimizing Service Delay in Edge Cloud Computing Through VM Migration and Transmission Power Control”. In: *IEEE Transactions on Computers* vol. 66, no. 5 (2017). GSCC: 0000353, pp. 810–819.

[118] Rundensteiner, E. A. et al. “Cape: Continuous query engine with heterogeneous-grained adaptivity”. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 1353–1356.

[119] Röger, H., Bhowmik, S., and Rothermel, K. “Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures”. In: *Proceedings of the 20th International Middleware Conference*. 2019, pp. 255–267.

[120] Röger, H. and Mayer, R. “A comprehensive survey on parallelization and elasticity in stream processing”. In: *ACM Computing Surveys (CSUR)* vol. 52, no. 2 (2019), pp. 1–37.

[121] Sahal, R., Breslin, J. G., and Ali, M. I. “Big data and stream processing platforms for Industry 4.0 requirements mapping for a predictive maintenance use case”. In: *Journal of manufacturing systems* vol. 54 (2020), pp. 138–151.

[122] Sakr, S. et al. “A survey of large scale data management approaches in cloud environments”. In: *IEEE communications surveys & tutorials* vol. 13, no. 3 (2011), pp. 311–336.

[123] Salama, M., Elkhatib, Y., and Blair, G. “IoTNetSim: A modelling and simulation platform for end-to-end IoT services and networking”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 2019, pp. 251–261.

[124] Shah, M. A. et al. “Flux: An adaptive partitioning operator for continuous query systems”. In: *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE. 2003, pp. 25–36.

[125] Sonmez, C., Ozgovde, A., and Ersoy, C. “Edgecloudsim: An environment for performance evaluation of edge computing systems”. In: *Transactions on Emerging Telecommunications Technologies* vol. 29, no. 11 (2018), e3493.

[126] Srivastava, U., Munagala, K., and Widom, J. “Operator placement for in-network stream query processing”. In: *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2005, pp. 250–258.

[127] Starks, F. and Plagemann, T. P. “Operator placement for efficient distributed complex event processing in manets”. In: *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE. 2015, pp. 83–90.

[128] Starks, F., Plagemann, T. P., and Kristiansen, S. “DCEP-Sim: An Open Simulation Framework for Distributed CEP”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS ’17. Barcelona, Spain: ACM, 2017, pp. 180–190.

[129] Suhothayan, S. et al. “Siddhi: A second look at complex event processing architectures”. In: *Proceedings of the 2011 ACM workshop on Gateway computing environments*. 2011, pp. 43–50.

[130] Sun, D. et al. “A multi-level collaborative framework for elastic stream computing systems”. In: *Future Generation Computer Systems* vol. 128 (2022), pp. 117–131.

[131] Sun, D. et al. “Dynamic redirection of real-time data streams for elastic stream computing”. In: *Future Generation Computer Systems* vol. 112 (2020), pp. 193–208.

[132] Taieb, S. B. et al. “A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition”. In: *Expert systems with applications* vol. 39, no. 8 (2012), pp. 7067–7083.

[133] To, Q.-C., Soto, J., and Markl, V. “A survey of state management in big data processing systems”. In: *The VLDB Journal* vol. 27, no. 6 (2018), pp. 847–872.

[134] Tucker, P. et al. *NEXMark—A Benchmark for Queries over Data Streams DRAFT*. Tech. rep. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[135] Tziritas, N. et al. “On improving constrained single and group operator placement using evictions in big data environments”. In: *IEEE Transactions on Services Computing* vol. 9, no. 5 (2016), pp. 818–831.

[136] Urgaonkar, R. et al. “Dynamic service migration and workload scheduling in edge-clouds”. In: *Performance Evaluation* vol. 91 (2015). Special Issue: Performance 2015, pp. 205–228.

[137] Vogel, A. et al. “Self-adaptation on parallel stream processing: A systematic review”. In: *Concurrency and Computation: Practice and Experience* vol. 34, no. 6 (2022), e6759.

[138] Volnes, E. et al. “EXPOSE: Experimental Performance Evaluation of Stream Processing Engines Made Easy”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2020, pp. 18–34.

[139] Volnes, E. et al. “Travel light: state shedding for efficient operator migration”. In: *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. 2022, pp. 79–84.

[140] Wang, C. et al. “Automating characterization deployment in distributed data stream management systems”. In: *IEEE Transactions on Knowledge and Data Engineering* vol. 29, no. 12 (2017), pp. 2669–2681.

[141] Wang, H. et al. “Service migration in mobile edge computing: A deep reinforcement learning approach”. In: *International Journal of Communication Systems* (2020). GSCC: 0000116 Publisher: Wiley Online Library, e4413.

[142] Wang, L. et al. “Elasticutor: Rapid elasticity for realtime stateful stream processing”. In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 573–588.

[143] Wang, S. et al. “Mobility-Induced Service Migration in Mobile Microclouds”. In: *2014 IEEE Military Communications Conference*. GSCC: 0000132. 2014, pp. 835–840.

[144] Wang, S. et al. “Dynamic service migration in mobile edge-clouds”. In: *2015 IFIP Networking Conference (IFIP Networking)*. GSCC: 0000294. IEEE. IEEE, 2015, pp. 1–9.

[145] Wang, W. et al. “Potential-driven load distribution for distributed data stream processing”. In: *Proceedings of the 2nd international workshop on Scalable stream processing system*. 2008, pp. 13–22.

[146] Wang, X. et al. “Convergence of edge computing and deep learning: A comprehensive survey”. In: *IEEE Communications Surveys & Tutorials* vol. 22, no. 2 (2020), pp. 869–904.

[147] Wei, J. et al. “SatEdgeSim: A toolkit for modeling and simulation of performance evaluation in satellite edge computing environments”. In: *2020 12th International Conference on Communication Software and Networks (ICCSN)*. IEEE. 2020, pp. 307–313.

[148] *Weka*. https://weka.cms.waikato.ac.nz/. [Online; accessed 28-August-2021].

[149] Wu, Y. and Tan, K.-L. “ChronoStream: Elastic stateful stream computation in the cloud”. In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 723–734.

[150] Xing, Y., Zdonik, S., and Hwang, J.-H. “Dynamic load distribution in the borealis stream processor”. In: *21st International Conference on Data Engineering (ICDE’05)*. IEEE. 2005, pp. 791–802.

[151] Xu, J. and Palanisamy, B. “Model-based reinforcement learning for elastic stream processing in edge computing”. In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE. 2021, pp. 292–301.

[152] Xu, L., Peng, B., and Gupta, I. “Stela: Enabling stream processing systems to scale-in and scale-out on-demand”. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2016, pp. 22–31.

[153] Yi, S., Li, C., and Li, Q. “A Survey of Fog Computing: Concepts, Applications and Issues”. In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata ’15. Hangzhou, China: ACM, 2015, pp. 37–42.

[154] Zacheilas, N. et al. “Elastic complex event processing exploiting prediction”. In: *2015 IEEE International Conference on Big Data (Big Data)*. IEEE. 2015, pp. 213–222.

[155] Zeng, Z. et al. “Efficient Edge Service Migration in Mobile Edge Computing”. In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. GSCC: 0000273. 2020, pp. 691–696.

[156] Zhang, C. and Zheng, Z. “Task migration for mobile edge computing using deep reinforcement learning”. In: *Future Generation Computer Systems* vol. 96 (2019). GSCC: 0000116, pp. 111–118.

[157] Zhang, L. et al. “Autrascale: an automated and transfer learning solution for streaming system auto-scaling”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2021, pp. 912–921.

[158] Zhou, Y., Aberer, K., and Tan, K.-L. “Toward massive query optimization in large-scale distributed stream systems”. In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2008, pp. 326–345.

[159] Zhou, Y. et al. “Efficient dynamic operator placement in a locally distributed continuous query system”. In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2006, pp. 54–71.

[160] Zhu, Y., Rundensteiner, E. A., and Heineman, G. T. “Dynamic plan migration for continuous queries over data streams”. In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. GSCC: 0000194. 2004, pp. 431–442.

Paper V

Travel light: state shedding for efficient operator migration

Espen Volnes, Thomas Plagemann, Boris Koldehofe, Vera Goebel

Published in *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, August 2022, pp 79-84, DOI: 10.1145/3524860.3539638.

Abstract

Operator migration is a crucial concept to adapt event processing systems to dynamic changes. When the placement of a stateful operator changes, the operator state must be migrated to the new host. However, operator state size and time constraints can make it impossible to migrate the operator without severe Quality of Service (QoS) degradation. As a relief, we propose to perform state shedding in such a situation. The core idea of state shedding is to partition the operator state, assign a utility to each partial state, and use the utility and size of each partial state to identify the most useful partial states that can be migrated in a given time frame. Thus, state shedding can maintain a substantially higher QoS with a lower impact on query results than state-of-the-art solutions targeting consistent state at the old and new host. In this paper, we define this novel approach and in a simulation environment evaluate state shedding in migration scenarios with pattern-matching queries.

V.1 Introduction

Stream and event processing systems are of fundamental importance and an integral part of Big Data systems. They support important requirements of Big Data applications to integrate and analyze in real-time high volume data streams which can stem from many distinct and highly distributed data sources. Current stream and event processing systems operate in a highly distributed manner, i.e., the operators in charge of analyzing data streams can be flexibly executed on systems resources in the cloud, at the edge, or even on connected (mobile) devices. This way they can support application requirements regarding Quality of Service (QoS).

System and application dynamics, like bursty input rates, resource contention, and mobility can lead to reduced QoS and require adapting the way operators are executed. Two established methods to react to such changes are *operator migration* [18] and *load shedding* [1, 16, 20]. In operator migration, the placement of operators on resources is changed by migrating one or several operators from their current host (further on referred to as old host) to a new host, which is better suited to meet the required QoS. Load shedding allows reacting to temporary overload situations, by dropping tuples in the input stream or dropping some state of the operator to ensure the operator can process fresh data tuples timely.

Both approaches are effective to deal with overload situations, but they also impose a cost for the distributed operator execution and therefore need to be carefully designed and applied. Operator migration allows changing the resources and this way also the performance, e.g., the processing rate for executing and operator or communication delays for input tuples. Operator migration requires (1) to set up a new resource, the new host of the operator, (2) transmit state from the old host to the new host, and (3) coordinate the handover between the old and new host. As such operator migration can consume temporally redundant resources and increase delays until the new host becomes operational. Load shedding reduces the time to react to overload situations, but dropping tuples and state reduce the accuracy of the results produced by the operators. For longer periods of overload situations, load shedding may therefore be costly in terms of ensured accuracy.

Adapting distributed operator execution approaches mostly treat these two mechanisms as alternatives performed in isolation. We propose and study the combined use of migration and load shedding mechanisms. In particular, we propose to apply state shedding in the course of operator migration to counteract unexpected long delays during operator migrations. State-of-the-art methods aim to atomically transfer the entire operator state based on good estimates of the transfer cost. Contrary, in this paper, we observe that the combination of state shedding and migration is promising for operator migration to better adapt to unexpected situations. We propose to counteract abrupt changes, such as reduced bandwidth and increased transmission latencies, by transferring only the most necessary state. This requires appropriate online migration procedures to prioritize the partial state to ensure a high utility in terms of accuracy and imposed migration and execution delays.

In this paper, we contribute to (1) a novel concept of combining state shedding and operator migration by maximizing the utility of partially migrated state, and (2) an analysis including a first empirical evaluation that illustrates possible advantages of utility-based load shedding in the context of two real-world data sets: the Citi Bike data set [5] and a bus GPS data set from Dublin [6].

V.2 Background

In this section, we introduce background on distributed operator execution, operator migration and load shedding.

V.2.1 Distributed Operator Execution

In stream and event processing systems, the logic and the computational functions to analyze and transform data streams are given in form of operators, e.g., filter, join, grouping, and pattern detection operators. The operators are commonly organized in a data flow graph, called the *operator graph*. The operator graph models dependencies between operators and data sources in receiving and producing *tuples* from/to specific *streams*. The operators are executed on hosts of the distributed infrastructure. They can also be dynamically migrated between hosts to meet the performance requirements of the application or react to other changes, such as failures. It is important to note that during the execution of an operator on a host, state is built up while performing processing steps on the received input tuples. Such state can be modeled in the form of (1) tuples in input and output queues and (2) so-called partial states [16, 20], which correspond to intermediate results needed to produce output tuples. When adapting the operator execution, e.g., performing migrations or load shedding, managing the operator state is highly important for the resulting accuracy and consistency.

V.2.2 Operator migration

Operator migration is a mechanism for exchanging the hosts engaged in the distributed operator execution. It requires organizing the state transfer between the old and new host and reorganizing the flow of data streams, also named *data stream management*. A major objective of current operator migration procedures is to ensure consistency, i.e., to ensure the migration of the entire state completes and the resulting migration has no impact on the operator results.

Approaches for performing operator migration can be classified according to their stream management during the state transfer, i.e., in a *single track* or *parallel track* [18]. In single-track migration, the tuples of upstream operators are buffered (at the upstream node, new host, or old host). Therefore, the migration procedure results in a temporary downtime during the handover between the new and old host until all upstream tuples and operator state are transferred consistently.

Parallel-track migration algorithms are able to migrate state without operator downtime by upstream nodes sending tuples to the old and new host [18]. Either the old host continues its executions until the state transfer has been completed or the old host gradually moves state to the new host.

These algorithms require temporary duplication of input streams and good connectivity. Under high system dynamics, e.g., slow communication links and drastically reduced bandwidth, these mechanisms can significantly reduce the performance of the distributed operator execution.

V.2.3 Load shedding

Load shedding is an established mechanism for operator execution to react to overload situations, e.g., as originally proposed for the data stream management system Aurora [1, 17]. In overload scenarios, part of the workload for an operator is dropped to stabilize the system. Most of the literature describes solutions where input tuples are dropped [2, 4, 7–10, 12, 13, 17]. For aggregation operators, the goal is to minimize the relative error of the calculated aggregate. For join operators, the goal is to drop the tuples that eventually join with the fewest tuples. Another method is to drop windows [16] internally, which reduces the number of produced aggregates instead of reducing the aggregates' accuracy. In pattern-matching operators, dropping input tuples is likely to distort the results completely, because individual tuples can determine whether a sequence fulfills a pattern or not. In such cases, a different state-based load shedding mechanism that drops partial states from the operator is a better option. A partial match might or might not result in an output complex event. If the likelihood of the partial match in producing output is low, the entire sequence of tuples might be dropped. This is done for the pattern-matching operator in a few recent works [3, 14, 19, 20]. As a result of load shedding, the consistency may be invalidated, but the accuracy and utility of the query may remain high.

V.3 Problem Statement

All state-of-the-art operator migration approaches aim to establish a consistent state at the new host. Unforeseen network conditions can prevent a timely transmission of the entire state between the old and new host. Consequently, operators can experience unexpected freeze times before the operator execution can be resumed. This is an inherent limitation of single-track operator migration algorithms.

Figure V.1 shows a VANET scenario executing with three roadside base stations running applications and collecting data from passing vehicles. The red base station has a critically high load and needs to reduce it by moving some operators to the green node that has sufficient capacity. In this scenario, the operators deployed on the colored nodes execute operators for detecting collisions, bottlenecks and other traffic situations which need to be timely reported to traffic participants to properly act. Clearly, freezing the operator execution can lead to the situation where traffic participants cannot

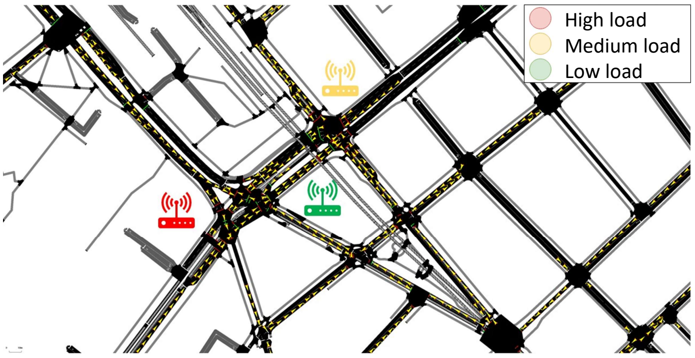


Figure V.1: Example VANET scenario

react while the operator execution is suspended during an unexpected long migration.

Therefore, a better strategy, which we study in this paper, is to limit the effect of delayed migrations by transmitting the most relevant state until the time the operator needs to be resumed. With the help of state shedding, the old host can decide on the most relevant partial state to be transmitted yielding the highest utility for the application, e.g., to react to a possible dangerous traffic event. In this paper, we address the following research questions (RQ):

- RQ1: How to partition operator state in such a way that each partial state is useful for further processing?
- RQ2: How to determine the utility of partial states?
- RQ3: How to select the partial states that can be sent in a given time frame and provide the highest accumulated utility?
- RQ4: How do different approaches for operator migration with state shedding perform?

In the next section, we present the overall approach of operator migration with state shedding. RQ1 and RQ2 are addressed in Section V.5. RQ3 is addressed in Section V.6 and RQ4 is answered in Section V.7.

V.4 Approach

In this section, we present the overall approach that combines operator migration with state shedding. It comprises six steps illustrated in Figure V.2 and Algorithm 7. (1) a monitor detects an overload situation or a network problem, which triggers (2) the placement module to determine the new placement and the maximum migration time, and triggers (3) the migration

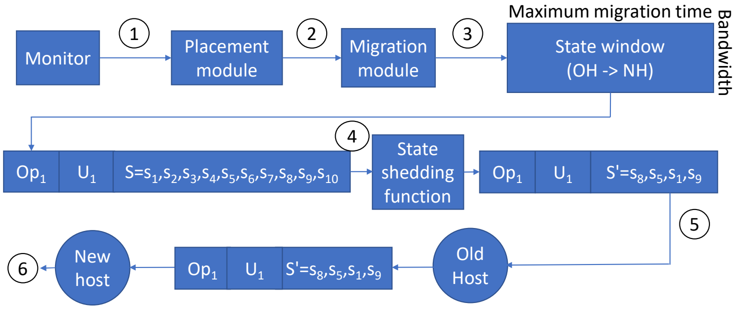


Figure V.2: Migration with state shedding in six steps

module to extract the current operator state S and to partition S into i partial states, i.e, $S1$ to $S10$. Each partial state is the smallest useful unit for resuming the operator at the new host. (4) The state shedding function determines the utility of each partial state u_i , and (5) selects the most useful partial states, i.e., $S8$, $S5$, $S1$, and $S9$, migrates them to the new host, and drops the remaining partial states. The final Step (6) is to resume the operator at the new host.

Algorithm 7 Operator migration from old host (oh) to new host (nh) with state shedding. Abbreviations: operator (op), partial states (p_states), available bandwidth (bw), latency (lat), shedded state (s_state), state shedding function (ls)

```
 1: $trigger\_migration \leftarrow monitor(load, resources)$
 2: $migration\_time \leftarrow calculate\_migration\_time(oh, nh, op)$
 3: $p\_states \leftarrow partition(state)$
 4: $p\_states\_util[S_i, U_i] \leftarrow calculate\_utils(p\_states[S_i])$
5a: $c \leftarrow calculate\_c(oh, nh, op, bw(oh, nh), lat(oh, nh))$
5b: $s\_state \leftarrow ls(start\_time, max\_time, p\_states, bw(oh, nh)lat(oh, nh))$
5c: $migrate(oh, nh, shed\_state, start\_time)$
 6: $resume\_operator(nh)$
```

The optimization problem of selecting the most useful partial states to migrate during the maximum migration time can be reduced to solving the knapsack problem. The objective function is to maximize the utility of the operator's partial states, each with utility u_i and size s_i, subject to a limited capacity c that represents the maximum amount of data that can be sent during the migration.

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} u_i \\ \text{s.t.} \quad & \sum_{j} s_j < c \end{aligned} \tag{V.1}$$

The success of the solution depends highly on the specific operator semantics which determine how to partition the state and assign utility.

V.5 State partitioning

This section explores how to partition the operator state into partial states of limited size and determine their utility (RQ1 and RQ2). We identify three common stateful operators that differ significantly in how their state manifests: aggregation, join and pattern-matching operators (see example queries in Figure V.3). Based on the established design procedures of these operators, we want to analyze how state needs to be represented in order to be be ready for partitioning, and its impact on utility.

An aggregation operator such as in Q1 can record the state as partial aggregates (Figure V.3a) that are updated for each tuple that is processed. If it uses a sliding window, it will update multiple aggregates for each tuple. Alternatively, all received tuples can be stored until the end of the window, and the tuples are aggregated (Figure V.3b). However, the latter method requires substantially more storage space and can increase the delay of the aggregation.

A pattern-matching operator looks for particular sequences of tuples that indicate a higher-level event. The stored tuples in the sequences might vary in size and the length of the partial matches may vary. A pattern-matching operator such as in Q2 looks for patterns in a single stream and groups the patterns by a key, leading to an internal state of a tuple sequence for each group (Figure V.3c). If the pattern-matching operator looks in multiple streams, such as in Q3, it is only able to keep one sequence in the internal state at the time, because a group is defined for one stream only (Figure V.3d). If a query joins and does pattern-matching with groups in the same query, the query must first join the streams as in Q4 before matching patterns.

A join operator such as in Q4 might store the internal state as tuples in a window and evict tuples when the window jumps (Figure V.3e). It may keep cached matches on filter predicates to match new tuples to stored tuples faster, using some lookup mechanism. Tuples often vary in size, especially tuples from two different streams that are being joined. Even within the same stream, some attributes, e.g., text attributes, can vary in size.

A state shedding function can drop random state, but there is a strong incentive to keep the most important states. What this means depends on the type of operator that is being assessed. The utility of a partial state is not trivial to define or calculate. It is an operator-specific function that depends also on the type of application that is executed.

In traditional aggregation queries, the shedding of input tuples reduces the accuracy of the results produced output, but keeps the number of produced tuples the same. As such, the goal has traditionally been to minimize the relative error in results [2]. On the other hand, the

```
Q1: Sliding window 10 seconds        Q2: Pattern A+B
    Jump every 1 second                  Define A as speed > max(A.speed)
    Group by id                          Group by S1.id

Q3: Pattern A+B                      Q4: Join S2
    Define A as T.temp > 45              on S1.id = S2.s1
    B as H.humidity < 25
```

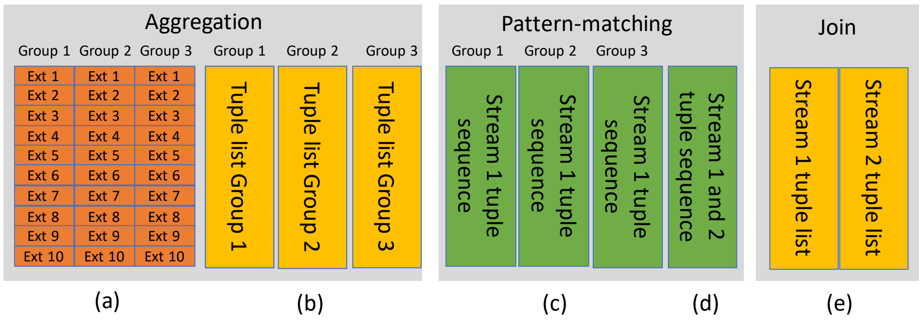


Figure V.3: Internal state of operators

number of produced tuples may be reduced when shedding tuples in a join operator, sequences in a pattern-matching operator or window extents in an aggregation operator. The accuracy of the produced tuples is retained, but the accuracy of the query is reduced.

A tuple for a join operator has utility if it joins with other tuples. Therefore, the overall goal is to maximize the number of tuples that are produced by the join operator. For the pattern-matching operator, the goal is the same. Either a match completes and produces a complex event, or it expires and never completes. A partial match has no utility until tuples are produced. For an aggregation operator, this is different. A window extent can be considered as just a few integers that indicate the start and stop of the window extent, the count of tuples in the window, and the current aggregate. An incoming tuple triggers an increment of the count in every window extent, and the aggregate is updated.

V.6 Partial state selection

This section discusses how to select the partial states to migrate (RQ3). The knapsack problem can be solved in a few ways, where the greedy approach of sending partial states in a descending order of utility density is the easiest way, but it can not guarantee to achieve the optimum. If all partial states have an identical size, such as for the aggregation operator in Figure V.3a, the greedy solution will perform exactly as the optimum. However, with variable partial state sizes in Figure V.3b, c, d and e, the state shedding solution might result in a significantly higher utility than the greedy solution. In some cases, the maximum migration time is uncertain, and therefore, the best effort provided with the greedy method might perform reasonably well.

However, if the maximum migration time is short or the state size varies significantly, finding the optimum or near-optimum might yield a significantly higher utility than the greedy solution. Since the knapsack problem is NP-hard, brute-forcing is unfeasible for even small input sizes. However, the knapsack problem has a polynomial-time approximation scheme that uses dynamic programming to find the optimum. This solution has a run-time and space complexity of $O(n^2 \cdot m)$, where n is the total number of partial states, m is the highest utility of the partial states, and $n \cdot m$ is the highest possible sum of utilities. A simplification can be done with the fully polynomial-time approximation scheme algorithm that scales down the utility with factor $\theta > 0$ to reduce the number of iterations. This reduces the run-time and space complexity significantly to $O(n^2 \lfloor \frac{m}{\theta} \rfloor)$.

These algorithms might still have a significant run-time and memory usage, which might be unfeasible if the migration must occur quickly. If we design the utility functions such that the utility values depend on each other, we can create a heuristic that binds the complexity of the optimal search without compromising the accuracy: (1) each partial state i gets assigned a utility through $U(i)$, (2) the utilities are updated as $U'(i) = \frac{U(i)}{m} \cdot 100$. Each partial state gets a utility between 0 and 100, depending on the most important partial state. Since the maximum utility is capped at 100 for each partial state, and the inner loop iterates through maximum $\sum_{i=1}^{n} u_i$, the worst case number iterations is $n \cdot 100$, which leads to a run-time and space complexity of $O(n^2)$.

V.7 Analysis

This section studies a practical application of the state shedding technique to compare the different approaches for partial state selection (RQ4). We apply a scenario similar to the VANET scenario from Section V.3 with two real-world VANET data sets: (1) a data set from Citi Bike [5] that describes bike trips of users, and (b) a data set containing GPS readings from buses in Dublin [6]. We describe three queries in Listing V.1, Q_{pm1} uses the Citi Bike data set and Q_{pm2} and Q_{pm3} use the bus data set. The configurations result in different state characteristics: Q_{pm1} produces many partial states; Q_{pm2} only produces eight partial states; and Q_{pm3} produces many partial states.

Listing V.1: Queries used in the simulations

Q_{pm1}
```
SEQ(A+B)
 GROUP BY BikeTrip.bikeid
 DEFINE B AS BikeTrip.end_station_id == 3116
 WITHIN 1 day
```

Q_{pm2}
```
SEQ(A+B)
 GROUP BY BusRecord.operator
 DEFINE B AS BusRecord.block_id == 67002
 WITHIN 1 hour
```

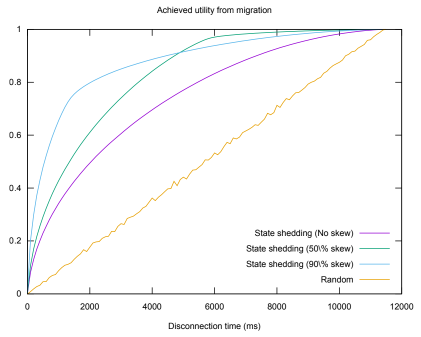
Q_{pm3}
```
SEQ(A+B)
 GROUP BY BusRecord.vehicle_id
 DEFINE B AS BusRecord.block_id == 67002
 WITHIN 1 hour
```

We simulate a connectivity issue scenario where the old host is about to lose connection or experience heavily degraded link connection with the upstream and downstream nodes, and has to complete the migration by a certain deadline. The connectivity failure can be due to imminent node failure or network disconnection. The deadline is estimated by the migration decision model Figure V.2, and the data it uses to predict this time is assumed to be collected throughout normal execution. The old host attempts to migrate the operator state using the state shedding function, and after the scheduled state is sent, it continues sending the remaining states as long as possible.

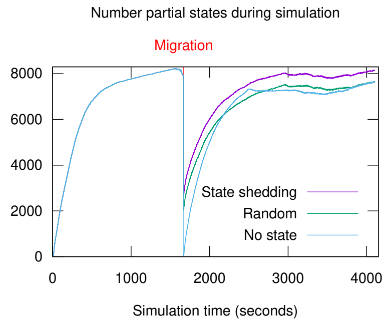
V.7.1 Simulations

Simulations are performed using the distributed stream processing simulator DCEP-Sim [15]. The simulator implementation incorporates a small-scale stream processing engine and builds upon the well-established discrete-event network simulator ns-3 [11]. Four nodes are deployed in DCEP-Sim: one upstream node, one downstream node, the old and new host. The upstream node produces tuples and sends them to the query that produces tuples for the downstream node. During the simulation, the old host migrates state to the new host, and the utility achieved from the migration is measured and analyzed.

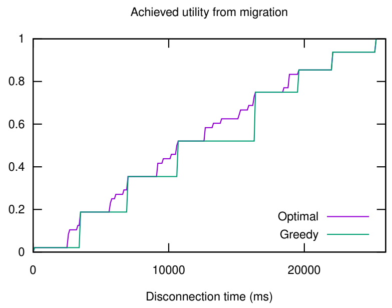
We do three runs, one for each query in Listing V.1. The first run compares optimal partial state selection with the random ordering, i.e., without prioritization, where partial states are sent until the old host disconnects. Three utility distributions are used: one balanced and two skewed distributions. In the first distribution, each partial state gets a random utility between 0 and 100. In the second distribution, the random utility is assigned and for 10% of the partial states, i.e., those with utility above 90, we introduce skew by multiplying the utility by ten. In the third distribution, utility values above 50 are multiplied by ten. State shedding is expected to have a more significant effect on skewed utility distributions. The random scenario is only executed with the balanced distribution, but with a high number of partial states, the random case without skew has a very similar utility result as the random case with skew. The second and third runs compare the optimal to the greedy solution, without any utility



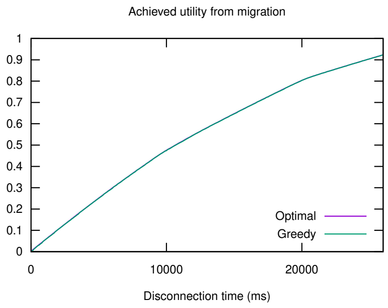
(a) Q_{pm1} with multiple utility distributions



(b) Size of internal state during simulation using Q_{pm1}



(c) Q_{pm2} with few big partial states



(d) Q_{pm3} with many small partial states

Figure V.4: Simulation results where (a) and (b) use Q_{pm1}, (c) uses Q_{pm2}, and (d) uses Q_{pm3}

skew. These runs aim to show how the size and number of partial states affect the achieved utility.

V.7.2 Results

The utility obtained with the Citi Bike data set are shown in Figure V.4a. For the state shedding technique, the utility achieved depends on the utility distribution of the partial states. In the case with skew for 10% of the partial states, 80% of the total utility is reached when the disconnection time is 2 s, less than 20% of the time it takes to send all partial states. When the skew is 50% of the partial states, it takes 3.6 s to send partial states with 80% of the total utility. At 5.5 s disconnection time, the 50% skew reaches >95% of total utility. Even without skew, there is a clear advantage to using state shedding.

Figure V.4b illustrates the internal state of the query throughout the simulation, including during and after the migration. The query reaches a bit over 8000 partial states. The migration goes on until the old host is disconnected, at which point, the remaining partial states are dropped. The configurations migrate approximately the same number of tuples, but a different number of partial states. Since the utility is random, the state shedding scheme drops fewer partial states than the random case to achieve a higher utility.

Figure V.4c and V.4d compare the optimal and greedy solutions. The utility is a function of the state size—the bigger the partial state is, the higher the utility is. The optimal solution performs visibly better than the greedy solution in Figure V.4c, but not in Figure V.4d. Figure V.4c illustrates a case with few big partial states of varying size and Figure V.4d is based on many small partial states of similar size. The main difference between the two runs is the size of the partial states compared to the disconnection time. This suggests that the benefit of optimization increases with the proportion of the partial state size variance to the total capacity.

V.8 Conclusion and Future Work

This work presents the first investigation of the opportunities and challenges of state shedding for operator migration. It is grounded in the insight that the triggers for operator migration, i.e., overload or network problems, can be limiting factors to successfully performing operator migration with state-of-the-art solutions. Instead of the prevailing all-or-nothing solutions, we propose to perform state shedding to migrate the most useful partial state under the given situation. Both partitioning operator state and estimating the utility of partial state depend on the particular operator. To maximize the aggregated utility of the migrated partial states, we present a solution to the given optimization problem with complexity $O(n^2)$. The simulation-based comparison of this optimal solution with the greedy approach reveals that the distribution of the partial size and the number of partial states play an important role. With few larger partial states the optimal solution outperforms greedy and with many partial states of similar size, they perform almost identically. Further simulation experiments confirm the intuition that the larger the skew in the distribution of the utility of partial states, the faster the aggregated utility at the new host increases. Random selection of partial states to migrate will in cases with utility skew and disconnection before all state is migrated result in lower utility achieved than when using state shedding.

To thoroughly investigate the full potential of state shedding, future work will address novel solutions for utility estimation, e.g., to consider application requirements and to use statistics about previous upstream data, as well as to combine state shedding with scheduling of operator migration, e.g., to delay migration.

Acknowledgments: This work was supported by the Parrot project (Research Council of Norway, 311197).

References

[1] Abadi, D. et al. “Aurora: a data stream management system”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 666–666.

[2] Babcock, B., Datar, M., and Motwani, R. “Load shedding for aggregation queries over data streams”. In: *Proceedings. 20th international conference on data engineering*. IEEE. 2004, pp. 350–361.

[3] Chapnik, K., Kolchinsky, I., and Schuster, A. “DARLING: data-aware load shedding in complex event processing systems”. In: *Proceedings of the VLDB Endowment* vol. 15, no. 3 (2021), pp. 541–554.

[4] Chi, Y. et al. “Loadstar: A load shedding scheme for classifying data streams”. In: *Proceedings of the 2005 siam international conference on data mining*. SIAM. 2005, pp. 346–357.

[5] *Citi Bike trip data set*. https://s3.amazonaws.com/tripdata/201810-citibike-tripdata.csv.zip. 2018.

[6] *Dublin bus GSP data from Dublin city council insight project*. https://data.smartdublin.ie/dataset/dublin-bus-gps-sample-data-from-dublin-city-council-insight-project. 2013.

[7] Gedik, B. et al. “Adaptive load shedding for windowed stream joins”. In: *Proceedings of the 14th ACM international conference on Information and knowledge management*. 2005, pp. 171–178.

[8] Gedik, B., Wu, K.-L., and Philip, S. Y. “Efficient construction of compact shedding filters for data stream processing”. In: *2008 IEEE 24th International Conference on Data Engineering*. IEEE. 2008, pp. 396–405.

[9] Gedik, B. et al. “A load shedding framework and optimizations for m-way windowed stream joins”. In: *2007 IEEE 23rd International Conference on Data Engineering*. IEEE. 2007, pp. 536–545.

[10] Kleiminger, W., Kalyvianaki, E., and Pietzuch, P. “Balancing load in stream processing with the cloud”. In: *2011 IEEE 27th International Conference on Data Engineering Workshops*. IEEE. 2011, pp. 16–21.

[11] Riley, G. F. and Henderson, T. R. “The ns-3 network simulator”. In: *Modeling and tools for network simulation*. Ed. by Wehrle, K., Güneş, M., and Gross, J. Berlin, Heidelberg: Springer, 2010, pp. 15–34.

[12] Rivetti, N., Busnel, Y., and Querzoni, L. “Load-aware shedding in stream processing systems”. In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. 2016, pp. 61–68.

[13] Slo, A., Bhowmik, S., and Rothermel, K. “hSPICE: state-aware event shedding in complex event processing”. In: *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 2020, pp. 109–120.

[14] Slo, A., Bhowmik, S., and Rothermel, K. “State-Aware Load Shedding from Input Event Streams in Complex Event Processing”. In: *IEEE Transactions on Big Data* (2020).

[15] Starks, F., Plagemann, T. P., and Kristiansen, S. “DCEP-Sim: An Open Simulation Framework for Distributed CEP”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS ’17. Barcelona, Spain: ACM, 2017, pp. 180–190.

[16] Tatbul, N. and Zdonik, S. “Window-aware load shedding for aggregation queries over data streams”. In: *VLDB*. Vol. 6. 2006, pp. 799–810.

[17] Tatbul, N. et al. “Load shedding in a data stream manager”. In: *Proceedings 2003 vldb conference*. Elsevier. 2003, pp. 309–320.

[18] Volnes, E., Plagemann, T., and Goebel, V. “To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing”. In: *IEEE Communications Surveys & Tutorials (in revision)* (2023).

[19] Zhao, B. “Complex event processing under constrained resources by state-based load shedding”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1699–1703.

[20] Zhao, B., Hung, N. Q. V., and Weidlich, M. “Load shedding for complex event processing: Input-based and state-based techniques”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE. 2020, pp. 1093–1104.

Appendices

Appendix A

Expose GUI Queries in YAML Format

A.1 Full Expose Configuration in YAML Format

Listing A.1: Expose configuration in YAML format

```
stream—definitions:
— stream—id: 0
  name: StopStream
  tuple—format:
  — {name: streamList, type: string}
  — {name: node, type: int}
— stream—id: 1
  name: Person
  tuple—format:
  — {name: dateTime, type: timestamp}
  — {name: id, type: int}
  — {name: name, type: string}
  — {name: emailAddress, type: string}
  — {name: creditCard, type: string}
  — {name: city, type: string}
  — {name: state, type: string}
— stream—id: 2
  name: Auction
  tuple—format:
  — {name: dateTime, type: timestamp}
  — {name: id, type: int}
  — {name: itemName, type: string}
  — {name: description, type: string}
  — {name: initialBid, type: long}
  — {name: reserve, type: int}
  — {name: expires, type: timestamp}
  — {name: seller, type: int}
  — {name: category, type: int}
— stream—id: 3
  name: Bid
  tuple—format:
  — {name: dateTime, type: timestamp}
  — {name: auction, type: int}
  — {name: bidder, type: int}
  — {name: price, type: long}
— stream—id: 4
  name: Category
  tuple—format:
  — {name: id, type: int}
  — {name: name, type: string}
  — {name: description, type: string}
  — {name: parentCategory, type: int}
— stream—id: 5
  name: BikeTrip
  tuple—format:
  — {name: a_tripduration, type: long}
  — {name: b_starttime, type: string}
  — {name: c_stoptime, type: string}
  — {name: d_start_station_id, type: long}
  — {name: e_start_station_name, type: string}
  — {name: f_start_station_latitude , type: double}
  — {name: g_start_station_longitude , type: double}
  — {name: h_end_station_id , type: long}
  — {name: i_end_station_name , type: string}
  — {name: j_end_station_latitude , type: double}
  — {name: k_end_station_longitude , type: double}
  — {name: l_bikeid , type: long}
  — {name: m_usertype , type: string}
  — {name: n_birth_year , type: long}
  — {name: o_gender , type: long}
— stream—id : 15
  name: BikeTrip_2
  tuple—format :
  — {name: a_tripduration , type: long}
  — {name: b_starttime , type: string}
  — {name: c_stoptime , type: string}
  — {name: d_start_station_id , type: long}
  — {name: e_start_station_name , type: string}
  — {name: f_start_station_latitude , type: double}
  — {name: g_start_station_longitude , type: double}
  — {name: h_end_station_id , type: long}
  — {name: i_end_station_name , type: string}
  — {name: j_end_station_latitude , type: double}
  — {name: k_end_station_longitude , type: double}
  — {name: l_bikeid , type: long}
  — {name: m_usertype , type: string}
  — {name: n_birth_year , type: long}
  — {name: o_gender , type: long}
— stream—id : 6
  name: BusRecord
  tuple—format :
  — {name: a_ts , type: long}
  — {name: b_line_id , type: long}
  — {name: c_direction , type: long}
  — {name: d_journey_pattern_id , type: string}
  — {name: e_time_frame , type: timestamp}
  — {name: f_vehicle_journey_id , type: long}
  — {name: g_operator , type: string}
  — {name: h_congested , type: long}
  — {name: i_longitude , type: double}
  — {name: j_latitude , type: double}
  — {name: k_delay , type: long}
  — {name: l_block_id , type: long}
  — {name: m_vehicle_id , type: long}
  — {name: n_stop_id , type: long}
  — {name: o_at_stop , type: long}
— stream—id : 18
  name: OutQuery
  tuple—format :
  — {name: avgPrice , type: long}
  — {name: maxPrice , type: long}
  — {name: minPrice , type: long}
  — {name: bidder , type: long}
  — {name: auction , type: long}
— stream—id : 19
  name: OutQuery2
  tuple—format :
  — {name: price , type: long}
  — {name: bidder , type: long}
  — {name: auction , type: long}
— stream—id : 20
  name: OutQuery3
  tuple—format :
  — {name: l_bikeid , type: long}
  — {name: h_end_station_id , type: long}
— stream—id : 21
  name: OutQuery4
  tuple—format :
  — {name: avg_price , type: long}
  — {name: count_tuples , type: long}
  — {name: description , type: string}
  — {name: itemName, type: string}
experiments:
— id: 4
  flow:
  — node: coordinator
    task: deployQueries
    arguments: [5, 144]
  — node: coordinator
    task: addPotentialHost
    arguments:
    — [2, 3, 4]
    — 144
  — node: coordinator
    task: addSinkNode
    arguments:
    — [5]
    — 144
  — node: coordinator
    task: addSourceNode
    arguments:
    — [1]
    — 144
  — node: coordinator
    task: adapt
    arguments: [144, 3, drop—state]
  — node: 1
    task: wait
    arguments: [1000000000]
  — node: coordinator
    task: loopTasks
    arguments:
    — 1
    — — node: coordinator
        task: loopTasks
        arguments:
        — 10
        — — node: 1
            task: sendNRowsDsAsSpecificStream
            arguments: [45, 2, 1000]
      — node: 3
        task: retWhenReceived
        arguments: [10000]
      — node: 3
        task: wait
        arguments: [1000000000]
      — node: 1
        task: sendNRowsDsAsSpecificStream
        arguments: [94, 3, 100000]
      — node: 3
        task: retWhenProcessed
        arguments: [ 100000, 144, join ]
      — node: coordinator
        task: wait
        arguments: [ 1000000 ]
      — node: coordinator
        task: adapt
        arguments: [144, 4, migration_mechanism]
      — node: 5
        task: retEndOfStream
        arguments: [1000000000]
datasets:
— {file: nexmark—dataset—1000—unique—bidders—100000—rows.csv,
   name: NexMark 100000 tuples (CSV), id: 94, type: csv}
— {file: auction—10000.csv, name: 10k Auction, id: 46, type: csv}
spequeries:
  outputs: []
  inputs: []
```

```
queries:
- operators:
  - name: Output 0
    type: output
    parameters: {stream-id: -1}
  - name: Select 0
    type: select
    parameters:
      fields: [avg(B.price) avg_price, count(B.auction) count_tuples,
               A.description, A.itemName]
  - name: input 0
    type: input
    parameters: {stream-id: 3, alias: B}
  - name: Print 0
    type: print
    parameters: {}
  - name: Window 1
    type: window
    parameters: {external-timestamp-field: '', size: 100000,
      emit-type: PROCESSING_TIME, emit-size: 100,
      size-type: PROCESSING_TIME, jump: 1000}
  - name: GroupBy 1
    type: groupby
    parameters:
      fields: [B.bidder]
  - name: input 1
    type: input
    parameters: {stream-id: 2, alias: A}
  - name: Join 0
    type: join
    parameters: {}
  - name: Window 0
    type: window
    parameters: {external-timestamp-field: '', size: 0,
      emit-type: TUPLE_COUNT, emit-size: -1,
      size-type: TUPLE_COUNT, jump: 1}
  - name: Window 2
    type: window
    parameters: {external-timestamp-field: '', size: 1000000,
      emit-type: TUPLE_COUNT, emit-size: -1,
      size-type: TUPLE_COUNT, jump: 1}
  - name: Filter 2
    type: filter
    parameters:
      op: larger_than_or_equals
      arg2: {value-type: int, type: attribute, value: A.id, offset: 0}
      arg1: {value-type: int, type: attribute, value: B.auction}
  - name: Filter 3
    type: filter
    parameters:
      op: smaller_than_or_equals
      arg2: { value-type: int, type: attribute, value: A.id, offset: 0 }
      arg1: { value-type: int, type: attribute, value: B.auction }
  name: NewName0
  edges:
  - stream: stream_name1
    from: {name: Window 0, type: intermediate}
    to: {name: Join 0, type: intermediate}
  - stream: stream_name0
    from: {name: Window 1, type: intermediate}
    to: {name: Select 0, type: intermediate}
  - stream: Auction
    from: {name: input 1, type: intermediate}
    to: {name: Window 2, type: intermediate}
  - stream: Output 0
    from: {name: Output 0, type: output}
  - stream: stream_name5
    from: {name: Join 0, type: intermediate}
    to: {name: Filter 2, type: intermediate}
    — stream: OutQuery4
      from: {name: Select 0, type: intermediate}
      to: {name: Print 0, type: intermediate}
    — stream: stream_name4
      from: {name: GroupBy 1, type: intermediate}
      to: {name: Window 1, type: intermediate}
    — stream: stream_name3
      from: {name: Window 2, type: intermediate}
      to: {name: Join 0, type: intermediate}
    — stream: Bid
      from: {name: input 0, type: intermediate}
      to: {name: Window 0, type: intermediate}
    — stream: Final
      from: {name: Print 0, type: intermediate}
      to: {name: Output 0, type: intermediate}
    — stream: stream_name330
      from: { name: Filter 2, type: intermediate }
      to: { name: Filter 3, type: intermediate }
    — stream: stream_name6
      from: {name: Filter 3, type: intermediate}
      to: {name: GroupBy 1, type: intermediate}
    id: 5
plots:
— {operator—name: Print 0, query—id: '144', control—experiment—id: '4',
  stream—id: '3', name: plot1, type: Input—latency}
network:
  bandwidth: {migration: '10000000', tuples: '10000000'}
  latency: {migration: '0', tuples: '0'}
```

A.2 Grouped Aggregation Query in YAML Format

Listing A.2: Grouped aggregation query in YAML format

```
spequeries:
  outputs: []
  inputs: []
  queries:
  — operators:
    — name: Output 0
      type: output
      parameters: {stream—id: —1}
    — name: Select 0
      type: select
      parameters:
        fields: [avg(B.price) avg_price, count(B.auction) count_tuples]
    — name: input 0
      type: input
      parameters: {stream—id: 3, alias: B}
    — name: Print 0
      type: print
      parameters: {}
    — name: Window 1
      type: window
      parameters: {external—timestamp—field: '', size: 700000,
        emit—type: PROCESSING_TIME, emit—size: 1000000,
        size—type: PROCESSING_TIME, jump: 1000000}
    — name: GroupBy 1
      type: groupby
      parameters:
        fields: [B.bidder]
    name: NewName0
    edges:
    — stream: stream_name0
      from: {name: Window 1, type: intermediate}
      to: {name: Select 0, type: intermediate}
    — stream: Output 0
      from: {name: Output 0, type: output}
    — stream: OutQuery4
      from: {name: Select 0, type: intermediate}
      to: {name: Print 0, type: intermediate}
    — stream: stream_name3
      from: {name: GroupBy 1, type: intermediate}
      to: {name: Window 1, type: intermediate}
    — stream: Bid
      from: {name: input 0, type: intermediate}
      to: {name: GroupBy 1, type: intermediate}
    — stream: Final
      from: {name: Print 0, type: intermediate}
      to: {name: Output 0, type: intermediate}
    id: 5
```

A.3 Non-equijoin Query in YAML Format

Listing A.3: Non-equijoin query in YAML format

```
spequeries:
  outputs: []
  inputs: []
  queries:
  — operators:
    — name: Output 0
      type: output
      parameters: {stream—id: —1}
    — name: Select 0
      type: select
      parameters:
        fields: [B.price, A.id]
    — name: input 0
      type: input
      parameters: {stream—id: 3, alias: B}
    — name: Print 0
      type: print
      parameters: {}
    — name: input 1
      type: input
      parameters: {stream—id: 2, alias: A}
    — name: Join 0
      type: join
      parameters: {}
    — name: Window 0
      type: window
      parameters: {external—timestamp—field: '', size: 0,
        emit—type: TUPLE_COUNT, emit—size: —1,
        size—type: TUPLE_COUNT, jump: 1}
    — name: Window 2
      type: window
      parameters: {external—timestamp—field: '', size: 1000000,
        emit—type: TUPLE_COUNT, emit—size: —1,
        size—type: TUPLE_COUNT, jump: 1}
    — name: Filter 2
      type: filter
      parameters:
        op: larger_than_or_equals
        arg2: {value—type: int, type: attribute, value: A.id, offset: 0}
        arg1: {value—type: int, type: attribute, value: B.auction}
    — name: Filter 3
      type: filter
      parameters:
        op: smaller_than_or_equals
        arg2: { value—type: int, type: attribute, value: A.id, offset: 0 }
      arg1: { value—type: int , type: attribute , value: B.auction }
  name: NewName0
  edges:
  — stream: stream_name1
    from: {name: Window 0, type: intermediate}
    to: {name: Join 0, type: intermediate}
  — stream: stream_name0
    from: {name: Filter 2, type: intermediate}
    to: {name: Filter 3, type: intermediate}
  — stream: stream_name9
    from: { name: Filter 3, type: intermediate }
    to: { name: Select 0, type: intermediate }
  — stream: Auction
    from: {name: input 1, type: intermediate}
    to: {name: Window 2, type: intermediate}
  — stream: Output 0
    from: {name: Output 0, type: output}
  — stream: stream_name5
    from: {name: Join 0, type: intermediate}
    to: {name: Filter 2, type: intermediate}
  — stream: OutQuery4
    from: {name: Select 0, type: intermediate}
    to: {name: Print 0, type: intermediate}
  — stream: stream_name3
    from: {name: Window 2, type: intermediate}
    to: {name: Join 0, type: intermediate}
  — stream: Bid
    from: {name: input 0, type: intermediate}
    to: {name: Window 0, type: intermediate}
  — stream: Final
    from: {name: Print 0, type: intermediate}
    to: {name: Output 0, type: intermediate}
  id: 5
```

A.4 Join Followed By Grouped Aggregation Query in YAML Format

Listing A.4: Join followed by grouped aggregation query in YAML format

```
spequeries:
  outputs: []
  inputs: []
  queries:
  — operators:
    — name: Output 0
      type: output
      parameters: {stream—id: —1}
    — name: Select 0
      type: select
      parameters:
        fields: [avg(B.price) avg_price, count(B.auction) count_tuples,
          A.description, A.itemName]
    — name: input 0
      type: input
      parameters: {stream—id: 3, alias: B}
    — name: Print 0
      type: print
      parameters: {}
    — name: Window 1
      type: window
      parameters: {external—timestamp—field: '', size: 100000,
        emit—type: PROCESSING_TIME, emit—size: 100, size—type:
        PROCESSING_TIME, jump: 1000}
    — name: GroupBy 1
```

```
   type: groupby
   parameters:
     fields: [A.id]
 — name: input 1
   type: input
   parameters: {stream—id: 2, alias: A}
 — name: Join 0
   type: join
   parameters: {}
 — name: Window 0
   type: window
   parameters: {external—timestamp—field: '', size: 0,
     emit—type: TUPLE_COUNT, emit—size: —1, size—type: TUPLE_COUNT, jump: 1}
 — name: Window 2
   type: window
   parameters: {external—timestamp—field: '', size: 1000000,
     emit—type: TUPLE_COUNT, emit—size: —1, size—type: TUPLE_COUNT, jump: 1}
 — name: Filter 2
   type: filter
   parameters:
     op: equals
     arg2: {value—type: int, type: attribute, value: A.id}
     arg1: {value—type: int, type: attribute, value: B.auction}
 name: NewName0
 edges:
 — stream: stream_name1
   from: {name: Window 0, type: intermediate}
   to: {name: Join 0, type: intermediate}
 — stream: stream_name0
   from: {name: Window 1, type: intermediate}
   to: {name: Select 0, type: intermediate}
 — stream: Auction
   from: {name: input 1, type: intermediate}
   to: {name: Window 2, type: intermediate}
 — stream: Output 0
   from: {name: Output 0, type: output}
 — stream: stream_name5
   from: {name: Join 0, type: intermediate}
   to: {name: Filter 2, type: intermediate}
 — stream: OutQuery4
   from: {name: Select 0, type: intermediate}
   to: {name: Print 0, type: intermediate}
 — stream: stream_name4
   from: {name: GroupBy 1, type: intermediate}
   to: {name: Window 1, type: intermediate}
 — stream: stream_name3
   from: {name: Window 2, type: intermediate}
   to: {name: Join 0, type: intermediate}
 — stream: Bid
   from: {name: input 0, type: intermediate}
   to: {name: Window 0, type: intermediate}
 — stream: Final
   from: {name: Print 0, type: intermediate}
   to: {name: Output 0, type: intermediate}
 — stream: stream_name6
   from: {name: Filter 2, type: intermediate}
   to: {name: GroupBy 1, type: intermediate}
 id: 5
```