# Network-centric Complex Event Processing

## Adaptive, Efficient, and Flexible Placement and Execution of Internet of Things Applications

at the Department of Electrical Engineering and Information Technology
of the Technische Universität Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

submitted in fulfillment of the requirements for the degree of
Doctor of Engineering (Dr.-Ing.)

**Dissertation by Manisha Luthra, M.Sc.,**
**born on 24.07.1990 in New Delhi, India**

First examiner: Prof. Dr.-Ing. Ralf Steinmetz
Second examiner: Prof. Dr. Boris Koldehofe
Submitted on 11th May 2021
Darmstadt

etit
ELEKTROTECHNIK UND
INFORMATIONSTECHNIK

Manisha Luthra, M.Sc.: *Network-centric Complex Event Processing,*
Adaptive, Efficient, and Flexible Placement and Execution
of Internet of Things Applications
Darmstadt, Technische Universität Darmstadt
Year thesis published in TUprints: 2021
URN: *urn:nbn:de:tuda-tuprints-192857*
Date of the viva voce: *5th August 2021*

*Dedicated to my mother,*
*whose loving spirit sustains me still.*

# Abstract

Complex Event Processing (CEP) is a widely used paradigm to detect and react to events of interest for various applications. Numerous companies, including Twitter and Google, build on CEP in a broad spectrum of applications to perform real-time data analytics. Many of these applications require to *efficiently adapt* to the dynamic environmental conditions and changes in the quality requirements. An essential building block to perform adaptations in CEP is through an *operator*, which encapsulates the event detection logic in the form of a *query* often coupled with an execution *state*. Despite significant contributions to concepts for operator specification, placement, and execution, there are multiple research gaps concerning *adaptivity*, *efficiency*, and *interoperability* in CEP. Thereby, this thesis identifies and contributes appropriate methods and their analysis to overcome these fundamental research gaps in CEP: *(i)* The lack of adaptivity between CEP mechanisms hinders meeting the changing quality requirements of applications. *(ii)* Absence of suitable network-centric abstractions that hinder efficient event processing. *(iii)* Absence of suitable programming abstractions that hinder reuse of CEP mechanisms across multiple programming models.

To close the first gap, we contribute a novel programming model, named TCEP, that enables transitions between so-called operator placement mechanisms. The programming model provides methods for the research questions '*when*' and '*how*' to perform a transition while ensuring crucial properties of transition such as *seamlessness*. In particular, we propose transition strategies that minimize the costs for operator migrations and ensure seamlessness in performing adaptations. A learning-based selection algorithm guarantees a well-suited operator placement mechanism for given quality requirements. By integrating and evaluating six operator placement mechanisms, we showed that the programming model allows the use of distinct mechanisms for adaptations, and it provides a better understanding of their cost and performance characteristics. Our extensive evaluation study using a real-world workload and implementation shows that TCEP can adapt to the dynamic quality requirements of applications in a quick, seamless, and low-cost manner.

To close the second gap, we propose a novel unified communication model named INETCEP. The proposed concepts of INETCEP contribute to the research question of '*how*' to enable efficient continuous event stream processing and network-centric CEP query execution. We build INETCEP using the concepts of Information-centric Networking, which has been proven to facilitate in-network programmability. As part of the unified communication model, we propose an expressive meta query language and query execution algorithms for CEP that efficiently place operators over Information-centric

Networks. Our detailed evaluation study of INETCEP shows that event forwarding can be achieved in a very short time of a few microseconds. Similarly, using our network-centric abstractions, CEP queries can be resolved at very high incoming event rates in a few milliseconds while incurring no event loss.

Finally, we propose a novel unified CEP middleware, named CEPLESS, based on the *serverless computing* principles to close the third gap. The middleware provides concepts for the research question of '*how*' to specify CEP queries independent of their programming and execution environment. Specifically, the middleware contributes a programming abstraction that hides away the complexity of heterogeneous CEP programming models from the application developers. Moreover, we propose mechanisms for an efficient exchange of events using so-called in-memory queues and allow event processing across different execution models. By extending the CEPLESS middleware programming abstraction with five different programming languages, we show extensibility as well as the platform and language independence of the concept. Our evaluation using a real-world workload and implementation shows that event processing using the CEPLESS middleware is equally performant as native CEP systems.

Overall, this thesis contributes *(i)* a novel programming model and methods for transitions in CEP systems to support changing quality requirements, *(ii)* a novel unified communication model and efficient algorithms that accelerate query execution using the concepts of Information-centric Networking, and *(iii)* a novel serverless middleware with programming abstractions to achieve efficient execution and reuse of multiple and heterogeneous CEP execution environments.

# Kurzfassung

Complex Event Processing (CEP) ist ein weit verbreitetes Paradigma, das Anwendungen unterstützt, Ereignisse zu erkennen und auf diese zu reagieren. Zahlreiche Unternehmen, unter anderem Twitter und Google, nutzen CEP in vielfältigen Anwendungsfeldern, um Echtzeit-Analysen in Form der Ereignisverarbeitung über Ereignisströme durchzuführen. Viele dieser Anwendungen erfordern die dynamische Anpassung an sich ändernde Rahmenbedingungen und Qualitätsanforderungen. Ein zentraler Baustein bei der Adaption von CEP ist der *Operator*, welcher Logik und entsprechenden Zustand der Datenanalyse umfasst. Trotz signifikanter Forschungsanstrengungen zu Konzepten und Mechanismen hinsichtlich Spezifikation, Platzierung und Ausführung von Operatoren, besteht eine erhebliche Forschungslücke bei der Unterstützung von *Adaptivität*, *Effizienz* und *Interoperabilität* von CEP. Entsprechend identifiziert und trägt diese Arbeit mit geeigneten Methoden sowie deren Analyse dazu bei, drei fundamentalen Problemen in der dynamischen Anpassung von CEP entgegenzuwirken: *(i)* die fehlende Adaptivität von CEP-Mechanismen, die das Erfüllen von sich dynamisch ändernden Qualitätsanforderungen solcher Anwendungen erschwert, *(ii)* fehlende netzwerkzentrische Abstraktionen, die eine effiziente Datenanalyse in Form der Ereignisverarbeitung behindern und *(iii)* fehlende Programmierabstraktionen, die die Wiederverwendung von CEP-Mechanismen über mehrere Programmiermodelle hinweg einschränken.

Entsprechend der ersten Forschungslücke trägt diese Dissertation zu einem neuen Programmiermodell namens TCEP bei, dass Transitionen zwischen Mechanismen der Operatorplatzierung unterstützt. Die als Teil des TCEP Programmiermodells konzipierten Verfahren leisten einen Beitrag zu Konzepten zur Durchführung von Transitionen, insbesondere hinsichtlich der Forschungsfragen '*wann*' und '*wie*' Transitionen nahtlos durchgeführt werden sollen. Die konzipierten Transitionen stellen sicher, die Kosten für Operator-Migrationen in Folge von Anpassungen zu minimieren. Ein auf Lernverfahren basierter Auswahlalgorithmus garantiert die Verwendung geeigneter Operator-Platzierungsmechanismen entsprechend gegebener Qualitätsanforderungen. Durch die Integration und Evaluierung von sechs Operator-Platzierungsmechanismen wird gezeigt, dass das Programmiermodell von TCEP die Verwendung unterschiedlicher Mechanismen zur Anpassung unterstützt und ein besseres Verständnis der Kosten- und Leistungsmerkmale ermöglicht. Als Teil einer umfangreichen Evaluierungsstudie mit realistischer Arbeitslast und Implementierung zeigt, dass TCEP schnell und kostengünstig Anwendungen an dynamische Qualitätsanforderungen anpassen kann.

Weiterhin wird in dieser Dissertation mit INETCEP ein neues einheitliches Kommunikationsmodell vorgestellt, dass beiträgt die zweite identifizierte Forschungslücke zu schließen. Die als Teil von INETCEP entwickelten Konzepte ermöglichen durch Nutzung und Erweiterung von Konzepten der Informations-zentrischen Netze (ICN) die Ausführung von Operatoren auf Ressourcen eines Rechnernetzes. Das auf die Anforderungen von CEP und ICN abgestimmte vereinheitlichte Kommunikationsmodell ermöglicht Effizienzgewinne in der Verarbeitung von kontinuierlichen Ereignisströmen. Durch eine eigens für ICN konzipierte ausdrucksstarke Meta-Sprache, können Operatoren einfach auf ICN Ressourcen von ICN abgebildet und platziert werden. Eine detaillierte Evaluierung von INETCEP zeigt, dass die Weiterleitung von Ereignissen über ICN mit geringer Verzögerung im Bereich von wenigen Mikrosekunden erreicht werden kann. In ähnlicher Weise werden CEP Anfragen, selbst bei einer sehr hohen Eingangslast, über die in INETCEP bereitgestellten netzzentrischen Abstraktionen in wenigen Millisekunden aufgelöst, ohne dass es zu Verlusten von Ereignissen kommt.

Schließlich werden als Teil der konzipierten CEPLESS Middleware Konzepte zur vereinheitlichten Nutzung verschiedener CEP Programmiermodelle und deren integrierte Nutzung in heterogenen Ressourcenumgebungen vorgeschlagen. Diese basieren auf Prinzipien des sogenannte Serverless Computing. Die Konzepte und Abstraktionen in CEPLESS ermöglichen eine Programmierung und Spezifikation von CEP Abfragen unabhängig von der Ressourcenumgebung, in der sie ausgeführt werden. Dies verbirgt die Komplexität der Vielfalt und Heterogenität bestehender CEP Programmiermodelle vor den Anwendungsentwicklern. Die Konzeption neuer Mechanismen, die den Austausch von Ereignissen über sogenannte In-Memory Queues erreichen, können Ereignisse effizient über verschiedenen Ressourcenumgebungen verarbeiten. Die Erweiterbarkeit und Unabhängigkeit der Spezifikationskonzepte werden durch die Integration von fünf verschiedenen Programmiersprachen in CEPLESS aufgezeigt. Die als Teil der Dissertation durchgeführten Studien unter realen Bedingungen belegen, dass durch die Konzepte der vereinheitlichten Ereignisverarbeitung über CEPLESS keine Leistungsnachteile gegenüber nativen CEP Systemen entstehen.

Insgesamt leistet diese Dissertation Beiträge *(i)* zu einem Programmiermodell und Methoden zur Durchführung von Transitionen, die hochdynamisch die Platzierung von CEP-Operatoren unterstützen, *(ii)* zu einem neuen einheitlichen Kommunikationsmodell und entsprechenden Konzepten, die eine Ausführung von Operatoren als Teil Informationszentrischer Netze beschleunigen und *(iii)* zu Verfahren und Programmierabstraktionen, die basierend auf dem Paradigma des Serverless Computing die effiziente Ausführung, Wiederverwendung und Zusammenführung verschiedenster CEP Systeme über heterogene Ressourcenumgebungen erreichen.

# Acknowledgments

Firstly, I would like to sincerely thank my advisor, Prof. Ralf Steinmetz, for giving me this opportunity to pursue a doctoral degree in the Multimedia Communications Lab (KOM) at TU Darmstadt. Without his guidance and support, this research work wouldn't have been possible. A special thanks go to my second advisor of the dissertation, Prof. Boris Koldehofe, for his continuous supervision, encouragement, and support to conduct research in this exciting research area of Complex Event Processing. I am incredibly grateful for the fruitful research discussions that resulted in the impactful publications of this dissertation. Next, I would like to express sincere gratitude to Prof. Carsten Binnig for agreeing to be the referee of this dissertation. I am looking forward towards many more interesting research discussions and publications with you in the Data Management Lab.

I am very grateful for the opportunity to conduct research within a Collaborative Research Center MAKI and for the funding provided by Deutsche Forschungsgemeinschaft (DFG) (German Science Foundation). Being a research scientist in MAKI, I got an opportunity to collaborate with several great people, including the principal investigators: Prof. Guido Salvaneschi, Prof. Bernd Freisleben, Prof. Lin Wang, Prof. Amr Rizk, and Prof. Andy Schürr. Furthermore, I got an opportunity to exchange ideas and work together with the research associates of different sub-projects in MAKI: Pascal Weisenberger, Jonas Höchst, Artur Sterz, Patrick Lampe, Markus Weckesser, Martin Pfannemüller, and Kamran Razavi. A big thanks go to them and all members of MAKI; I am grateful to all the collaborators for their support.

I would like to thank all the present and past members of KOM for providing a friendly and comfortable environment to work in. A big thank you goes to Rhaban Hark for helping me out in the final days of my thesis and patiently listening to me :-). A special thanks go to all the ATMs: Frau Scholz-Schmidt (the know-it-all), Karola Schork-Jakobi (my German language practice partner, thank you for the nice time we spent talking to each other), Frank Jöst, Sabine Kräh, Frau Ehlhardt, Julia Müller, and Michaela Bock (also for MAKI) for always helping in the administrative things. Many thanks to the people (past and present) whom I worked with either research-wise or other duties (in no order): Sounak Kar, Ralf Kundel, Tim Steuer, Tobias Meuser, Polona Caserman, Denny Stohr, Wael Alkhatib, Yassin Alkhalili, Leonhard Nobach, The An Binh Nguyen, Patrick Lieser, and Rahul Wermund. I would also like to thank the students whom I supervised, for their involvement in this work: Sebastian Hennig (who received an EXIST grant and founded a start-up called arryve GmbH), Ali Haider Rizvi, Johannes Pfannmüller (both now working in Software AG), Benedikt Lins (working in Acxiom GmbH), Raheel Arif (working in 360 Trading Networks), and Niels Danger (who recently submitted his master thesis). I also would like to extend my gratitude to my mentor and master

thesis supervisors Prof. Wolfgang Effelsberg and Stefan Wilk, for motivating me to pursue this work and supporting me throughout this journey.

Further, I would like to thank my colleagues (who became friends): Anam Tahir, Rhaban Hark, Wasiur Khudabukhsh, and Sounak Kar (not only for proof-reading thesis and papers) but also for the nice time we spent together. A big thank you also goes to all who proof-read one or more chapters of this dissertation: Rhaban Hark, Boris Koldehofe, Pratyush Agnihotri, Utkarsh Agnihotri, Mansi Kakkar, Julian Zobel, Wasiur Khudabukhsh, Sounak Kar, Anam Tahir, Sebastian Hennig, Johannes Pfannmüller, and Trevor Ballard.

Finally, I would like to dedicate this dissertation to my mother, Smt. Late Gunita Luthra who is my inspiration and the one behind what I am today. If it were not for all her morals, values, and principles, I would not have been the person I am today; thank you, mother, for everything. A special and big thanks go to my loving husband and my best friend Pratyush Agnihotri for supporting me in literally everything in every way. You are and will be my source of motivation and lifelong support. My extreme gratitude goes to my father and sister for their great support, brother-in-law, who supported me also in many ways, especially for proof-reading this dissertation, and my parents-in-law, who love me as much as my parents do. Thank you for all the love, encouragement, and support you have given to me.

# Author's Publications

## Major Publications

The author of this thesis published major parts of the presented content in the following peer-reviewed publications:


[1] Manisha Luthra, Boris Koldehofe, Niels Danger, Pascal Weisenburger, Guido Salvaneschi, and Ioannis Stavrakakis. "TCEP: Transitions in Operator Placement to Adapt to Dynamic Network Environments." In: *Journal of Computer and Systems Sciences (JCSS), Special Issue on Algorithmic Theory of Dynamic Networks and its Applications (accepted for publication)* (2020), pp. 1–76. URL: https://luthramanisha.github.io/TCEP/ (cit. on pp. xv, xvi, xviii).

[2] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. "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 (DEBS)*. 2018, pp. 136–147 (cit. on pp. xv–xviii).

[3] Manisha Luthra, Johannes Pfannmüller, Boris Koldehofe, Jonas Höchst, Artur Sterz, Rhaban Hark, and Bernd Freisleben. "Efficient Complex Event Processing in Information-centric Networking at the Edge (under submission)." In: (2021), pp. 1–17. URL: https://arxiv.org/pdf/2012.05070.pdf (cit. on pp. xv–xvii, xix).

[4] Manisha Luthra, Boris Koldehofe, Jonas Höchst, Patrick Lampe, Ali Haider Rizvi, Ralf Kundel, and Bernd Freisleben. "INetCEP: In-Network Complex Event Processing for Information-Centric Networking." In: *Proceedings of the 15th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–13. URL: https://luthramanisha.github.io/INetCEP/ (cit. on pp. xv–xvii, xix).

[5] Manisha Luthra, Sebastian Hennig, Kamran Razavi, Lin Wang, and Boris Koldehofe. "Operator as a Service: Stateful Serverless Complex Event Processing." In: *Proceedings of the IEEE International Conference on Big Data Workshops (BigData Workshop)*. 2020, pp. 1–10. URL: https://luthramanisha.github.io/CEPless (cit. on pp. xv–xvii, xix).

[6] Bastian Alt, Markus Weckesser, Christian Becker, Matthias Hollick, Sounak Kar, Anja Klein, Robin Klose, Roland Kluge, Heinz Koeppl, Boris Koldehofe, Wasiur R. Khudabukhsh, Manisha Luthra, Mahdi Mousavi, Max Mühlhäuser, Martin Pfannemüller, Amr Rizk, Andy Schürr, and Ralf Steinmetz. "Transitions: A Protocol-Independent View of the Future Internet." In: *Proceedings of the IEEE* 107.4 (2019), pp. 835–846 (cit. on p. xv).

[7] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. "Quality-aware Runtime Adaptation in Complex Event Processing." In: *Proceedings*

*of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017, pp. 140–151 (cit. on pp. xv, xvii, xviii).

[8] Manisha Luthra, Boris Koldehofe, and Ralf Steinmetz. "Transitions for Increased Flexibility in Fog Computing: A Case Study on Complex Event Processing." In: *Informatik Spektrum* 42.4 (2019), pp. 244–255 (cit. on pp. xv, xvii).

[9] Manisha Luthra and Boris Koldehofe. "ProgCEP: A Programming Model for Complex Event Processing over Fog Infrastructure." In: *Proceedings of the 2nd International Workshop on Distributed Fog Services Design (DFSD@Middleware)*. 2019, pp. 7–12 (cit. on pp. xv, xviii).

[10] Manisha Luthra. "Adapting to Dynamic User Environments in Complex Event Processing System Using Transitions." In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS), Doctorial Symposium*. 2018, pp. 274–277 (cit. on pp. xv, xviii).

[11] Manisha Luthra, Sebastian Hennig, and Boris Koldehofe. "Understanding the Behavior of Operator Placement Mechanisms on Large-Scale Networks." In: *Proceedings of the 19th ACM/IFIP Middleware Conference: Posters and Demos (Middleware)*. 2018, pp. 19–20 (cit. on p. xv).

[12] Manisha Luthra, Boris Koldehofe, and Ralf Steinmetz. "Adaptive Complex Event Processing over Fog-Cloud Infrastructure Supporting Transitions." In: *Proceedings of the GI/ITG KuVS-Fachgespräch (Fog-Computing)*. 2018, pp. 1–4 (cit. on pp. xv, xvii).

[13] Manisha Luthra and Boris Koldehofe. "Highly Flexible Server Agnostic Complex Event Processing Operators." In: *Proceedings of the 20th ACM/IFIP International Middleware Conference: Posters and Demos (Middleware)*. 2019, pp. 11–12 (cit. on pp. xv, xix).

## Additional Publications

Furthermore, the author of this thesis co-authored the following peer-reviewed publications, which are not part of this thesis:

[14] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. "Emu: Elastic Multi-stage Deep Learning Inference with SLA Guarantees." In: *Submitted*. 2021, pp. 1–12.

[15] Ralf Kundel, Christoph Gärtner, Manisha Luthra, Sukanya Bhowmik, and Boris Koldehofe. "Flexible Content-based Publish/Subscribe over Programmable Data Planes." In: *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 2020, pp. 1–5.

[16] Martin Pfannemüller, Markus Weckesser, Roland Kluge, Janick Edinger, Manisha Luthra, Robin Klose, Christian Becker, and Andy Schürr. "CoalaViz: Supporting Traceability of Adaptation Decisions in Pervasive Communication Systems." In: *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 590–595.

[17] Martin Pfannemüller, Janick Edinger, Markus Weckesser, Roland Kluge, Manisha Luthra, Robin Klose, Christian Becker, and Andy Schürr. “Demo: Visualizing Adaptation Decisions in Pervasive Communication Systems.” In: *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 335–337.

[18] Pratyush Agnihotri, Manisha Luthra, and Sascha Peters. “UrbanPulse: Adaptable Middleware to Offer City and User Centric Smart City Solution.” In: *Proceedings of the 20th International Middleware Conference Demos and Posters (Middleware)*. 2019, pp. 29–30.

[19] Yassin Alkhalili, Manisha Luthra, Amr Rizk, and Boris Koldehofe. “3-D Urban Objects Detection and Classification From Point Clouds.” In: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems (DEBS)*. 2019, pp. 209–213.

[20] The An Binh Nguyen, Pratyush Agnihotri, Christian Meurisch, Manisha Luthra, Rahul Dwarakanath, Jeremias Blendin, Doreen Böhnstedt, Michael Zink, and Ralf Steinmetz. “Efficient Crowd Sensing Task Distribution Through Context-Aware NDN-Based Geocast.” In: *Proceedings of the 42nd IEEE Conference on Local Computer Networks (LCN)*. 2017, pp. 52–60 (cit. on p. xviii).

[21] Stefan Wilk, Manisha Luthra, and Wolfgang Effelsberg. “One Sensor is Not Enough: Adapting and Fusing Sensors for the Quality Assessment of User Generated Video.” In: *Proceedings of the 24th ACM International Conference on Multimedia (ACM MM)*. 2016, pp. 626–630.

[22] Emmanouil Vasilomanolakis, Jörg Daubert, Manisha Luthra, Vangelis Gazis, Alex Wiesmaier, and Panayotis Kikiras. “On the Security and Privacy of Internet of Things Architectures and Systems.” In: *Proceedings of the International Workshop on Secure Internet of Things (SIoT)*. 2015, pp. 49–57.

# Previously Published Material

This thesis contains material that has been previously published in various scientific journals and conferences. Table 1 summarises the relationship between the former publications and the chapters of this thesis. None of the text in this thesis is directly taken from the publications. However, figures, tables, algorithms, notations, and results that exclusively represent the core concepts, have been used consistently in this thesis to prevent discrepancies with the true data that is formerly published. The list of all the author's scientific publications is available in the previously presented *Chapter Author's Publications*.

| Chapters | Publications |
|---|---|
| *Chapter 2, Fundamentals and State-of-the-art* | Luthra et al. [1, 2, 3, 4, 5] |
| *Chapter 3, Scenario and System Architecture* | Luthra et al. [1, 2, 3, 5, 9] |
| *Chapter 4, Mechanism Transitions in Operator Placement* | Luthra et al. [1, 2, 8, 9, 10, 11, 12], Weisenburger et al. [7], Alt et al. [6] |
| *Chapter 5, Network-centric Query Execution* | Luthra et al. [3, 4] |
| *Chapter 6, Unified Serverless Middleware* | Luthra et al. [5, 13] |

Table 1: List of peer-reviewed scientific publications used in the corresponding chapters of this thesis.

High-quality scientific work is a result of a collaborative environment. This thesis is no exception, which is carried out in the scope of the DFG (German Research Foundation) funded Collaborative Research Center 1053 – MAKI (Multi-Mechanism Adaptation for the Future Internet)[1]. MAKI combines interdisciplinary research areas of computer science, electrical engineering and information technology, and economics to achieve a highly adaptive and efficient Future Internet. Therefore, the below-mentioned publications result from a collaborative effort of researchers from the computer science, electrical engineering, and economics departments either associated with the MAKI project or the Multimedia Communications Lab of the Technical University of Darmstadt. In the following, the pronoun "I" is used exclusively in this chapter to describe the contributions of the author of this thesis to each publication. The contributions of the co-authors, and their affiliations, are also described. The co-authors where no dedicated institution is provided are col-

[1]MAKI – Multi Mechanism Adaptation for the Future Internet `https://www.maki.tu-darmstadt.de/` [Accessed in May 2021].

leagues at the Multimedia Communications Lab of the Technical University of Darmstadt or associated with the MAKI project. In the remaining thesis, the pronoun "we" is used instead, which refers to all the co-authors of the respective contribution.

The contributions provided in this thesis were continually supervised by Prof. Dr. Boris Koldehofe (Professor at the University of Groningen, Netherlands) and Prof. Dr.-Ing. Ralf Steinmetz. The author of this thesis got feedback from Prof. Koldehofe on the methodology, system model, and written publications. Furthermore, Prof. Steinmetz supervised the work conducted in this thesis as well as the publications. In the below description, I have discussed their concrete contributions. However, wherever not explicitly mentioned, they generally contributed via continuous supervision.

Chapter 2, *Fundamentals and State-of-the-art*, presents an extensive analysis of the background and literature work associated with the three core problems addressed by this thesis, *adaptivity*, *efficiency*, and *interoperability* in the context of Complex Event Processing (CEP) systems. In this chapter, I *(i)* identify the central requirements for the problems and *(ii)* perform an extensive analysis to discover the fundamental research gaps addressed by this work towards the aforementioned problems. Concerning the *adaptivity* problem, I conducted a review of the existing Operator Placement mechanisms, Self-Adaptive Systems in the domain of CEP, and currently available open-source and commercial CEP systems. The discussion related to the review is partially published in [1], which is an extension of our previous work in [2]. Concerning the *efficiency* problem, the systematic review of the current networking architectures related to the Information-centric Networking paradigm concerning the categorisation of pull- and push-based architectures is under submission in [3], which is an extension of the work published in [4]. Concerning the *interoperability* problem, the thesis work in [1] helped to get a common understanding of the system model of serverless platforms within the Cloud Computing paradigm presented in this chapter. I improved the literature work analysis based on serverless computing and CEP systems published in [5]. All the co-authors of the previous works [1, 3, 5] helped review the written manuscript.

Chapter 3, *Scenario and System Architecture*, I present a common scenario, system model, and architecture related to the three core contributions of this work. The analysis of the traffic scenario related to the *adaptivity* problem was previously published in [1], and the fraud detection scenario was discussed in [5]. I developed a common system model and scenario for this work that serves as a foundation for the later proposed methods and concepts. Notably, I modelled the event processing concepts such as an event, operator graph, operator state, and query, the deployment infrastructure such as fog-cloud, nodes, and quality requirements for the CEP system. Moreover, the

specialised modelling of concepts related to the contributions like transitions, unified communication and queue models are presented in the respective chapters. Parts of the system model corresponding to the respective problems of *adaptivity*, *efficiency*, and *interoperability* were previously published in [3, 4, 5], respectively. Prof. Koldehofe reviewed and gave feedback to improve the system model discussed in this work.

Chapter 4, *Mechanism Transitions in Operator Placement* proposes a solution to the *adaptivity* problem based on mechanism transitions methodology. The original idea related to mechanism transitions originated from the MAKI project[1] headed by Prof. Steinmetz, where numerous researchers investigate the transition methodology in the context of communication systems. I conceptualized a programming model, named TCEP, that enables specification of adaptable CEP mechanisms. Moreover, I conceptualized the methods for cost-efficient transitions between Operator Placement mechanisms and an adaptive and performance-based selection of mechanisms for transitions. Employing transitions, TCEP can fulfill the conflicting quality of service requirements of consumer queries. Towards the contribution to this chapter, I identified the scenarios in the context of the Internet of Things (IoT) applications such as traffic control and smart health monitoring that could benefit from the concept of transitions in between mechanisms of CEP systems. Moreover, I identified and analyzed mechanisms in CEP systems, particularly Operator Placement, Elasticity and Reliability mechanisms, as suitable candidates for transitions. These findings got published in [12, 8]. Prof. Koldehofe reviewed and gave feedback to improve the scenario and the written manuscript. The publication got contribution from the B.Sc. thesis of Niels Danger [2] in terms of scenario description, which was motivated and supervised by the author of this dissertation.

In a subsequent publication [2], I addressed the research problem of *adaptivity* for the Operator Placement mechanism by employing cost-efficient transitions. In particular, I modelled the system and proposed the design, implementation, and evaluation of transition execution algorithms that enables cost-efficient and seamless transitions while maintaining the correctness of the results. Furthermore, I proposed a learning-based selection of mechanisms for a transition. Prof. Koldehofe described the initial idea in the subproject C2 of MAKI[1], contributed to the formal definition of transition, its costs, and the written manuscripts. This work was conducted in collaboration with Prof. Dr. Guido Salvaneschi (former PI in subproject C2 of MAKI and currently Assistant Professor at the University of St. Gallen) and his Ph.D. student Pascal Weisenburger from the Software Technology Group in the Technical University of Darmstadt (Research Associate in the subproject C2 of MAKI). Prof. Salvaneschi and Pascal developed a domain-specific language for CEP systems in [7] that allows the specification of the quality of service requirements for the Operator Placement mechanism. TCEP uses this

specification language to signify changes in the quality of service requirements to execute cost-efficient transitions that fulfill the defined quality requirements. I contributed to the design, development, and evaluation of TCEP as well as the Operator Placement mechanisms in TCEP in order to improve the language and its semantics based on the required mechanisms in the TCEP system. TCEP received contributions from the M.Sc. thesis of Raheel Arif [3] that led to the implementation of a Docker-based virtualisation and naive transition strategies for comparison, and a lab project of Sebastian Hennig [1] that led to the extension of the work with GENI and CloudLab infrastructure, which was motivated and supervised by the author of this dissertation. All the co-authors of the aforementioned works [2, 7] helped in the review of the manuscript. I published the overall idea for transitions in the CEP mechanisms established in this doctoral thesis in a doctoral symposium in [10]. Follow-up work on TCEP is accepted and publlished in [1]. In this article, I developed a programming model that led to an analysis of *six* distinct Operator Placement mechanisms and a cost-efficient selection of Operator Placement mechanism based on the observed performance at runtime. I worked on the programming model used for the analysis [9], cost analysis of the proposed algorithms, and the extensive evaluation of the proposed algorithms by utilizing the deployment model based on the distributed resources. Niels Danger helped in the implementation, testing, and bug fixing of the mechanisms as a student assistant. Furthermore, Prof. Koldehofe, Prof. Dr. Ioannis Stavrakakis, and anonymous reviewers (single-blind revision process) helped in the review of the written manuscript [1].

Chapter 5, *Network-centric Query Execution* introduces network-centric processing of CEP operators and solves core research issues related to realising network-centric CEP on the edge networking architectures of Information-Centric Networking (ICN). The original idea of leveraging ICNs for CEP resulted from a discussion with The An-Binh Nguyen after a collaboration [20]. As a result, I performed a literature review on the currently available ICN architectures and found that none of the existing traditional ICN architectures look into the problem of continuous query execution for streaming applications. Subsequently, I supervised the M.Sc. thesis of Ali Haider Rizvi [4], where the student performed a further analysis of existing work and implemented a proof of concept using an existing ICN architecture called Named Function Networking with simple CEP functionalities such as filters and joins. I modelled and conceptualized a unified communication model, INETCEP, towards the following contributions: *(i)* unified communication mechanisms to enable continuous data stream processing, *(ii)* query execution algorithms to resolve CEP queries, and *(iii)* deploy queries over the ICN substrate. Moreover, two IoT benchmarks based on the DEBS Grand Challenge 2014 and a disaster field test were established, which was partially supported by the lab project of Pfannemüller et al. [2]. Simultaneously, I collaborated on INETCEP with Jonas Höchst and Patrick Lampe (Research Associates in MAKI

subproject A3 and C5). Subsequently, we published INETCEP [4] in collaboration with Prof. Dr. Bernd Freisleben (PI in MAKI subprojects A3 and C5 and Professor at the Philipps-Universität Marburg), where Jonas and Patrick helped in improving the software to be generally applied to distinct topologies. Prof. Koldehofe and Prof. Freisleben reviewed the written manuscript. In subsequent work, I increased the efficiency of INETCEP by providing a flow control algorithm, application of the unified communication model on the data plane and improved its wide applicability for different scenarios in a heterogeneous cloud setup. This work is under submission [3]. The manuscript received support from Johannes Pfannemüller, who implemented the concept under my supervision during his M.Sc. thesis [5], Jonas Höchst, who worked on the implementation of the evaluation setup using the CORE emulator, and Artur Sterz, who reviewed the implementation at the kernel level. Prof. Koldehofe and Dr.-Ing Rhaban Hark supervised the work and improved the written manuscript. All co-authors contributed to the review of the written manuscript [3].

Chapter 6, *Unified Serverless Middleware for Query Execution* presents a middleware called CEPLESS that decouples the specification language of a CEP system from the runtime by building on the serverless computing principles. With the help of the seminar work of Sebastian Hennig [3], I formulated the challenges that hinder a CEP system from being integrated into a serverless platform. I worked on the system model, the problem statement, and the proposed initial solution in a poster publication [13]. The publication received contributions from the B.Sc. work of Sebastian Hennig [1], where the student implemented the CEP middleware based on the serverless principles and evaluated it for TCEP (cf. Chapter 5) and Flink, a widely used CEP system, which was motivated and supervised by the author of this dissertation. I developed the concept and formalized the solution with a queuing model, system- and user-defined operators, and improved the system design. Moreover, I performed an extensive evaluation of the proposed middleware concepts and mechanisms. The work resulted in a publication [5] in collaboration with Prof. Dr. Lin Wang (PI in subproject C7 of MAKI) and his Ph.D. student Kamran Razavi. Prof. Koldehofe supervised this work, and all the co-authors contributed to the review of the written manuscript.

# Content


1 Introduction and Motivation **1**
- 1.1 Research Challenges . . . 4
- 1.2 Research Questions and Contributions . . . 6
- 1.3 Structure of the Thesis . . . 11

2 Fundamentals and State-of-the-Art **13**
- 2.1 Event-Based Systems . . . 14
  - 2.1.1 Stream Processing . . . 15
  - 2.1.2 Complex Event Processing . . . 16
  - 2.1.3 Discussion . . . 23
- 2.2 In-Network Processing . . . 25
  - 2.2.1 The Push and Pull Dilemma . . . 26
  - 2.2.2 Content-Centric Networking . . . 27
  - 2.2.3 Discussion . . . 30
- 2.3 Deployment Infrastructures . . . 31
  - 2.3.1 Cloud Computing . . . 31
  - 2.3.2 Serverless Computing . . . 34
  - 2.3.3 Discussion . . . 39
- 2.4 IoT and its Applications in Event Processing . . . 43
- 2.5 Summary . . . 44

3 Scenario and System Architecture **45**
- 3.1 Scenario Description . . . 45
- 3.2 System Architecture . . . 49
  - 3.2.1 System Model . . . 50
  - 3.2.2 Architecture and Contributions Overview . . . 59

4 Mechanism Transitions in Operator Placement **61**
- 4.1 Analysis of Adaptivity in OP Mechanisms . . . 64
- 4.2 Transition Problem Formulation . . . 67
  - 4.2.1 Extended System Model . . . 67
  - 4.2.2 Transition Problem Statement . . . 69
- 4.3 The TCEP System Design . . . 71
  - 4.3.1 Placement Performance Evaluator . . . 74
  - 4.3.2 Transition Engine . . . 79
  - 4.3.3 TCEP Programming Model . . . 88

4.4 Evaluation . . . . . 93
4.4.1 Evaluation Environment and Methodology . . . . . 94
4.4.2 Operator Placement Mechanisms . . . . . 98
4.4.3 Evaluation of Mechanism Transitions . . . . . 104
4.4.4 Evaluation of Transition Execution Strategies . . . . . 105
4.4.5 Learning Costs of Placement Selection . . . . . 111
4.5 Summary . . . . . 112

5 Network-centric Query Execution **113**
5.1 Analysis of Efficiency in Network-centric Query Execution . . . . 116
5.1.1 Extended System Model . . . . . 116
5.1.2 Problem Space . . . . . 117
5.2 The INetCEP Architecture . . . . . 121
5.2.1 Core Ideas . . . . . 122
5.2.2 Unified Communication Model . . . . . 124
5.2.3 CEP Query Engine . . . . . 134
5.3 Evaluation . . . . . 139
5.3.1 Evaluation Environment . . . . . 139
5.3.2 Evaluation of Unified Communication . . . . . 144
5.3.3 Evaluation of INetCEP Query Engine . . . . . 148
5.4 Summary . . . . . 154

6 Unified Serverless CEP Middleware **155**
6.1 Analysis of Flexibility in Operator Specification . . . . . 158
6.1.1 Extended System Model . . . . . 158
6.1.2 Problem Statement . . . . . 161
6.2 The CEPless System Design . . . . . 162
6.2.1 Serverless Middleware . . . . . 164
6.2.2 Programming Interfaces . . . . . 168
6.3 Evaluation . . . . . 171
6.3.1 Evaluation Environment and Methodology . . . . . 172
6.3.2 Evaluation of Dynamic Operator Update . . . . . 174
6.3.3 Evaluation of CEPless Middleware . . . . . 175
6.3.4 Runtime and Language Independence . . . . . 180
6.4 Summary . . . . . 182

7 Conclusion and Outlook **185**
7.1 Contributions Revisited . . . . . 186
7.2 Key Results . . . . . 187
7.3 Future Outlook . . . . . 189

Bibliography **191**

A Appendix **209**
A.1 Supplementary Material to Chapter 4 . . . . . 209
A.1.1 Additional Insights on the Performance Evaluation . . . . 209
A.1.2 Illustration of TCEP System . . . . . 211
A.1.3 Selection Method for OP mechanism . . . . . 213
A.2 Supplementary Material to Chapter 5 . . . . . 215
A.2.1 Query Grammar . . . . . 215
A.2.2 Extensibility . . . . . 215
A.2.3 Additional Insights on the Performance Evaluation . . . . 217
A.3 Supplementary Material to Chapter 6 . . . . . 221
A.3.1 Additional Insights on the Performance Evaluation . . . . 221
A.3.2 Programming Interface of CEPLESS . . . . . 223
A.4 Supervised Student Theses . . . . . 225
A.5 Supervised Student Labs and Seminars . . . . . 225

B Erklärung laut Promotionsordnung **227**

# 1

# Introduction and Motivation

> "*The whole of science is nothing more than a refinement of everyday thinking.*"
>
> – Albert Einstein

*The world is relying on digitalization using IoT during COVID...*

As per Satya Nadella, CEO of Microsoft Corporation, "two years of digital transformation occurred in just two months since the pandemic started"[2]. A continuous acceleration towards digital transformation has become the need of the hour due to the COVID-19 pandemic. The omnipresence of the Internet of Things (IoT) plays a crucial role in this digital transformation, known to enable the interconnection of heterogeneous devices [1]. Some examples of such digital transformation using the IoT are in healthcare, such as remote patient monitoring, vaccine cold chain monitoring, and delivery using healthcare drones [2]. Furthermore, IoT in healthcare is being investigated to monitor symptoms and avoid spreading COVID-19 [3]. Besides healthcare, other industries like e-commerce, finance, and manufacturing also continue to sustain because of the digital transformation using IoT. Even prior to the pandemic, digitalization using IoT continued to emerge in enabling a broad spectrum of applications across multiple areas, including smart cities, smart industry, smart healthcare, to name a few [4, 1]. Cisco predicted a revenue estimation of a hundred billion dollars to be generated alone by the smart city market until 2025 [5].

*... and before*

Complex Event Processing (CEP) systems [7, 8] are widely used to deliver the information required by continuously evolving IoT applications (cf. Figure 1). It is currently used in many companies, including Apache Storm at Twitter [9][3] used for real-time analytics of tweets, Millwheel at Google [10] used to power the search engine, and Rabobank, one of the three largest banks in the Netherlands use Kafka Streaming [11] for fraud alerts related to financial events. Despite its promises, a complete digital transformation

[2] 2 years of digital transformation in 2 months. Article by Jared Spataro, Corporate Vice President for Microsoft 365. https://www.microsoft.com/en-us/microsoft-365/blog/2020/04/30/2-years-digital-transformation-2-months/ [Accessed in May 2021].

[3] More recently known as Heron [9].

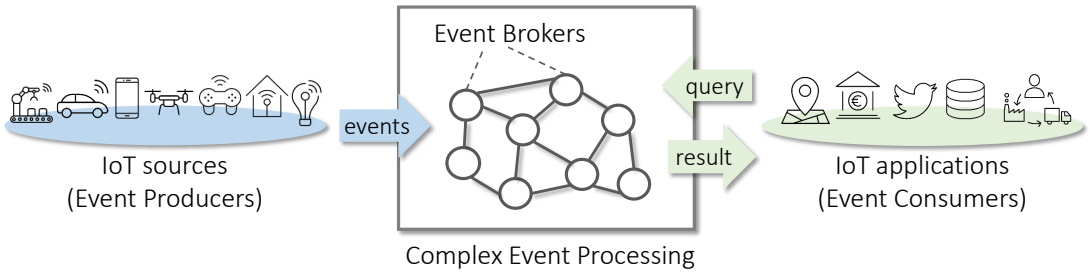



Figure 1: High-level overview of Complex Event Processing (adapted from [6]). It consumes low-level events from the IoT sources, named producers, to deliver events of interest to the IoT applications, named consumers, using resources for computation, named brokers.

using existing CEP systems is a far-off vision due to numerous challenges as follows.

*The adaptivity problem*

1. *Adaptivity:* IoT applications are inherently exposed to the dynamics of the surrounding environment that impacts the performance of the applications, for instance, mobility of smart vehicles and varying workload of incoming sensor data. CEP systems need to continuously adapt their mechanisms to the dynamics of the environment and the Quality of Service (QoS) requirements of the applications. For instance, a CEP system used in the financial context has to timely deliver a fraud event to the credit card institution and react to it by blocking the card. While it is important to react to fraud in a timely manner, during high load, it becomes crucial to react in a reliable manner [12]. Such changes in the environmental conditions and the QoS requirements are hard to be met at runtime.

*The efficiency problem*

2. *Efficiency:* Often, IoT applications have to deliver information with very low latency, for instance, a fraud event has to be detected not in seconds or milliseconds, but in order of a few microseconds. The difference between milliseconds and microseconds in such applications could lead to big profits or losses to the financial institutions [13]. The demands for such latency-critical applications are low latency, high dynamics of the environment, and substantial network load, which are hard to be met using existing CEP mechanisms.

*The interoperability problem*

3. *Interoperability:* Although many CEP programming models exist, often looking into disparate problems, there is no means to benefit or reuse solutions across the programming models. This is problematic as the demands of current IoT applications is extremely diverse and, hence, might need more than one CEP execution environment to work together. Moreover, due to this inflexibility, it is currently not possible to exchange query at runtime that is often a requirement of IoT applications. For

instance, fraud detection application needs to continuously update machine learning methods to deal with newly observed patterns of fraud. Therefore, the use of a single CEP execution environment is not sufficient to meet the needs of a broad spectrum of IoT applications.

*CEP aims towards a potential solution...*

An essential aspect in an IoT application is detecting situational changes in the environment and trigger actions to deal with those changes within the deadline. CEP is well-suited to model this behavior as it encapsulates the happening of such a situational change using the notion of an *event* as seen in Figure 1. IoT applications interact with a CEP system to transform low-level events that correspond to raw data from sources such as sensors into high-level events or *complex events* that represent situational changes [14]. In a CEP system, complex events are expressed using a continuous query which is processed in a distributed manner by so-called *Operator Placement* (OP) mechanisms. Typically, queries specify certain QoS requirements that have to be fulfilled by an OP mechanism in a CEP system. For instance, a fraud event is a complex event that has to meet the latency requirements of the credit card institution in order to block the card.

*...however, it is extremely challenging*

Despite many advancements, current work on CEP has significant limitations in its support for the *adaptivity*, *efficiency*, and *interoperability* problems. First, concerning the *adaptivity* problem, existing CEP systems use a *single* OP mechanism, which restricts their ability to fulfill conflicting and changing QoS requirements at runtime. Some CEP systems employ self-adaptation of OP mechanisms [15, 16, 17, 18] to deal with the dynamics in the environment. Other authors use a single OP mechanism to encapsulate distinct QoS requirements [19, 20]. Still, these approaches are either unable to fulfill conflicting QoS requirements of applications or unable to specify and meet changes in the QoS requirements at runtime. Second, concerning the *efficiency* problem, classical CEP systems are limited in terms of deployment infrastructure to commodity hardware, which resolves a query by forming an overlay network. This imposes high inefficiency in terms of meeting challenging QoS requirements of IoT applications. Finally, concerning the *interoperability* problem, though a plethora of highly diverse and complex CEP programming models [21, 22, 23, 24, 9, 25, 26, 27, 28] have been proposed, hardly any of them provide reusability. This makes it difficult for the application developers to realize IoT applications since they have to combine multiple CEP programming models, which is very complex. Recent work on benchmarking CEP systems [29] and simulating OP mechanisms [30] unifies a couple of CEP systems. However, these approaches lack the reusability of mechanisms across CEP programming models and do not support changes in the query at runtime.

In essence, the existing CEP systems exhibit crucial shortcomings with respect to the three aforementioned problems. Therefore, this thesis proposes

concepts and algorithms to realize an *adaptive*, *efficient*, and *flexible* architecture to solve the challenges encountered by IoT applications in the face of an extremely dynamic and heterogeneous environment. In particular, we propose *(i)* a programming model for transitions between CEP mechanisms, *(ii)* a communication model and algorithms for efficient network-centric execution of operator graphs, and *(iii)* a middleware to allow interoperability between diverse CEP systems. In the following, first, Section 1.1 details the three research problems. Second, Section 1.2 formulates the research questions addressed by this thesis, followed by the contributions. Finally, Section 1.3 presents an outline of this thesis document.

## 1.1 Research Challenges

Based on the previous motivation, we elaborate on the three core problems in the context of CEP systems that we solve in this thesis.

### *The adaptivity problem*

*Research Challenge 1: Dynamic environmental conditions*

A CEP system is inherently exposed to a multitude of dynamics in the environment of the IoT application, namely *(i)* the characteristics of the data stream such as event rate, *(ii)* the characteristics of the communication network such as latency and bandwidth, and *(iii)* the underlying environmental conditions of the IoT scenario such as mobility of vehicles, may drastically vary. As a result, the QoS requirements of an application change significantly during the lifespan of a continuous query. For instance, in the fraud detection application, the QoS requirement depends on the transaction workload. To better understand the requirements, let us consider an example as follows. During Alibaba's[4] Double 11 Online Shopping Festival (similar to Black Friday Day in the US), the peak workload reaches 6,000,000 transactions per second, in contrast to a normal day workload of a few thousand transactions [31]. While it is important to react to a fraud event in a timely manner under both environmental conditions, under large workloads, it becomes crucial to detect a fraud additionally in a highly reliable manner. Similar changes in the QoS requirements are also noted in the application of autonomous cars, where dynamics in the environment are prominent. Thus, in the face of dynamic changes in the environment and the changing QoS requirements like above, a placement that was efficient to process a continuous query before the change may become very inefficient all of a sudden. It is therefore crucial for a CEP system to provide adaptivity in the OP mechanism. Here,

[4] A leading Chinese multinational technology company specializing in e-commerce `https://www.alibaba.com/` [Accessed in May 2021].

*adaptivity* stands for how well a CEP system deals with the dynamics of the environmental conditions.

Existing CEP systems have dealt with the dynamics in either the stream or communication network characteristics by self-adaptation of an OP mechanism [15, 32, 33, 18], or by explicitly scaling the resources and migrating the operators between in-network resources [16, 24, 27, 34]. Some adaptive approaches even embed distinct QoS requirements in a multi-objective optimization function to find a Pareto-optimal solution to the OP problem [19, 20]. However, the CEP system should be prepared to change the OP mechanism for a continuous query, depending on the change of QoS requirements at runtime when the QoS requirements are not known beforehand, which none of the above approaches does. Without the adaptivity of OP mechanisms, query performance in the CEP system may drop drastically in some situations when the CEP system is not able to fulfill the changed QoS requirement.

*The efficiency problem*

*Research Challenge 2: Efficiency requirements of IoT*

The second challenge is to meet the efficiency requirements of IoT applications in the face of dynamic environmental conditions. For instance, a fraud detection application has to react by blocking a card very quickly [13]. Efficiency in such applications is of extreme importance since a late reaction to an can lead to huge monetary loss– to a bank in the fraud detection example. Similarly, other applications may require efficiency in terms of different QoS requirements, such as bandwidth, throughput, availability, etc.

Early research on CEP systems has focused on resolving queries in an overlay network on top of commodity hardware or cloud-based infrastructure using an OP mechanism [23, 15, 24, 35, 36, 28, 9]. The focus of these works has been to optimize for a single or a combination of different metrics using an OP mechanism such as latency [37, 15, 27, 38], bandwidth [39, 40], throughput [41, 42], load [43, 44], availability [19], and trust [45] in the execution of queries. However, these works rely on geo-distributed cloud infrastructures that cannot meet the efficiency requirements of many latency-critical applications like fraud detection. Some of the approaches have been proposed using fog-computing infrastructures that offer low latency by processing queries near the end-user [46, 27, 47, 48]. A few approaches have leveraged hardware acceleration using FPGAs [49], using parallel hardware in GPUs [50], and in-network switches [51] for the execution of queries. However, none of them has addressed the ability to process continuous streams as part of the networking architecture while efficiently performing in-network query execution. Without efficiency in performing in-network execution of operators, the CEP system is not able to meet the challenging QoS requirements leading to

dire consequences. Here, *efficiency* is referring to the query performance in the above example in terms of latency to retrieve a complex event.

*The interoperability problem*

*Research Challenge 3: Lack of reusability*

A plethora of CEP programming models exist, each offering numerous standard functionalities such as operator abstractions, yet in different programming languages and execution environments. These CEP models can benefit from each other heavily in terms of operator abstractions and OP mechanisms, yet there is no means to reuse functionalities across CEP systems. This is because the CEP specification language of the query and the runtime environment responsible for executing the event detection logic are highly interdependent. This poses many limitations for building IoT applications using CEP, detailed as follows. *(i)* Application developers need to have extensive knowledge of the runtime system, which is relatively complex. *(ii)* Dependence on the specification language requires the IoT applications to be rewritten. *(iii)* Due to the interdependence, it is difficult to update the queries dynamically. *(iv)* Although current CEP models have plenty of similarities, operators lack reusability across CEP models.

Despite the fact that a multitude of CEP programming models have been proposed, many times focused on providing common functionalities both in academia [21, 22, 23, 24, 9, 25, 26, 27, 28] and industry [35, 36, 52, 28, 9], there has been only a small amount of effort in reusing their functionalities or enabling cross-compatibility. Furthermore, attempts to benchmark CEP systems [29] and CEP mechanisms in simulation [30] unifies a couple of CEP systems. Nevertheless, they do not allow reusability of mechanisms across those CEP systems. More recently, Apache Beam [53], proposed by Google, aims to develop a unified programming model above multiple CEP runtime engines similar to our work. However, in all the aforementioned approaches, operators are bound to the respective runtime environments, and therefore no reuse or exchange is possible. In summary, these CEP systems fail to fulfill the flexibility requirements of IoT applications that require multiple systems to interact with each other. Here, *interoperability* refers to the ability to reuse functionalities of CEP systems across each other, and flexibility refers to the ability to use these distinct models.

## 1.2 Research Questions and Contributions

The ultimate goal of this thesis is to provide models and methods for network-centric complex event processing to enable *adaptive*, *efficient*, and *flexible* operator placement and execution of continuous queries. In Figure 2, we

illustrate the contributions of this thesis using a three-tier architecture comprising the underlay representing the deployment infrastructure; a middleware that provides a common platform to execute on diverse CEP systems available in the execution layer and to execute transitions in operator graphs over the underlay; and the execution tier illustrating the proposed and existing CEP systems.

*Overall Contributions and RQs*

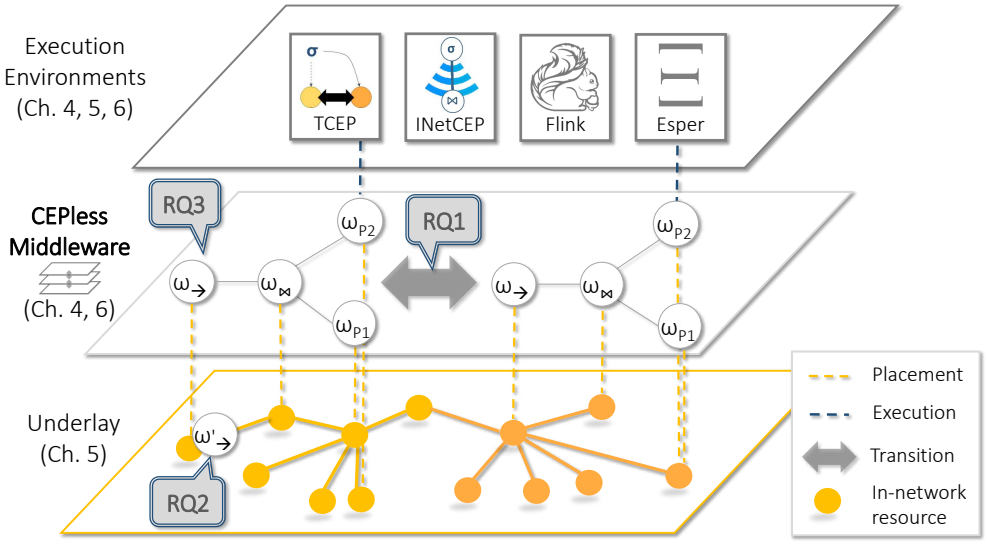



Figure 2: Contributions and research questions (RQ) of this thesis shown in a three-tier architecture comprising the underlay, the middleware, and the execution tier. Here, the underlay represents the resources involved for processing queries such as a fog-cloud infrastructure [48]. The middleware shows the overlay network comprising operator graph for placement that aid in transitions and interoperability of execution environment using the proposed CEPless system (cf. Chapter 6). Last, the execution tier comprise the proposed Tcep (cf. Chapter 4), INetCEP (cf. Chapter 5), and widely used CEP systems such as Flink [54] as well as Esper [35].

In the foremost research question, we solve the *adaptivity* problem in two steps. *(i)* We propose the *first programming model* to develop OP mechanisms that deploy continuous queries in a heterogeneous deployment infrastructure. *(ii)* We propose *methods for transitions in CEP mechanisms* with Tcep, the first Transition-capable CEP system that enables cost-efficient transitions between the OP mechanisms to meet the changing QoS requirements of IoT applications (as seen in the middleware). In the second research question, we solve the *efficiency* problem by proposing *efficient network-centric* execution of CEP queries in the underlay using a unified communication model of INetCEP. Finally, in the third research question, we propose CEPless, the *first CEP middleware* that is *serverless*, which *(i)* provides common pro-

gramming interfaces for developing CEP operators, which can be executed on any CEP execution environment of choice and *(ii)* allows reuse of CEP mechanisms across a wide range of CEP systems (as seen in the execution tier and the middleware in the figure) to address the *interoperability* problem. In the following, we list the research questions and the contributions towards them.

***Research Question 1:*** *How to specify and adapt between OP mechanisms in the face of dynamic environmental conditions and QoS requirements?*

*Solution to RQ1: mechanism transitions*

To provide adaptivity in a CEP system, we propose *transitions* in the OP mechanism using TCEP, a *Transition-capable CEP* system that facilitates the dynamic change of OP mechanisms. The TCEP system can meet the QoS requirements of a continuous query in the face of dynamic changes in the environment using transitions. Introducing transitions in a seamless and non-disruptive manner, i.e., without any interruption in delivering the output of the continuous query, is a highly challenging task and requires a careful choice of system mechanism. Another critical issue is that a transition can be costly in terms of resources it consumes – to change an OP mechanism – the operator state may have to be migrated for stateful operators. Naively approaching the problem will lead to low performance in terms of *(i)* time and overhead taken for state transfer of operators, *(ii)* incorrect query output, and *(iii)* interruption in the delivery of the query output. These problems eventually lead to a failure in terms of fulfillment of QoS requirements of the query, which accounts for a major research challenge. In our contributions, we target these problems as follows.

1. We formalize the problem of transitions for an Operator Placement in the context of a CEP system, considering distinct QoS requirements of applications, and present the definition of the cost that needs to be considered in performing transitions between the OP mechanisms.

2. We present a programming model that supports the development of OP mechanisms with specific QoS requirements, which is used to support seamless transitions.

3. We propose and analyze a genetic learning method for adaptively planning transitions between OP mechanisms to meet dynamically changing QoS requirements and changes in environmental conditions.

4. We propose and analyze two transition strategies to facilitate a dynamic change of OP mechanisms in a non-disruptive and seamless manner while maintaining the correctness of the results.

5. Finally, we present an open-source implementation of TCEP[5]. Moreover, we provide an extensive evaluation to analyze the performance of *six OP mechanisms* using distinct queries and analyze the performance of transitions and the learning algorithm on the distributed set of fog-cloud infrastructure, including GENI [55], CloudLab [56], and MAKI [57] resources. We show using a realistic workload comprising multiple traffic congestion detection queries shows that seamless transitions can be realized in a CEP system in the range of $850$ms - $2$ seconds and as low as $0.5$ ms for a single operator. Simultaneously, minimum costs are maintained for transitions in terms of time and overhead – $5\times$ better than a widely used migration strategy – while maintaining $100$% throughput in the delivery of complex events.

***Research Question 2:*** *How to increase the efficiency in executing queries using in-network architectures at the edge?*

Edge infrastructure provides a means to process operator graphs by reducing the time taken to send and receive the data, thereby reducing the overall latency in the delivery of events. Edge devices are typically in near proximity to the producers and the consumers of the data. In this work, the hierarchical infrastructure of edge and fog-clouds plays a dominant role in fulfilling QoS requirements for the consumers of IoT applications. Novel edge network architectures such as ICN provide promising concepts for network-centric event processing in the data plane, which can possibly address the efficiency problems for demanding IoT applications [58, 59]. Yet, current ICN architectures pose strong limitations in their support to process continuous data streams as desired by current IoT applications.

These limitations are due to *(i)* missing communication strategies to continuously process periodic data streams; *(ii)* missing abstractions to represent operators on the data plane and process them in an efficient, robust, and accurate manner; and *(iii)* missing abstractions to deal with high input event rates as part of the networking architecture. Therefore, in our second contribution, we present a unified communication model, INETCEP, that enables network-centric execution of continuous queries in the data plane of ICN. To this end, we make the following contributions:

*Solution to RQ2: network-centric operator execution*

1. We propose a unified communication model that allows the processing of continuous data streams in the substrate of ICN. As part of the unified communication, we provide a rate-based flow control mechanism that mitigates the issue of flooded links and ensures flow balance.

[5]Webpage of TCEP, including the programming model and cost-efficient transitions `https://luthramanisha.github.io/TCEP/` [Accessed in May 2021].

2. We introduce a meta query language that is able to express complex event queries on the data plane over the ICN substrate.

3. We propose network-centric query execution algorithms that efficiently performs event processing while maintaining zero event loss.

4. Finally, we provide an open-source implementation [60] of INETCEP[6] and its an evaluation using a widely used ICN architecture, namely Named Function Networking (NFN) [61] and CCN-lite [62]. Our evaluation study based on two IoT case studies and open datasets from the DEBS Grand Challenge 2014 [63] and a disaster communication scenario [64] shows that INETCEP maintains a very low latency in the forwarding of events of up to $73\mu$s under a very high workload of up to $50,000$ events per second, $15\times$ better than the CEP system Flink [54]. At the same time, it maintains $100\%$ throughput, and it is $32\times$ faster than Flink and over $100\times$ faster than the naive pull-based reference communication strategy in the delivery of complex events.

***Research Question 3:*** *How to achieve interoperability across different and diverse CEP programming models?*

So far, there is no substantial work on reusing mechanisms of multiple CEP programming models for different use cases of IoT applications. Furthermore, due to the tight dependency of the specification of an IoT use case with the runtime of existing CEP models, application developers face significant restrictions in developing novel applications. A possible solution to provide decoupling in the execution models of CEP is by using *Serverless Computing*. Serverless computing or Function-As-A-Service (FaaS) paradigm is being increasingly adopted by the cloud providers to limit developer access to the cloud resources to shift their focus towards developing applications rather than worrying about deployment decisions. CEP systems can benefit from the principles of serverless computing to address the interoperability problem. However, integrating CEP systems on the FaaS model is not possible out of the box because it lacks many abstractions for building an event-based system like CEP. While current CEP programming models have significant limitations in terms of interoperability because of the tight dependencies in terms of execution. This dependency further complicates the integration of diverse CEP systems. Hence, in our third contribution, we propose a middleware called CEPLESS, which allows, on the one hand, to specify operators without any knowledge of the underlying runtime system. On the other hand,

*Solution to RQ3: serverless middleware*

[6] Webpage of INETCEP including all the contributions stated here. `https://luthramanisha.github.io/inetcep/` [Accessed in May 2021].

multiple CEP runtimes can be reused for deploying and processing operator graphs. In the following, we enumerate the specific contributions of CEPless:

1. We propose in-memory queue management and batching mechanisms to enable stateful processing and ensure correctness and fast delivery of events, extremely important requirements for CEP systems.

2. We define a unified programming interface that enables the specification of novel user-defined operators that work independently of the runtime system and the specification language. Most importantly, using this interface, the operators can be specified in any existing programming language and be deployed on any execution environment.

3. We introduce a simple user-defined operator interface that integrates highly diverse CEP runtime systems into the CEPless system.

4. We present an open-source implementation and evaluation of CEPless[7] on widely used CEP system *Apache Flink* [28] and TCEP [25] (cf. Chapter 4) using a real-world credit card transaction dataset [65]. Our evaluation for Apache Flink and TCEP shows that CEPless can easily integrate multiple existing CEP systems while attaining similar throughput under a high workload of up to 100,000 events per second. Simultaneously, operator graphs can be dynamically updated in a mean time of 238 ms for the example fraud detection query.

In summary, Figure 2 provides an overview of the aforementioned contributions of this thesis altogether in developing an adaptive, efficient, and flexible placement and execution of queries for CEP systems. The contributions that are not reflected in this thesis are a network-centric publish-subscribe framework [66], introducing adaptivity using Context-Feature Models in middleware and its demonstration [67, 68], an adaptable middleware solution for a smart city, which is developed within the industry [69], object classification in autonomous vehicles using point clouds [70], adapting video quality assessment algorithms using multiple sensors in smartphones [71], and analysis of IoT frameworks in terms of security and privacy [72]. Some publications are under submission that proposes an auto-scaling framework for deep learning [73].

## 1.3 Structure of the Thesis

The thesis is structured into six further chapter and appendices. Chapter 2 presents the preliminaries and related work analysis for the contributions of

[7] Webpage of CEPless Github including all the contributions stated here `https://luthramanisha.github.io/CEPless` [Accessed in May 2021].

this thesis. Chapter 3 provides a common scenario, system model, and an overview of the contributions. Chapter 4 presents the methods for transition and its related concepts towards the adaptivity problem. Chapter 5 presents a unified communication model and algorithms to facilitate network-centric query execution targeted towards the efficiency problem. Chapter 6 presents a middleware that provides unified programming interfaces to represent a wide range of IoT applications and abstract the complexity of CEP systems from application developers. Finally, Chapter 7 provides a summary of the contributions of this work and highlights future directions.

# 2

# Fundamentals and State-of-the-Art

In this chapter, we provide background information and related work on the concepts and paradigms referred in this thesis. As shown in Figure 3, this chapter comprises two parts focused on push- and pull-based communication mechanisms. Firstly, we focus on Event-based System which relies on push-based communication mechanisms. Specifically, we detail Stream Processing and Complex Event Processing systems as the two main aspects relevant for this thesis. Based on this foundation, we present an overview of related approaches on Operator Placement focused on the *adaptivity problem* and In-Network Processing focused on the *efficiency problem* as earlier discussed in Chapter 1. Secondly, we discuss related concepts on deployment infrastructures such as Cloud Computing, Fog Computing, and Serverless Computing, and we provide an overview of related work on Complex Event Processing on Serverless platforms focused on the *interoperability problem*. Afterwards, we provide a brief background into IoT and its applications using the event processing paradigm.

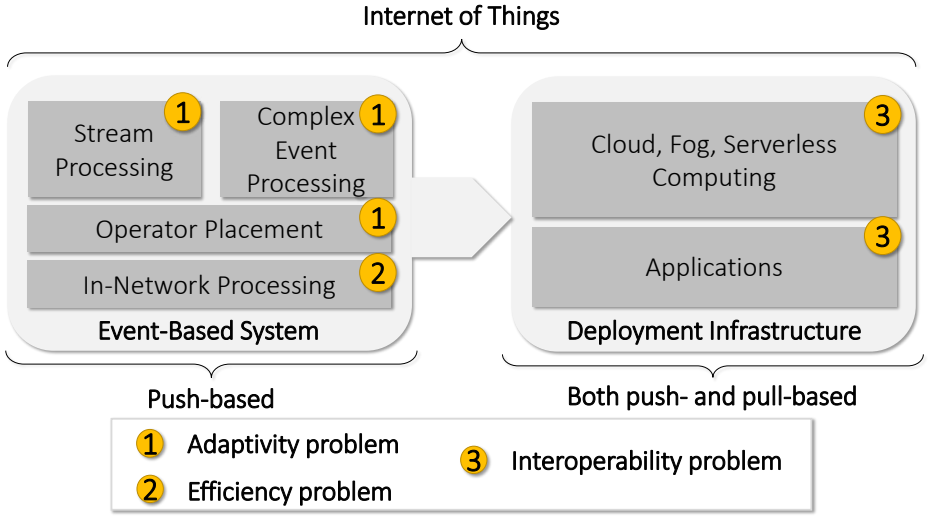



Figure 3: Chapter structure: background and related work based on the core problems addressed in this thesis.

## 2.1 Event-Based Systems

An *event* is defined as an occurrence of a situation that happened in the real world or in a particular system. An event can be a low-level event or a complex event also often referred as a higher-level event. A low level event is the one which directly arrives from the sources such as sensors. A complex event can be a result of a combination or a derivation based on two or more *(i)* low-level events, *(ii)* low-level events with complex events, or *(iii)* complex events [74]. For instance, a combination can be an aggregation of events such as min, max, etc.

To better understand the definitions, let us consider an example of a financial institution that wants to detect frauds in credit cards. In this example, an event occurrence could be a credit card transaction event in the financial institution. Thus, the low level transaction events contribute into the generation of a higher level or complex event such as a *fraud event*. The *fraud event* is derived by performing an aggregate over multiple low level attributes in the transaction event, such as credit card terminal location, and card transaction amount. An event processing engine is responsible for filtering, aggregating, and combining events from *event producers*, such as credit card terminals. Those complex events are then notified to the fraud detector applications, which act as *event consumers*, such as financial institution. Given these fundamentals, Definition 1 summarizes an event-based system as used in this thesis.

**Definition 1.** *Event-Based System.*
A software system that observes events from event producers and causes reactions in the system to derive complex event notifications for the event consumers. It enables the consumers to act upon the derived complex events [74].

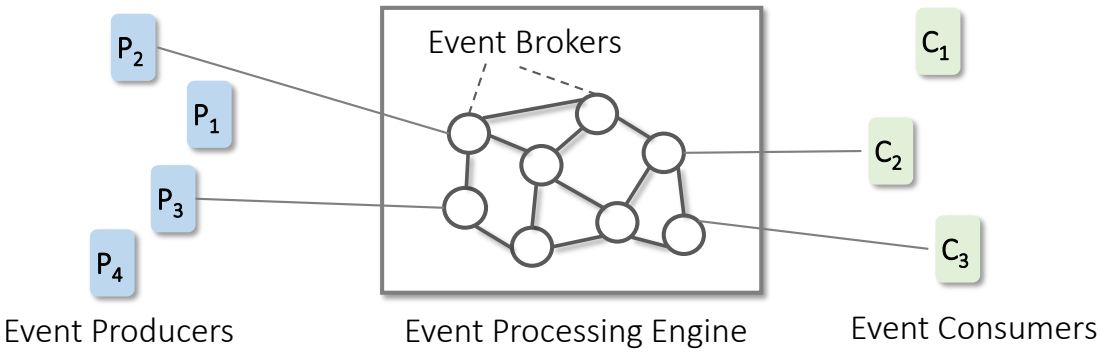



Figure 4: A high-level view of an event-based system (adapted from [6]).

An *Event-Based System* typically consists of three essentials components: a *monitoring component*, a *communication mechanism*, and a *reactive mechanism*. The *monitoring component* is responsible to observe, represent, and compose events. The *communication mechanism*, also commonly referred as *event-based communication*, handles the event notifications to the consumers. An event-based system's communication mechanism is typically push-based, which means consumers of the application continuously receive the notifications. A detailed analysis of such mechanisms is done in Section 2.2.1. Such communication mechanisms require a strict decoupling of producers and consumers. Thus, a notification service involving a single or a network of *event brokers* (as seen in the middle of the figure) is needed. The *event brokers* provide an execution environment for processing the events. The *event consumers* specify a complex event by means of a *query*. In a Distributed Event-Based System, the query is distributed among a set of event brokers that collaboratively processes it. Finally, the *reactive mechanism* of an event-based system expresses the query in the form of rules triggered by the corresponding events. Generally, the *event-condition-action* (ECA) rules are triggered when a corresponding event is raised (low level or complex). The condition represents the complex event detection logic, and if the condition is met, the action is executed. The event detection logic is defined by a specification language used to express the conditions for event detection [75].

Having discussed the background information on Event-Based Systems, we will now detail two primary classes of Event-Based Systems, Stream Processing and Complex Event Processing[8] in the next two subsections.

### 2.1.1 *Stream Processing*

Active database systems bring reactive behavior into the databases, allowing them to *continuously* respond independently to database-related events. Active databases are Event-Based Systems developed to overcome the challenges faced in the classical database systems that are passive in nature. The reactive behavior is described by ECA rules as described in Section 2.1. Yet, data in an active database is *(i)* persistent in nature, *(ii)* assumed to be persisted in order, and *(iii)* fixed in size and there are no time constraints. Due to these limitations, the active database community developed a new class of Event-Based Systems oriented towards processing large data streams in a timely manner, commonly referred to as Stream Processing Systems [6].

Stream Processing Systems processes events from unbounded data streams (that may arrive out of order) to derive query results within low latency. In

[8] *Data Stream Processing* and *Complex Event Processing* paradigms are used as synonyms in this thesis, with a focus on their similarities in consistent to related work [6].

Stream Processing, users or applications install continuous query in such systems that produce results until the query explicitly removed. However, detecting and notifying the complex event patterns that involves operators like sequence are usually out of the scope of these systems [6][9]. For this reason, CEP systems were introduced, as discussed in the next subsection.

### 2.1.2 *Complex Event Processing*

*CEP is a methodology to...*

Complex Event Processing (CEP) was first coined by David Lukham as "*a defined set of tools and techniques for analyzing and controlling complex events for modern distributed systems*" [7]. In the late 2000s, it emerged as a powerful paradigm that can detect complex patterns in the incoming data streams to derive higher level events for different applications. Since then, it is still used in multiple applications, starting from financial trading in stock market, fraud detection, supply chain management, location-based services, e-commerce, and more recently, in social networks like Twitter and smart cities.

*...detect and react to higher-level events*

To better understand the functionality of a CEP system, consider a fraud detection scenario for a financial institution, as seen in Figure 5. The financial institution collects events from a credit card terminal and analyzes them. An event tuple of this form is shown in Figure 5 with the attributes $< ts, card\_id, terminal\_id, card\_amount, terminal\_loc >$. Each event tuple carries a timestamp, and values of attributes such as a card identifier represent the credit card's identifier. Similar to an event-based system, the CEP system comprises of event producers, such as IoT devices that generate continuous events of the form above (as seen on the left-hand side in the figure). The event consumers specify interest in these incoming data streams using a query as seen in the figure, specified in EPL[10]. The query dictates a fraud event if two credit card transactions are performed at different locations; however, at almost the same time. Intuitively, the same person cannot be at different terminals simultaneously, hence, the transaction is detected as a fraud. A complex event with the transaction amount of `$450` is delivered to the event consumer or the fraud detection department of the bank encapsulated in an event notification as soon as this fraud is detected. The query is transformed into an intermediate representation of an operator graph by the CEP engine, where vertices represent the operators and the edges represent the flow of information.

[9] Modern stream processors like Apache Flink [28] and Storm [76] provide plugins for CEP, but these systems are still known as stream processing systems [6]. This thesis does not focus on the differences between these two classes, but rather similarities, such as both uses an operator placement mechanism for operator graph mapping.

[10] A specification language based on SQL used by Esper [35] for event processing.

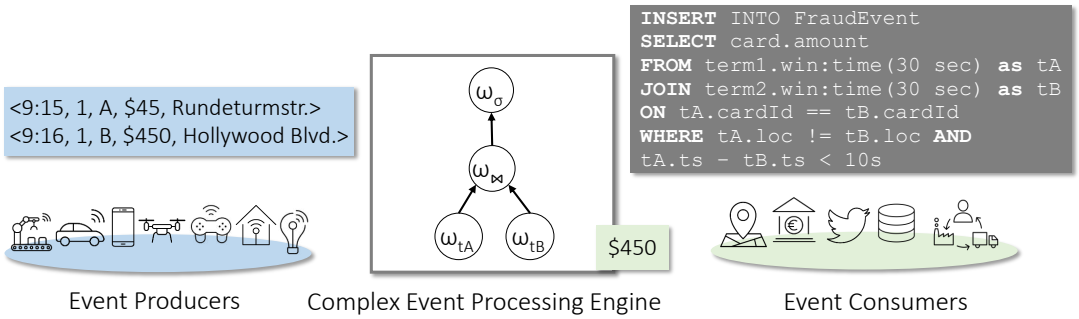



Figure 5: A fraud detection scenario using Complex Event Processing Engine.

Although Stream Processing and CEP Systems differ in multiple aspects including the ability to specify and detect patterns in data and implementation of operators, the underlying deployment model for both is the same [14]. Hence, the concepts defined in this thesis can be applied to both the classes of Event-Based Systems — each of them can process operator graphs in two ways: *centralized* or *distributed*. In a centralized system, an operator graph is processed on a single event broker. However, when the scale of incoming events is too high, or the number of queries supersedes the computational power of the central event broker, the processing of the operator graph is distributed among multiple event brokers as seen in Figure 4 (in the center). A so-called operator placement mechanism does the operator graph distribution on event brokers explained in the next subsection.

### *Operator Placement*

For better scalability, a CEP engine maps the operator graph on a set of event brokers – each running on different physical devices – which collaborate to process the events from multiple producers. Consequently, two problems arise as a result of this distribution: *(i)* Where to place the operators? *(ii)* How to perform this assignment, mainly how the processing devices interact and in what order in the operator graph, the assignment takes place? To date, many Operator Placement (OP) mechanisms have been proposed [15, 40, 77, 19], focusing on the assignment of operators on processing devices; however, the latter problem got only a little attention. In this work, we define OP mechanism in a broader way, as the following.

**Definition 2.** *Operator Placement mechanism.*
An Operator Placement mechanism defines the mapping of an operator graph on to a set of distributed event brokers. It dictates *(i)* where to map each operator, and *(ii)* how to perform the mapping.

An OP problem defines an optimization problem towards one or more QoS requirements such as latency, bandwidth, availability, etc. [14]. Typically, in a dynamic environment the CEP system has to replace operators to the event-brokers that can be highly resource consuming. In such scenarios, finding an optimal placement is desirable as it offers the best assignment, but it is highly inefficient in large scale and dynamic scenarios due to a very large search space and computational complexity [78, 42, 19, 79]. For instance, in wide-area networks like fog-cloud infrastructure as introduced in Chapter 1, it is expected to take a higher amount of time to find an optimal assignment, which is not desirable as it increases the time until the complex event is processed and retrieved. For this, multiple sub-optimal solutions or heuristics have been proposed [80, 77, 15]. Table 2 summarizes the OP mechanisms and adaptive Stream Processing and CEP Systems concerning the following characteristics.

1. *Runtime reconfiguration* is defined as the ability to self-adapt operator placement at runtime given the dynamic environment.

2. *Type* defines the kind of system for which the mechanism is proposed, where SPS stands for the Stream Processing System, and CEP stands for the Complex Event Processing system.

3. *Mechanism structure* of an OP mechanism is majorly divided into central and decentral mechanisms based on whether the placement information is shared with the coordinator globally (central) or locally (decentral), respectively.

   *Infrastructure* defines the resources on which the event-based system is deployed. The selection of the infrastructure highly influences the system's performance depending on the type of processing devices and the memory architecture of the infrastructure. It majorly comprises a single node, cluster, cloud, or fog infrastructures.

4. *QoS metrics* are the performance characteristics of a system in terms of delivery of complex events [81]. These are widely studied in the context of OP mechanisms [82] for different parameters of a CEP system. In the following, we present majorly studied metrics from the related work.

   *Latency* is defined as the overall delay in receiving the complex event at the event consumer since its generation at the event producer.

   *Bandwidth* refers to the amount of data transferred at a given time through the links between the nodes (in the infrastructure). In other words, bandwidth refers to the rate at which bits can be inserted into a medium [15].

| Year | Runtime reconfig. | Name, Ref. | Type | Mechanisms | | QoS metrics | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Structure | Infrastructure | Latency | Bandwidth | Input Queue Size | Throughput | CPU | Availability | Certainty | Trust and Privacy |
| **Operator Placement** | | | | | | | | | | | | | |
| 2004 | | Ahmad [37] | SPS | Decentral | Cluster | 👍 | 👍 | | | | | | |
| 2005 | | Srivastava [39] | SPS | Decentral | Cluster | | 👍 | | | 👍 | | | |
| 2006 | | Pietzuch [15] | SPS | Decentral | Cluster and Cloud | 👍 | 👍 | | 👍 | | | | |
| 2006 | 👍 | Zhou [43] | SPS | Decentral | Cluster | 👍 | | | | 👍 | | | |
| 2007 | | Stanoi [78] | SPS | Central | Cluster | 👍 | 👍 | | 👍 | 👍 | | | |
| 2010 | | Benoit [42] | SPS | Central | Cluster | | 👍 | | 👍 | 👍 | | | |
| 2010 | | Rizou [40] | CEP | Decentral | Cluster | 👍 | 👍 | | | | | | |
| 2014 | | Xu [83] | SPS | Decentral | Cluster | 👍 | 👍 | | | 👍 | | | |
| 2014 | 👍 | Ottenwälder [27] | CEP | Decentral | Cloud and Fog | 👍 | 👍 | | | 👍 | | 👍 | |
| 2015 | 👍 | Starks [80] | CEP | Decentral | Fog | 👍 | | | | | | | |
| 2015 | 👍 | Peng [41] | SPS | Central | Cluster | | 👍 | | 👍 | 👍 | | | |
| 2015 | 👍 | Keeffe [38] | SPS | Decentral | Fog | 👍 | 👍 | | | | | | |
| 2016 | | Cardellini [19] | SPS | Central | Cloud and Fog | 👍 | 👍 | | 👍 | | 👍 | | |
| 2016 | | Eidenbenz [44] | SPS | Decentral | Cluster | | | | | 👍 | | | |
| 2017 | | Dwarakanath [45] | CEP | Decentral | Cluster | | | | | | | | 👍 |
| 2019 | | Nardelli [20] | SPS | Central | Cloud and Fog | 👍 | 👍 | | 👍 | | 👍 | | |
| 2020 | | Souza [84] | SPS | Central | Cluster, Cloud, and Fog | 👍 | 👍 | | | 👍 | | | |
| 2021 | | Eskandari [79] | SPS | Central | Cloud and Fog | 👍 | 👍 | | | 👍 | | | |
| **Adaptive SPS and CEP systems** | | | | | | | | | | | | | |
| 2005 | 👍 | Sutherland [33] | SPS | Central | Cluster | 👍 | | 👍 | 👍 | | | | |
| 2013 | 👍 | Annielo [32] | SPS | Central | Cluster | | 👍 | | | 👍 | | | |
| 2014 | 👍 | Heinze [85] | SPS | Decentral | Cloud | 👍 | 👍 | 👍 | | 👍 | | | |

*continue on next page...*

*continued from previous page...*

| Year | Runtime reconfig. | Name, Ref. | Type | Mechanisms | | QoS metrics | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Structure | Infrastructure | Latency | Bandwidth | Input Queue Size | Throughput | CPU | Availability | Certainty | Trust and Privacy |
| 2017 | 👍 | Matteis [86] | SPS | Decentral | Cluster | 👍 | | | 👍 | 👍 | | | |
| 2018 | 👍 | Weisenburger [87] | CEP | Central | Cluster | 👍 | 👍 | | | | | | |
| 2019 | 👍 | Russo [88] | SPS | Decentral | Cluster, Cloud, and Fog | | 👍 | | | 👍 | | | |
| 2019 | 👍 | Liu [18] | SPS | Decentral | Cluster | 👍 | | | 👍 | 👍 | | | |
| 2020 | 👍 | Jonathan [89] | SPS | Central | Cluster, Cloud, and Fog | 👍 | 👍 | | | 👍 | | | |
| 2021 | 👍 | Vogel [90] | SPS | Central | Cluster, Cloud, and Fog | | | | 👍 | | | | |
| **Open Source and Commercial SPS and CEP systems** | | | | | | | | | | | | | |
| 2006 | | EsperTech Inc. [35] | Both | Decentral | Cluster | 👍 | | | | | | 👍 | |
| 2010 | | Apache Spark [91] | SPS | Decentral | Cluster, Cloud, and Fog | 👍 | | | | 👍 | | | |
| 2014 | | Apache Storm [76] with changes in [41] | SPS | Decentral | Cluster, Cloud, and Fog | 👍 | 👍 | | 👍 | 👍 | | | |
| 2015 | | Heron [9] | Both | Decentral | Cluster, Cloud, and Fog | 👍 | | | 👍 | 👍 | | | |
| 2015 | | WSO2 [36] | Both | Decentral | Cluster, Cloud, and Fog | 👍 | | | 👍 | 👍 | | | |
| 2015 | | Apache Flink [92, 54] | Both | Decentral | Cluster, Cloud, and Fog | 👍 | | | 👍 | | | | |

Table 2: Overview of related work on OP mechanisms and Adaptive Stream Processing and Complex Event Processing Systems. 👍 means the system or mechanism fulfills the requirement whereas blank means either it doesn't or is unknown.

*Input Queue Size* is the total number of tuples stored in the input queue for a query.

*Throughput* is the number of event tuples processed and transmitted in a given time frame for a query.

The *CPU* utilization is considered as system load in many SPS and CEP systems. In particular, the load is defined as the ratio of CPU time required by an operator over a fixed amount of time [23].

*Availability* is the accessibility of the nodes and the links involved in the processing and transfer of the data streams [19].

*Certainty* is the measurement on how accurate are the input data streams. This could happen due to faulty sensor sources that result in data uncertainty. Approaches such as [27] consider this inaccuracy in the data for effective placement decisions based on the location sensor data.

*Trust and Privacy* in sensor data often is required especially for applications like fraud detection, which includes sensitive information that must not be revealed to unknown processing devices during operator placement. The authors in [45] used trust categories and sensitivity parameters to measure information's trust and sensitivity.

In the following, we explain the approaches compared in Table 2 that have considered adaptivity in the mechanisms and in the event processing system. ADAPTIVECEP [87] is a programming model and a CEP system that provides a means to specify QoS requirements at runtime and fulfill them using so-called operator migrations (movement of operators between nodes as a consequence of change in placement). Elasticity in data Stream Processing System provides adaptivity towards workload. The authors [85] provide an adaptive Stream Processing System where the nodes can be added and removed at runtime using operator migrations. Later, the authors [93] utilize an online learning approach for auto-scaling. Based on this work, the authors [94] extend their work and proposed an adaptive replication scheme for fault tolerance in stream processing systems, which trades off recovery time for overhead.

*Adaptive Stream Processing and CEP systems*

Furthermore, the authors [95] look into the trade-off between monetary costs against the offered QoS. Several surveys on elasticity [96, 97, 98, 99] highlight the importance of a streaming system's adaptivity towards the changing workload in terms of adding and removing resources during runtime. For instance, Lorido et al. argue that the auto-scaling process in elastic streaming systems resembles the MAPE loop for autonomous systems similar to our work. Marcos et al. discuss the advantages of the online approach over static approaches in adaptivity.

Röger et al. [98] highlight the importance of distributed elasticity solutions by handling multiple operator at the same time. Proactivity in elasticity was proposed by Matteis et al. [86] using so-called *Model Predictive Control* method, which accounts for system behavior over a future time horizon to predict the best reconfiguration to be executed.

Several approaches look into the problem of adaptivity in scheduling tasks (operator graph) using methods such as operator migration. Sutherland et al. [33] proposed an adaptive selection algorithm for continuous queries in data stream processing systems. Aniello et al. [32] proposed adaptivity between two placement mechanisms in Apache Storm. Liu et al. [18] advance the work on state migration to look into the problem of colocating stateful and stateless operators. Jonathan et al. propose an adaptive system for wide-area stream processing [89]. Vogel et al. proposes self-adaptive strategies to regulate the degree of parallelism in a stream processing system. In Table 2, we summarize the characteristics and the QoS objectives for the adaptive systems discussed in the above paragraph.

*CEP Programming models*

Many specification languages for defining queries have been proposed in the past years for CEP and Stream Processing Systems such as SASE [100], Cayuga [101], CQL [102], TESLA [103] for specifying complex events and detecting them by triggering notifications. Modern CEP programming models and systems like Apache Storm [76], Apache Flink [28], Heron [9] and Google Dataflow [10] used in Apache Beam [53] provide extensive APIs to specify complex events for both batch and stream processing. Recently proposed benchmarking frameworks such as [29] and DCEP-Sim [30] unify CEP systems and simulate a CEP environment and operator placement, respectively.

*Adaptivity using Mechanism Transitions*

The abstract concept of mechanism transitions within the context of a communication system is formalized by the researchers of the Collaborative Research Centre "MAKI" [104, 57, 105, 106]. A communication system comprises mechanisms operating on different layers of the network stack, enabling different functionalities. Similar mechanisms, however, focusing on providing specific functionality, may function differently depending on the current environmental conditions. For instance, both WiFi and LTE focus on providing Internet connectivity in stationary and mobile scenarios, respectively. A simple yet useful transition from WiFi to LTE mechanism is advantageous to deal with such changes in the environmental conditions. The adaptation cycle of a transition is inspired by the well-known MAPE-K cycle [107]. It mainly comprises four processes pf *monitoring* the system's and mechanism's conditions, *analysing* the monitored information, *planning*, and *executing* the transitions strategically.

*MAPE-K loop*

The Collaborative Research Centre "MAKI" investigates the concept of mechanism transitions for a broad range of communication mechanisms [108, 109,

110, 111, 112, 113, 114]. Mechanism transitions are first studied in the context of video streaming systems [108, 109, 115, 116]. Afterwards, in publish-subscribe systems, mechanism transitions between filtering schemes [110] and event dissemination mechanisms [111] are studied [117]. Frömmgen et al. [113, 105] propose transition strategies to execute the best suitable search overlay in publish-subscribe systems. Mechanism transitions are also introduced in the context of different schedulers in MPTCP [118]. Richerzhagen et. al. [114, 119] propose a transition-enabled monitoring service that issues transition between monitoring mechanisms. Softwarized networks also benefit from mechanism transitions in monitoring [120].

*Previously studied mechanism transitions*

### 2.1.3 *Discussion*

Both the Stream Processing and CEP research areas focus on the OP problem, as discussed in Section 2.1.2. OP mechanisms are extensively researched considering different QoS requirements such as to minimize latency [37], to reduce load [43, 80, 44, 84], to minimize bandwidth-delay product [15, 40, 77], and to preserve trust and privacy [45].

The fulfillment of QoS requirements, however, is only feasible under limited changes in the environmental conditions. For instance, most existing works [20, 121, 37, 19, 40] build on stationary networks. The approaches that considering dynamics in the environment in some form, for instance, mobility, introduce *(i)* redundancy using duplication [80] or checkpointing [122], *(ii)* periodic operator placement updates [15], or *(iii)* explicit operator migrations to deal with the dynamics [38, 123, 27, 45, 79].

It is important to note that current approaches for CEP, so far build on a *single* placement mechanism. In contrast, we target to adaptively use *multiple* existing OP mechanisms by supporting transitions to increase the range at which a CEP system can meet changing QoS requirements. Although existing adaptive CEP systems [87, 85, 76, 86, 32, 89] benefit from integrating multiple mechanisms in another context than OP. In the above CEP systems, however, mechanisms are not actually changed depending on the environmental conditions. In contrast, in this thesis we offer a methodology of mechanism transitions to provide a means for CEP systems to easily integrate new OP mechanisms to be highly extensible.

*Key Research Challenge*

Furthermore, none of the approaches above provide adaptivity between distinct OP mechanisms based on the QoS requirements in a seamless manner and at a minimum cost. Adaptations between different mechanisms is investigated using mechanism transitions within Collaborative Research Centre "MAKI" for other application domains, such as live video streaming [108, 115],

user-generated videos [109, 116], publish-subscribe systems [110, 111, 117], network monitoring [114, 119] in softwarized networks [120], and topology adaptation in wireless sensor networks [124]. In contrast, in this thesis we switch between mechanisms in the context of CEP systems, while aiming to minimize the state transfer cost incurred in terms of transitions. Furthermore, we aim for optimal selection of CEP mechanism by understanding the performance of respective mechanisms, which is not targetted in any of the above works.

*Open source SPS and CEP systems*

In the last few years, several open-source and commercial SPS as well as CEP systems emerged, such as Esper [35], Apache Spark [91], Apache Storm [76], Heron [9], WSO2 [36], and Apache Flink [54]. Each has its way of implementing CEP mechanisms and placement decisions. Hence, their architectures vary a lot in terms of their design, making it a lot challenging to compare them. In Table 2, we summarize the findings related to the features defined in Section 2.1.2. Many of the above systems either don't provide dedicated OP mechanisms or don't incorporate adaptivity. Apache Storm [76] provides placement mechanisms that assign tasks comprising operator graphs to the worker nodes before the topology starts. Flink [54] makes use of the query optimizer proposed in Stratosphere [92] that performs OP while considering an objective function of network traffic and CPU load for the cost. For Event-Based Systems besides Storm and Flink, OP mechanism is missing so the QoS entry is based on the monitoring information available in the system. This means the runtime characteristics such as QoS requirements are not considered in the OP mechanisms. Several extensions to Storm including, R-Storm [41] and T-Storm [83] provide custom OP mechanisms based on resources and traffic at runtime. But only R-Storm [41] seems to be integrated as per the latest release to-date. Altogether, the above modern Event-Based Systems either do not provide dedicated OP mechanisms or lacks adaptivity in OP mechanisms.

*Programming Models*

Besides integrating OP mechanisms and adapting between them, we provide a novel programming model to develop OP mechanisms in this thesis. We analyze existing CEP programming models [87, 103, 9, 28]; however, none of them enable development and execution of distinct OP mechanisms in a heterogeneous environment of physical nodes like we do in this thesis. Close to our work, DCEP-Sim [30] enables development of OP but failed to deploy it in a real-world fog-cloud deployment infrastructure as they focus on only a simulation environment. In contrast, we study the effect of distinct operator placement mechanisms, perform adaptations between them, and analyze the cost of adaptations in a real-world network infrastructure.

In summary, Table 2 and the above discussion shows that there is no *one size fits all* OP mechanism or an adaptive CEP system that can fulfill conflicting and changing QoS requirements of the consumers. Although mod-

ern and open-source Event-Based Systems such as Apache Storm [76] provides interface to extend OP mechanisms allowing selection of servers for the Storm tasks, still those strategies do not allow fulfilling conflicting QoS requirements using adaptivity. By proposing mechanism transitions for OP mechanisms, this thesis focuses on the *(i)* modelling of QoS requirements, *(ii)* design and understanding of the behavior of OP mechanisms, *(iii)* learning-based adaptive selection of OP mechanisms, and *(iv)* cost-efficient transition strategies, which facilitate stateful mechanism transitions at runtime comprising a multitude of interdependent distributed components.

*1st Research Gap: No one size fits all OP mechanism*

## 2.2 In-Network Processing

The term *In-Network Processing (INP)* was first used in the context of distributed query processing in wireless sensor networks [125]. The act of offloading query processing to the sensor nodes was referred to as INP in early 2000s. The authors highlighted multiple significant research problems, including aggregation of data [126], query languages [102], query optimization, multi-query optimization [8], and adaptive query processing [127]. Most of the concerns mentioned above are still active research areas in stream processing and CEP systems.

Nowadays, modern Event-Based Systems refer to INP in a different context than it was done in the primal work [125]. It is commonly referred to as a method that accelerates operator processing on programmable hardware such as switches, SmartNICs, FPGAs, to name a few. Novel networking architectures such as Software-Defined Networking [128] and Content-Centric Networking [129] provide an opportunity to perform INP on programmable hardware, for instance, using OpenFlow protocol [130, 131], P4 switches [51, 66] or ICN enabled switches [132]. The idea of INP for Event-Based Systems is to enable event processing inside modern hardware to deliver very low latency requirements (order of microseconds - milliseconds). This ability can be used fulfill the latency requirements of many IoT applications such as autonomous vehicle control.

In the following sections, we first present the key differences between the communication mechanisms of Event-Based Systems and conventional Request-Reply Systems. Second, we present related concepts to the novel edge computing paradigm Content-Centric Networking followed by a classification of these architectures concerning formerly presented communication mechanisms. Lastly, in the discussion sub section, we highlight the research gaps that we focus on in the second contribution of this thesis.

### 2.2.1 *The Push and Pull Dilemma*

Many modern IoT applications like traffic monitoring and fraud detection are inherently push-based applications, i.e., data continuously flows to the CEP system. Still, various IoT applications rely on traditional pull-based communication mechanism for scenarios which do not encounter continuous flow of events, such as database applications. Intuitively, traditional pull-based data processing systems fall short in dealing with the continuous flow of events for push-based scenarios, and vice versa. In comparison to pull-based data processing, for instance, where queries are issued against a database system at discrete times, push-based data processing requires continuous queries to be processed, as seen in the IoT scenarios described above. Many a times applications require coexisting push- and pull-based communication. For instance, live video streaming applications need push-based delivery of video frames, however, also need to support pull-based requests in case of packet loss.

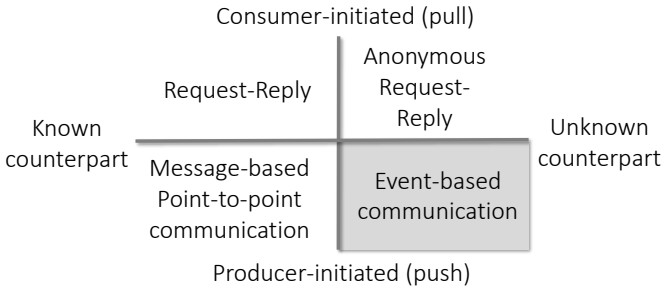



Figure 6: Communication mechanisms based on producer and consumer interaction adapted from [133].

To understand the difference between the scenarios that require pull-based or push-based data processing, let us analyze the communication mechanisms based on the interaction patterns illustrated in Figure 6. We consider two dimensions to categorize the interaction patterns: *(i)* who initiates interaction, and *(ii)* the knowledge about the counterpart. On the first dimension (y-axis), we distinguish on the initiator of the communication – consumer and producer of the information. On the second dimension (x-axis), we distinguish between knowing the counterpart of the communication – having knowledge of the counterpart's identity or having no knowledge at all [133].

In the request-reply interaction pattern, the interaction is initiated by the consumer or the client of the information– for instance, classical web-based applications or the interaction in a database system. In anonymous request-reply interaction, the identity of the producer is unknown. In contrast, if the producer initiates the interaction and the consumer identity is known, we have a classical messaging application interaction. If the consumer is unknown, we have the event-based communication (highlighted in gray in Fig-

ure 6) that depends on a mediator, typically known as a broker who connects the interested parties [134]. This thesis focuses on enabling both event-based communication (push) and request-reply communication (pull) to enable a wide range of IoT applications in the presence of novel networking architectures such as Content-Centric Networking, discussed in the next subsection.

### 2.2.2 *Content-Centric Networking*

Content-Centric Networking (CCN) networking architecture was proposed by Jacobson et al. [129] in 2009[11], built on the concept of named data. CCN has no notion of the host, and thus a packet "address" names the content and not the location. CCN communication mechanism is *consumer-initiated*, consisting of two packet types: `Interest` and `Data`. A consumer interested in a content broadcasts its interest over all the available channels. Any node receiving the interest and having the content responds to it with a `Data` packet. Both `Interest` and `Data` packets identify the content using a *name* that is typically a hierarchical object. It identifies the packet content such that the data object name is in the name subtree specified by the `Interest` packet. The match is successful, if the name in the `Interest` packet is a prefix of the name in the `Data` packet. Once a packet arrives on a face[12] of a node, the longest prefix match as explained above is performed and the data is returned based on a lookup. In the following, we first introduce the packet structure in CCN, the packet handling in the data plane, and the two emerging network architectures under the umbrella of CCN.

#### ***Packet Structure***

Each CCN packet includes "*a fixed-size header depending on the packet type, a variable-length data object name, the optional name TLV (type length value format), some more optional TLVs, and finally, the payload*" (cf. Figure 7) [135]. The maximum packet size of a CCN packet is 65 KBytes, which is fragmented to fit the MTU of the communication protocol. The header comprises of *(i)* the versioning field to combine protocol version, *(ii)* the packet type (cf. Table 3) and its length, *(iii)* the hop limit used to limit the scope of the packet, *(iv)* the flags to extend the fixed size header, and *(v)* the header length indicating the length of the hop by hop header. After the fixed-sized header, Msg type and Msg length fields are the same as Pkt type and length for the reconstructed CCN packet to be self-describing after fragmentation. The variable-

[11] In the remainder of the thesis, we will use the terms ICN and CCN interchangeably.
[12] Face stands for interface in CCN terminology.

length name represents the name prefix followed by the type length value format to encode variable-length fields like the name in CCN packets[136].

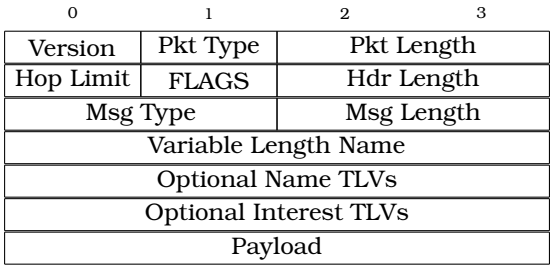



Figure 7: CCN packet structure adapted from [136].

| Packet Type code | Packet Type |
|---|---|
| 0x00 | `Interest` |
| 0x01 | `Data` |

Table 3: Default Packet Types as shown in [136].

### *CCN Data Plane*

Every CCN node comprises three major data structures: Forwarding Information Base (FIB), Content Store (CS) (also called in-network cache), and Pending Interest Table (PIT). When an `Interest` packet arrives on a face, the node first checks its CS for a matching data object by its name. If a match is found, the data object encapsulated in a `Data` packet is sent via the same arriving face. If not, the node searches for the matching name entry in PIT. If an entry already exists in the PIT, the face list is updated, and the interest packet is discarded. This is because an interest packet has already been sent upstream. If an entry in PIT is not found, the node continues looking for a matching FIB entry, creates a PIT entry and forwards the interest packet to the potential source of the data.

### *Named Data Networking (NDN) and Named Function Networking (NFN)*

Named Data Networking (NDN) [137] emerged as an architecture based on the principles of CCN *named data*. NFN enhances the packet addressing using the concept of *named functions* by extending the foundations of NDN that utilizes just *named data*. NFN provides data computations with network forwarding by executing computational tasks over the CCN network substrate [61]. It represents *named functions* on the data as so-called $\lambda-$calculus expressions [138]. However, the main focus of NFN is only on resolving pull-based (discrete) computations on top of the CCN substrate. In contrast, this thesis focuses on continuous and discrete computations (push and pull), an expressive representation to specify the computations, and its efficient and robust distribution. Having understood the basic principles of ICN architectures we now classify them based on the push-pull dilemma discussed in Section 2.2.1.

***Classification of ICN architectures based on the Push-Pull Dilemma***

In the following, we present a classification of ICN architectures based on the communication mechanisms utilized as: *pull-based, push-based, or both push- and pull-based* [139].

*Pull-Based*

Classical architectures in ICN such as NDN [137] and NFN [61] use a consumer-initiated communication mechanism. Specifically, the consumer initiates sending an interest packet towards the ICN network for each data object. The ICN network forwards this request to producers that produce data items in interest of the consumer. In the next step, producer forwards the data items to the consumer. Classical ICN architectures are inherently consumer-initiated and, hence, do not support processing periodic data streams. A possible approach to tackle the problem is by combining NDN [140, 141] and NFN [142, 143, 132] while continuously polling the producer to fetch the data packets. This approach can be meaningful when the sending rate of the producer is low, but higher sending rates need higher number of polling interest requests and, hence, a lot of unnecessary traffic.

*Push-Based*

The second class of ICN architectures [144, 145] are *push-based*, which support producer-initiated communication mechanism similar to publish-subscribe systems. In contrast to the pull-based architectures, these architectures lack support for request-reply interaction applications such as web applications. To solve this issue, Ahmed et al. [135] use so-called beacon messages indicating the content that could be interesting for the consumer. Those messages act as a callback to initiate interest in the content, and if the consumer is interested in the content, then it sends an interest packet. Another solution solves this problem by using long-lived interest packets [146]. This means that the entry related to the interest packet stays for a longer time in the PIT so that the consumer keeps receiving the corresponding data objects. However, the beacon approach incurs a high amount of message overhead (three-way exchange of what accounts for a one way). The other approach with long-lived interests poses a bottleneck for large-scale IoT applications, due to the centralized architecture.

*Both Push- and Pull-Based*

Some approaches like CONVERGENCE [147], GreenICN [148], Carzaniga et al. [149], and HoPP [146] provide support for both *push-* and *pull-* based communication mechanisms. The CONVERGENCE system combines the publish-subscribe communication pattern with an Information-centric Network layer. Another ICN architecture named GreenICN combines NDN (pull-based) with publish/subscribe based architecture COPSS [145] (push-based). Carzaniga et al. [149] present a unified network approach similar to the second contribution of this thesis. However, the authors propose a theoritical preliminary approach without any real-world implementation. In a subsequent work [150], the same authors presented routing protocols for the unified approach with no focus on how they deal with distribution aspect of the approach. HoPP [146] is closest to our approach that also enables co-existing push and pull functionality. Yet, the authors propose to implement push semantics in a centralized control plane, which could be a problem for a large-scale IoT application. A recent NDN implementation on the P4 switch [151] modifies NDN architecture to support both push and pull interaction patterns. However, this work inherits the limitations of the P4 language, such as it does not allow loops. Furthermore, all the aforementioned approaches that combines both push- and pull-based communication mechanisms are restricted in the ability to process continuous queries over the ICN substrate as we do.

### 2.2.3 *Discussion*

We organize the above introduced ICN architectures in taxonomy concerning the communication mechanisms, push-based, pull-based and both push- and pull-based. As discussed above, existing ICN architectures entirely rely on pull-based communication mechanism [137, 132, 61, 140, 142, 141] or push-based communication mechanism [144, 145, 135]. On the one hand, relying only on the pull-based communication mechanism to handle periodic data streams poses several issues as follows. *(i)* It results into significant overhead in terms of interest request packets required to fetch each data object. *(ii)* Because of the regular polling mechanism there could be packet loss which eventually could result into high delay until fresh data becomes available. *(iii)* Due to the regular polling traffic, the resulting data could be stale, for example, when served from in-network cache. *(iv)* Lastly, consumers might end up in busy-waiting for data, for example, when a producer does not have any data to deliver, thus, resulting into wastage of resources. On the other hand, changing to a push-based communication mechanism is problematic for IoT applications based on request-reply interactions. For example, traditional web applications and database systems still need personalized request-

*Key Research Challenge*

reply interaction. Existing work [135, 146] trying to accomplish request-reply interaction while using push-based semantics not only add additional complexity but also incur very high message overhead.

In this thesis we argue that ICN architectures should provide efficient support for both communication mechanisms push and pull enabling any type of IoT application. However, the aforementioned approaches pose one or the other limitations *(i)* while implementing push-based communication, the authors [152, 148] do not deal with flow imbalance that eventually results in event loss; *(ii)* their approach [146] poses large in-network state causing significant overhead; or *(iii)* inherits limitations of the underlying technology, e.g., P4 language constructs [151]. We mitigate the above limitations in the second contribution of this thesis by providing an efficient CEP-based unified communication model that enables flow control as well as reliable and in-order processing of data over the ICN architecture. By enabling the development of IoT applications that may use any communication mechanism push or pull, we increase the range of applications developed atop ICN architectures. Moreover, our unified communication model deals with the implications of push-based communication in an ICN architecture such as *flow imbalance*, *event-loss*, and *out-of-order event arrival* while providing an efficient in-network complex event processing.

*2nd Research Gap: missing both pull- and push-based mechanisms*

## 2.3 Deployment Infrastructures

This section defines the deployment infrastructure considered in this thesis, followed by a related work analysis of current serverless platforms pertaining to the third contribution of this thesis.

### 2.3.1 *Cloud Computing*

Cloud computing is a paradigm coined in late 2007, which lately emerged as an important utility, nowadays offering flexible and dynamic IT infrastructures worldwide [153]. Cloud computing offers a promising infrastructure for building, deploying, and operating pull- and push-based IoT applications. It offers resources for compute, network, and more recently data management services over the Internet on a pay-per-use basis [134]. In cloud computing terminology, resources are lent to *tenants* by *cloud providers*. Each tenant uses the cloud resources and provides functionality to the *end-users*, such as end-users of IoT applications. Cloud computing essentially offers a threerd fold deployment model. *(i)* In a *public* cloud, multiple tenants share the physical resources provided by the cloud provider. *(ii)* In a *private* cloud, all

the resources are exclusively dedicated to a single tenant. *(iii)* In a *hybrid* cloud, a composition of two or more cloud setups (public and private) are used. For instance, a sudden burst in demand is compensated by adding a public cloud to private cloud resources. Clouds majorly offers different service models depending on the level of abstraction of the resources that are accessible to the tenants[13] [154].

*Service models of cloud*

1. *Infrastructure as a Service* (IaaS) offers the capability to provision machines (virtual or hardware) involving privileges for computing and processing power, storage, network as well as other required resources. IaaS gives the tenants the highest level of privileges over the cloud resources. The pricing is based on the number, specification, and runtime of the managed resources [134].

2. *Platform as a Service* (PaaS) enables the tenants to focus on the deployment and management of their application by providing only the platform– for instance, a database management system through a defined API. The cloud provider is responsible for managing the infrastructure (hardware and operating system) [155].

3. *Software as a Service* (SaaS) enables the tenants with a complete working product that is administered by the cloud provider [134]. For instance, Amazon Translate provides a translation service for different languages powered by the deep-learning technology of Amazon[14]. The cloud provider is responsible for all the underlying resources, and the pricing is based on usage and subscription.

4. *Function as a Service* (FaaS) provides the tenants with a programming model to create applications on the cloud infrastructure that abstracts away all the operational concerns. It is a relatively new service model being adopted by many cloud providers. It lies in between the two extremes, PaaS and SaaS models, where the tenant is unaware of the infrastructure (cf. Figure 8). Instead, they focus on a packaged component or full applications. The tenants are allowed to host code using the provider's programming model that may be tightly coupled to the platform. This service model emerged nowadays to a paradigm known as Serverless Computing, elaborated in Section 2.3.2.

[13] AWS offers over 160 cloud services with the pay-as-you-go approach as of December 2020. https://aws.amazon.com/pricing/ [Accessed in May 2021].

[14] Amazon Translate Service. https://docs.aws.amazon.com/translate/ [Accessed in May 2021].

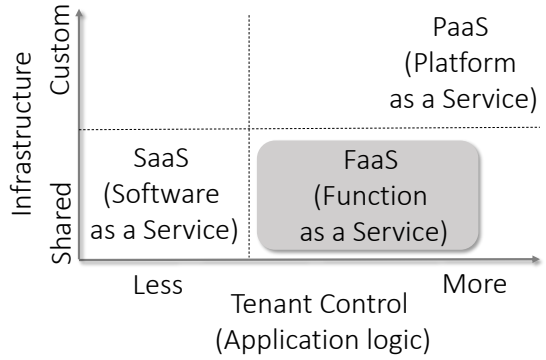



Figure 8: Control over application and infrastructure for the different service models adapted from [156].

### *Fog Computing*

*Fog Computing* is another trend close to cloud computing known for its benefits, such as low latency. It extends the cloud computing paradigm to bring computation and storage towards the edge near to the consumers. Cloud providers such as Amazon and Google launched several fog locations known as AWS CloudFront [157] and Google Cloud CDN [158], respectively. Additionally, companies like AT&T, Microsoft, Intel, and Verizon are investing in edge hardware and software infrastructure to fulfill low latency demands and maximize overall throughput [159].

*Fog Computing is ...*

Fog Computing have been defined in the literature in different ways. For instance, Cisco described it as a synonym to edge computing. Specifically, it is defined as *"analyzing data at the network edge (near to the end-users)"* [160]. Another definition of Fog Computing is proposed by the OpenFog consortium standards as *"as an architecture that distributes resources and services like computing, storage, control, and networking anywhere along the continuum of Cloud to IoT devices (edge)"* [161].

*... processing data at the network edge.*

In this thesis, we proceed with the former view, defining Fog Computing as a paradigm that enables processing at the network edge as this definition is more prominent and widely accepted [162, 160, 163]. More precisely, we proceed with the fog-cloud hierarchy illustrated in Figure 9. The fog-cloud infrastructure provides a platform to meet distinct requirements of IoT applications. For instance, latency requirements can be met using the infrastructure at the fog or the edge layer that is near to the user, while computation requirements can be met using the infrastructure at the cloud layer that is computationally powerful. To meet the distinct requirements of the applications, we aim to utilize the deployment infrastructure to deliver the quality requirements. Another essential paradigm that is studied widely in the context of Cloud Computing is Serverless Computing explained in the next subsection.

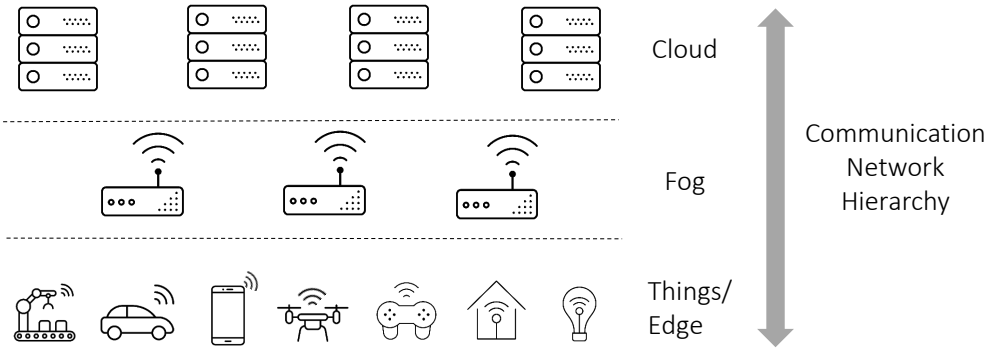



Figure 9: Deployment infrastructure comprising of IoT devices at the edge, fog and cloud resources (adapted from [48]).

### 2.3.2 *Serverless Computing*

*Serverless Computing* is a term adopted by industry cloud providers for the FaaS service model introduced in the above section. It is recently popularized by Amazon in the re:Invent session of AWS Lambda [164], and other providers with the Google Cloud Functions [165], Microsoft Azure Functions [166], and IBM OpenWhisk [167]. A common misconception with the name is that the server is no more needed, which is not valid, but the decisions on the number of servers and its capacity are taken by the serverless platform. It provides an abstraction so that a computation such as a stateless function is disconnected from where it could be executed [156]. It allows running applications on pre-configured resources abstracted away from the developer so that configuration time is minimal and billing is only applied to the runtime of the functions itself. The underlying hardware and operating system of the server are unknown to the user, as it is assumed to be independent of the function that is placed on it. Furthermore, serverless functions are isolated from other functions and are ephemeral. Therefore, every execution of a function is independent of the state of previous or parallel running executions [168].

In this work, we benefit from Serverless Computing for deploying a CEP system. Therefore, in the following description, we investigate if the existing serverless platforms provide concepts required by a CEP system. First, we define crucial CEP concepts for a serverless platform that could deploy a CEP system. Second, we define a generic model for Serverless Computing at the cloud. Third, we define serverless relation to microservices paradigm and the numerous serverless platforms. Finally, we present a discussion of the most prominent serverless platforms concerning the CEP concepts.

*Defining features for later comparison*

1. *Continuous data streams.* The support for continuous data streams in serverless functions is crucial when implementing stream processing and complex event processing systems. While the basic support is suffi-

cient for the need of CEP systems, there are numerous challenges when implementing continuous streams. For instance, correctness (accuracy) of the results has to be ensured by processing the data streams in-order.

2. *Statefulness.* As a CEP system can utilize the sliding window operator, for example, to join data received from multiple producers, there is a need to keep these values received in the window saved for further processing. Serverless functions are by definition cold-started, which means that every time we want to execute the function, the service starts a new instance of the application, effectively clearing all data in current memory. After the function is finished executing, the data is deallocated. Therefore, it is needed to keep the state of currently received data to perform a sliding window protocol. Furthermore, cold-started serverless functions can highly influence the overall latency of the system. Also, in dynamic environments like mobile applications it is also possible for a CEP system to migrate operators between nodes. In this case the operator state has to be kept in order to fulfill consistency of data.

3. *Reusing existing operator graphs.*

   Another challenge that is related to the operator state is the reuse of existing operators. Operator reuse, also referred to as operator graph reuse, is a mechanism that allows the CEP system to utilize operators from already mapped queries that are deployed on nodes in the current network to be reused by another CEP query currently in deployment. The operator reuse reduces latencies and CPU load for CEP systems that have to fulfill changing QoS requirements for queries or transit the operator placement between different QoS demands as shown in the first contribution of this thesis (cf. Chapter 4)

4. *Operator graph processing and Operator Placement.* In traditional distributed CEP systems, the operator graph is formed by connecting operators in an overlay network, which can have physically different underlying network topologies [169]. The operator graph is processed in a centralized and a distributed manner as dictated by the CEP and the OP mechanism. An equivalent functionality has to established in a serverless architecture that can be challenging.

5. *Flow control.* A flow control mechanism makes sure that everything in transit will have the capacity at the receiver's end to handle. Since CEP systems deal with continuous data streams that, for some use cases, amount to several thousand and even million of events per second, it is indispensable for a CEP system to implement flow control to handle the influx of enormous data.

6. *Auto scalability* is the capability of the provider to automatically detect workload changes and scale the requested resources effectively so that the systems support elasticity.

Given the fundamentals of serverless platform, let us review the generic deployment model of serverless computing and related concepts before going into the discussion.

### *Generic Serverless Deployment Model of Cloud Providers*

The steps of deployment of functions are often not equal for every serverless framework provider, but it can be assumed that the same assumptions hold for most. We will look into the basic system model of AWS Lambda [164] as it was one of the first and most prominent commercial serverless platform. After defining a function, it can be placed solely on the serverless platform either through a web interface or by a command line (Serverless Platform Interface), which is shown in Figure 10. Besides the function itself, the developer should specify how many resources the function needs while being executed such as RAM or CPU credits. If no resource bounds are set, the platform often executes the function with only minimal resources available. After the deployment request is received, ① the provider chooses a machine internally that provides enough resources for the requested computation. ② The user do not have visibility of this process. The only received feedback from the platform is presented when the function is in execution or when the execution is finished. When the machine is ready for placing a function, a virtual machine (VM) is started, which is needed to securely execute the function in an encapsulated environment. ③ Amazon uses its own open-source implementation called Firecracker that manages the creation, maintenance, and termination of VMs. Firecracker defines VMs as so-called microVMs, as their execution has only a lightweight footprint in the hosted environment. To execute functions in the created VM, most serverless platform providers, encapsulate the defined function by wrapper code containing the entry point of the application that is defined by the provider. The entry point of the application creates the seen function context and sends status updates back to the web interface. After the setup of the application is finished, it invokes the function body with the context and the given input parameters of the function. As the function is running, the wrapper collects and stores logs emitted by the function for later debugging and monitors the health of the currently running function. ④ When execution is finished, it retrieves the result from the function and sends it back to the issuer. Afterwards, the VM is released, making space for other functions to be executed on the same host machine. An essential characteristic of a serverless function is that all the computed intermediate state, such as variables in the heap memory, are considered lost after exe-

cution. This is a severe limitation for applications that need to save state for later computations. The state has to be saved on other resources such as cloud storage which in turn have higher access latencies and lower throughput. Besides reducing configuration overhead for cloud resources, Serverless Computing also allows developers to extract encapsulated parts of the code into single applications, effectively reducing code complexity.

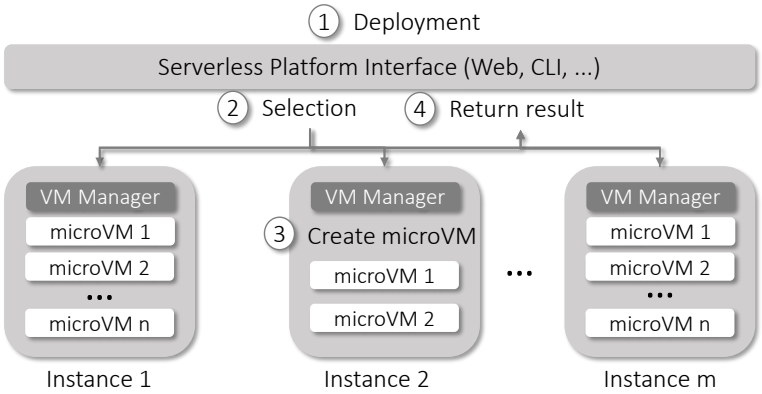



Figure 10: Generic deployment model of current serverless platforms [170].

### *Relation to Microservices*

In contrast to Serverless Computing, an often used paradigm to achieve similar functionalities is the *Microservices Architecture* that could be seen as a similar architecture to the serverless paradigm but needs to be differentiated. A *microservice* is considered to be a part of a bigger application handling only logic related to this specific service. An example service could be a mail service with its only purpose being the sending and receiving of emails, while content and formatting is left to other services. Multiple services can interact together to build a whole application, which is considered to be better scalable and maintainable as developers can focus on single services at a time. It would surely be possible to implement the same logic with a serverless architecture but with a main difference in the deployment overhead. While the application logic is defined in the serverless paradigm in functions, the deployment and configuration of these functions are left to the provider of the FaaS framework. In contrast, a microservice can be seen as a monolithic application and therefore has to be deployed by the user that also configures the underlying OS. This, of course, enables users of microservices greater flexibility at the deployment of applications but is also more time consuming and error-prone. Many distributed applications, such as the content streaming platform by Netflix [171], are considering a hybrid solution of microservices and serverless. Only specific services are written as serverless functions as not all services can be written in a serverless manner. This is often the case

*Microservices Architecture*

when a service is considered to be stateful, which is a feature that serverless applications do not provide by design.

### *Other Serverless Platforms*

Additionally, besides commercial providers like AWS Lambda [164], we also look into OpenFaaS [172], an open-source implementation of a serverless platform. Instead of using microVMs, OpenFaaS uses docker containers combined with the container orchestration framework Kubernetes [173] to manage servers and functions. A container orchestrator enables a distributed network of docker container managers on different nodes and configures communication between those containers by providing an overlay network. Furthermore, the network of nodes is manageable from one or multiple manager nodes that have the privilege to deploy and manage containers on other nodes. Because of this system architecture, OpenFaaS provides the advantage of deploying every valid docker container as a serverless function by extending it with the so-called Function Watchdog. The extension is a light HTTP server that transforms incoming function requests, redirects them to the application in the container, and sends the application back to the caller [172].

*OpenFaaS*

Moreover, because every docker image can be deployed as a serverless function, the application language is independent of the serverless provider platform. This is a key difference to the commercial solution from AWS, as it provides only some languages for writing custom functions. Besides language independence, OpenFaaS provides auto-scaling for functions to increase or decrease the number of functions as soon as a load change is detected. Nastic et al. [174] proposes a serverless real-time data analytics platform mainly focused on edge computing that also shows how serverless functions can be used for processing streams of data. The business logic of functions that manipulate data is encapsulated in containers to dynamically control the deployment of functions. The framework provides a wrapper API that controls access from and to the streaming data pipeline in execution to provide a consistent ability to access functions. Therefore, the user-defined function is abstracted away from the stream pipeline. An open-source alternative to AWS Lambda is Kubeless [175], which is based on the container orchestration framework Kubernetes [173]. When submitting a function through the command-line interface to Kubeless, it packages the function in a docker container that serves as a runtime environment. Kubeless provides a number of runtime environments each supporting individual programming languages using a custom docker image. As AWS Lambda [164], Kubeless supports processing streams by receiving executing a function every time the stream provider publishes a record.

*Serverless @Edge*

*Kubeless*

### 2.3.3 *Discussion*

While there are numerous serverless FaaS providers in the industry and open-source we want to concentrate on the mature and developed solutions. In recent years, many commercial serverless platforms evolved, e.g., AWS Lambda [164], Google Cloud Functions [165], Azure Functions [166], and IBM OpenWhisk [167]. An open-source alternative to AWS Lambda is Kubeless [175], which provides a number of runtime environments.

*3rd Research Gap: Missing abstractions for reuse of CEP mechanisms*

In Table 4, we compare the following platforms comparing the functionalities presented in the above section: AWS Lambda [164], IBM Cloud Functions [176] based on OpenWhisk [167], Google Cloud Functions [165], and Microsoft Azure Functions [166]. It is important to note that although the reviewed serverless platforms provide one or other functionality of CEP, it cannot be used out of the box for realizing a CEP system. There are multiple research challenges listed below in terms of their support in processing continuous data streams, state management, and operator placement in a distributed infrastructure that needs to be considered.

*State support*

For a system to enable stateful operators, is the first and foremost issue that provides the storage of data. While serverless functions do not have a persistent storage model before and after execution by design and as the execution space is provisioned as soon as the function is requested, it is necessary to find a suitable location with a high I/O throughput.

The advantage of state is also underlined by the work of Jonas et. al. [177], as they identified the most effective storage method for serverless functions are currently cloud-enabled storage spaces such as Amazon S3 [178] or parallelized SSDs in larger compute instances.

*State migration*

Another comparison is curated by López et al. [179] based on Amazon Lambda and IBM OpenWhisk. Their work shows that Amazon Step Functions [180] can run multiple function invocations that keep the state in the following executions. Although this is practical for environment variables, etc., it is limited to a maximum of *32 KB*, at least for Amazon. Moreover, they evaluate the cost of state migrations in terms of state migration time. They show that IBM has equal state migration time to Amazon but that IBM transition times significantly increase when using larger amounts of data. State migration, as considered in FaaS, is the ability of the function to synchronize the state to the other functions that are currently in execution. Besides their findings they also show a comparable solution to Microsoft's parallel functions [166] and show that IBM is currently not offering this as part of their IBM BlueMix portfolio [167]. When looking at the state migration of FaaS in dynamic environments, we found that the FaaS approaches do not provide

| Provider | AWS Lambda [164] | IBM Cloud Functions [176] | Google Cloud Functions [165] | Azure Functions [166] |
|---|---|---|---|---|
| **Serverless Features** | | | | |
| Execution time Limit | **15 min** | 5 min | 9 min | 10 min/ 60 min/ **unlimited** |
| Memory Limit | **Up to 3 GB** | Up to 2 GB | Up to 2 GB | 1.5 GB |
| Deployment Package size | 250 MB | 500 MB | 500 MB | **unlimited** |
| Temporary Disk Space | 500 MB | **2GB** | **2GB** | 1.5 GB |
| Extended Cloud Storage | S3, DynamoDB | IBM Cloud Object Storage | Google Storage | Azure Storage |
| Billing | Duration and declared memory | Duration and declared memory | Duration, declared CPU and memory | Duration and average memory use |
| Auto scalability | 👍 | 👍 | 👍 | 👍 |
| **CEP Features** | | | | |
| Continuous event streams | Amazon Kinesis | Apache Kafka | Cloud Pub/Sub | Azure Stream Analytics |
| Flow control | 👍 | 👍 | 👍 | 👎 |
| Operator graph processing | 👍 | 👎 | 👍 | 👍 |
| Operator Placement | 👎 | 👎 | 👎 | 👎 |
| Deployment on Edge | AWS IoT Greengrass [182] | IBM BlueMix [167] | 👎 | Azure Functions Edge [183] |
| State support and migration | 👍 | 👍 | 👍 | 👍 |
| Operator reuse | 👍 | 👎 | 👍 | 👍 |
| Parallel execution support | 👍 | 👎 | 👎 | 👍 |
| Query notification | AWS SNS service | Push notifications | Firebase | Azure Notification hub |

Table 4: Comparison of support for CEP and serverless functionalities in the FaaS platforms considered. 👍 refers to the presence, while 👎 refers to the absence of a functionality in the serverless platform. Text in **bold** refers to the best feature configuration possible in the serverless system.

a major disadvantage in startup time and migration time than using docker containers. Aske et al. [181] identified that cold starts of serverless functions by common providers like AWS or Google, which take two seconds to complete while warm starts take about one second. This is comparable to the startup time by docker containers, although there are no specific measurements as it depends on the application.

*Operator reuse*

As we can utilize the aforementioned storage method also for operator reusability, a decentralized management of the diverse operator placement has to be done to support elasticity and the basic functionality of placement algorithms. Using OP mechanisms [82], the node placement coordinator decides which node gets an operator deployed in a CEP system. Most CEP systems use decentralized mechanisms to propagate messages to control the operator graph and identify operators to be reused. In a serverless implemented CEP system, we would need to implement an equal functionality.

*Continuous data stream support*

In terms of continuous data stream support, we found that with AWS Lambda, it is possible to transport data streams in functions using Kinesis [184] that is the stream processing and analysis service developed by Amazon. Also, IBM OpenWhisk supports continuous streams using streaming systems like Apache Kafka [11]. The support of streaming functions is not surprising as serverless functions are executed in a containerized environment with the same possibilities as a virtual machine. Most of these providers support streaming in some form of input data towards serverless functions. However, those services are often limited to provider-specific streaming solutions In contrast to basic stream support, an issue persists regarding maximum execution time and serverless functions' maximum deployment size. We looked into all providers having an upper limit for serverless function execution except Azure Functions that recently launched a premium plan with unlimited time. Furthermore, the size of the deployment package that contains the code is also unlimited (cf. Serverless Features in Table 4).

*Operator graph processing*

IoT frameworks aid in keeping operator state for reuse when rebuilding the operator tree and compositing it. Besides AWS also Google and Microsoft have established almost equal IoT frameworks with Google IoT [185] and Microsoft IoT Edge [183]. As IBM is utilizing the open-source serverless framework OpenWhisk [167], it would also be possible to enable IoT execution serverless functions by simply installing OpenWhisk on the nodes and manually developing the cloud connection. This is an approach taken by the LEON prototype system for distributed image processing [186]. Currently, IBM does not offer a solution like AWS Greengrass [182] that makes the process simpler. The architecture of locally deployed serverless functions, which we saw could also be developed using AWS Greengrass, was discovered to be more efficient in terms of latency and throughput. When the graph is composited, the operator dependencies are built by chaining operators in the query and

linking the following operation as successor operator in the graph. To enable operator graph in a serverless based CEP system, we need persistent addressing, as presented in the features above. Therefore, when deploying the graph, the system developed in, e.g., AWS Greengrass [182], could deploy the lambda functions on nodes, effectively placing static addresses of the successor operators in the code because the centralized cloud has knowledge of the overall network topology and operator placement.

*Deployment at Edge*

Recent research on involving multiple serverless providers to complete the computation of data on edge [181] is also reflected by recent research where the execution round-trip time of code executed was dramatically reduced when executed utilizing multiple serverless providers, and local compute power of an edge node. In summary, this also shows that modern applications do not have to focus on only offloading computational work to the cloud, which is also involved with higher latencies, but can also utilize the mix of local/edge computed and cloud FaaS.

*Operator Placement*

Despite the above serverless features, what remains an open problem would be the deployment of an operator graph. As OP mechanisms can constantly optimize the operator graph in order to fulfill QoS requirements. With the current solution using AWS Greengrass or comparable providers, the lambda function executing the operators would need to be replaced with updated functions that encapsulate an operator. Current CEP systems provides a placement coordinator that monitors the operator graph and the nodes used for deployment for the placement decisions. This functionality has to be established in a serverless CEP middleware.

*Parallel execution*

Parallel execution of operators allows the CEP system to utilize other nodes in the network with lower machine usage to minimize latency and maximize throughput. Therefore, parallelization support is also a feature supported by serverless platforms that the CEP system can benefit from greatly.

In the above discussion and in Table 4, we analyzed the existing serverless platforms in terms of their ability to support CEP concepts. Although current serverless platforms provide several promising features, which can provide high performance in CEP, they lack core concepts in support of event processing such as operator placement, state management, and other optimizations. Furthermore, the CEP programming models presented in Table 2 lack of interoperability because of the tight coupling between the programming model and the execution environment. This research gap leads us to the final contribution of this thesis, a unified serverless middleware that provides *(i)* flexibility in CEP to reuse mechanisms such as operator specification across distinct execution environments, *(ii)* flexibility in exchanging operator specification at runtime facilitates IoT applications to change execution semantics, which often becomes necessary.

## 2.4 IoT and its Applications in Event Processing

Although there is a wide range of definitions of IoT proposed in the literature [187, 188], we adopt a user-centric definition focusing on its application deployment across fog-cloud infrastructures.

**Definition 3.** *Internet of Things (IoT).*
"IoT interconnects sensing and actuating devices to share data across the platforms through a unified framework, such as fog-cloud infrastructures, and forming a common operating picture for enabling innovative applications [1]."

*CEP used in real world IoT applications*

Event processing is used in a wide range of IoT applications in different sectors. It has become even easier using open-source event processing platforms such as Apache Storm [9], Apache Flink [28], and Apache Kafka [11]. These platforms are used in thousands of companies globally, including more than 60% of the Fortune 100 list[15]. To name a few prominent IoT applications, Alibaba Group[16], a leading Chinese multinational technology company specializing in e-commerce and retail, recently acquired Apache Flink to analyze trillions of online transactions per day. Storm[17] powers a wide variety of Twitter applications related to real-time analytics of tweets, personalization, search queries, and many more. Finally, Apache Kafka[18] is used at LinkedIn to power activity stream data and various other operations. Furthermore, cloud providers like Amazon, IBM, Google, and Microsoft are gradually starting their own streaming solutions for IoT, as shown earlier in Table 4.

In summary, event processing paradigm is powering many IoT applications presently, and is expected to do so in the future. In this thesis, we present methods and concepts that enables deployment of IoT applications atop fog-cloud infrastructures in an *adaptive*, *efficient* and *flexible* manner to overcome the challenges in *(i)* meeting changing QoS requirements, *(ii)* missing abstractions in realizing CEP on the in-network processing infrastructure, and *(iii)* missing abstractions in dynamically updating queries and reusing functionalities across CEP programming models.

[15] 100 fastest growing companies https://fortune.com/100-fastest-growing-companies/2020/ [Accessed in May 2021].

[16] Tech crunch Alibaba acquires German big data startup Data Artisans. https://techcrunch.com/2019/01/08/alibaba-data-artisans/ [Accessed in May 2021].

[17] Companies using Apache Storm. https://storm.apache.org/Powered-By.html [Accessed in May 2021].

[18] Apache Kafka powered by 60% of Fortune 100 companies. https://kafka.apache.org/powered-by [Accessed in May 2021].

## 2.5 Summary

This chapter introduces the preliminaries to better understand the contributions as well as introduces existing approaches that solves similar problems of *adaptivity*, *efficiency* and *interoperability*. By reviewing the existing work in the fields of Event-Based Systems, Adaptive Systems, Information-centric Networking, as well as Cloud Computing and Serverless Computing, we found out three key research gaps that are solved in this thesis. First, there is no single Operator Placement mechanism that can fulfill changing QoS requirements of a highly dynamic network environment that indicates the issues concerning the *adaptivity* problem. We solve this in our first contribution in Chapter 4. Second, existing in-network processing abstractions fall short in efficiently meeting the challenging QoS requirements of IoT applications. Especially, Information-centric Networking architectures have crucial shortcomings in processing continuous data streams from heterogeneous sources. We target the efficiency problem in Chapter 5. Finally, current CEP programming models are tightly coupled in their design to the execution environment, which makes it difficult to benefit from multiple CEP programming models and to reuse their mechanisms. The dependency worsens the ability to specify operators at runtime which are highly essential for the functioning of dynamic IoT applications. We base our proposed concept on the Serverless Computing principles, however, current serverless platforms are restricted in providing CEP related functionalities, that we solve in our final contribution in Chapter 6.

# 3

# Scenario and System Architecture

This chapter describes a common scenario and the overall system architecture proposed in this thesis. Section 3.1 defines the overall scenario used in this thesis to highlight the research problems addressed in this thesis. Section 3.2 defines the overall system architecture, including formal definitions of the concepts later used in the contributions of this thesis.

## 3.1 Scenario Description

This thesis considers a smart city scenario comprising of multiple applications necessary for users. Figure 11 illustrates such a smart city majorly comprising four applications: ① traffic control, ② post-accident management, ③ smart grid, and ④ fraud detection for a financial institute. We consider different applications to show the wide applicability of our concepts and system model for different scenarios, as well as address specific research challenges arising in those. For each application scenario we define the assumptions, used infrastructure, and specific research challenges targetted in this thesis. Many cities worldwide are becoming *smart*, such as Masdar in UAE, Barcelona in Spain, Bristol in the UK, and Darmstadt city in Germany, to name a few[19] [189, 190]. Similar to these smart cities, the fictional smart city introduced in this work is equipped with modern sensors to allow derivation of higher-level complex events of interest to the respective applications. In particular, we define the applications within this smart city in detail as follows.

### ① *Traffic Control Application*

The first application is a traffic control application from the domain of Intelligent Transportation Systems. To improve route planning for smart vehicles and reaction time for emergency services related to traffic accidents, the traffic on a crossroad is monitored by a CEP system that allows the detection of complex events such as traffic congestion and accidents. Such complex

[19] Titled "Digitalstadt Darmstadt" that means digital city Darmstadt.

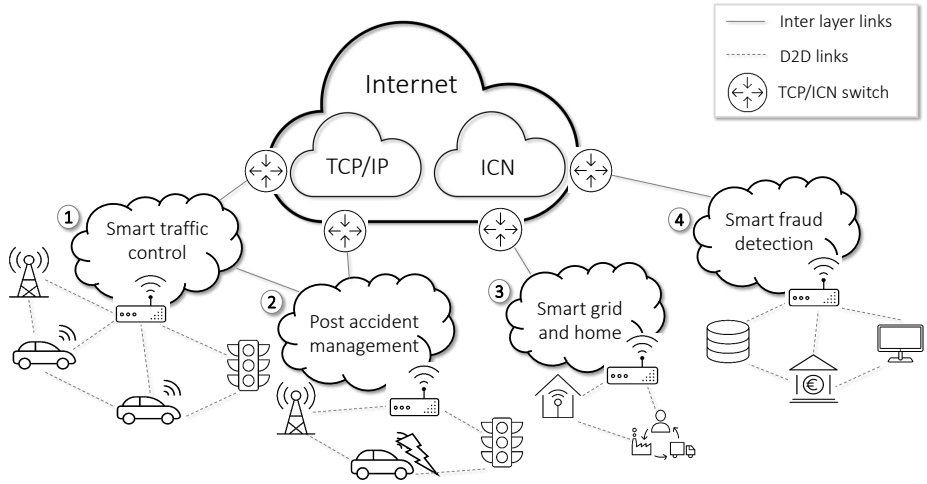



Figure 11: A smart city scenario with the four applications: ① traffic control where a traffic congestion and accident is detected, ② post accident management where a live heatmap is tracked, ③ smart grid and home where the load is predicted from smart plugs, and ④ fraud detection where credit card fraud is detected. These applications communicate over a communication substrate of TCP/IP and ICN protocols.

events can be derived by combining the low-level events from sensor sources (such as radar, lidar sensors, video cameras, induction loops, noise level sensors and smoke sensors) along with smart vehicle sensor events (such as speed information) (cf. Figure 11 ①). Reliable and timely detection of a complex event is important depending on the environment so that emergency services and vehicles before the accident road section can benefit from this information. As a basis for fast and reliable communication between the vehicles, the roadside network devices such as the 802.11p standard, Wireless Access in Vehicular Environments (WAVE) [191], can be used. This technology defines the necessary communication medium for wireless vehicular communication and supports a bandwidth of up to 27 MBps in a range of up to 1000m. WAVE defines the communication between the On-Board Units in vehicles and stationary Road Side Units enabling both vehicle-to-vehicle and vehicle-to-infrastructure communication. Besides, cellular communication using 4G LTE can be used as a fall-back when wireless communication cannot be established. A significant difference between 802.11p and cellular communication is that the former establishes a device to device communication while the latter operates using a network operator. Thus, a cellular message needs to pass the uplink and downlink channels of the cellular network to reach the destination, whereas 802.11p messages are directly sent to the destination user. A swift switch between such mechanisms and timely response under dynamic environmental conditions is crucial in this scenario.

*Assumptions and Infrastructure*

In the scenario, we consider an edge-fog-cloud infrastructure as motivated earlier, for event processing. The edge infrastructure includes vehicles, Road Side Units and stationary sensor sources such as a radar. The fog infrastructure is more powerful than the edge, comprised of processing nodes from nearby fog locations, such as micro data centers. The cloud infrastructure available at the end of emergency services for traffic management and data centers with more computational power are considered for more complex operations such as predictions required in the later applications. In this application, the CEP system is exposed to many dynamics of the environment. For instance, depending on the vehicle density, event rates, and mobility of vehicles may vary a lot. Furthermore, depending on the location of the vehicle, whether it is near the traffic congestion area or to a highly mobile highway, the QoS requirements may also vary. For instance, in a highly dense location, it is critical to quickly retrieve the traffic congestion event to make timely decisions on re-routing the vehicle or changing the lane. While in a highly mobile highway location, due to very mobile vehicles with a high speed, it is needed to detect the event in a highly stable manner. In this scenario, the key issues are *how* and *when* to adapt mechanisms to deal with the dynamic environment and QoS requirements. For this, we propose a methodology on *transitions* for CEP mechanisms, provide adaptable OP mechanisms and transition execution algorithms in a transition-capable CEP system, named TCEP (cf. Chapter 4).

*Research Challenge: Dynamic Environment*

② *Post-Accident Management Application*

An accident at a highway can be very devastating and it is imperative to carry out rescue activities in time to bring the situation back to normal. A widely used method to know the location of victims is using a live heatmap of the area with the rescue team for timely rescue operations. We have similar assumptions on the communication and available infrastructure for processing as in the traffic control application (cf. Figure 11 ②) for this application. The extremely dynamic environment of this application and the stringent QoS requirements of such an emergency scenario makes it difficult to deliver information promptly to the rescue team. The requirements for such safety-critical applications are low latency (in the range of milliseconds), high relative speeds between transceivers (up to 200 km/h and above), high dynamics of the environment (density and mobility of the vehicles), and substantial network load (period data streams from vehicles) requirements, which are hard to be met with existing Communication and Event-based Systems [192]. Therefore, in this scenario the ICN paradigm can be beneficial. ICN principles such as named data, in-network processing, and in-network caching simplify mobility support and allow vehicles to retrieve information in very low latency and network traffic [193]. However, existing ICN architectures fall short in support of processing of continuous data streams as intended in push-based

*Assumptions and Infrastructure*

*Research Challenge: Stringent QoS requirements*

Event-based Systems. Furthermore, current architectures only limitedly support in-network processing as provided by CEP. Here, the key issues are to *unify* communication mechanisms in a single ICN architecture and to *enable* event processing in such programmable in-network architectures. To deal with these challenges, we propose unified communication model that complements a state-of-the-art ICN architecture and algorithms for event processing over ICN substrate in a INETCEP system (cf. Chapter 5).

③ *Smart Plug Load Prediction Application*

*Assumptions and Infrastructure*

Load prediction for households is a topic of concern for the smart grid to assess if the power grid needs upgrading or a temporary boost in power. The technicians have to feed more power into the grid at specific times to prevent it from collapsing. Usually, in the morning time when people watch television, prepare tea and use a water heater at the same time, it results in additional three gigawatts of power for roughly 3-5 minutes. The phenomenon is widely known as television pickup, and in former times power grid had to increase the power by monitoring television channels manually. In this scenario, load prediction can be a huge benefit to handle the problem without monitoring television channels manually. A CEP system can model this scenario by monitoring load from smart plugs and deriving predictions for the future regarding the load so that power grids can proactively plan to feed more power automatically. Like the traffic control application, an edge-fog-cloud infrastructure comprising smart plugs at the edge, in-network devices like switches and routers at the fog and high-end servers available at the power grid location are assumed to be used for event processing (cf. Figure 11 ③). Predictions in a timely and reliable manner are fundamentally important in this application to avoid power failure and unnecessary costs on the smart grid. In addition to ② application, INETCEP system can be used to deal with the reliability and efficiency requirements of this ③ application.

*Research Challenge: Low latency & reliability*

④ *Fraud Detection Application*

*Assumptions and Infrastructure*

Credit card fraud is an open issue in the financial industry. According to Federal Trade Communication, Americans reported losing over $1.9 billion to credit card fraud in 2019 alone. With an increase of credit card fraud year up to 50% since 2017, financial providers heavily invest in new technologies to prevent further damages. According to International Data Corporation, banks spent $5.3 billion on AI in 2019, growing to $12.4 billion in 2023, on fraud detection analysis alone [194]. While the development of new detection algorithms for identifying fraud cases is getting more and more attention with the use of machine learning, the announcement by Visa shows that not only the development of an algorithm itself is challenging, but also the de-

ployment of such unstable systems is getting increasingly complex. As fraud patterns can change quickly, systems bound to detect these patterns need to be constantly updated while still maintaining a no-downtime policy. Existing financial providers use event processing systems such as Apama CEP [195] to handle the billions of credit card events in a fraction of milliseconds (cf. Figure 11 ④). However, requirements of such large scale applications are much larger than what existing CEP systems offer. Custom operators such as machine learning functions are hard to be integrated in existing CEP systems and require extensive knowledge of the underlying CEP system. Furthermore, because of the tight dependency on the underlying CEP runtime, custom operators cannot be updated dynamically without re-compiling and deploying the entire CEP system again. This poses a severe problem for high availability applications, such as fraud detection in the context of financial systems. Redeployment of a CEP system would require multiple seconds or event minutes before being able to be used again, which is unacceptable for such applications, where multiple fraud transactions can be repeated within this period. As a solution to these problems, we propose a CEPLESS middleware in Chapter 6 based on Serverless Computing principles that allows flexible custom implementation of operators and dynamic update of operators ensuring zero downtime.

*Research Challenge: Lack in interoperability*

Besides, according to its original definition, a middleware should not be application dependent or system dependent. Ideally, it should allow seamless integration of many diverse IoT applications within a smart city, which needs cross-compatibility of multiple CEP systems. We enable such integration using CEPLESS by providing custom operator abstractions that allows execution of operator graphs on multiple CEP systems in the execution layer. In the next section, we define the overall system architecture proposed in this thesis that enables this flexibility.

*Beyond the specific applications and systems*

## 3.2 System Architecture

This section explains the overall system architecture proposed in this thesis. Section 3.2.1 discusses a system model, including the standard definitions used for the contributions of this thesis in the rest of the chapters. Section 3.2.2 discusses the overall architecture, including the different software components of the overall architecture and their interactions.

### 3.2.1 *System Model*

This section provides a formal description of the entities involves in the architecture proposed by this thesis. Table 5 provides a summary of all the common notations used in this thesis for completeness.

| Notation | Meaning | Notation | Meaning |
|---|---|---|---|
| $P$ | Set of event producers ($p \in P$) | $t(e)$ | Event timestamp |
| $D$ | Continuous data stream (comprising set of events $e$) | $B_I$ | Input buffer of an operator |
| $C$ | Set of event consumers ($c \in C$) | $B_O$ | Output buffer of an operator |
| $B$ | Set of brokers ($b \in B$) | $\phi_\omega$ | State of an operator |
| $\Omega$ | Set of CEP operators ($\omega \in \Omega$) | $C_\omega$ | Operator container |
| $G$ | Operator graph | $f_\omega$ | Processing function of an operator |
| $N$ | Set of nodes ($n \in N$) | $en(t)$ | Environment conditions dependent on time $t$ |
| $L$ | Set of links connecting the nodes ($l \in L$) | $Q$ | Continuous query including the QoS requirements ($Q = \{q, QoS, T_{QoS}\}$) |
| $E$ | Set of events ($e \in E$) | | |

Table 5: Notations and their meaning.

#### *Event and Operator Graph Model*

*Event processing components*

The proposed system architecture consists of *(i)* a set of *event producers* ($P$), which produce continuous *data streams* ($D$), *(ii)* a set of *event consumers* ($C$), which express a *complex event* by means of a *query* $Q$ on the incoming data streams, and *(iii)* a set of *event brokers* ($B$), which acts as host to a set of *operators* ($\Omega$) effectively processing and forwarding the events. A data stream comprises multiple events ($E$) – low-level or complex events – where each event has an attached *timestamp* ($t(e)$). Formally, an event represents a tuple of key-value pairs, of the form $e = \{(k_1, v_1), \ldots, (k_{num^e}, v_{num^e})\}$.

Before processing, the query is transformed into an intermediate representation called of a directed acyclic *operator graph*. The operator graph dictates an execution plan specific to the query given by the event consumer. Formally, an operator graph is $G = (\Omega \cup P \cup C, D)$, comprising of a set of operators ($\Omega$), producers ($P$), consumers ($P$) and data streams ($D$), formally, $D \subseteq (P \cup \Omega) \times (C \cup \Omega)$.

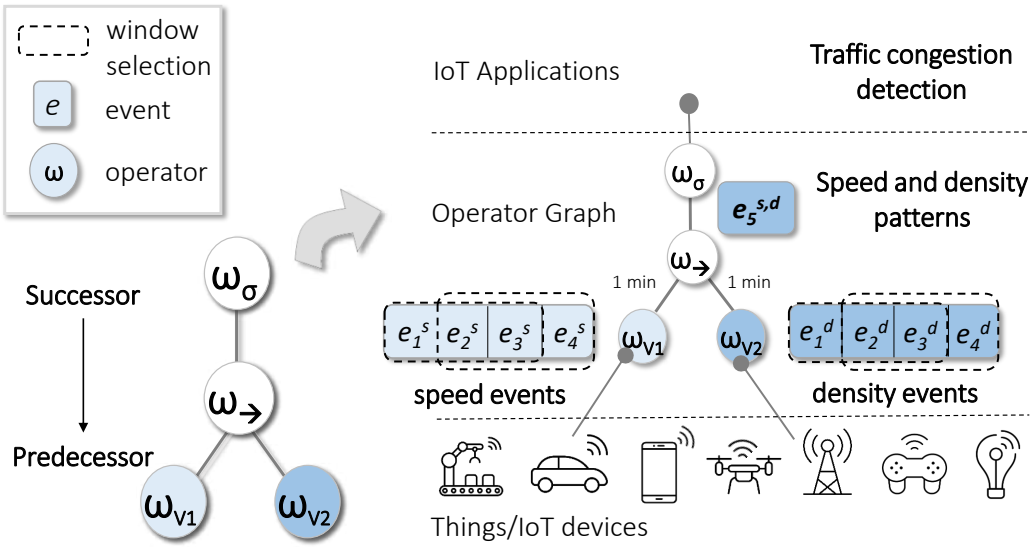



Figure 12: Simple CEP operator graph example for a traffic congestion query (shown in the left figure). An operator graph is used to detect meaningful events for the IoT applications [196].

Figure 12 illustrates an operator graph comprising four operators from the traffic control application introduced in Section 3.1 ①. In the operator graph, the data flow of the events is from bottom to top of the graph, where the bottom operators are the predecessors while the top are the successors. Here, the bottom-most level are producers, while the top-most are the consumers. Operators $\omega_{V1}$ and $\omega_{V2}$ refer to the window operators on the two input data streams based the road sections $V1$ and $V2$, comprising low-level events on *speed* and *density* of vehicles arriving from the *incoming data streams*. Operators $\omega_{\rightarrow}$ and $\omega_{\sigma}$ denote sequence and selection operators, respectively, that reacts on the *outgoing data streams* from operators $\omega_{V1}$ and $\omega_{V2}$. The execution of an operator graph is composed of step-wise transformation based on the processing function $f_{\omega}$.

Each operator $\omega$ applies its specific processing function $f_{\omega}$ to produce a set of complex events either forwarded to the successor operators in the operator graph or the consumer. Each transformation step takes an input a *selection* $s$ of events from its incoming data streams and applies an operator-specific processing function $f_{\omega}$ to produce a set of complex events. The selection happens by means of a *selection policy* that determines the events for processing. For instance, the operator $\omega_{V1}$ specifies a selection policy for a sliding window of three subsequent speed events $\{e_1^s, e_2^s, e_3^s\}$ on the incoming speed data stream. In a subsequent transformation step, operator $\omega_{V1}$ apply the processing function on the updated selection of events $\{e_2^s, e_3^s, e_4^s\}$ after sliding one event. Thus, each transformation step produces zero or more events as output. Events are evicted from the incoming data streams after each transformation step by

means of a *consumption policy*. For instance, the slide size of a sliding window determines the events that can be evicted, e.g., $e_1^s$ is evicted when the subsequent transformation step with $\{e_2^s, e_3^s, e_4^s\}$ is performed.

### *Timestamps*

In this thesis, we assume that the low-level events arrive in the order indicated by the event timestamp $(t(e))$ [24], and the system nodes are equipped with clocks that can be synchronized using a clock synchronization protocol such as Network Time Protocol [197]. Although, there can be late events and common ways to deal with them by specifying a wait threshold for those events [54] or using predictions [198], dealing with late events is out of scope for this thesis. A complex event depends on a number of low-level or other complex events in selection occurring in different order and timestamps. Literature adopts different ways to timestamp the complex event by assigning maximum, minimum or even an interval-based timestamp [199]. In this thesis, we adopt a timestamping scheme that assigns the maximum timestamp of the selection to each complex event, but the results of this thesis can be easily adapted for other timestamping schemes.

**Definition 4.** *Operator Buffers* ($B_I(\omega)$ and $B_O(\omega)$). For each operator ($\omega$), the incoming events arrive in the input buffer $B_I(\omega)$ of an operator, which are processed by the function $f_\omega$ and the output complex event is sent to the output buffer $B_O(\omega)$ of an operator.

### *Kind of Operators*

There are two kinds of operators in CEP: *stateful* and *stateless*. A *stateless* operator works based on the fixed computational parameters that are immutable (e.g., filter and stream operators). In contrast, a *stateful* operator works on a dynamically changing computational state $\phi_\omega$ that is mutable (e.g., window and sequence operators), depending on the internal logic of the operator [200]. For instance, a mutable operator can change the selection of events at runtime based on an operator-specific *selection policy* and *consumption policy* [126].

### *Query Model*

Event consumers specify complex events that represent multiple event patterns using a continuous *query*[20], $Q = \{q, QoS, T_{QoS}\}$. Here, $q$ denotes the semantic query specified by the CEP system[21], $QoS$ denotes a set of QoS requirements that the query must satisfy and $T_{QoS}$ denotes a set of thresholds ($\tau_{qos} \in T_{QoS}$) for the respective QoS requirements ($qos \in QoS$).

### *Deployment Infrastructure Model*

This thesis focuses on an edge-fog-cloud infrastructure commonly considered to deploy IoT applications, though this work is not only for this specific network topology. The hierarchical network infrastructure comprises three layers as illustrated in Figure 13. *(i)* Static or mobile "Things" represent IoT devices connected over cellular or wireless communication. *(ii)* Infrastructure at the fog layer, such as switches and routers that are TCP/IP and ICN enabled in micro data centers offering a low-latency link to the Things in physical proximity. *(iii)* The cloud layer comprises distributed infrastructure such as in data centers interconnected over a fixed network. The infrastructure can communicate over the standard TCP/IP architecture or using the novel ICN architecture [129]. Furthermore, the IoT and edge resources can communicate wirelessly using device-to-device communication [34] or WAVE communication [191], cellular communication or ICN protocols.

*Edge-fog-cloud*

In the infrastructure model, producers and consumers (specific applications shown in the top of the figure) are placed on Things. In contrast, the operators can be placed on any of the three layers. The end-to-end latency for this resource model is influenced by the physical proximity of resources and the computational power of resources. In general, we assume higher resource availability and computational power in the cloud layer. In contrast, Things are resource-constrained because they are battery-powered. Fog nodes are computationally more powerful than nodes in the Things layer. Moreover, the availability of a fog location near an IoT device is not ascertained. For placement on such heterogeneous infrastructure, each operator $\omega$ is encapsulated in a container to place it on the computational resources of the distributed deployment infrastructure as defined later in Definition 5.

[20] Note that the query need not neccessarily be specified by a consumer. An autonomous CEP system or a system administrator can also specify a query and respective QoS requirements depending on the current context.

[21] We use AdaptiveCEP domain specific language [87] to specify continuous queries which uses the mainstream programming language Scala. Nevertheless, the model proposed is not dependent on a specific programming language.

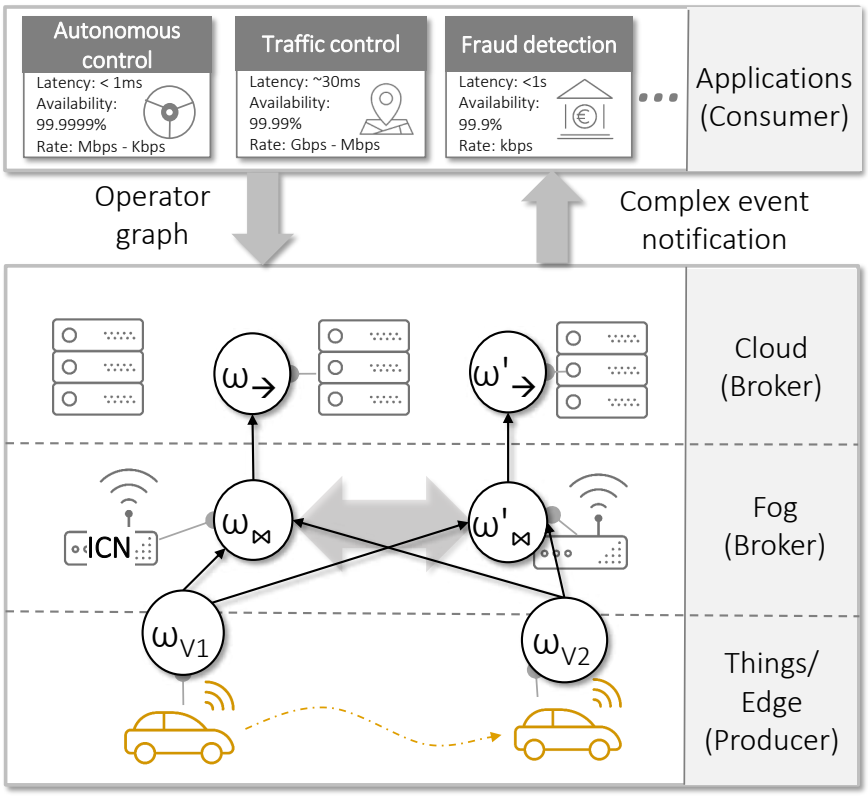



Figure 13: An example operator graph employing adaptation on the edge-fog-cloud infrastructure. The applications specify QoS requirements which must be met while processing operator graphs.

### *Node Model*

A node ($n \in N$) is referred to a physical resource where a producer, consumer, or broker can be executed. It acts as a host to the system entities such as brokers. Since the mapping of operators on the broker nodes ($b \in B$) can change at runtime due to the dynamics in the environment, we need to move operators on broker nodes flexibly. The nodes are connected through a set of communication links ($l \in L$), which are defined by the underlying communication technology, e.g., wireless or cellular communication.

**Definition 5.** *Containers* ($C_\omega$). A container acts as an execution environment for an operator. Using a container, an operator can execute on the heterogeneous infrastructure and flexibly migrate between nodes in the heterogeneous infrastructure.

Each container can hold more than one producer, consumer or operator. To distinguish between pinned and unpinned system entities, we introduce two kinds of containers *static* and *dynamic*. A static container is pinned to

one node, while a dynamic container is unpinned, which means operators assigned to a dynamic container can freely move between different nodes at runtime. Producers and consumers are usually pinned to the static container, while brokers can be pinned on any static or dynamic container. Each dynamic container executes an *Empty App* that defines the necessary interface for an operator to interact with the system. Although Empty App can hold more than one operator, operators are free to move between other Empty App without harming the other operators being executed on the same node. In this way, the model provides a way to flexible operator deployment and operator migrations on a heterogeneous deployment infrastructure.

### *Quality Requirements Model*

In this section, we define the quality requirements that is needed to be delivered by a CEP system while processing the continuous query. We divide the requirements into two main classes: Quality of Service (QoS) and Quality of Results (QoR) as seen below.

#### *Quality of Service (QoS)*

*Meeting QoS is an essential goal of CEP*

An essential goal of an OP mechanism is to find a mapping of an operator graph to brokers such that it satisfies an objective function of QoS requirements, such as end-to-end latency, bandwidth, and control message overhead. The consumer can specify one or more QoS requirements ($QoS$) and change them at run time. The dynamics in the environmental conditions ($en_1, en_2, \ldots,$), such as varying workload and mobility, influence the fulfillment of such QoS requirements.

In this work, we consider the following crucial performance metrics in the context of IoT applications that influence the decision of operator placement in a highly dynamic environment: *end-to-end latency*, *control message overhead*, *network usage*, *number of hops*, *throughput*, and *loss rate*.

#### *Latency*

Although there is no unique definition of latency, most of the related work distinguishes between two notions of latency: *event-time* and *processing-time* latency [29], [10], [201]. Here, the former refers to the interval between the timestamp of an event when it was first generated at the producer and its emission time at the last output operator. While the latter refers to the interval between the event ingestion at the first input operator and its emission time at the last output operator. However, none of the above definitions cover

the time taken by an event in the entire end to end path. We consider this end-to-end path to compute the latency of a complex event as defined in the following.

**Definition 6.** *End-to-end latency.* The time taken to *(i)* retrieve low-level events from producers, *(ii)* process the events, *(iii)* generate a complex event, and *(iv)* transmit the complex event through the network path between the given event producers to the given consumers.

*Example:* In accordance to the maximum timestamping in the complex event, we consider the maximum path latency as an end-to-end latency. To better understand the definition of end-to-end latency, let us revisit the example scenario introduced in Figure 12. We assume that two producers $\omega_{V1}$ and $\omega_{V2}$, and a single consumer is interested in detecting congestion. Let us consider the path from $V_1$ and $V_2$ via broker nodes $b_1, b_2, \ldots, b_k$ to the consumer $c$. We assume the position of the consumer is near to producer $V1$, when it triggers the query and the operator graph is mapped to the broker network path. Thus, the end-to-end latency is the sum of the network delay observed on the path $V_2, V_1, b_1, b_2, \ldots, b_k, c$, and the execution time of the query on these nodes in the path. The expected network latency between the broker nodes is well known, e.g., by using Vivaldi coordinates [202]. Furthermore, the network latency can vary because of the dynamic nature of the network.

*Control Message Overhead*

This metric is important as in a highly dynamic and resource constrained environment involving IoT devices, a high overhead is not desirable. We adapt the definition using existing work in this direction in the context of OP mechanisms for a scenario involving MANETs (Mobile Ad-hoc Networks) [80].

**Definition 7.** *Control message overhead.* The number of control messages sent to assign all the operators of a query to the brokers. In other words, the overhead in terms of exchanging messages to place a query on the deployment infrastructure nodes.

*Example:* Using the above definition of control message overhead and the assumptions on the traffic congestion scenario in Definition 6, let us demonstrate the meaning of control message overhead. To fulfill an objective, such as end-to-end latency, usually OP mechanisms, maintain a latency cost space to find out network paths with minimum end-to-end latency. However, to build such a cost space, many messages have to be exchanged between the considered nodes for placement and the OP coordinator. Furthermore, to place an operator graph, acknowledgements on the assignment of opera-

tors on nodes are sent. We refer to the number of such control messages for OP as control message overhead.

*Network Usage and Hops*

We adapt the definitions of network usage and hops as defined in previously proposed OP mechanisms literature [15, 40, 80].

**Definition 8.** *Network usage* ($NU(Q)$). The amount of data that is on the wire including the low-level events, complex events, and coordination messages for placement for a given query $Q$ at a given time [82].

Formally, it is measured as:

$$NU(Q) = \sum_{l \in L} \mathcal{B}(l) lat(l). \tag{1}$$

In Equation (1), $L$ is the set of all the links utilized to place query $q$, $\mathcal{B}(l)$ is the bandwidth over link $l$, and $lat(l)$ is the latency over link $l$. This metric captures the bandwidth-delay product of the query.

**Definition 9.** *Hops.* The total number of nodes required to place a query.

*CPU Load*

The performance of a query may get affected by the CPU load of the node used for operator placement. In some CEP systems, this metric is considered to achieve both good performance and for load fairness [23].

**Definition 10.** *CPU load.* The amount of CPU utilization of the node hosting the operator or the operator graph.

*Throughput*

In particular, a CEP system has to consider the arrival event rate of the incoming data stream while processing the events. When the event rate increases and the processing rate cannot cope with the arrival rate, most of the CEP systems adapt to the increased rate by exhibiting backpressure [92]. Specifically, backpressure queues the events in the input buffer to match the processing

rate of the CEP engine. To deal with this effect, we use the metric *sustainable throughput* to measure throughput as widely done in the literature [29, 203].

**Definition 11.** *Sustainable throughput.* The maximum amount of event load that a CEP system can handle at a given time without exhibiting prolonged backpressure.

*Quality of Results (QoR)*

While the QoS metrics measure the overall performance of query in terms of different service metrics [81], Quality of Results (QoR) metrics measures quality in terms of accuracy or loss of the output complex events. For QoR, we refer to the *accuracy* metric from the information retrieval domain as the "*proximity of the complex events to the true complex events due to false positives and false negatives*" [139]. A complex event can be false positive or false negative because of the event loss, e.g., due to congestion in the network, defined as follows.

**Definition 12.** *False positives and negatives.* A false positive (FP) is defined as a "*complex event that is falsely derived when it should not have been derived*", while a false negative (FN) is a "*complex event that is falsely not derived when it should have been derived*" [139].

In contrast, the true positives (TP) and true negatives (TN) are correctly derived complex events and correctly not detected complex events, respectively.

*Loss Rate*

Packet loss rate is a well-known metric used to measure the performance of the computer networks [204]. We refer to the packet loss rate to define the loss rate for events as follows.

**Definition 13.** *Loss rate.* It is measured as the drop of events while forwarding the data stream as the ratio of output events to the total number of events forwarded by the producer.

Formally, it is measured as:

$$\textit{Loss rate} = \frac{(\textit{total events} - \textit{processed events})}{\textit{total events}}. \tag{2}$$

In Equation (2), *total events* are the number of events in a data stream ($D$) that were generated by the producers and should be received by the consumers

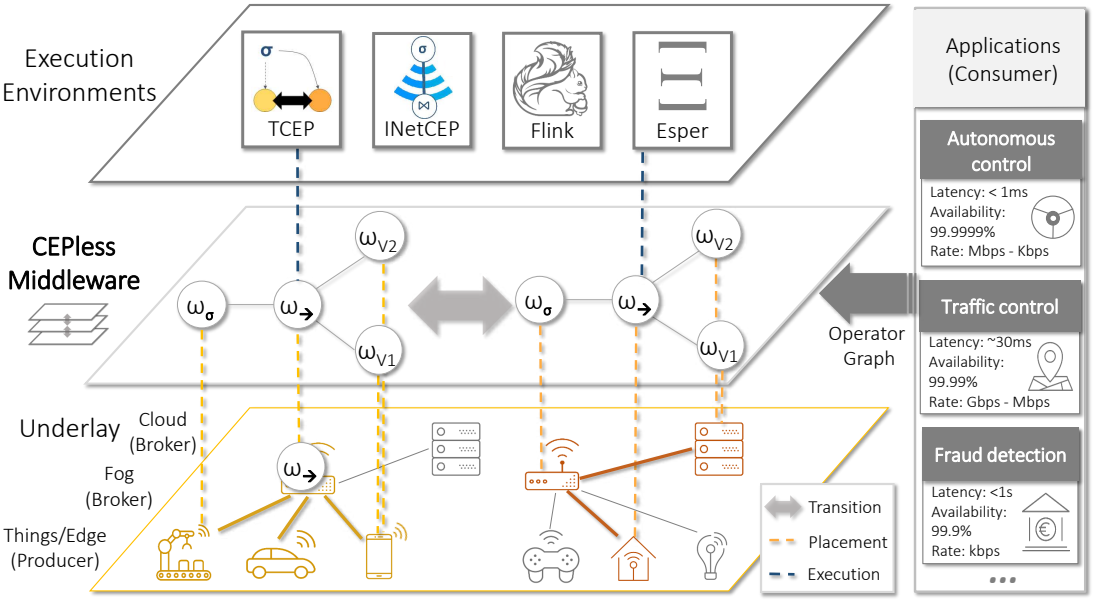



Figure 14: Overall system architecture illustrating the major components involved. *(i)* Applications, which act as consumers (right side), *(ii)* the edge-fog-cloud infrastructure introduced as underlay resources (bottom tier), *(iii)* and the operator graph mapping for placement and execution on distinct CEP systems (middle tier). Nodes and links in yellow and orange are the ones selected before and after transition in placement.

if there is no event loss, and *processed events* are the number of events forwarded by the broker in a data stream, which are eventually received at the consumer. When no transformation (in the case of forwarding) on the input events is performed, the processed events and total events should be equal (in case of no event loss). Of course, transformation steps in a query will reduce the output complex events depending on the processing logic of an operator ($f_\omega$); for this, we refer to the *accuracy* metric as defined above.

### 3.2.2 *Architecture and Contributions Overview*

*High level overview of contributions*

This section discusses the logical overview of the architecture proposed in this thesis, as shown in Figure 14. It was first introduced in Chapter 1, this section refines the architecture based on the proposed system model and briefly state the contributions.

*Methods for transitions*

1. The dynamicity in the environment of the IoT applications and the changes in the QoS requirements leads the current OP mechanism inefficient. This can be seen in the underlay tier that comprises the hierarchical edge-fog-cloud infrastructure, where the Things layer of mobile devices are very dynamic, consequently leading the current OP mecha-

nism inefficient in the middle tier. To deal with this issue, we propose a programming model and methods for transitions between OP mechanisms as shown in the figure at the middle tier. The operator graph is migrated to another set of broker nodes in the edge-fog-cloud infrastructure after a transition. The TCEP system seen in the execution tier details the contributions related to the methodology of OP mechanism transitions (cf. Chapter 4).

*Network-centric CEP*

2. A CEP system also has to efficiently meet the QoS requirements that are challenging to meet due to the dynamics in the environment. In challenging scenarios like fraud detection seen on the right of the figure, where latency-critical decisions are taken to prevent fraud, operator graph processing in an overlay network is inefficient due to high latencies. To deliver performance in such scenarios, a CEP system must be able to execute operator graphs directly on the in-network infrastructure available in the underlay. Such a deployment of operator $\omega_{\rightarrow}$ is illustrated in the fog layer in the figure. The INETCEP system seen in the execution tier describes the respective contributions related to network-centric operator execution (cf. Chapter 5).

*Serverless Middleware*

3. Finally, to deal with the limitations of current CEP programming models and fill the gap of enabling reuse of CEP execution environments, a common middleware is required that enables this integration as shown in the execution tier in the figure. For this purpose, we propose a novel CEPLESS middleware as seen in the middle tier, which provides programming interfaces to develop operators independently of the execution environments and to update them at runtime (cf. Chapter 6).

# 4

# Mechanism Transitions in Operator Placement

This chapter provides a solution to the *adaptivity problem* in the context of Complex Event Processing. By introducing adaptivity in the Operator Placement mechanisms of CEP, we aim to meet the changing Quality of Service requirements in the face of dynamic environmental conditions. Mechanism transitions is a promising and widely applied concept used in the research project Collaborative Research Centre "MAKI" to meet the quality requirements for communication systems [106, 57]. So far, it has been extensively applied for different problems [108, 109, 110, 111, 112, 113, 114]. However, it is still not known if transitions can fulfill *changing* QoS requirements in a highly challenging and dynamic environment of Internet-of-Things comprising of multiple distributed and stateful entities. This chapter aims to fill this gap by proposing a programming model and methodology for mechanism transitions in CEP systems to adapt to the dynamic environmental conditions and QoS requirements.

*Key Research gap: Changing QoS requirements*

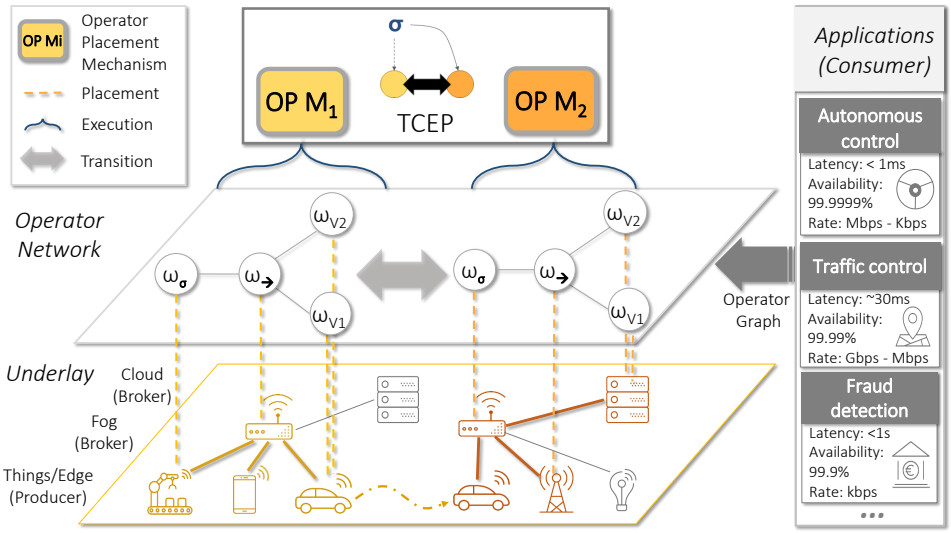



Figure 15: Overall system architecture with the focus on TCEP showing mechanism transitions from OP mechanism $M_1$ to $M_2$ to meet the changes in the QoS requirements.

*Solving the adaptivity problem*

Therefore, to solve the *adaptivity problem*, we propose a first Transition-capable CEP system, named TCEP, that integrates the proposed programming model and algorithms into a distributed CEP system. To understand the research challenges addressed in this chapter, recall the overall architecture introduced in Chapter 3 with TCEP-centric components as seen in Figure 15. On the right side of the figure, we illustrate the exemplary IoT applications with the respective QoS requirements, such as latency that has to be fulfilled by the TCEP system. In particular, we focus on an Operator Placement (OP) mechanism that aims to map operator graph, shown in the Operator Network tier, to the in-network resources, shown in the Underlay, aiming to meet the specified QoS requirements. However, conflicting QoS requirements that changes at runtime cannot be met using a single OP mechanism.

*Quick recap on state-of-the-art*

Moreover, OP mechanisms are specialized for given environmental conditions, such as a mechanism developed for a fixed network might not work well under highly mobile conditions. As pointed out in Chapter 2: Section 2.1.3, the existing OP mechanisms [43, 80, 44, 84] and adaptive CEP systems [87, 85, 76, 86, 32, 89] fall short in adaptively meeting the conflicting and changing QoS requirements of IoT applications. In addition, current CEP programming models [103, 9, 28] are either focused on providing better operator abstractions [103, 9] or combining push- and pull-based communication mechanisms [28]. Nevertheless, hardly any existing CEP programming model focused on providing programming abstractions for specifying OP mechanism to deal with heterogeneous needs of applications. These observations lead us to our first research question and sub questions, answered in this chapter.

*First Research Question and its sub RQs*

*RQ1: How to specify and adapt between OP mechanisms in the face of dynamic environmental conditions and QoS requirements?*

*RQ1.1* How to adaptively select an OP mechanism based on the QoS requirements?

*RQ1.2* How to realize mechanism transitions in a live and seamless manner?

*RQ1.3* When to perform a transition such that its costs are minimal?

*RQ1.4* How to specify distinct OP mechanisms and its performance characteristics?

*Challenge 1: Careful choice of OP mechanism*

*(RQ1.1)* A careful choice of OP mechanism is required so that the transition is able to meet the specified QoS requirement after a change. Naively approaching this problem would alleviate the costs of transition due to a frequent change to a new mechanism, as well as result into a violation of the QoS requirements. Therefore, as a first contribution a *lightweight learning*-based adaptive selection of OP algorithm is proposed. The selection algorithm

aims to capture the performance of OP mechanisms for the observed QoS requirements to indicate the *best* OP mechanism that fulfills the specified requirement. *(RQ1.2)* Changing an OP mechanism at runtime requires us to move operators, commonly referred as operator migrations, to new broker nodes that might lead into disruption in the output events. Such disruptions during the operator migrations result into: *(i)* QoS requirement violations, *(ii)* missing important events that eventually reduces the correctness of the output events, and *(iii)* monetary costs for applications such as fraud detection due to missing fraud events. Thus, as a second contribution, we provide novel *transition strategies* that ensures the delivery of output events is done in a live, seamless, and correct manner such that no disruption is observed and consistency in delivery output events is maintained. We analytically prove that the strategies possess the aforementioned properties of liveness, seamlessness, and correctness. *(RQ1.3)* A transition to a new OP mechanism is costly due to operator state migrations that consume ample amount of resources. These costs have to be minimized, otherwise the transition to a new OP mechanism might not be worth due to the consequences of the resources consumed. For this, we propose transition strategies that minimizes the costs in terms of operator migrations by determining *optimal* and *discrete time steps* to perform operator migrations. The time steps are selected such that the operator state that has to be migrated at that time is minimal. *(RQ1.4)* Finally, an integration of novel and existing OP mechanisms is required to allow swift transitions between them. So far, OP mechanisms for specific applications with a specific QoS requirement have been proposed, which are not transition-capable. In our final contribution, we provide a novel programming model to specify OP mechanisms, regardless, how different they are in terms of their functionality, QoS requirements and application domains. Moreover, their deployment on the heterogeneous fog-cloud infrastructure is provided such that operator graphs can be executed independently on any underlying infrastructure.

*Challenge 2: Operator migrations might cause output disruption*

*Challenge 3: Transitions are costly*

*Challenge 4: Specification of adaptable OP mechanisms*

The findings presented in this chapter are based on previous publications in [196, 205, 25, 206, 12, 207]. The structure of this chapter is explained as follows. Section 4.1 explains the need for transitions using a traffic congestion query referred consistently in the chapter to explain the relevant concepts. Section 4.2 formalizes the mechanism *transition* problem and the cost associated in performing a transition. Section 4.3 presents the overall system design of TCEP. Section 4.4 presents the evaluation of transitions. TCEP and its placement programming model are publicly available at `https://luthramanisha.github.io/TCEP/` together with the state-of-the-art OP mechanisms and novel transition strategies for reuse by researchers and industry application developers.

*Key publications on TCEP and structure*

*Publicly available, try it out!*

## 4.1 Analysis of Adaptivity in OP Mechanisms

This section analyzes the ability of OP mechanisms to adapt for QoS requirements using the traffic control scenario previously introduced in Chapter 3: Section 3.1 ①. In the following, first, we use a query of traffic congestion to explain the dynamicity in the environment conditions and hence the QoS requirements. Second, using a preliminary analysis on the query, we show for the scenario important shortcomings of current OP mechanisms that prove our hypothesis.

*Illustrative example of traffic congestion query*

Figure 16 illustrates the running example of traffic congestion highlighting the different environmental conditions. We use a continuous query[22] to specify that a road section on a crossing is congested, as seen in Listing 1. Any event consumer can register a query to a specific road section on the crossing, say `SectionV1`. Consumers could be, for instance, nearby vehicles, emergency services, or intelligent traffic lights. The query specifies conditions, *Condition 1:* high traffic density and low vehicle speed on `SectionV1` (shaded in red with lines) and *Condition 2:* low traffic density and high speed in its crossed road section, say `SectionV2` (shaded in green). In the query, the complex data streams `vehiclesAtSectionV1` and `vehiclesAtSectionV2` are presumed to contain information on the average speed and density, for instance, by using aggregated data from radar sensors and road side units (cf. Section 3.1). The query uses a sequence (Line 9) of conditions for `SectionV1` (Lines 5–7) and `SectionV2` (Lines 10–12) in order to generate a complex event "congestion of road `SectionV1`". The complex event: "congestion of road `SectionV1`" is detected whenever the sequence of conditions on `SectionV1` and `SectionV2` in a temporal timespan of one minute (Line 14) indicating (i) dense traffic and slow vehicles for `SectionV1` (indicating a congestion) and (ii) sparse traffic and fast vehicles for `SectionV2` (indicating a smooth flow of vehicles), respectively, are met.

The query execution can be distributed to the available deployment resources such as on to IoT infrastructure like vehicles. The communication can be established using techniques like V2X [209] and cellular communication [191] or using nearby fog-cloud resources to deliver the required performance. The mapping of the query to such in-network resources is done using an OP mechanism that must account for the `QoS_REQT` specified within the query (Line 16). The latency requirements with a threshold ($\tau_{qos}$) such as `latency < 30ms` can be provided in the query specification[22] (e.g., given in Figure 15).

[22] Using the ADAPTIVECEP query language [208] detailed later in the TCEP programming model (cf. Section 4.3.3).

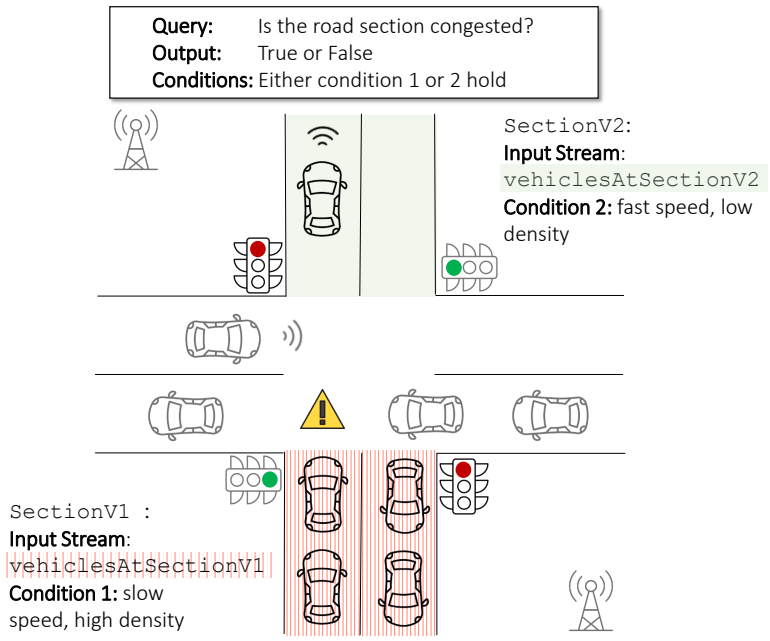



Figure 16: Traffic control scenario representing dynamic environmental conditions with *Condition 1:* slow moving cars and high density (congested road) and *Condition 2:* fast moving cars and low density in road sections (smooth running vehicles) `SectionV1` and `SectionV2`, respectively. The traffic congestion is represented using a query specified in Listing 1.

```
1   case class VehiclesAtSection(sectionId: Int, avgVehiclesDensity: Long,
        avgVehiclesSpeed Long, time: Long)
2   val vehiclesAtSectionV1: Stream[VehiclesAtSection] = ...
3   val vehiclesAtSectionV2: Stream[VehiclesAtSection] = ...
4   val congestedAdjacentRoadSections = Query[RoadSections]
5     ((vehiclesAtSectionV1 where { v1 =>
6        v1.avgVehiclesSpeed < NormalSpeedThreshold &&
7        v1.avgVehiclesDensity > HighTrafficThreshold
8      })
9      ->
10     (vehiclesAtSectionV2 where { v2 =>
11       v2.avgVehiclesSpeed > NormalSpeedThreshold &&
12       v2.avgVehiclesDensity < HighTrafficThreshold
13     })
14     within 1.min
15     where { case (v1, v2) => v2.time > v1.time }
16     demand QOS_REQT)
```

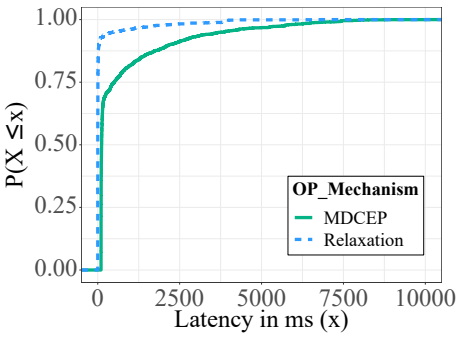
Listing 1: Traffic congestion detection query used to express change in environmental conditions and the user requirements.

According to our hypothesis the same OP mechanism cannot accommodate conflicting QoS requirements. Thus, we analyze the ability to fulfill two conflicting QoS requirements for the query provided in Listing 1 using two popular state-of-the-art OP placement mechanisms: *Relaxation* [15] and *MD-*

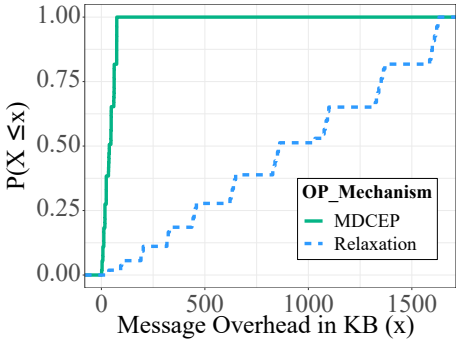
*CEP* [80]. The basic idea of the *Relaxation* OP mechanism is to use a so-called latency space to map operators using Vivaldi coordinates [202]. The latency space allows estimation of communication delays between resources used for operator placement, which is used to find a near-optimal mapping of the operator graph. In contrast, *MDCEP* avoids maintaining such cost space and concentrates on minimizing overhead, focusing on applications with high dynamics. Hence, the OP decision is based on resources that are within the proximity of the data sources to achieve mapping of the operator graph with low control message overhead.

*Hypothesis: single OP mechanism cannot meet conflicting QoS*

We perform a preliminary analysis of the OP mechanisms for the traffic congestion query in a dynamic environment with mobile Things (such as vehicles) and varying workload (number of queries). The performance is measured in terms of two QoS metrics concerning the OP mechanisms: *(i)* end-to-end latency defined as the overall time to perform the congestion detection (cf. Definition 6), and *(ii)* control message overhead defines the number of control messages sent to assign all the operators of a query to the brokers (cf. Definition 7).




(a) Relaxation supersedes in terms of end-to-end latency.




(b) MDCEP supersedes in terms of control message overhead.

Figure 17: Performance comparison of Relaxation [15] and MDCEP [80] OP mechanisms for 50 incrementally deployed queries [25, 196].

*Preliminary analysis verifies no one size fits all*

Figure 17 shows the measurements for the two metrics under a dynamic environment for 50 incrementally placed traffic congestion queries given in Listing 1. Further details on the specification and evaluation with multiple OP mechanisms can be found in Section 4.4.2. Figure 17a illustrates the end-to-end latency results using a cumulative distribution function (CDF) under incrementally placed traffic congestion queries. It shows that the *Relaxation* OP mechanism consistently performs better than *MDCEP* with latency lower than 100 ms for the maximum number of query workload (up to 80 % workload).

These findings are consistent with the article where the original OP mechanism is proposed [15]. However, Figure 17b shows that *Relaxation* incurs

higher control message overhead compared to *MDCEP* due to the maintainence of the latency space. In contrast, *MDCEP* incurs only a little overhead for all the queries in the order of a few bytes, making it suitable for highly dynamic environments, yet suffer from a higher latency, ∼7.5 s on average.

Therefore, using the above preliminary evaluation, it can be derived that conflicting QoS demands are hard to be fulfilled using the same OP mechanism. In principle, dynamic environmental conditions, as seen in this example, need different OP mechanisms to fulfill the changes in QoS requirements at runtime. For a fixed network environment or relatively static scenario, we measured a significantly lower latency for the *Relaxation* mechanism and hence it can be used to achieve low latency in condition 1 (cf. Figure 16). In contrast, when the condition changes to be highly mobile, we measured significantly lower message overhead for the *MDCEP* mechanism, and thus it can be used to deliver complex events with less overhead in condition 2. This proves our hypothesis that as the environment condition changes, a transition from *Relaxation* to *MDCEP* becomes imperative.

## 4.2 Transition Problem Formulation

This section presents an extension to the common system model defined in Chapter 3: Section 3.2.1. We model the entities required to define the design of TCEP system in Section 4.2.1 and provide the transition problem statement in Section 4.2.2.

### 4.2.1 *Extended System Model*

In the following, we provide the overall system model for TCEP and the model for OP mechanisms and transitions.

*Place operators on heterogeneous fog-cloud resources*

In the TCEP system, the operator graphs can be deployed on to a set of distributed nodes comprising the fog-cloud infrastructure referred to as broker nodes. The availability of this infrastructure allows serving many queries at once and change between resources to fulfill the dynamic QoS requirements of applications. The infrastructure possesses realistic resources such as CPU, memory, etc., and are known in advance as in a real-world fog-cloud setup. It comprises of the nodes that continuously execute the *Empty Apps* (cf. Chapter 3: Section 3.2.1). An Empty App serves as an execution environment for the operators executing in TCEP to readily execute and migrate operator graphs. Furthermore, we formally define the transition model and the cost metrics to measure the influence of transition ($C_{Time}(T)$ and $C_{Overhead}(T)$) on

the performance of the CEP system and the overall optimization problem are formulated in the following section.

### *OP Mechanism and Transition Model*

The TCEP system follows a modular design and allows transitions between multiple distinct CEP mechanisms to be integrated. Hence, we modelled TCEP such that it can execute transitions between *any* CEP mechanism. However, we focus in this chapter only on OP mechanisms. Therefore, TCEP provides a composition of multiple OP mechanisms $M_1, M_2, \ldots, M_{max^M}$. In the following we present the model for OP mechanism and a transition.

#### *Operator Placement Model*

**Definition 14.** *Operator Placement mechanism.* An OP mechanism determines *where* and *how* to map an operator graph $G$ onto a number of broker nodes $B = \{b_1, b_2, \ldots, b_{num^b}\}$ in the deployment infrastructure.

After the mapping of operators ($\omega \in \Omega$) on the brokers ($b \in B$), it forms a so-called operator network. We define the mapping on the operator network as follows:

$$\alpha : \Omega \times B \rightarrow \{0,1\}, s.t.$$
$$\alpha_{i,j} = \begin{cases} 1, & \text{if } \omega_i \text{ is placed on } b_j \\ 0, & \text{if } \omega_i \text{ is not placed on } b_j. \end{cases} \tag{3}$$

#### *Transition Model*

As pointed out earlier in Chapter 2: Section 2.1.2, the abstract concept of a *transition* between the mechanisms within a communication system is formalized by the members of Collaborative Research Centre "MAKI" in [104, 57, 105, 106]. In this work, we enhance the transition methodology to deliver important properties required for an adaptive and distributed CEP system, such as *liveness*, *seamlessness*, and *correctness* in the delivery of complex events. Furthermore, we address key challenges related to performing operator migrations as discussed later in Chapter 4. In the following, first we define the concept of transition using the standard definition in Collaborative Research Centre "MAKI" [57, 105, 106]. Afterwards, we introduce specific properties for an adaptive CEP system that this thesis contributes.

**Definition 15.** *Transition.* A transition $T$ performs a switch from a mechanism $M_A$ to $M_B$, e.g., OP mechanisms at run time. Formally, it is denoted as $T : M_A \rightarrow M_B$.

As a consequence of a transition, the given set of operators migrate from the previously deployed set of brokers, so-called source brokers ($b_{src}$) to the new so-called target brokers ($b_{trg}$), that are selected by the changed OP mechanism. While executing a transition and performing operator migrations, the delivery of complex events must not be interrupted. Thus, a transition must ensure some properties like *liveness* and *seamlessness*. We define these properties in consistent with the definitions used in the context of distributed systems [210] as follows.
*Liveness.* A transition ensures the *liveness* property iff once triggered; the system continues to deliver complex events, and that the transition eventually terminates.
*Seamlessness.* A transition is *seamless* iff events are delivered to the consumers with the required QoS specified by the consumer, such that there is no downtime or a violation in the QoS requirement.

Since the transitions are assumed to be resource consuming, there might occur latency or throughput spikes during the delivery of events. Such occurences are not desired and, while the transition still fulfills the liveness property it is not seamless in every case. Yet, if there are interruptions, for example, downtime during the execution, the transition is neither live nor seamless. Therefore, if a transition ensures seamlessness then it should ensure liveness but not everytime vice versa is true.

### 4.2.2 *Transition Problem Statement*

*Determine optimal time steps so that...*

Consider a query $Q = \{q, QoS, T_{QoS}\}$ with QoS requirements that can be placed using $max^M$-different OP mechanisms on the fog-cloud infrastructure. Here, $q$ represents the continuous query, $QoS$ is the set of QoS requirements $qos \in QoS$, $T_{QoS}$ is the set of threshold $\tau_{qos} \in T_{QoS}$ for each QoS requirement and $max^M$ is the number of OP mechanisms available for transition. Based on the environmental conditions $en_1(t)$ at time $t$ and $en_2(t+1)$ at time $t+1$, the QoS requirements change, say $qos_{1|en(t)}$ to $qos_{2|en(t+1)}$. Consequently, the cost of a placement in terms of resource demands such as network costs and the ability to fulfill the QoS requirements also changes over the time.

The TCEP system aims to ensure that the given QoS requirements are fulfilled despite the changes in the environmental conditions by utilizing different OP mechanisms and the fog-cloud infrastructure. Thus, the system

determines for changing environmental conditions with time, say $en_1(t)$ to $en_2(t+1)$ and corresponding QoS requirements $qos_{1|en(t)}$ and $qos_{2|en(t+1)}$, a sequence of points in time, say $t_1, \ldots, t_{num^t}$ and a sequence of OP mechanisms $M(t_1), \ldots, M(t_{num^t})$, on which a transition $T_i : M(t_i) \rightarrow M(t_{i+1})$ is initiated at time $t_i$.

Typically, a pair of OP mechanisms will use different nodes for placement. Therefore, the transition between OP mechanisms requires several operator migrations, which move the operators from one physical node to another. Consequently, transitions impose a significant cost when the state of an operator graph that has to be migrated is very large. Furthermore, to ensure live and seamless transition, state migrations have to occur in a cost-efficient manner.

*...the costs are minimum*

We formalize the *transition problem* using an objective function comprising two key cost factors, namely, the costs incurred in terms of $C_{Time}(T_i)$: transition time and $C_{Overhead}(T_i)$: transition overhead. Here, the transition time $Time(T_i)$ is defined as the total time taken to perform a transition. This comprises of *(i)* $Time_{select}$: time to select a new target OP mechanism $M(t_{i+1})$, *(ii)* $Time_{\alpha}$: time to find a node for placement $\alpha$ (cf. Equation (3)) dependent on $M(t_{i+1})$, and *(iii)* $Time_{mig.(\omega_j)}$: time to migrate $j$ operators $\omega_j \in \Omega$, $\forall j \in [1, max_{\omega^j}]$ to the target brokers dependent on the mapping function $\alpha$. Therefore, the total cost in terms of transition time is given as follows:

$$C_{Time}(T_i) = Time_{select} + Time_{\alpha} + \sum_{j=1}^{max_{\omega^j}} Time_{mig.(\omega_j)}. \quad (4)$$

Similarly, the transition overhead is derived as the total number of messages exchanged while performing a transition. This comprises of overhead in *(i)* $Overhead_{select}$: selection of a placement mechanism, *(ii)* $Overhead_{\alpha}$: performing the operator placement, and *(iii)* $Overhead_{mig.(\omega_j)}$): migration of the operators including their state.

$$C_{Overhead}(T_i) = Overhead_{select} + Overhead_{\alpha} + \sum_{j=1}^{max_{\omega^j}} Overhead_{mig.(\omega_j)}. \quad (5)$$

In this work, the optimization transition problem minimizes a weighted sum of normalized transition time ($\hat{C}_{Time}(T_i)$) and transition overhead ($\hat{C}_{Overhead}(T_i)$), such that the QoS requirements under the changing environmental conditions can be met. Given the (normalized) costs of operator migrations for $j$

operators, $j \in [1, max_{\omega^j}]$ in terms of time $\hat{C}_{Time}(T_i)$ and overhead $\hat{C}_{Overhead}(T_i)$, the optimization problem aims to minimize the costs as follows:

$$\begin{aligned} &\min \ \left[\beta_t \cdot \hat{C}_{Time}(T_i) + \beta_o \cdot \hat{C}_{Overhead}(T_i)\right] \\ &s.t.\ \alpha(t) \text{ satisfies } QoS_{|en(t)} \text{ under the execution of } T_1, \ldots, T_n \\ &C_{Time}(T_i), C_{Overhead}(T_i), QoS_{|en(t)} \in \mathbb{R}^+. \end{aligned} \tag{6}$$

Here, $\beta_t,\ \beta_o \geq 0,\ \beta_t + \beta_o = 1$, represent weights for transition time and overhead, respectively.

## 4.3 The TCEP System Design

*Four goals: Low costs, seamlessness, correctness, and easy specification*

TCEP proposes the following models and algorithms to accomplish the four main research goals addressing each research question discussed previously. *(i) (RQ1.1)* A *lightweight* genetic learning-based algorithm that keeps the learning costs as minimum and predicts the best possible OP mechanism that meets the given QoS requirement. *(ii) (RQ1.2)* Transition execution algorithms that perform operator migrations such that the output events are delivered without any interruption (*live* and *seamless* manner while maintaining the correctness). *(iii) (RQ1.3)* A strategy to find out optimal time steps to perform operator migrations when migration costs are *minimal*. *(iv) (RQ1.4)* A programming model to specify queries and adaptable OP mechanisms and enables integration of different OP mechanisms into the given placement library provided by the system.

In the following, before introducing the specific models and algorithms as listed above, we first provide a conceptual overview on the TCEP system components. Then, we introduce the decentralized MAPE-K adaptation loop used for transitions in TCEP.

### *Conceptual Overview*

*Introduction to the main components of* TCEP

Figure 18 illustrates the four key components of the TCEP system to solve the transition problem. The *Deployment Infrastructure* layer includes the event consumers, the event producers and the event brokers. Recall, it comprises of the edge-fog-cloud infrastructure that is used for the deployment of operator graphs. The *TCEP Engine* layer provides a programming model to specify and initiate queries and OP mechanisms that can be used to place operator graphs on the deployment infrastructure. The OP mechanisms are stored in the so-called placement library explained in the *Control* layer. Furthermore, the *Engine* layer includes mechanisms to monitor *(i)* the environmental con-

ditions of the IoT applications that trigger adaptations, *(ii)* QoS requirements and the threshold defined by the query ($\tau_{qos}$), and *(iii)* the performance of the OP mechanisms in terms of the QoS metrics.

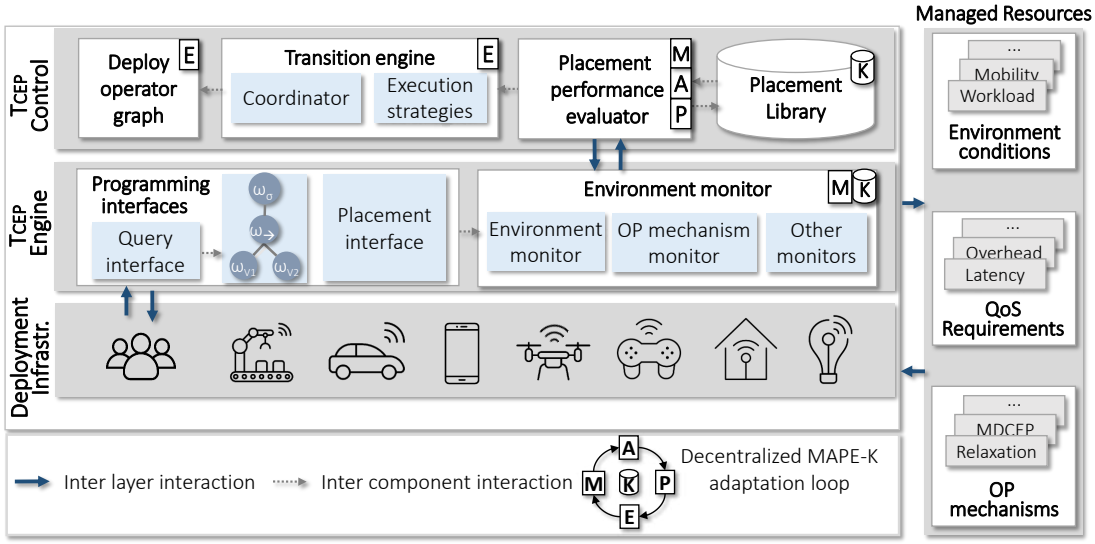



Figure 18: The TCEP system design [196].

The *TCEP Control* layer uses and manages a library of OP mechanisms curated by the programming model. Moreover, the state-of-the-art OP mechanisms can be integrated in the placement library for adaptations. It also provides a *transition engine* that includes a transition coordinator, which initiates and manages a transition, and execution strategies specify how to perform a transition. The *placement performance evaluator* decides on the target OP mechanism for the transition that fulfills the QoS requirement using the input from the monitoring component of the *Engine* layer. The *deploy operator graph* component performs the actual deployment of the operator graph on the IoT infrastructures.

Finally, the *Managed Resources* component specifies the resources monitored and controlled by the TCEP components, particularly the environmental conditions, the QoS requirements of the query, and the OP mechanisms. Once a continuous query $Q$ containing the QoS requirements with thresholds (e.g., $qos < \tau_{qos}$) are specified by an event consumer, the TCEP system initiates the placement of the query to deliver the results while meeting the QoS requirements efficiently. The query is first transformed into a logical operator graph by the TCEP engine, which monitors the performance of the query along with other criteria such as environmental conditions. Once the *placement performance evaluator* has chosen an appropriate OP mechanism, the operator graph is deployed on the infrastructure. As the TCEP system detects a change in the environmental condition and the specified QoS require-

ment of the continuous query $Q$, the *transition engine* triggers a transition that is managed by a decentralized MAPE-K adaptation loop (cf. Chapter 2: Section 2.1.2) explained as follows.

*Decentralized MAPE-K Adaptation Loop*

For the adaptation decisions in a transition, TCEP follows the well-known *MAPE-K* [107] loop. The four processes of the loop, *Monitor* (M), *Analyze* (A), *Plan* (P), *Execute* (E), and *Knowledge* (K) are realized in a decentralized manner (cf. Figure 18) in the control layer and within the TCEP engine to manage the resources depicted in the lowest layer. In the following, we provide the definitions of these components in TCEP.

*Decentralized MAPE-K loop for transitions*

*Monitor (M)* function provides the required mechanisms to collect, aggregate, filter, and report details on the managed resources [107] (given by the *Managed Resources* component). Examples of monitoring information are environmental conditions such as mobility of cars and workload; performance metrics related to the query such as latency and bandwidth observed on the links; and performance characteristics of the transition such as time and overhead. Hence, decentralized monitoring components lie within the *environment monitor* and *placement performance evaluator* components, which are responsible for collecting and aggregating the above monitoring information.

*Analyze (A)* provides mechanisms that correlate and provides basis for adaptations in a system. These mechanisms allow the *transition engine* to learn about the managed resources and predict future situations. For instance, the *placement performance evaluator* implements a fitness score mechanism that measures the performance of the OP mechanism, which is used to predict the next suitable OP mechanism for the respective environmental conditions.

*Plan (P)* provides mechanisms that construct the actions needed to fulfill the QoS requirements of the query. For instance, the *placement performance evaluator* determines if a change to a new OP mechanism would help fulfill the QoS requirements.

*Execute (E)* provides mechanisms to manage the necessary changes required for the transition. This component is responsible for carrying out the transition itself. For instance, the transition coordinator generates a plan on the operator graph transition, and the execution strategies perform the transition.

*Knowledge (K)* component stores the data shared across the above four functions. This includes OP mechanisms in the placement library and monitoring information on the performance, among others.

In relation to Figure 18, in the following Section 4.3.1 explains the *placement performance evaluator*, Section 4.3.2 explains the *transition engine* and Section 4.3.3 explains the *programming interfaces* component.

### 4.3.1 *Placement Performance Evaluator*

This component measures and maintains the performance statistics on the OP mechanism execution. A heuristic fitness score is proposed that aims in the selection of the best OP mechanism that suits the current environmental conditions and meets the QoS requirements. After all OP mechanisms received a score, a learning-based selection method is proposed that selects an OP mechanism based on the statistics collected during the current execution. In the following, we discuss the design decisions and highlight alternatives. Afterwards, we present the heuristic fitness score and the adaptive selection method.

*Why not alternative approaches?*

A plausible way to determine an OP mechanism for execution is to apply offline learning. Given a query $Q$, QoS requirements and the environmental conditions, a performance-influence model of the OP mechanism can be developed that predicts the best OP mechanism with respect to the given QoS requirements and environmental conditions [211]. However this method has significant limitations as follows. *(i)* The performance influence model developed offline would be very specific to the characteristics of the deployment infrastructure used for placement such as network characteristics that changes frequently over the time in a dynamic environment. Such changes in the infrastructure at runtime would influence the behaviour of the OP mechanism and hence render the model useless. *(ii)* Offline learning of such a performance influence model is unrealistic for IoT environment which is resource constrained. Often such learning models need long training time and large amount of training data, which consumes many resources. A way to mitigate the first limitation would be to update the performance influence model using incremental learning [212, 213]. However, this method still has to maintain the learning model that has to be updated with time, which takes quite many resources, and it relies on the offline learned model, which again consumes resources to be obtained. Other online learning methods like reinforcement learning that do not need a prior performance model cannot be used as they become computationally expensive as the state and action space becomes larger.

In contrast, we need a learning algorithm that is able to determine the best OP mechanism in an online way based on the recent performance of the mechanisms while being less computationally expensive. Therefore, we employ a *lightweight online learning* algorithm to statistically determine the

target OP mechanism that best meets the QoS requirements based on the genetic learning algorithm [214]. It is an efficient and robust search algorithm that combines the survival of the fittest[23] with a structured yet randomized information exchange between the entities involved in the population, in our problem this entity is an OP mechanism [214]. Here, *lightweight* refers to the fact that our learning method does not rely on offline learning or any prior training dataset. In contrast, it uses statistics that are collected online during the query execution. The online learning method is complemented by a ranking method based on the selection strategy of genetic algorithms, which determines the OP mechanism with the best performance compared to other mechanisms in the placement library. The transition (and the selection of OP mechanism) is triggered by the *environment monitor* component of the TCEP *Engine* layer (cf. Figure 18), which reports a violation of the QoS requirement to this component. During the initial placement, when no empirical statistics on the performance are available, the target OP mechanism is determined based on the specified QoS requirement of the query and the objective function of the OP mechanism. In case the objective function of more than one OP mechanism matches, the selection is performed in a round-robin fashion until sufficient statistics on the available OP mechanisms are available.

*Lightweight learning to select best OP mechanism*

The following section defines the heuristic fitness score that makes the available OP mechanisms comparable. After defining the score, a ranking method utilizing the fitness score to derive the best available OP mechanism for the QoS requirement is defined.

### *Heuristic Fitness Score for OP Mechanism*

We describe the derivation of the fitness score by measuring the performance of the OP mechanism for the currently executing continuous query. The measurement is taken at periodic intervals and is used as a basis for comparison to the other OP mechanisms. The performance of the OP mechanism is measured in terms of the current QoS requirement of the query. Precisely, the heuristic fitness function captures an objective to maximize the function of the number of times the QoS requirement is fulfilled. Formally, for each OP mechanism, the fitness function measures the number of times the QoS requirement is fulfilled. *Example:* say $num_{qos_j}$ is the measurement of number of times a given QoS requirement $qos_j$ is fulfilled by OP mechanism $M_i$ and $max_{qos_j}$ is the maximum count of QoS requirement fulfillment from all the available statistics $max^M$ of OP mechanisms. If $num_{qos_j} > max_{qos_j}$ holds, the respective OP mechanism will be given the highest possible rank. The measurement is performed between the time interval when the query was first submitted ($t_s$) until when the transition is triggered ($t_t$). The fitness score is

*Fitness score to make OP mechanisms comparable*

[23] Fittest is the measure of how fit a given OP mechanism is based on the fitness score.

updated at regular intervals until the next transition to consider all the QoS requirements of the query, for instance, due to the change in environmental condition. In essence, the score measures how well the OP mechanism performs throughout query execution time, compared to the OP mechanisms in execution before (starting when the query was first submitted $t_s$).

The ultimate goal for the score is to find the best OP mechanism for the respective QoS requirements by utilizing the collected performance information. In particular, the goal is accomplished by maintaining the scores for the respective OP mechanism $M_{i,qos_j}(t_t)$ for each QoS requirement $qos_j$, such that it is updated at the occurrence of each transition at time $t_t$. Since, an OP mechanism can aim for multiple QoS requirements (e.g., in a multi-objective function), the score is updated separately for each QoS requirement. A score function for each OP mechanism $M_i$, $Score_{M_{i,qos_j}}(t_t)$ is obtained based on the performance statistics for the respective query and each QoS requirement $qos_j$. The *mean normalization method* is applied to normalize the score $M_{i,qos_j}(t_t)$ for each OP mechanism to make them comparable. The fitness score is computed based on the statistics collected during the execution of OP mechanism $M_i$ (with subscript $i$), which is then compared to the statistics of other OP mechanisms since the query was first submitted (during the time interval $t_s$ – $t_t$) given as $t_{s,t}$ in the following Equation 7:

$$M_{i,qos_j}(t_t) = \frac{\mu_{i,qos_j}(t_{s,t}) - \mu_{qos_j}(t_{s,t})}{max_{qos_j}(t_{s,t}) - min_{qos_j}(t_{s,t})} \cdot (1 - decay) + M_{i,qos_j}(t_t - 1) \cdot decay. \tag{7}$$

In the above equation, $\mu_{qos_j}(t_{s,t})$, $max_{qos_j}(t_{s,t})$, and $min_{qos_j}(t_{s,t})$ denote the mean, maximum and minimum score for *all* the OP mechanisms, respectively, during the time interval $t_s$ – $t_t$ for the QoS requirement $qos_j$. The notation $\mu_{i,qos_j}(t_{s,t})$ indicates the mean score of OP mechanism $M_i$ for the time interval $t_s$ – $t_t$ during the QoS requirement $qos_j$. $M_{i,qos_j}(t_t - 1)$ is the previous score of OP mechanism $M_i$ with a $decay$ factor that is used to exponentially reduce the influence of older statistics so that recently collected statistics is given more priority. Therefore, the decay factor ranges $[0, 0.5]$, such that more preference is given to current statistics. The initial value of decay is set to $0$, which is updated once a transition is performed by a factor dependent on the number of OP mechanisms to be explored. For instance, if there are $10$ OP mechanisms, then the decay is incremented by $0.05$.

The overall score is computed by combining the statistics collected for each QoS requirement $qos_j$ by the OP mechanism. Again, it is the normalized score

for each QoS requirement $qos_j \in [qos_1, qos_2, \ldots, qos_{max^q}]$, where $max^q$ is the total number of QoS requirements considered by OP mechanism $M_i$, as follows:

$$Score_{(Mi)}(t_t) = \sum_{j=1}^{max^q} M_{i,qos_j}(t_{s,t}). \tag{8}$$

### *Adaptive Selection of an OP Mechanism*

This section describes a selection method that enables the *placement performance evaluator* to change an OP mechanism, once all of the mechanisms have received a fitness value (as discussed in the previous section). This enables to keep the overall transition cost low in terms of time taken by appropriately balancing between the exploration vs exploitation of OP mechanisms. In the following, we first describe the issues tackled during the selection process. Second, we explain the input to the selection method, followed by the description of the selection method.

*Main issues addressed while selecting an OP mechanism*

After each OP mechanism got assigned a score as described in the previous section, the *placement performance evaluator* can decide if the current OP mechanism $M_i$ can be used again or changing to another OP mechanism yields better performance. This is done by comparing the score of $M_i$ with the scores of the other mechanisms that were in execution so far. To do this, we use a simple Radix sort algorithm that sorts the OP mechanisms in linear time so that the comparison is cheap. The following issues are considered while selecting the next OP mechanism. *(i)* In the beginning, we allow some degree of exploration so that all the OP mechanisms get a chance to prove themselves. Therefore, a round-robin selection is used for the adaptive selection of an OP mechanism initially. Moreover, we allow exploration of alternate OP mechanisms at random intervals during the execution to give a chance to perhaps better-performing OP mechanism. *(ii)* Adapting too often might cause oscillations (back and forth) while also skewing the results of the used OP mechanism. Therefore, we empirically set the delay threshold between consecutive transitions to give the new OP mechanism enough time so that the *performance evaluator* can correctly assess its behaviour.

There are several learning-based selection methods in Genetic Algorithms that can be applied to decide if it would be beneficial to change the OP mechanism. These methods take as an input only the fitness score values. In the following analysis, a larger fitness value is considered better and hence a fitness maximization is assumed. Since the OP mechanisms in the placement library are constant, the number of fitness values is a finite value $f_1, f_2, \ldots, f_F, (F \leq max^f)$. The state of the placement library can be described by the function $s(f_k)$ that represents the number of occurences of the fitness

value $f_k$ in the OP mechanisms of the placement library. The fitness distribution is defined as a function $s(f)$ that assigns each OP mechanism $M_i$ in the placement library, a fitness value $f_k$. Moreover, $max^f$ is the size of the fitness distribution $s(f)$.

We use a *Linear Ranking Selection method* for this purpose [214] because: *(i)* it enables a relative analysis of the OP mechanisms suitable for the proposed heuristic fitness score, and *(ii)* it avoids selecting worse OP mechanism by preferring exploitation over exploration. In particular, the ranking method compares the fitness values, which are used to decide the best OP mechanism. Furthermore, it prefers exploitation by applying an appropriate selection pressure defined as the intensity of search focused on finding the best OP mechanism [215]. Reducing the selection pressure focuses on increasing the diversity of OP mechanism selection, for instance, by the selection of worse OP mechanism. Contrarily, increasing the selection pressure focuses on the reduced search space of selected best OP mechanisms. This explains the idea of exploration vs exploitation using the ranking method. Theoretically, using the linear ranking method, we can compute the appropriate selection pressure $\mathcal{S}$ using the average fitness distribution $\overline{\mathcal{M}}$ before selection, and expected average fitness distribution $\overline{\mathcal{M}^*}$, for the given fitness values $f_1, \ldots, f_F, (F \leq max^f)$ as follows:

*We maintain balance between exploration and exploitation*

$$\overline{\mathcal{M}} = \frac{1}{max^f} \sum_{k=f_1}^{f_F} \overline{s}(f)$$

$$\overline{\mathcal{M}^*} = \frac{1}{max^f} \sum_{k=f_1}^{f_F} \overline{s^*}(f)$$

$$\mathcal{S} = \frac{\overline{\mathcal{M}^*} - \overline{\mathcal{M}}}{\overline{\sigma}}. \tag{9}$$

Here, $\overline{s}(f)$ and $\overline{s^*}(f)$ are the fitness distribution and expected fitness distribution of the OP mechanisms, respectively. The notation $max^f$ denotes the size of the fitness distribution, and $\overline{\sigma}$ denotes the standard deviation of the fitness distribution $\overline{s}(f)$. All the functions assumed to be continuous are denoted with an $\overline{overline}$, and the fitness values for the OP mechanisms are assumed to be sorted ($f_1 < f \leq f_F$). Afterwards, the OP mechanisms are assigned ranks based on the sorted fitness values. The best OP mechanism is assigned a rank of $\mathcal{N}$, while the worst is assigned rank 1. For selection, the OP mechanisms are linearly assigned the selection probability $P_i$ according to the rank as follows:

$$P_i = \frac{1}{\mathcal{N}} \left( \eta^- + (\eta^+ - \eta^-) \frac{i-1}{\mathcal{N}-1} \right); i \in [1, \mathcal{N}]. \tag{10}$$

Here, $\frac{\eta^-}{\mathcal{N}}$ denotes the selection probability of the worst OP mechanism, while $\frac{\eta^+}{\mathcal{N}}$ denotes the selection probability of the best OP mechanism. At a given time the OP mechanisms are fixed in the placement library, therefore the conditions $\eta^+ = 2 - \eta^-$ and $\eta^- \geq 0$ apply. The selection probability also provides a means to provide distinct probabilities if the OP mechanisms are ranked the same, for instance, when they have the same fitness score [215]. Hence, the selection probabilities are directly proportional to the fitness scores of the OP mechanism as given by the probability distribution function $\eta$ in the following:

$$\eta_i = \frac{Score_{(M_i)}}{\sum_i Score_{(M_i)}}. \tag{11}$$

The selection probabilities of the worst and the best OP mechanism is calculated as the minimum and maximum of the probability distribution function $\eta$, respectively. The reduced search space is updated each time an OP mechanism is selected. As a result, the OP mechanism, which is well-performing, gets a higher probability of selection that ensures exploitation is preferred over exploration. However, at times, we also select the worst OP mechanisms to update their scores (exploration). This is achieved by applying an appropriate selection pressure depending on the collected statistics so far. The selection pressure for the ranking method can be computed [215] using the assumption that the fitness distribution follows a Gaussian distribution, the Equation (13) and the Proof as shown in Appendix A.1.3.

After selecting an appropriate OP mechanism, the realization of the transition concept is performed by the *transition engine* detailed in the next subsection.

### 4.3.2 *Transition Engine*

*Main goals for a transition...*

The main goals of the transition engine are *(i)* the coordination of the transition, i.e., preparing to start, end and plan the transition, such that the time taken to transit is minimum, *(ii)* execute a transition by performing cost-efficient, live, and seamless operator migrations[24], and *(iii)* ensure that delivered output events to the consumers are correct. We present the lifecycle of a transition based on previous work [105] with a distinction of handling

[24] Recall, that it is a process to move operators from one physical node to another.

state migrations in a cost-efficient manner, ensuring that the transition process is seamless, and ensuring the correctness in delivery of complex events. In the following, first, we look into the high-level requirements for the transition process and then present the two transition execution strategies that fulfills the above goals. A transition in a CEP system between OP mechanisms involves multiple distributed entities because of the distributed nature of the OP problem. Hence, a transition has to be coordinated consistently between these entities. Therefore, a *coordinator* maintains and orchestrates the transition lifecycle. The *coordinator* interacts with the transition execution strategies that carry out the process of transition. The main novelty of the proposed transition execution strategies lies in the properties *liveness*, *seamlessness* and the low *cost* (cf. Section 4.2.2) incurred in terms of time and overhead of the transitions.

*... cost-efficient, seamless and correct delivery*

As soon as the target OP mechanism is selected by the *placement performance evaluator* component (cf. Section 4.3.1), a set of target broker nodes are identified for the new placement. All the operators in the graph have to migrate to the target broker nodes to comply with the new placement logic. It is primary to keep satisfying the QoS requirements of the query while performing the operator migrations. Operator migrations in this realm have been widely studied in the literature, such as stop and restart strategies [16, 52] and partial pause and resume strategies [24, 216]. Here, the former completely stops the execution to migrate the operator to start executing at a target broker, while the latter partially pauses the execution of the concerned operator only. However, none of the approaches addresses seamless and cost-efficient operator migrations while using multiple OP mechanisms. We specifically optimize the costs in terms of performing the transitions for two metrics: *time* (cf. Equation (4)) and *overhead* (cf. Equation (5)) to achieve cost-efficient operator migrations, which is achieved by the transition execution strategies. We propose two strategies for this purpose which comprise of two steps: *(i)* coordinate transition and *(ii)* determine and perform operator migrations.

#### *Moving Fine-Grained State (MFGS) Sequential Transition*

This strategy initiates operator migrations in a specific order, bottom-up, to handle the dependencies between the operators in the operator graph (cf. Algorithm 1: Lines 1-14). An operator is migrated iff all its predecessors in the operator have been migrated. The dependency order in the operator graph follows a bottom-up order, where leaf operators (e.g., stream operator) are predecessors of their connected operators that are successors, as we go up in the operator graph (cf. Operator graph model in Chapter 3: Section 3.2.1). Since multiple operators have to be migrated, the migrations occur in a se-

quential and breadth-first manner such that only one operator is migrated at a time to the target broker (Lines 2-3).

In the next step, the strategy determines the target broker where the operator is to be migrated. This happens with the help of the target OP mechanism, which is previously determined (cf. Section 4.3.1) (Line 5). Therefore, depending on the decision of the OP mechanism, an operator $\omega$ may or may not be transferred to a new target broker (Line 6-7). A minimal state of the operator is retained referred to as *intermediate-state*, to keep the operator migration time and overhead as very low as discussed in detail in the next paragraph (Line 8). Finally, the state is migrated to the target broker, and the input data stream is received at the target broker, which executes the operator using the minimal state. To receive the incoming data stream from the producers and the predecessors that were previously migrated, a subscription from the target broker is sent to them (Line 9). Once the operator is successfully migrated, an acknowledgment of the migration is sent by the target broker alongwith the sequence number of the first complex event to the source broker and the coordinator (Line 10). Afterwards, the source broker stops executing the operator as it continues its execution on the target broker (Line 11).

*Maintain an intermediate state for efficient migration*

**Algorithm 1 :** Moving Fine-Grained State Sequential Transition [196].

**Variables :**

| | | |
|---|---|---|
| $\Omega List$ | $\leftarrow$ | bottom-up list of set of operators |
| $\omega_{cur}$ | $\leftarrow$ | current operator to be migrated |
| $PList$ | $\leftarrow$ | list of producers connected to $\omega$ |
| $targetOPMechanism$ | $\leftarrow$ | target OP mechanism |
| $Trg$ | $\leftarrow$ | target broker host of $\omega$ |
| $\phi_{Int}$ | $\leftarrow$ | intermediate state of $\omega$ |

```
1  function INIT-MFGS-SEQUENTIALTRANSITION()
2      ΩList ← BOTTOMUPASLIST(Ω);
3      MFGS-SEQUENTIALALGORITHM(ΩList.HEAD, targetOPMechanism)
4  function MFGS-SEQUENTIALTRANSITION(ω_cur, targetOPMechanism)
5      Trg ← targetOPMechanism.FINDTRG(ω_cur);
6      if Trg ≠ ω_cur.SOURCEBROKER then
7          ω_cur.COPYEXECUTIONENVIRONMENT(Trg);
8          φ_Int. ← ω_cur.COMPUTEINTERMEDIATESTATE();
9          Trg.STARTEXECUTIONWITHDATA(PList, φ_Int.);
10         if ω_cur.NEXT().RECEIVEDACK(timeout, retries) then
11             STOPEXECUTION(ω_cur.SOURCEBROKER);
12             MFGS-SEQUENTIALTRANSITION(ω_cur.NEXT(), targetOPMechanism);
13         else
14             MFGS-SEQUENTIALTRANSITION(ω_cur, targetOPMechanism);
```

This strategy performs sequential migrations until all the operators are migrated, which marks the end of the transition. This happens using a recursive method that traverses the operator graph in a bottom-up order while

migrating the operators (Line 12). In case the operator migration is not successful for some reason, for instance, the target broker becomes unavailable, and the acknowledgement is not received, the operator migration is repeated (Line 14). In paragraph below we detail how the intermediate buffer is used to keep the cost of operator migrations minimum. In Line 14, we assume a consumer specified parameter $m$ that determines the maximum number of repetitions[25] of this loop and guarantees termination after $m$ tries.

*Cost-efficient Operator Graph Transition*

This strategy computes a minimal *intermediate state* to reduce the costs for the transition of an operator that must be migrated. The strategy builds on the operator state model proposed by Wermund et al. [34, 217], which allows retrieving an intermediate state required for the transfer. We improve on the operator state model by concurrent coordination of operator graphs that enable minimal cost for operator migrations as discussed in the following.

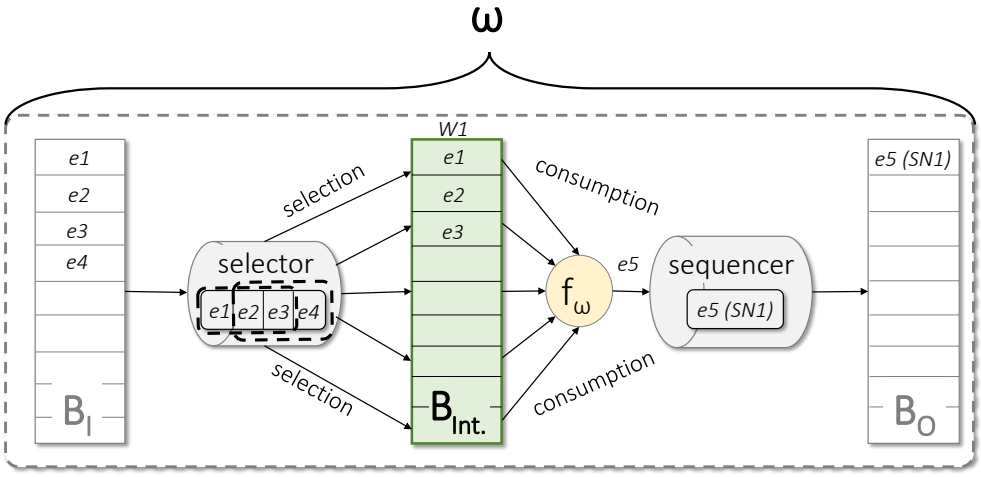



Figure 19: Intermediate buffer represented in the operator state model adapted from [217]. Here, $\{e_1, e_2, e_3, e_4\}$ are incoming events in the data stream selected based on the window size $w = 3$ by the selector. $W1$ is the first window processed by the $f_\omega$ resulting in outgoing event $e_5$ with sequence number $SN1$ [196].

Figure 19 explains the operator state model, comprising an input buffer $B_I$, output buffer $B_O$, selector, sequencer, and processing function $f_\omega$ of the operator. The input events $\{e_1, e_2, e_3, e_4\}$ are retrieved from the incoming data stream by the producer in the input buffer $B_I$. The selector indicates events to be processed based on the selection policy of the operator, as explained in Chapter 3: Section 3.2.1, e.g., using a sliding window, say $\{e_1, e_2, e_3\}$. The function $f_\omega$ processes the selected events of window $W1$, and the output

[25]This is very unlikely to happen that the target node is not found again and again.

events are evicted based on the consumption policy, e.g., using the slide size of the sliding window. The sequencer appends each output or complex event with a sequence number inserted into the output buffer. The complex event is then delivered to $\omega$'s successor. Once the successor acknowledges the event, it is removed from the output buffer $B_O$.

Traditionally, in a CEP system, the state of an operator $\phi_\omega$ comprises: *(i)* the input buffer $B_I$, *(ii)* the *selector*, *(iii)* the correlation function $f_\omega$, *(iv)* the *sequencer*, and *(v)* the output buffer $B_O$. Instead, we reduce the amount of state to only the content of the intermediate buffer $B_{Int}$. The intermediate buffer $B_{Int}$ stores the events based on the selection policy that are required at the target broker to resume the operator and start its execution. We explain the role of the above state components using a sliding window operator example, which is one of the most important *stateful* operator (cf. Chapter 3: Section 3.2.1).

*Example:* Figure 19 gives an example of a sliding window operator of size $w$, say three events $\{e_1, e_2, e_3\}$ which are selected by the selector and stored in the $B_{Int}$. In the first transformation step these events are processed by the processing function $f_\omega$ releasing the output event $e_5$. In the next transformation step, the slide size $s$ is applied, say one event, and the next window is stored in the intermediate buffer $B_{Int}$, i.e., $\{e_2, e_3, e_4\}$. Therefore, the events required to be processed in the next transformation step are only sent to the target broker to resume processing.

Furthermore, it is essential that the target broker timely subscribes to the incoming data streams at the producer or the node of the predecessor operator in the operator graph. This is to ensure the correctness of the events as well as to keep the transition time low. Note that the intermediate state $\phi_\omega(t_{opt})$ migrated at time $t_{opt}$ includes the intermediate buffer $B_{Int}$, the processing function $f_\omega$, and the sequence number of the last evicted complex event (Line 9). The $B_{Int}$ stores the state of the operator to be processed at the next transformation step. After the last complex event is evicted (identified by matching the sequence number), the target broker starts processing the operator in parallel. In particular, at time $t_{opt} - \delta_M$, the target broker starts executing the operator based on the input events received from the producer or the predecessor operator. Here, $\delta_M$ is a small value to make sure that the target broker starts the execution beforehand. However, the duplicate events, due to the concurrent processing by both the source and target brokers, are discarded (Line 10). It is important to note that a careful selection of $\delta_M$ value is essential so that the target broker does not miss any input event. In case the value is very big, there will be an overlap in the execution of the source and the target broker. The duplicates are discarded; however, it results in an unnecessary overhead that should be avoided.

On the other hand, if the $\delta_M$ value is very small, there is a slight chance that the target broker might miss some of the input events. However, this is very unlikely to happen. Nevertheless, we address this problem by proposing a seamless transition algorithm where the state overhead is further minimized and the correctness of the events is guaranteed, as discussed in the following subsection.

*Cost Analysis*

*Linear time complexity*

We analyze the transition time and present an asymptotic upper bound on the cost ($C_{Time}(T)$). The transition time is bounded by the time required by the algorithm to iterate over all operators sequentially and to transfer the intermediate state of each operator (Lines 9 to 12). Therefore, the overall transfer time can be bounded by the transfer time of the entire intermediate operator $\phi_\Omega$ plus the time to iterate over all operators which yields $\mathcal{O}(|\Omega| + |\phi_\Omega|)$ Here, $\phi_\Omega$ is the intermediate state, and $\Omega$ is the set of operators[26]. The strategy reduces this time by transferring a minimum amount of state $\phi_{Int}$ using an intermediate buffer. Although only a minimal state is required to be transferred, the state transfer still involves costs in time and resources. Furthermore, the sequential transfer of operators is time consuming. While transferring operators in a sequential manner consumes fewer network resources, it is another factor that takes time. Therefore, to solve these issues, we propose the following transition strategy.

*but.. sequential and requires state transfer*

***Seamless Minimal State (SMS) Concurrent Transition***

*Mitigate issues by concurrent and seamless migrations*

This strategy differs from the MFGS transition in two ways: *(i)* concurrent operator migrations and execution and *(ii)* seamless execution of migrations. Algorithm 2: Lines 1-16 presents the strategy where multiple operators are migrated to the target broker simultaneously. In particular, the coordinator performs at most $2^l$ operator migrations (for binary operator graph) at each level $l = 0$ to $m$ in a bottom-up manner (Line 2). This order ensures the integrity of the operator graph execution. The concurrent operator migrations contributes to the cost in terms of runtime. The coordinator starts operator migration by first transferring the execution environment (Line 5). This is readily available by using the TCEP container as provided in Definition 5 in the form of *Empty Apps*. Afterwards, the coordinator determines an optimal discrete time step $t_{opt}$ for each operator $\omega$. The time signifies when the operator state comprises only the new event tuples, which are already available at the target broker so that the operator execution can be resumed from the

[26] $\Omega$ here stands for the set of operators as previously defined in the notations, not to be confused with the generic notation on asymptotic lower bound of an algorithm.

target broker (Line 7). Essentially, the optimal time $t_{opt}$ is determined by waiting for each operator $\omega$ to purge from its old state (Line 8), i.e., until $B_{Int}$ and $f_\omega$ do not hold any old state. To better understand how $t_{opt}$ is calculated, consider an example of a window operator.

*Select optimal time steps when anticipated costs are low*

**Algorithm 2 :** Seamless Minimal State Concurrent Transition [196].

**Variables :**

| | | |
|---|---|---|
| $PList$ | $\leftarrow$ | list of producers connected to $\omega$ |
| $OGlevel$ | $\leftarrow$ | operator graph level for migration |
| $targetOPMechanism$ | $\leftarrow$ | target OP mechanism |
| $Trg$ | $\leftarrow$ | target broker node of $\omega$ |
| $\phi_{sequencer}$ | $\leftarrow$ | state of sequencer |
| $waitTime$ | $\leftarrow$ | wait time for the current operator until it is purged from its old state |

```
1  function SMS-ConcurrentTransition(OGlevel, targetOPMechanism)
2      for all ω ∈ OGlevel do in parallel
3          Trg ← targetOPMechanism.findTrg(ω);
4          if Trg ≠ ω.sourceBroker then
5              ω.copyExecutionEnvironment(Trg);
6              NTPClockSynchronization(Trg, ω.sourceBroker);
7              minimalStateTime ← ω.determineMinimalStateTime();
8              waitTime ← waitUntil(minimalStateTime));
9              φ_sequencer ← ω.lastSequenceNumber;
10             Trg.StartExecutionWithData(PList, φ_sequencer);
11             Trg.determineReferencePoint(minimalStateTime);
12             if ω.parent().receivedACK(timeout, retries) then
13                 StopExecution(ω.sourceBroker);
14                 SMS-ConcurrentTransition(OGlevel.next(), targetOPMechanism);
15             else
16                 SMS-ConcurrentTransition(OGlevel, targetOPMechanism);
```

*Example:* A window operator waits until all the old event tuples have been processed, say $w + \delta_S$. Here $w$ is the size of the window and $\delta_S$ is a small value such that $t_{opt}$ time is selected to be after a given time when events are still received from the source broker. This allows execution of the window at the target broker after the source broker. The time $t_{opt}$ is chosen to be the *transition start time*. In other words, the target broker waits until the last old event of the window is processed, and this time is chosen as *transition start time* for a given operator ($t_{optmin}(\omega)$).

It is important to note, to ensure the correctness of complex events, both source and target brokers run concurrently until all events in the input buffer are new. In particular, the target broker of the current operator starts executing with the minimal state (the last $SN$ of the complex event) concurrently at the transition start time $t_{optmin}(\omega)$, while the successor operators in the operator graph hierarchy still execute on source broker using the placement logic

of former OP mechanism. Thus, in this strategy, the transition coordinator also allows coexisting execution of two OP mechanisms.

*Seamless and Concurrent Operator Graph Transition*

Consider an operator graph from our example scenario in Figure 20 to understand the concurrent operator graph transition. In the figure, *Src* represents the source broker, and *Trg* represents the target broker in Step 1 – 5, through the initial placement at the source to the final placement at the target after migration. The operator migration starts with the leaf operators in Step 1, which starts operating at target brokers in Step 2. This is because the operators $\omega_{V1}$ and $\omega_{V2}$ are stateless and do not have previous state to resume execution at the target broker. We assume clock synchronization using standard Network Time Protocol (NTP) [197] at both source and target brokers (Line 6) to ensure that the two clocks do not diverge. Steps 3 and 4 show the $B_{Int}$ buffer of the sequence operator with the event tuples being processed. In these steps, the operator graph is concurrently processed at the source and target brokers until the intermediate buffer has all new tuples. Therefore, the concurrent execution of operator graph and coexisting OP mechanisms at source and target brokers enable seamlessness in transition. Furthermore, the concurrent operator migrations do not interfere with each other, and the transition is accomplished atomically in the TCEP transition engine.

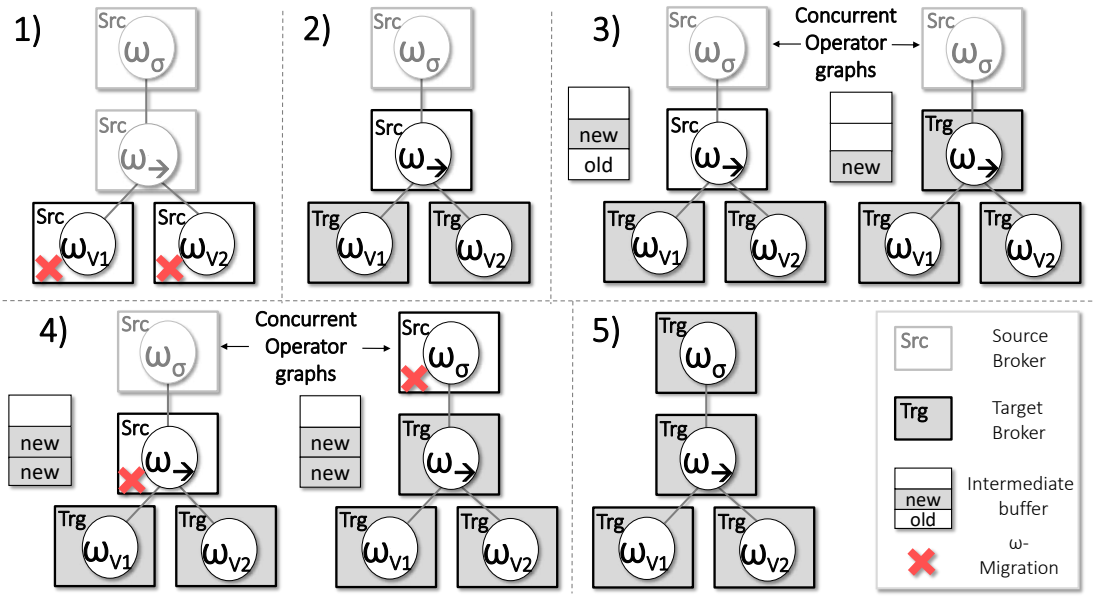



Figure 20: The order in which concurrent operator migrations and execution are performed in SMS strategy. In step 3 and 4, we have concurrently running operator graphs. The red cross (✕) indicates an operator migration from source (Src) to target (Trg) broker [196].

The basic idea of this transition strategy is that at the transition start time $t_{optmin}(\omega)$, the input buffer $B_I$ and the output buffer $B_O$ are shared among the source and the target brokers until it is safe to discard the source broker. Both source and target broker for a stateful operator $\omega$ run concurrently so that all the old tuples in the intermediate buffer $B_{Int}$ of the source broker are gradually purged (cf. step 3 and 4). For instance, in a sliding window operator with a window size of $w$ events and slide size of $s$ events, the old tuples ($w - s$ events) have to be processed until the target broker has received a full window size of $w$ events to ensure completeness. During this time, the output is continually produced by both the brokers, while duplicates are discarded using the reference point method [218]. When the intermediate buffer is purged completely, then the source broker is discarded. This is because the target broker now has all the new tuples that exist in the source broker. The source brokers of stateless operators are gradually replaced by their targets, marked with red cross (✕) in Figure 20.

*Correctness Properties*

We assess correctness on two aspects as widely done in the literature [16], [219]: the output is complete, and there are no duplicates in the output. Figure 20 shows the transfer of the operator graph in 1) through 5) steps using the SMS algorithm. The stateless operators are transferred straightaway, while stateful operators run in parallel using the SMS algorithm until all the old tuples are purged. Furthermore, while the predecessor operators are migrated, successors still use the former OP mechanism for resolving the query. We must also ensure that there are no duplicate output tuples, as we can see in step 3): the sequence operator leads to duplicate output tuples from the source and target operator, respectively. A naive approach is to discard all the input as old tuples that results from the source broker. However, this would lead to incorrect results, as seen in step 3): the old tuple might be a true sequence that will remain undetected if dropped. To solve this issue, though we have the source and target brokers in execution concurrently, we drop events from target brokers unless all the events in the $B_{Int}$ buffer are new and the source broker could be stopped. For instance, in Step 3, we retrieve the output result from the source broker holding the sequence operator, while in step 4, we can safely discard the source broker since all the tuples in the state are new.

*Correct delivery of events and no duplicates*

*Cost of Concurrent Operator Graph Transition*

We present an analysis of the cost of concurrent operator graph transition by comparing the position of the system before and after the transition. In particular, we look at the input events and the state at the $b_{src}$ and $b_{trg}$ brokers after

*Log time complexity*

the transition start time $t_{optmin}(\omega)$. In this transition strategy, for each operator $\omega$, the state $\phi_\omega(t_{optmin}(\omega))$ at $t_{optmin}(\omega)$ consists of the sequencer state (Algorithm 2: Line 9). It contains the $SN$ of the last event delivered by the $b_{src}$ broker, which hints at the $SN$ of the first event that the $b_{trg}$ broker must deliver. This is achieved by partitioning the transition at discrete time steps such that each operator migration $m_i$ in a transition $T$ determines the *transition start time* as explained before. Thus, ensuring a live and seamless transition because of the minimal consumption of resources. Based on the concurrent migrations of operators, the number of operators in the $b_{trg}$ broker increases exponentially over time, with the increase in the size of the operator graph $G$. Therefore, the total transition time is bounded within $\mathcal{O}\left(log(|\Omega|) + C\right)$, where $C$ is $|\phi_{sequencer}|$ that is the state of sequencer, which is constant for a fixed size of operator graph $G$.

### 4.3.3 *TCEP Programming Model*

*Programming model for adaptable OP mechanisms*

The placement programming model of TCEP provides a means to specify adaptable OP mechanisms that play a key role in the performance of a deployed query in the transition-capable TCEP system. Furthermore, it provides a way for researchers and application developers to develop CEP queries for distinct IoT applications. Existing works [121, 30, 82] focus on proposing OP mechanisms optimizing for a diversity of QoS requirements. The growing interest in novel OP mechanisms becomes obvious due to their application in many practical problems in the field of CEP and Stream Processing [20, 84, 79]. Nevertheless, there is no substantial work focused on providing a common programming model to develop adaptable OP mechanisms and validating them in heterogeneous infrastructure, such as fog-clouds. Modern CEP systems such as Apache Flink [54], Storm [9], and Spark [52] focus on providing programming abstractions for operators. However, none of them provides a programming model for specifying and executing a diverse range of OP mechanisms across large-scale IoT infrastructures[27].

This section introduces the major components of the TCEP programming model as follows. *(i) QoS monitors* that is an integral part of the programming model as each OP mechanism observes some QoS metrics. *(ii) OP interface* provides methods to develop unique OP mechanism. *(iii) Query interface* provides standard operators from stream processing and CEP. In addition, so-called QoS operators are provided that helps specifying QoS requirements.

[27] For a detailed discussion, we refer the readers to Table 7.

| Method | Description |
|---|---|
| `getPlacementMetrics()` | Determines the QoS requirements that must be optimized |
| `configurePlacement()` | Resets placement parameters (called initially and on reconfiguration) |
| `findPlacementNode()` | Finds placement node based on the QoS metrics |
| `findPossibleNodesToDeploy()` | Retrieves all nodes that can host operators |
| `initialVirtualOperatorPlacement()` | Centralized mechanisms treat all operators at once during the initial placement instead of one-by-one by using a heuristic to find optimal locations in the virtual latency space |

Table 6: Novel TCEP placement API for developing placement mechanisms [205].

### *QoS Monitors*

The TCEP placement programming model characterizes the OP mechanisms based on the placement decision into two main categories: *(i)* centralized and *(ii)* decentralized as prominently done in the literature [82, 30, 220]. A centralized placement mechanism assumes global knowledge on the network and the nodes. In contrast, a decentralized mechanism assumes only partial knowledge of the network. For instance, a cluster head assigns an operator on each node of the cluster. It is known that finding an optimal placement from the number of possible resources is an NP-complete problem [39]. Furthermore, the assignment varies with the QoS requirements in consideration for the cost objective function. Hence, there exist many solutions and heuristics towards the OP problem.

Both centralized and decentralized placement heuristics assume monitoring knowledge of the network and node information. The TCEP programming model provides explicit extensible monitors for commonly used network and node information metrics, such as latency, bandwidth, and CPU load. These metrics are measured from end-to-end, meaning the cumulative latency or bandwidth observed while data streams traverse the path from producer to consumer. The measurements are accumulated step by step and, hence, individual measurements can also be fetched easily. The monitoring information is collected by every node separately and aggregated on the decision node based on the placement characteristics. In a centralized OP mechanism, the QoS monitors transfer the observed metric to a centralized node responsible for the placement decision. Whereas for decentralized mechanisms, we provide monitoring based on coordinate system, such as Vivaldi [202], which is prominently used in several OP mechanisms [15, 40, 19, 121]. It allows handling the dissemination of monitoring information for better placement decisions.

### *Operator Placement Mechanism Interface*

Table 6 lists the foremost API of the TCEP programming model used to implement OP mechanisms in TCEP. The `PlacementStrategy` API defines these methods for OP mechanism in order *(i)* to formulate a single objective and multi-objective optimization function for centralized OP mechanism, *(ii)* to define heuristics for decentralized OP mechanism, and *(iii)* to make OP mechanism exchangeable at runtime to enable transitions. An OP mechanism represents a cost objective function dictating the QoS requirements. An example of a cost objective function is to minimize the end-to-end latency from the producers to the consumers. Each mechanism, centralized or decentralized, must define a cost objective function for the QoS requirements that need to be optimized. The cost objective function can comprise a single or multiple QoS requirements, e.g., latency, CPU load, and bandwidth utilization. The objective function depends on the runtime measurements from the QoS monitors defined above, which are used to determine placement decisions on physical hosts of the fog-cloud infrastructure. The placement coordinator fetches this information from the QoS monitors for placement using `getPlacementMetrics` (cf. Table 6). This serves as an input to the cost objective function. The specific way in which the optimization problem is solved optimally or sub-optimally using heuristics is defined in the specific implementations of the OP mechanism. In Table 7, we define the currently available implementations of OP mechanisms in TCEP. Different OP mechanisms use different ways to optimize. For instance, Relaxation uses a spring relaxation method [15], while MOPA uses an approximation for the Weber problem [40]. However, both optimized for the same QoS metric bandwidth-delay product. Also, in optimal solutions, the optimization problem can be solved using different methods depending on the nature of the objective function (convex or concave) and the scenario at hand. Hence, in the TCEP programming model, we segregate the implementation of a specific optimization approach of the OP mechanism from the common interfaces.

The placement parameters are initialized using `configurePlacement` method, which is invoked in the beginning and each reconfiguration, e.g., during periodic updates of the same OP mechanism. The `findPossibleNodesToDeploy` and `findPlacementNode` methods determine the possible nodes where the operator can be deployed depending on the cost function. Some centralized OP mechanisms behave differently when performing placement initially and on reconfiguration, such as the Relaxation [15] mechanism. This mechanism places all operators of the query at once inside the virtual coordinate space using `initialVirtualOperatorPlacement()` and the physical placement is performed using `findNode()` since no operator is physically deployed using virtual placement. However, on reconfiguration, only physical placement is changed. In

| OP Mechanism | Placement Decision | Optimization Goal | Approach | Section 4.4.2 |
|---|---|---|---|---|
| Relaxation [15] | Centralized | Bandwidth-delay$^2$ product (BDP) | Spring relaxation technique | *(i)* |
| MOPA [40] | Centralized | BDP | Approximation for Weber Problem | *(ii)* |
| Global Optimal | Centralized | BDP | Optimally finds node with minimum BDP | *(iii)* |
| MDCEP [80] | Decentralized | control message overhead, latency | Host operator on the nearest neighbors unless producer or consumer | *(iv)* |
| Producer-Consumer | Decentralized | hops | Always host on producer or consumer | *(v)* |
| Random | Decentralized | - | Random allocation | *(vi)* |

Table 7: Design space of OP mechanism [205].

contrast, decentralized mechanisms only implement the `findNode()` since their behaviour is the same during initial placement and transitions.

### *Query Interface*

TCEP builds on AdaptiveCEP [208] to specify CEP queries with QoS demands that represents IoT applications in TCEP. The queries provide expressiveness by enable specification of standard operators used in stream processing and CEP systems, specifically, *(i)* typical operators in event algebra, such as conjunction, disjunction, window, and sequence; *(ii)* aggregation operators, such as minimum, maximum, and average; and *(iii)* relational operators, such as selection, projection, and joins [208]. Queries are composed of one or more aforementioned operators forming a logical operator graph that is adaptively processed in TCEP. Section 4.3.3 lists major operators and their syntax as they are implemented in TCEP. Listing 1 in Section 4.1 gives an example of such a query.

In addition, we provide QoS operators that allow specification of QoS requirements in the query language. For instance, latency in the delivery of a complex event can be declared as `vehiclesAtSectionV1 demand (latency < 50.ms)`. Table 9a lists some examples for the supported QoS operators. The first row explains the general syntax to indicate QoS demands. The remaining rows list specific examples on QoS operators.

The QoS demands can be *conditional* as explained in Table 9b, so that the demand applies only when the given condition is met. For instance, in the

| Operator | Operator Syntax | Description |
|---|---|---|
| Stream source | *stream* | A `stream` operator is applied on a single producer which used to forward the data stream to the rest of the operators in the graph |
| Join | (*stream0* window *win0*) join (*stream1* window *win1*) on *join-condition* | A `join` operator combines two data streams from producers by bounding it using a `window` and a `join` condition |
| Filter | *query* where *predicate-function* | A filter operator (or `where` clause) is used to apply a condition specified by the *predicate* on a sub query, such as window. |
| Map | *query* map *mapping-function* | A `map` operator takes one attribute in a data stream and produces one attribute by applying a custom function. |
| Aggregation Min | *query* min *numeral-selection-function* | Aggregate operator like `min` finds a minimum value using a sub query, such as in a window. |
| Aggregation Max | *query* max *numeral-selection-function* | A `max` operator finds a maximum value using a sub query. |
| Aggregation Avg | *query* avg *numeral-selection-function* | A `avg` operator finds a aggregate average using a sub query. |
| Conjunction or Logical AND | *query0* $\wedge$ *query1* | A conjunction ($\wedge$) operator express the detection of an event if either event from `query0` or `query1` occur. |
| Disjunction or Logical OR | *query0* $\vee$ *query1* | A disjunction ($\vee$) operator specifies an exclusive-OR under the assumption that simulataneous events from `query0` and `query1` cannot occur. |
| Negation or Logical NOT | ! *query* | A negation operator (!) signifies no event resulting from `query` can occur. |
| Temporal Sequence | *query0* $\rightarrow$ *query1* | A sequence operator ($\rightarrow$) specifies events from both `query0` and `query1` should occur but in time order, i.e., event from `query0` happens before event from `query1`. |

Table 8: CEP Operators [208].

earlier latency example `vehiclesAtSectionV1 demand (latency < 50.ms) when proximity within 100.m`, the vehicles which are in 100 m proximity are only considered for inclusion in the generation of the complex event. Again the first row presents a general syntax to specify a condition for a QoS demand using the `when` clause and the second row specifies condition using the `only` clause. The remaining table presents examples on the conditions based on `proximity` and `frequency` of events. The conditions are used to limit the number of producers considered for processing the low-level events based on the location information or the event rate information.

| Demand | Demand Syntax |
|---|---|
| Demand on stream[a] | *stream* demand *predicate* $\tau_{qos}$ |
| Latency | *stream* latency < $\tau_{latency}$ |
| Throughput | *stream* throughput > $\tau_{throughput}$ |
| Bandwidth | *stream* bandwidth > $\tau_{bandwidth}$ |

(a) Quality demands.

[a]Recall, $\tau_{qos}$ is the threshold on QoS requirement as introduced previously in Section 3.2.1.

| Condition | Condition Syntax |
|---|---|
| Condition on quality demands | *demand* when *condition* |
| Condition on event producers | *stream* only *condition* |
| Proximity | proximity within *length*<br>proximity < *count* |
| Frequency | rate > *event-rate-threshold* |

(b) Quality demand conditions.

Table 9: QoS Operators [208].

## 4.4 Evaluation

We answer the following evaluation questions in relation to our main contributions in Section 4.3.1, 4.3.2, and 4.3.3.

*Main evaluation questions*

1. Is the programming model able to express transition-capable OP mechanisms?
2. Can changing and conflicting QoS requirements be satisfied using transition betweens OP mechanisms?
3. Do the transitions deliver complex events in a live and seamless manner?
4. What are the transition costs in terms of learning, transition time, and overhead?

The following sections answer the aforementioned evaluation questions. Section 4.4.2 evaluates the TCEP programming model in terms of development of OP mechanisms and validating their performance. Section 4.4.3 evaluates transitions between OP mechanisms in terms of meeting changing and conflicting QoS requirements. Section 4.4.4 evaluates the properties of transition and the costs imposed by the proposed transition execution strategies. Finally, Section 4.4.5 evaluates the cost imposed by transitions in terms of learning the performance of OP mechanism and selecting a mechanism.

Before that, the following sections present the evaluation environment and setup related to the implemented system TCEP, scenario modelling, and the deployment setup, as well as finally the evaluation findings.

| | |
|---|---|
| Number of producers | $1-8$ |
| Number of brokers | $1-10$ |
| Number of consumers | $1-2$ |
| Number of queries | $1-50$ |
| Type of queries | Stream, Filter, Conjunction, Join, Accident detection (Listing 1) |
| WINDOW_SIZE | $5, 10, 20, 30, 40, 50$ secs |
| QOS_REQTS | latency, message overhead, network usage, hops |
| OP mechanisms | Relaxation [15], MOPA algorithm [40], Mobile DCEP [30], Global Optimal, Producer Consumer, Random |
| Transition execution strategies | MFGS-Sequential, MFGS-Concurrent, SMS-Sequential, SMS-Concurrent |
| Selection strategies | Lightweight learning, Requirement-based |

Table 10: Configuration parameters for the evaluation. *Default values are underlined.*

### 4.4.1 *Evaluation Environment and Methodology*

This section describes the TCEP implementation, the dataset, queries, and the overall evaluation configuration and plots.

#### *Implementation*

TCEP *uses Akka and Esper CEP engine*

TCEP is implemented using Scala in a total of 749,047 lines of code. The runtime environment of TCEP is based on the Akka actor system [221] and Akka Cluster to build a distributed network for easy deployment in the fog-cloud scenario. TCEP uses AdaptiveCEP specification language [208] for dictating QoS requirements at run time (cf. Listing 1), and Esper [35] as a backend to run the queries. TCEP uses docker tools such as container, service, and swarm for the distributed deployment. We use Akka version 2.6.0 [221], the Esper CEP engine version 5.5.0 [35], and Docker version 19.03.8-ce [222]. TCEP is publicly available for use[28].

#### *Evaluation Platform*

We deploy Docker services on 8 virtual machines with 8 GiB of memory and 8 processors each on different physical machines as denoted in Figure 21.

[28] TCEP webpage: `https://luthramanisha.github.io/TCEP/` [Accessed in May 2021].

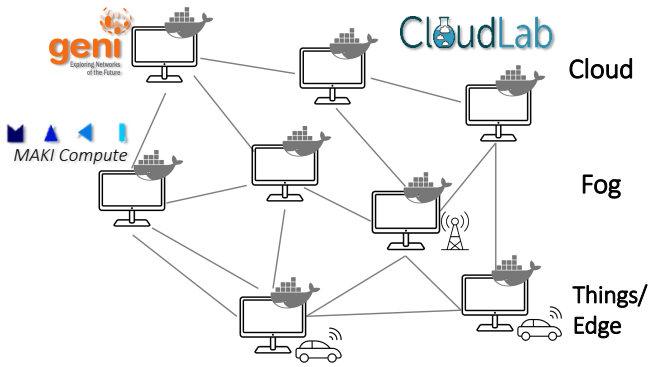



Figure 21: Our setup comprises of 8 physical machines using publicly available network infrastructures running our virtualized TCEP system in docker containers.

TCEP's Docker image is build upon the Alpine Linux distribution[29], which is much smaller in size (base image size of only 5 MB) and lightweight than other Linux based images. This is extremely useful that TCEP has a small memory footprint so that it can run on resource constrained IoT infrastructure. We consider different physical machines comprising deployment infrastructures GENI [223], CloudLab [56], and the onsite MAKI [57] compute machines. This infrastructure together provides a realistic deployment environment similar to the fog-cloud infrastructure resource model previously introduced in Section 3.2.1 and hierarchically illustrated in Figure 21. With resources dispersed in North America (Ohio and UCLA) and Europe (Darmstadt), we have introduced geographical diversity, realistic network latencies, and packet loss environment for our experiments [55]. The infrastructure nodes communicates using a Docker network setup.

*GENI and CloudLab distributed testbeds for evaluations*

### *Dataset*

We use a realistic dataset for a highly mobile vehicular network scenario from Madrid [224] comprising the input data stream with the event tuples: $< time, position, lane, speed >$. This helps in generating complex data streams of `vehiclesAtSectionV1` and `vehiclesAtSectionV2` (cf. Listing 1), as well as to evaluate the congestion detection query. In aggreement with our assumptions, the dataset includes at the rush hours a high in-flow of traffic and reduction in speeds indicating a traffic congestion. While during the normal hours, the traffic is regular with sparse traffic and high speeds.

*Real-world traffic scenario workload*

[29] Alpine Linux distribution in Docker `https://github.com/gliderlabs/docker-alpine` [Accessed in May 2021].

### *Queries*

Figure 22: Q1–Q5 illustrate the operator graphs of the queries used for the evaluation. Queries Q1–Q4 define the standard CEP queries written using the TCEP programming model (Section 4.3.3).

(Q1) `Stream` Operator

```
1     Stream => stream[StreamData](speedPublishers(1), demand QoS_REQT)
```

Q1 represents a Stream operator and acts as a forwarding query that forwards the data stream from the speed publisher or the vehicle to the consumer.

(Q2) `Filter` Operator

```
1     Filter => stream[StreamData](speedPublishers(1), demand QoS_REQT).where { v1 =>
2     v1.avgVehiclesSpeed < NormalSpeedThreshold}
```

Q2 is a filter query that detects a complex event indicating one of the conditions of a congestion. It verifies if the average speed from the vehicles contributing to a stream is less than a threshold.

(Q3) `Conjunction` Operator

```
1     Conjunction => stream[StreamData](speedPublishers(0)).and(stream[StreamData](
          speedPublishers(1)), requirement QoS_REQT)
```

Q3 indicates a Conjunction query on two data streams arriving from two publisher vehicles.

(Q4) `Join` Operator

```
1     Join => stream[StreamData](speedPublishers(0)).join(stream[StreamData](
          speedPublishers(1)), slidingWindow(5.seconds), slidingWindow(5.seconds)).
          where{ case (v1, v2) =>
2     v2.time > v1.time }, requirement QoS_REQT)
```

Q4 represents a complex event indicating that the second vehicle is behind the first vehicle useful to study traffic patterns. It performs a Join on a Window of two data streams such that the second vehicle enters after the first vehicle at the road section.

Finally, Q5 illustrates the traffic congestion detection query previously introduced in Section 4.1: Listing 1. We extend the query to take input as low level events from the dataset on speed and density and generate the complex data streams `vehiclesAtSectionV1` and `vehiclesAtSectionV2` used to detect a congestion. In the figure, Q5 comprises eight publishers each representing

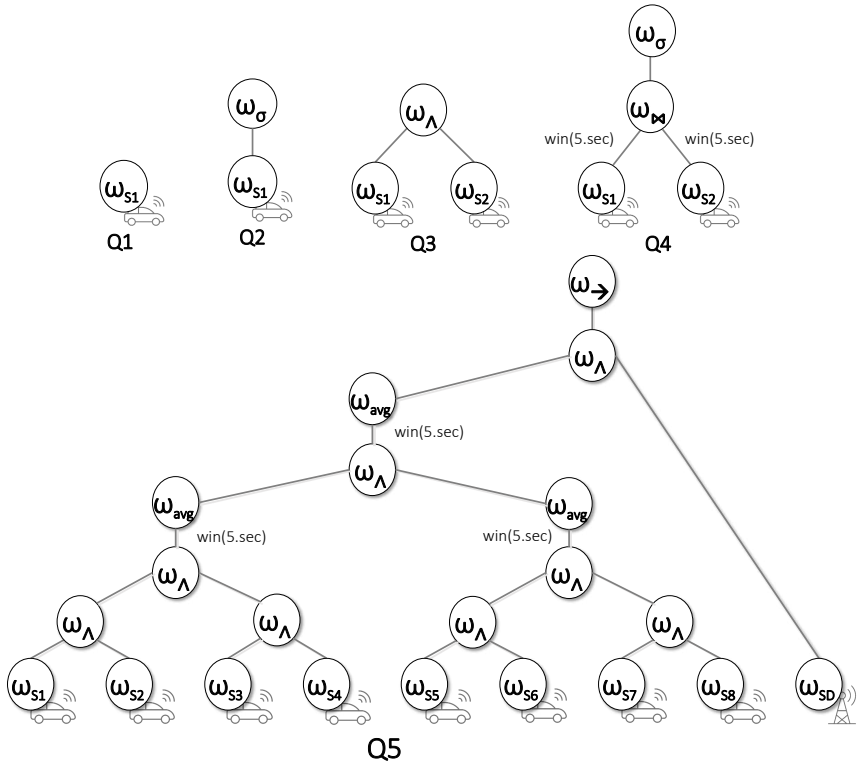



Figure 22: Operator graph for queries Q1 to Q5.

a Stream operator ($\omega_{S1}$ to $\omega_{S8}$). In the operator graph, the speed information related to the vehicle from the Stream operators are analyzed to get the Average speed of the two road sections each comprising four publishers. Another Stream operator ($\omega_{SD}$) contributes the density information related to the two road sections which is used to detect a sequence for the congestion detection using a Sequence operator ($\omega_{\rightarrow}$).

### *Evaluation Configuration and Plots*

Table 10 summarizes the above configuration used for evaluating the performance of TCEP. Each evaluation is done for $20$ minutes and the performance measurements are sampled after two minutes at a periodic interval of five seconds. To introduce dynamicity in the environment, we incrementally increase the query workload upto $50$ queries. In addition to the real-world deployment at the fog-cloud infrastructure, we use Mininet emulation [225] to introduce publisher mobility in the scenario. We divide the road into two sections with a length of 2500m each. The link between a publisher and the current base station is changed to the base station in the next section when it leaves the section. The mobility factor influences the end-to-end latency of an event considerably when the publisher moves to the next section. In the eval-

uations, we study the impact of multiple configuration parameters, including the number of queries, the mobility, and the window size. We evaluate the performance using six distinct OP mechanisms, four transition execution strategies, and two selection strategies. To gain the confidence interval, each evaluation is repeated 30 times if not stated differently. We use the Cumulative Distribution Function (CDF), line plots, and bar plots to illustrate the results. CDF shows the cumulative distribution of the investigated metric. It represents the probability distribution function $P(X \leq x)$ such that the value of the investigated metric is the probability that $X$ takes a value less than or equal to $x$. The line plot shows a thin line that depicts the mean ($50$th percentile) while the shaded areas report the percentiles including the values of the investigated metric between the $5$th and the $95$th percentile. Finally, the bar plots show the mean of the investigated metric alongwith the error bar on the top denoting the values between the $5$th and $95$th percentile.

### 4.4.2 *Operator Placement Mechanisms*

*Integrate six OP mechanisms*

This section shows the expressiveness of our programming model proposed in Section 4.3.3 by providing an implementation of *six distinct* OP mechanisms with a minimal effort. We focus on OP mechanisms with the objective function of the QoS requirements, which are widely investigated, end-to-end latency (cf. Definition 6), bandwidth-delay product or network usage (cf. Definition 8), CPU load (cf. Definition 10), hops (cf. Definition 9), and control message overhead (cf. Definition 7). There are two reasons behind this. First, most CEP systems concentrate on minimizing network costs while maintaining load so that operator graph processing is performed with minimum resource consumption. Second, highly dynamic scenarios require stable placement that can be achieved by minimizing the overall control overhead and performing OP as near to the producers and consumers as possible. In the following, we start by giving a brief description of the essential design characteristics of the implemented mechanisms. Afterwards, we provide an extensive evaluation of their performance to show *(i)* similar performance characteristics of OP mechanisms implemented using our programming model as originally proposed and *(ii)* our hypothesis that conflicting QoS requirements are hard to be fulfilled by a single OP mechanism.

#### *Adaptable Operator Placement Mechanisms*

All the OP mechanisms are made adaptable by integrating them to the MAPE-K adaptation loop. The main challenge therein is to plan and execute transitions by migrating the operator state, and to maintain the state consistently

across multiple distributed entities as previously discussed in the transition engine Section 4.3.2. In the following, we discuss the key differences between the selected OP mechanisms, their implementation, and the reason for their selection. The same is important to understand the performance of these mechanisms defined in the later sections. In addition, Section A.1.2 shows the realization of the OP mechanisms and the specified queries in a web interface that is used to demonstrate the behavior of transitions and the OP mechanisms.

*(i) Relaxation [15]*[30]. It is based on a so-called cost space that considers latency and bandwidth together as two dimensions. In this method, authors achieves placement of operators in two main steps. The first step is a virtual placement using a cost space representing network usage metric, while in the second step the operators are physically placed using a KNN algorithm. The basic idea behind the first step or the virtual placement, is a physics analogy revolving around springs. The distance by which a spring is extended resembles a link's latency, and the spring constant specifying its stiffness is the bandwidth of the link. The product of spring extension and spring constant is the force needed to extend the spring by Hooke's Law. Similarly, the product of latency and bandwidth is the bandwidth-delay product (BDP) or network usage metric. Another important point is that the Relaxation OP mechanism uses a squared latency bandwidth product to ensure a unique solution if the bandwidth observed is equal. Operators are connected by springs that pull and push them into place inside the virtual coordinate space until the system has "relaxed" completely, or more specifically, until the sum of forces inside the operator graph is zero. The operators are then mapped to the nodes closest to their respective physical locations that are not overloaded. Through this heuristic, the overall BDP or network usage is minimized while ensuring load fairness among the physical nodes.

*(ii) MOPA (Multi-Operator Placement Algorithm) [40]*[31]. MOPA is a variant of the Relaxation algorithm to minimize the bandwidth-delay product; hence instead of squared delay, this mechanism considers delay as an optimization criterion. Besides the optimization goal, this algorithm finds the local optimal solution using a gradient method, terminating when the current network usage, given by the above optimization criteria, becomes smaller than a threshold.

[30] Open implementation of this algorithm can be found and tried here `https://github.com/luthramanisha/TCEP/blob/master/src/main/scala/tcep/placement/sbon/PietzuchAlgorithm.scala` [Accessed in May 2021].

[31] `https://github.com/luthramanisha/TCEP/blob/master/src/main/scala/tcep/placement/mop/RizouAlgorithm.scala` [Accessed in May 2021].

*(iii) Global Optimal*[32]. Compared to the previous two mechanisms that find a sub optimal solution, we implemented a global optimal mechanism that chooses the best possible operator placement with minimum network usage or BDP product based on an exhaustive search of the possible placements. This OP mechanism uses the global knowledge of the entire network, in contrast to the decentralized mechanisms explained previously.

*(iv) Mobile DCEP [80]*[33]. The basic principle behind this mechanism is to maintain low overhead to deal with the high dynamics of mobile scenarios. To achieve this, the placement coordination is kept local and no cost information is shared between the nodes in the topology. In particular, only the next hops towards the data sources are considered for placement. Hence, the objective function of message overhead and latency is considered in this mechanism. By considering only the next hops and keeping the communication low, the mechanism aims to minimize the overhead, while for latency, the shortest path towards the data source is considered for placement.

*(v) Producer Consumer*[34]. For comparison to the above approaches, we consider placement on the randomly chosen producer or consumer *only*. This is chosen to complement Mobile DCEP mechanism that places operators near to the producers or consumers.

*(vi) Random*[35]. This mechanism chooses a physical host for each operator randomly and serves as a naive comparison.

### *Evaluation of Operator Placement Mechanisms*

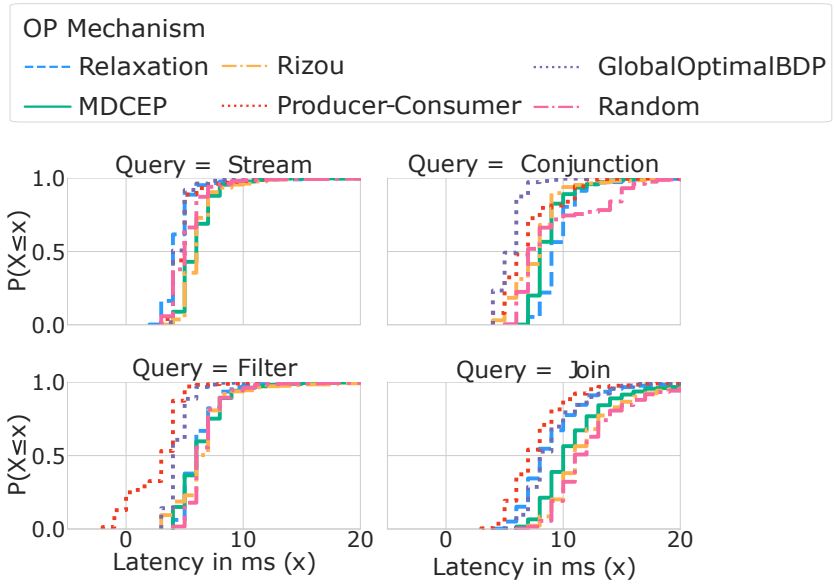
To understand the design space of OP mechanisms with distinct and conflicting optimization criteria, we evaluate them using the TCEP programming model. We take the QoS requirements, queries, and OP mechanisms as stated in Table 10 for comparison. The performance metrics, including the QoS requirements used are as follows: *(i)* Mean end-to-end latency or simply *latency*: Recall, latency is the time interval from when the event was first generated at the producer until the reception of the complex event at the consumer (cf. Definition 6).

[32]Open implementation of this algorithm can be found and tried here `https://github.com/luthramanisha/TCEP/blob/master/src/main/scala/tcep/placement/GlobalOptimalBDPAlgorithm.scala` [Accessed in May 2021].

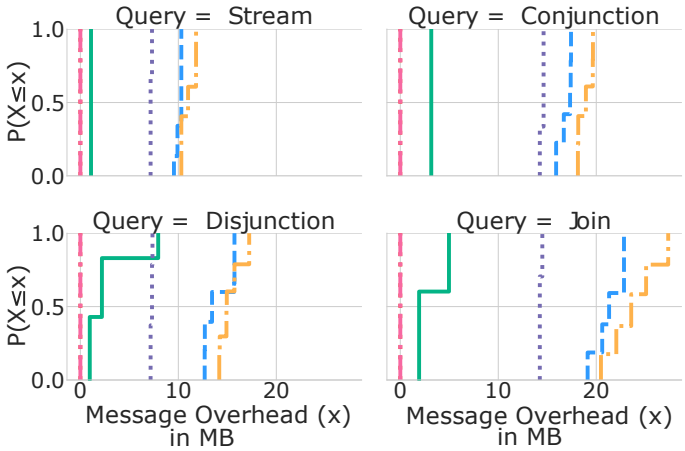
[33]`https://github.com/luthramanisha/TCEP/blob/master/src/main/scala/tcep/placement/manets/StarksAlgorithm.scala` [Accessed in May 2021].

[34]`https://github.com/luthramanisha/TCEP/blob/master/src/main/scala/tcep/placement/MobilityTolerantAlgorithm.scala` [Accessed in May 2021].

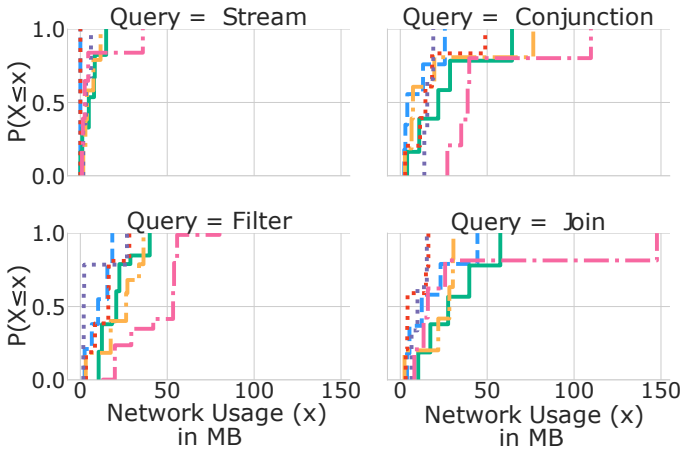
[35]`https://github.com/luthramanisha/TCEP/blob/master/src/main/scala/tcep/placement/RandomAlgorithm.scala` [Accessed in May 2021].



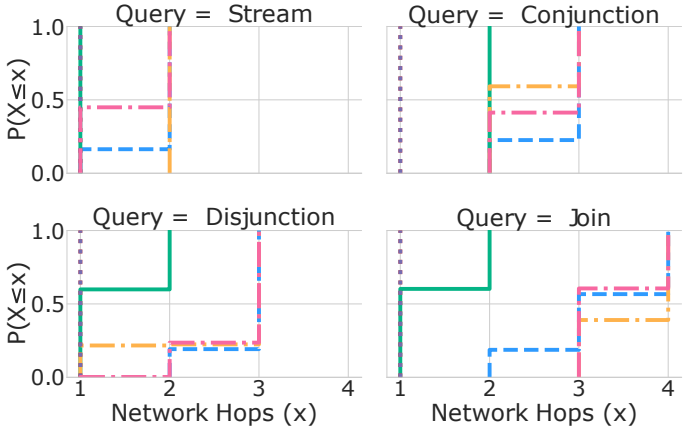

(a) Comparison in terms of latency (in ms).




(b) Comparison in terms of message overhead (in MB).




(c) Comparison in terms of network usage (in MB).



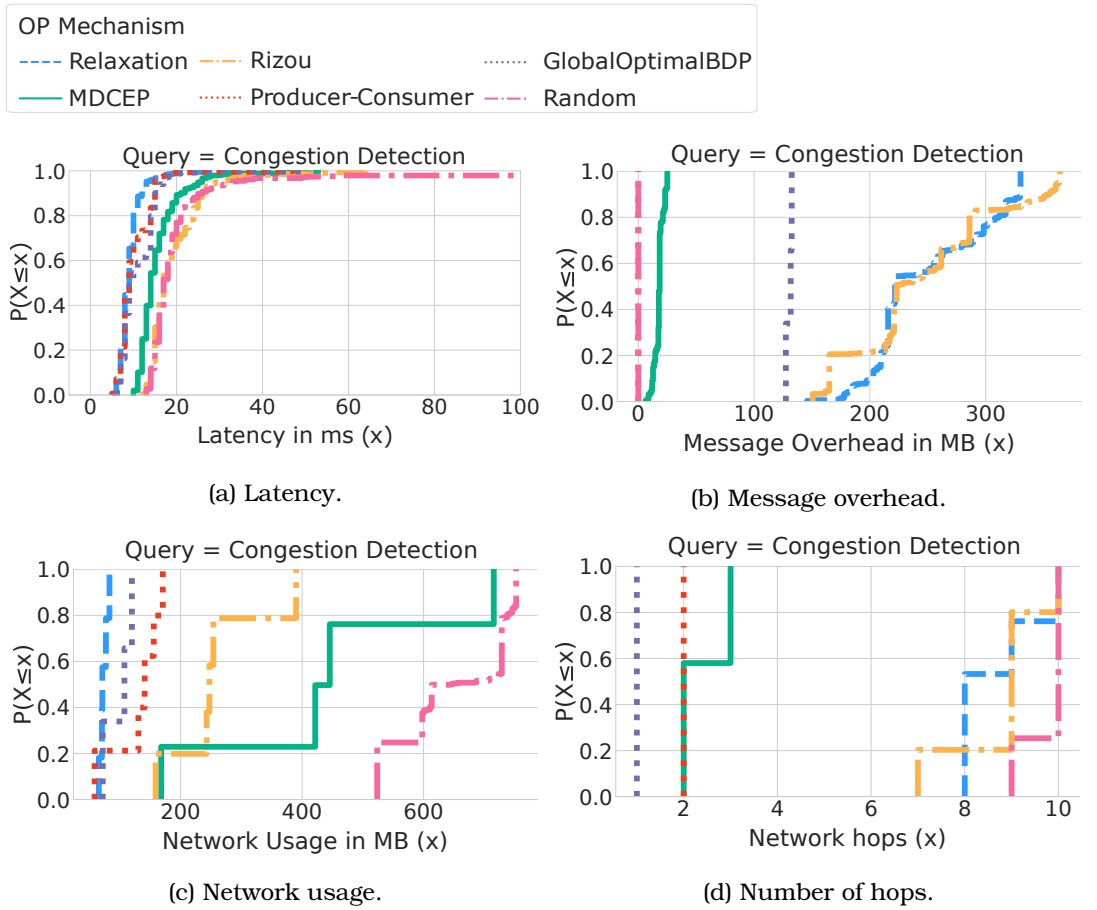

(d) Comparison in terms of number of hops involved for placement.

Figure 23: Performance evaluation of OP mechanisms using TCEP programming model for the standard CEP queries Stream, Conjunction, Filter and Join. All OP mechanisms perform differently for distinct queries and scenarios.

*(ii)* Mean message control overhead or simply *message overhead*: Recall, network usage is the amount of coordination overhead (in size of exchanged messages (in MB)) associated in performing operator placement. This includes establishing the broker network, exchanging network or node information for placement, and performing the placement (cf. Definition 7). *(iii)* Mean network usage: Recall, this is the amount of data in transit through the node network, commonly referred to as BDP product (cf. Definition 8). *(iv)* Number of hops: Recall, these are the number of hops or physical hosts used for an operator placement (cf. Definition 7).

Figure 23 shows the performance of the presented OP mechanisms (cf. Section 4.4.2) and standard CEP queries Q1 - Q4 (cf. Figure 22). The figure classifies the performance of the queries in terms of (a) latency, (b) message overhead, (c) network usage, and (d) number of hops. One important observaton is that OP mechanism behaves differently for different queries. For instance, in terms of latency, Relaxation, and Global Optimal mechanisms perform best for Stream and Conjunction operators, while Producer-Consumer supersedes them when executing Filter and Join queries. This is because the main objective of Relaxation and Global Optimal OP mechanisms is to minimize overall latency. The Producer Consumer mechanism can also achieve similar performance because of its proximity to the event sources and the end-users. In terms of message overhead, MDCEP and Random mechanisms perform the best because of the low coordination overhead in both OP mechanisms. In terms of network usage or the BDP product, we again see a difference in the performance of Relaxation and Global Optimal mechanisms in different queries. While for simple operators like Stream, the Producer-Consumer mechanism supersedes the former by a small magnitude for more complex queries like Conjunction, the Global Optimal and Relaxation mechanisms are better. Since we focus on more complex queries, those applied in IoT application scenarios, we further look into their performance for a traffic congestion detection query introduced in the setup (cf. Section 4.4.1) in the next paragraph.

Figure 24 presents the performance evaluation of the different OP mechanisms while executing a traffic congestion query for (a) latency, (b) message overhead, (c) network usage, and (d) number of hops. Similar to the other queries analyzed above, Relaxation performs well in terms of latency. However, it possesses much high message overhead due to the maintenance of the latency cost space. In contrast, MDCEP possesses much low message overhead while it suffers from very high latency for a high workload of queries. The variant of Relaxation, the MOPA, and Optimal mechanisms also suffer in performance in terms of message overhead. Contrarily, the Producer Consumer and Random mechanism suffer in terms of network usage.




(a) Latency.

(b) Message overhead.

(c) Network usage.

(d) Number of hops.

Figure 24: Performance evaluation of OP mechanisms using TCEP programming model. The key observation is that there is *no one size fits all* mechanism.

Table 24 in Appendix A.1.1 summarizes the observed mean, minimum, maximum, and percentiles (90, 95, 99%) for the metrics latency and message overhead. The table presents the results for Q1, Q4, and Q5 execution using the different OP mechanisms. From Figure 24 and Table 24 together, we show that Relaxation and MDCEP mechanisms stand representatives for the metrics latency and message overhead, respectively. This further solidifies our hypothesis that no *one size fits all* mechanism can satisfy both the optimization criterion network usage and message overhead. The same has been also recently investigated by other works on placement [20].

*No one size fits all!*

The mechanism transition methodology is suitable for such scenarios facing dynamics in the environment and QoS requirements. Based on the aforementioned results, in the rest of the evaluations, we will focus on the two representative OP mechanisms Relaxation and MDCEP and investigate the performance of mechanism transitions.

### 4.4.3 *Evaluation of Mechanism Transitions*

This section analyzes the performance of OP mechanism transition and its ability to fulfill changing QoS requirements at runtime. We use the two QoS metrics in relation to the used OP mechanisms mean network usage (objective function for Relaxation) and mean control message overhead (objective function for Mobile DCEP) as defined before. Furthermore, we consider a traffic congestion detection query for the rest of the evaluations because it is representative of our scenario (cf. Section 4.1), it captures the major standard CEP operators, and we have observed significant variation in the performance of the OP mechanisms as shown in Figure 24.

Figure 25 shows the mean network usage on the first y-axis and control message overhead on the second y-axis for 5 runs in TCEP. At around 45 seconds (shown by a red dotted line), we observe a change in QoS requirement from message overhead to network usage.

*TCEP seamlessly transits between OP mechanisms*

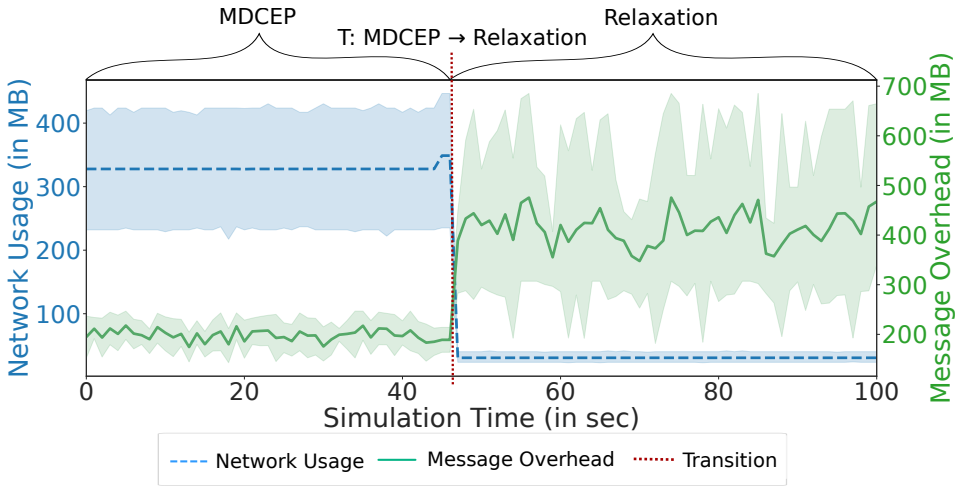



Figure 25: Network usage (y1-axis) (in blue) and message overhead (y2-axis) (in green) measurement over a transition from MDCEP to Relaxation OP mechanism. TCEP system seamlessly transits to another OP mechanism without incurring any overhead in terms of the specified QoS requirements.

TCEP handles this by executing a transition between MDCEP to Relaxation $T$ : MDCEP $\rightarrow$ Relaxation at run time. TCEP triggers a transition automatically and selects an appropriate placement mechanism that fulfills the QoS requirement. It is noticeable that TCEP does not induce any interruption or costs in terms of the optimized metrics while performing a transition to a completely new OP mechanism. Hence, it can be seen that the execution of transition takes place in a *live* and *seamless* manner. In the next sections, we

further investigate the transition cost for the different algorithms for transition and selection of placement mechanism and evaluate seamlessness based on the throughput (number of output events).

### 4.4.4 *Evaluation of Transition Execution Strategies*

This section evaluates the *(i)* cost of transition execution strategies and *(ii)* the seamlesssness of the proposed startegies. For this, we consider the following metrics: *(i)* transition time, which is measured by the total time taken from the start of transition until its end (cf. Equation (4)), *(ii)* transition overhead, which is measured as the total amount of overhead (in MB) to perform a transition (cf. Equation (5)), and *(iii)* throughput in delivery of complex events, which is measured as the ratio of the actual vs expected number of complex events (in %) (cf. Definition 11). We extend the transition execution strategies explained in Algorithm 1 and Algorithm 2 to migrate the operator graph both concurrently and sequentially, respectively. The four evaluated approaches are enlisted in Table 10. Furthermore, we increase the query load up to 10 queries in order to impose changes in the environmental conditions that trigger transitions in the TCEP system.

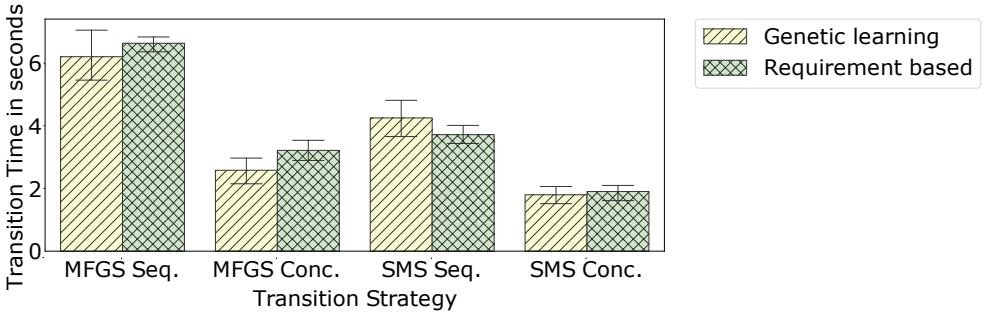
*Cost performance analysis*

#### ***Cost of Transition Strategies with Learning-Based Selection***

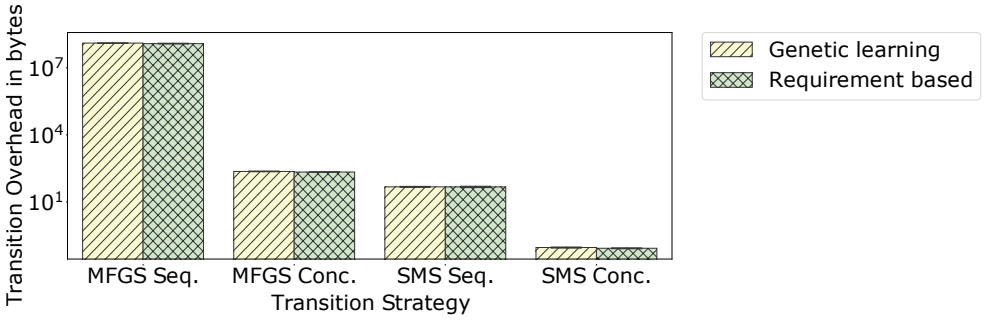
We analyze the cost of the different transition strategies proposed in Section 4.3.2. The transition strategies work together with the learning algorithm that is responsible for selecting the OP mechanism for a transition.

Besides the different transition strategies, we implemented a requirement-based selection algorithm that selects a placement mechanism by matching the QoS requirement with the optimization criteria for comparison with our learning algorithm. If there exists more than one mechanism matching the QoS requirement, there is a random selection. In contrast, the genetic learning-based selection algorithm takes into account the performance of the OP mechanism, as explained in Section 4.3.1.

Figure 26 shows the transition (a) time and (b) overhead incurred by the transition strategies using different selection algorithms.




(a) Transition time for the proposed algorithms.




(b) Transition overhead for the proposed algorithms.

Figure 26: The plot shows the transition cost in terms of time and overhead for the proposed transition algorithms. SMS transition strategies possess a negligible overhead in state transfer during operator migrations, and hence, can perform the transition in a mean time of 1.82 seconds compared to 6.29 seconds required by MFGS Sequential strategy. Moreover, the SMS Concurrent strategy has only a negligible overhead of 0.72 bits, thanks to the cost-optimal algorithm (cf. Line 14).

*Concurrent transitions substantially better in time*

Here, the costs of transition includes the cost in time and overhead as represented earlier in Equation (4) and Equation (5), respectively in Section 4.2.2. It is noticeable that MFGS algorithms possess higher transition times than SMS algorithms. This is due to the state involved that is to be transferred by the MFGS algorithms, whereas the SMS algorithms optimize for the minimal amount of state transfer (cf. Equation (6)). There is a substantial improvement in the transition time for concurrent strategies compared to sequential.

Finally, using the SMS concurrent strategy, we achieve an effective mean transition time of $1.82$ seconds for the load of ten congestion detection queries involving multiple stateful operators. We see an effective reduction of around $4$ seconds compared to the MFGS sequential transition strategy that takes $6.29$ seconds to finish a transition with state transfer. The only cost parameter involved in SMS transition strategies is in terms of selection of the OP mechanism and transition coordination costs in terms of communication between the distributed nodes.

In the second plot (b), we observe the total transition overhead incurred for the selection of a OP mechanism, transition coordination, and operator migrations due to the transition (Equation (5) in Section 4.2.2). Consistent with the transition time results, we observe lower overhead of SMS algorithms due to no state involvement for the migration. Note the scale of the y-axis is logarithmic to show the amount of overhead involved for SMS algorithms that is substantially lower. In particular, we have only a mean overhead of $0.72$ bits for SMS concurrent and $379.79$ bits for SMS sequential algorithm, where the former is more than $2000\times$ better and the latter is around $5\times$ better than the MFGS concurrent strategy.

*Transition overhead is only a few bits*

The second observation from these plots is that the genetic learning-based selection algorithm performs equally well compared to the requirement-based selection algorithm. This is because of no offline training costs and negligible learning costs. In Section 4.4.5, we elaborate on the learning costs of these algorithms. In summary, the SMS strategies both sequential and concurrent perform the best in both transition time and overhead, with a mean transition in the range of $0.85 - 2.83$ seconds, ten queries, compared to $35$ seconds when the transition is performed naively using the stop and start migration strategy for the congestion detection query. We analyze costs per operator in the next section.

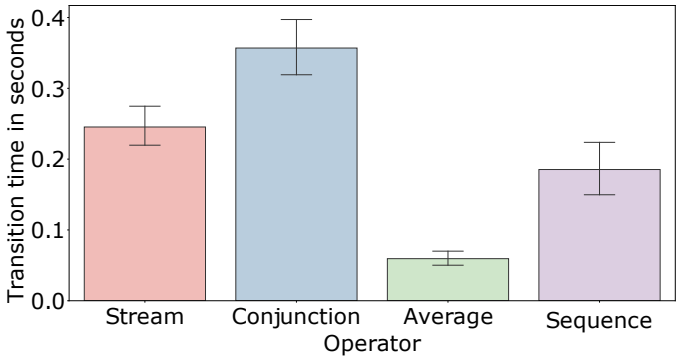
### *Cost of Transition Strategies for Different Operators*

In this section, we analyze the cost incurred by the transition execution strategies in detail.

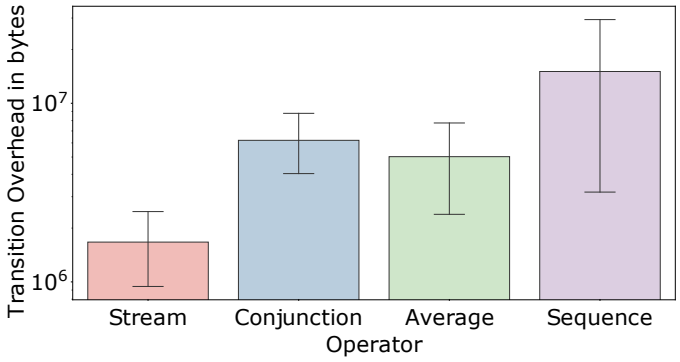
The transition time includes time taken to migrate the operator graph as well as the time taken to wait for the previous operators or predecessors to migrate and start executing at the target broker (cf. Section 4.3.2). For example, the Conjunction operator waits for migration until Average and Stream operators start their operation at the target brokers. Leaf operators (Stream or producers) have no wait time as they have no predecessors. The operator transition overhead involves the cost for first, the state involved in migration for stateful operators like Window-Aggregates, Joins, and Sequences, and *(ii)* second, the coordination overhead for the operator graph migration in terms of communication, such as acknowledgments (cf. Section 4.3.2: Algorithm 1 and Algorithm 2). Stateful operators have costs in both dimensions, communication as well as migration costs depending on the transition strategy – MFGS or SMS, while stateless operators like Filters and Stream do not have any state migration costs, but do have a small communication cost again depending on the transition strategy – sequential or concurrent.

*Stateful operator transition in few milliseconds*

Figure 27 shows the (a) mean transition time and (b) overhead using the transition strategies for all the operators while executing ten incrementally deployed congestion detection queries Q5 (cf. Figure 22). The total migration time correlates with the number of operators and the transition overhead in the second plot. Table 11 summarizes the mean, minimum, and maximum values of the distribution. It can be seen that the stateless operators like Stream, although high in number (90 operators), can be migrated in around $245.3$ ms (first column in table). While other operators like Conjunction and Sequence need slightly higher mean transition times of $356.89$ and $185.32$ ms, respectively, with a mean and maximum transition overhead of $6.2 - 130.98$ MBs, and $15.06 - 129.9$ MBs, respectively.




(a) Transition time.

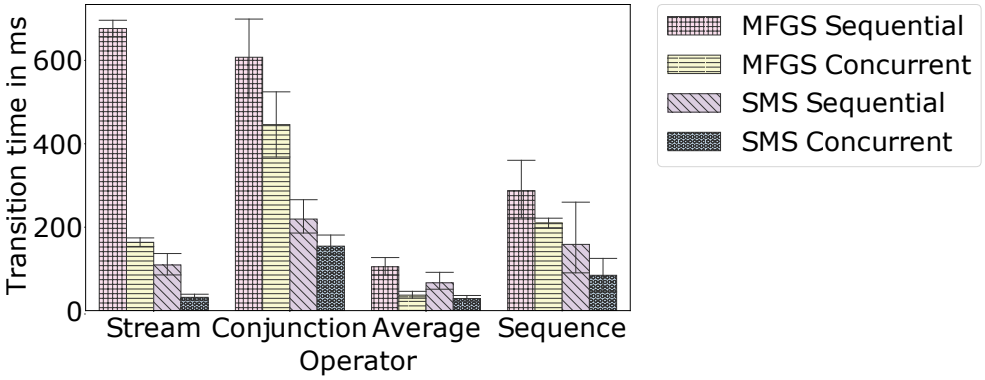



(b) Transition overhead.

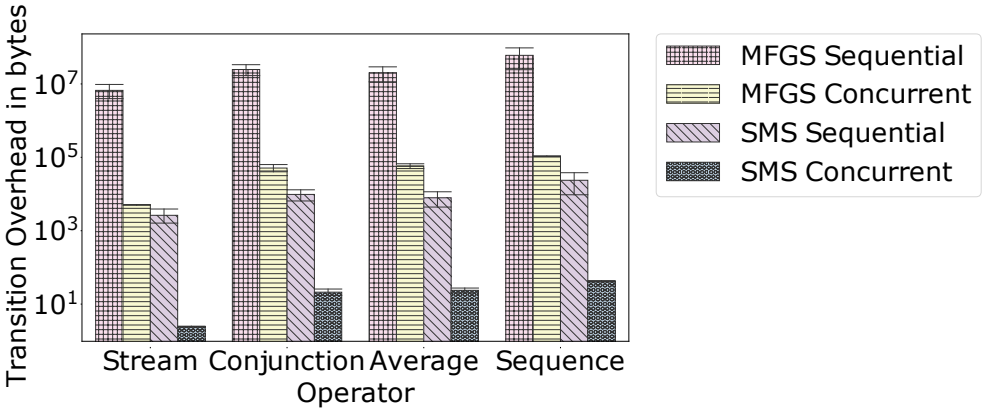
Figure 27: Transition time and overhead observed for the different operators in the congestion detection query. Operator transitions are performed in the order of few milliseconds and with very low overhead using our transition algorithms.

| Operator | Transition time (in ms) | | | Transition overhead (in MB) | | |
|---|---|---|---|---|---|---|
| | mean | min | max | mean | min | max |
| Average | 59.43 | 15 | 411 | 5.02 | 13.679 bytes | 64.72 |
| Conjunction | 356.89 | 119 | 1404 | 6.2 | 5.53 bytes | 130.98 |
| Sequence | **185.32** | 15 | 556 | **15.06** | 35.507 bytes | **129.9** |
| Stream | 245.38 | 8 | 913 | 1.67 | 1.315 bytes | 64.72 |

Table 11: Mean, min, and max values of transition time and overhead per operator for 10 incrementally deployed Q5 queries. Here **bold** values (for Sequence operator) incur low overhead among stateful operators.




(a) Transition time observed for different operators and transition algorithms.




(b) Transition overhead observed for different operators and transition algorithms.

Figure 29: Transition time and overhead measurement for different operators for ten incrementally deployed Q5 queries. SMS strategies possess minimum migration time and overhead due to the minimal amount of transition overhead.

| Operator | # $\omega$ in 10 Q5 |
|---|---|
| Stream | 90 |
| Conjunction | 80 |
| Average | 30 |
| Sequence | 10 |

Table 12: Number of operators in ten congestion detection query (Q5 in Figure 22) used for evaluation.

Figure 29 classifies the transition cost analyzed in Figure 27 based on the transition strategies. The MFGS sequential strategy performs clearly the worst because of the high amount of state overhead amounting 60.12 MB (mean value for the Sequence operator). In contrast, the SMS strategies require very short time to transit, a mean time of 41.1 ms for 30 Average operators and 85.1 ms for 90 Stream operators. Table 12 denotes the number (#) of respective operators per ten queries of congestion detection (Q5) in a single run. The Conjunction operator takes the highest amount of time to migrate due to high number of operators involved. The same applies to the Stream operator. The number of Sequence operators to be migrated is less; however, due to the high amount of state (~60 MB) to be transferred it takes longer to transit. Table 25 in Section A.1.1 summarizes the mean transition time and overhead required by the different strategies shown in the earlier Figure 29.

In summary, we analyzed the cost per operator for the transition strategies. Consistent with our findings in the previous section, the MFGS strategies take longer to transit than SMS strategies. The SMS concurrent strategy performs the best since the costs are minimal because of the optimization performed in the algorithm.

### *Seamlessness in Transition Execution*

*Transition strategies consistently deliver events*

In this section we aim to verify the seamlessless in the transition execution for the proposed transition strategies. The measurements were taken when ten congestion queries were in execution. Figure 30 measures the throughput in the delivery of complex events while TCEP's transition strategies are in execution. We observe a slight output disruption (mean of 0.02%) for MFGS Sequential and Concurrent algorithms, because of the large state transfer. In contrast, both SMS transition strategies exhibit live and seamless properties and, hence, delivers complex events while ensuring throughput of 100% for both the selection algorithms.

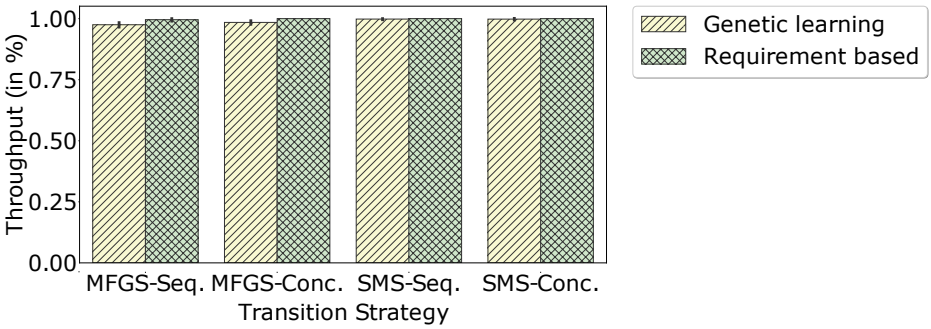



Figure 30: Throughput measurements using the different transition strategies and selection strategies for OP mechanisms. It can be seen that all the transition strategies consistently deliver complex events and, hence, enabling a *seamless* execution of transitions.

### 4.4.5 *Learning Costs of Placement Selection*

This section aims to understand the learning costs of the adaptive placement selection algorithm introduced in Section 4.3.1.

*Learning costs are negligible*

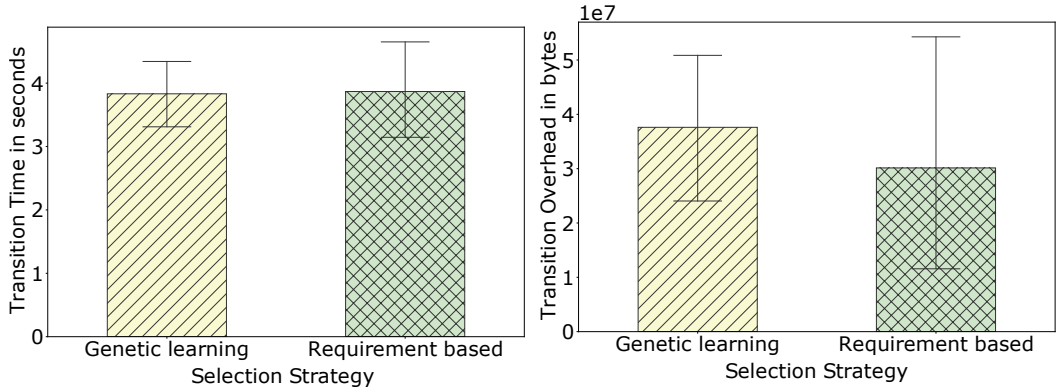



Figure 31: Learning cost comparison with a requirement based selection algorithm.

The following metrics are considered for the costs *(i)* the time taken to learn the performance characteristics, in other words, to update the learning model and *(ii)* communication cost for the placement selection. The genetic learning-based learning algorithm has no training costs since the algorithm is based on online learning. Hence, it induces only a negligible overhead on time within a range of $2.5 - 3.15$ ms (95% confidence interval). Often the update of the learning model induces no overhead at all. Furthermore, the algorithm *does not* induce any communication overhead due to local handling of operator placement selection.

Finally, to understand the influence of genetic learning-based selection algorithm on the performance mechanisms transitions, we analyze the transition costs in terms of time and overhead. We compare the learning algorithm with a requirement-based algorithm where the selection of a mechanism is done based on QoS requirements.

In Figure 31, we observe that due to the negligible overhead of genetic learning algorithm, the cost is comparable to the requirement based algorithm. In fact, the transition time observed with the genetic learning-based is slightly less than requirement-based algorithm. In terms of overhead, we see a slight increase due to the exploration of a suitable placement algorithm that is not performed in the requirement based algorithm.

## 4.5 Summary

This chapter proposes the first contribution of this thesis solving the adaptivity problem to meet the changing Quality of Service requirements in the face of dynamic environmental conditions. All applications including Internet-of-Things are inherently exposed to the dynamics of the environment. These applications have to abide to the changes the Quality of Service requirements in the face of dynamic environmental conditions. In this chapter, we aim to meet those changes in QoS requirements by providing adaptation between Operator Placement mechanisms, so-called *transitions*, in a CEP system. To this end, we proposed *(i)* a programming model that can specify OP mechanisms, *(ii)* a learning-based selection mechanism to select OP mechanisms for a transition, *(iii)* transition strategies that enable *live* and *seamless* transitions while ensuring *correctness* in results, and *(iv)* a transition-capable CEP system, named TCEP, that integrates the aforementioned mechanisms and models for evaluation. Our extensive real-world evaluation of TCEP using queries in the context of an IoT scenario and state-of-the-art OP mechanisms shows that *(i)* transition-capable OP mechanisms can be specified using our programming model in a minimum effort, *(ii)* transition algorithms can fulfill changing QoS requirements while seamlessly delivering results, and *(iii)* transition costs can be reduced to a few milliseconds in terms of time and a few Bytes in terms of overhead while incurring negligible learning costs using the proposed methods.

# 5

# Network-centric Query Execution

*Why network-centric?*

This chapter provides a solution to the *efficiency problem* in the context of the Internet of Things applications. In many IoT applications, efficiency in delivering the events is of extreme importance, for instance, in terms of latency, if violated would lead to dire consequences. An example of this is a fraud detection application that would lead to heavy monetary loss to the financial institutions if fraud is left undetected or if detected not in a timely manner. To avoid such situations, a CEP system has to detect such complex events with very high efficiency.

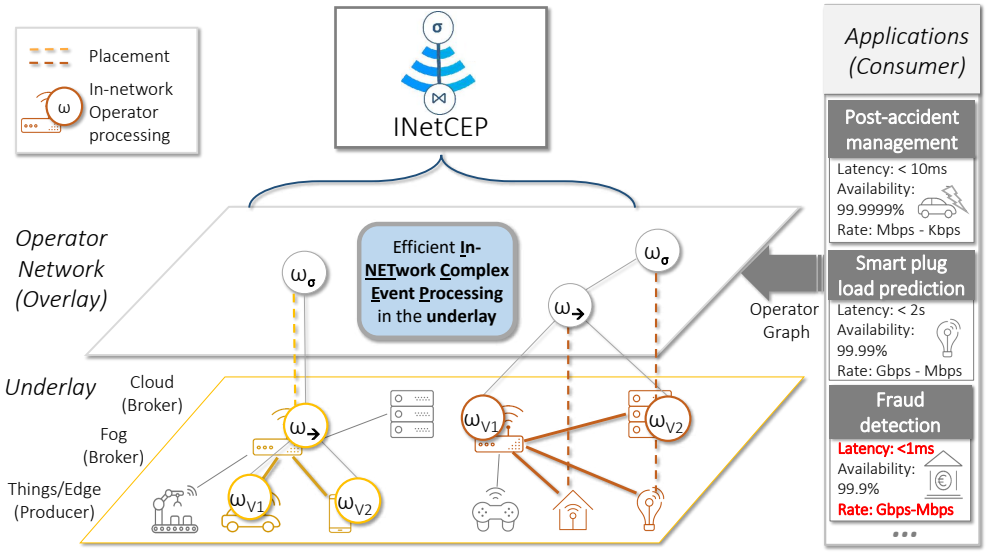



Figure 32: Overall system architecture with the focus on INETCEP showing hybrid execution of operator graph on underlay as well as on the overlay network.

Classical CEP systems [23, 22, 100, 54] performs operator graph processing in the form of an overlay network, which costs additional latency to ship events to the commodity hardware where actual processing is performed. A solution to mitigate this inefficiency is by offloading event processing to in-network resources, such as switches and routers. As pointed out in Chapter 2: Section 2.2.3, in-network processing in this realm has been investigated, for instance, using programmable data plane languages

like P4 [51, 66, 151], using other Software-defined protocols [131], and using Information-centric Networking (ICN) paradigm [61]. However, none of the above approaches model event processing and resolve continuous data streams, as well as queries using in-network processing resources while efficiently meeting the quality requirements.

*Solution to the efficiency problem*

To this end, we provide *efficiency* in event processing by proposing a unified communication model, named INETCEP, over an Information-Centric Networking architecture to enable processing of continuous data streams. Furthermore, we provide an expressive meta-query language that complements the data plane language of ICN to specify complex events. The efficient query execution algorithms enables fast and robust processing of events by utilising the distributed set of ICN nodes. To understand the research challenges addressed in this chapter, recall the overall architecture introduced in Chapter 3, with INETCEP-centric components as seen in Figure 32.

*ICN provides network-centric abstractions, but...*

We focus on challenging Quality of Service (QoS) requirements of exemplary IoT applications (seen on the right side of the figure), such as in terms of latency. For instance, in a fraud detection application, a very high input event rate (~100K events per second) has to be processed with very low latency in the order of a few microseconds [13]. We aim to utilize ICN networking architecture to realize network-centric CEP because it provides numerous benefits as follows. *(i)* In ICN, data is the first class citizen as it shifts the focus of addressing from *named-host* to *named-data*. *(ii)* Some core concepts of ICN such as in-network programmability and caching are highly beneficial to realize event processing on top of ICN architectures. Moreover, as explained in Chapter 2: Section 2.2.3, previous work has shown advantage of data processing in the underlay using ICN [61, 140, 142, 141]. However, current ICN architectures pose strong limitations in their support to process *continuous* data streams or the so-called push-based communication, which is required to realize CEP on top of ICN. Ideally, a networking architecture should enable support for both the communication mechanisms pull- and push-based to enable a broad spectrum of applications. Furthermore, ICN architectures need a way to resolve and process queries to derive complex events while ensuring the QoS requirements of the applications. These observations lead us to our second research question and its sub-questions answered in this chapter.

*... doesn't support continuous streams*

*Second Research Question and its sub RQs*

*RQ2: How to increase the efficiency in executing queries using in-network architectures, particularly ICN?*

*RQ2.1* How to enable continuous data stream and query processing over the ICN substrate?

*RQ2.2* How to improve efficiency in executing CEP queries on top of ICN?

We discuss the key research challenges pertaining to the above research questions in the following. *(RQ2.1)* An efficient design of a unified communication mechanism (coexisting push and pull) is challenging because of two reasons. First, classical ICN architectures are inherently pull-based, which requires a major redesign of the communication model of ICN. Second, the continuous data stream might disturb the flow balance of the networking architecture that can cause congestion and eventually packet loss or in the worst case can disrupt the stream completely. Therefore, the first contribution proposes *a unified communication model* that enables coexisting push- and pull-based communication mechanisms for ICN architectures while efficiently providing flow balance using a proposed rate-based lossless flow control mechanism. *(RQ2.2)* Existing ICN architectures fall short in expressing complex events. Moreover, existing CEP programming models cannot be easily reapplied on top of ICN because of heavy dissimilarities in event as well as processing layer semantics. Another challenge is to realize distributed event processing using Operator Placement on top of ICN that is quite different than overlay network processing as done in classical CEP systems. Thus, the second contribution proposes a CEP Query Engine for the data plane of ICN with the following concepts: *(i)* A *meta query language*[36] that complements the data plane language of ICN with the standard CEP operators and the core event processing semantics required to express CEP operator graphs; *(ii) Network-centric query execution* algorithms that enables core CEP functionalities like query reuse and Operator Placement mechanisms. The algorithms reactively handle the data stream and process them in a parallel manner to derive the complex events in an efficient manner while ensuring the correctness.

*Challenge 1: ICN is inherently pull-based*

*Challenge 2: lack of query abstractions*

The findings presented as part of this chapter are based on the author's previous publications in [139, 26]. The structure of this chapter is presented as follows. We start by introducing the extended system model and the problem statement in Section 5.1. Section 5.2 explains the design decisions, demonstrating the need of unified communication mechanism as well as the query engine for ICN substrate. Afterwards, we present the design of the proposed INETCEP networking architecture comprising the three main contributions enlisted above. Section 5.3 provides an extensive evaluation of INETCEP using a real-world system implementation of ICN architecture, Named Function Networking [61]. INETCEP and its unified communication mechanism, along with the query engine, are publicly available at `https://github.com/luthramanisha/INetCEP` together with the evaluation datasets and emulation environment used for reproducing the results shown in this chapter.

*Key publications on* INETCEP *and structure*

*Publicly available, try it out!*

[36] We use "meta" for our query language as the semantics of the language complements the standard language of the used ICN architecture (Named Function Networking).

## 5.1 Analysis of Efficiency in Network-centric Query Execution

This section discusses the extended system model in Section 5.1.1 with specifics to the contributions of this chapter, in addition to the previously introduced common system model in Chapter 3: Section 3.2.1. Later, Section 5.1.2 details the problem space by considering alternative approaches and concludes with a summary of limitations.

### 5.1.1 *Extended System Model*

This section describes the network-centric entities corresponding to ICN architectures. We consider the deployment infrastructure of edge-fog-cloud for the execution of queries. Here, we assume ICN-capable hardware or so-called routers along with standard TCP/IP switches at the fog layer. At the cloud layer, we assume the presence of more powerful machines from the data centers. Each node in the infrastructure can act as a *producer*, *consumer*, and *broker*. A node is an ICN node if it complies with the ICN protocols such as NDN (Named-Data Networking) and CCN (Content-centric Networking) (cf. Chapter 2: Section 2.2.2). An ICN node is situated at the fog layer and can take a broker role, while the producers and consumers are end-devices situated at the edge, such as a sensor or a mobile device, or at the cloud layer inside a data center. Each ICN node comprises of *(i)* a Pending Interest Table (PIT), which stores the requests from the consumer, *(ii)* a Forwarding Information Base (FIB) table, which acts as a routing table to forward requests towards the source, and *(iii)* a Content Store (CS), which acts as a cache for the data objects. Here, each event tuple from IoT devices is referred to as a single data object. In terms of QoS requirements, we focus to provide latency (cf. Definition 6), throughput (cf. Definition 11) and accuracy demands (cf. Definition 13 and Definition 12) of the applications, which are the most critical objectives for nearly all applications considered for event processing.

*In-network elements of ICN*

#### *Unified Communication Model*

For the unified communication, we assume that the communication can be initiated by both ends, consumer or producer. The consumer can initiate a pull request using the `Interest` as conventionally done in an ICN architecture (cf. Chapter 2: Section 2.2.2). Moreover, a consumer can also send a continuous request or a query $Q = \{q, qname, QoS, \tau_{qos}\}$ as described in the common system model Section 3.2.1. Here, $qname$ is used to identify the query in PIT similar to the name prefix $name$ is used to identify the request served by an `Interest` packet in classical ICN architectures. The producer can initiate

the communication by sending a continuous data stream that has to be processed by the CEP engine presented in this contribution.

An ICN flow consists of packets bearing the same data object $name$. Moreover, all the flows are considered to be independent of each other and can change dynamically. For instance, the event rates are known to vary in a dynamic environment. Finally, we assume a FIFO order for the packets in a flow.

### 5.1.2 *Problem Space*

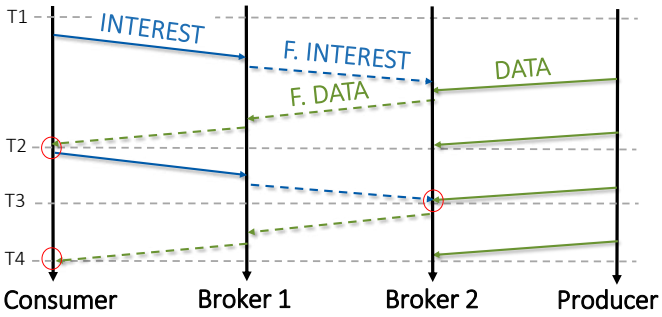
*Limitations of alternative approaches*

This section, first, discusses the limitations of plausible solutions for achieving both push- and pull-based communication functionalities using existing consumer-initiated [129], [132], [142] or producer-initiated [226], [227], [135] communication mechanisms as present in existing ICN architectures. Afterwards, we discuss the limitations of using existing naming schemes of ICN [129], [132] in expressing complex events.

#### *Communication Mechanisms*

A potential solution to enable continuous data stream and query processing over the ICN substrate is by polling queries using standard `Interest` packets at regular intervals. We call this *Periodic Request* mechanism [142] or mimic push by pull since push functionality is imitated while using pull-based communication. **First limitation**: Figures 33 (a) and (b) illustrate the communication mechanism and the problems with this approach, respectively. The consumer initiates the communication at $T1$ by continuously issuing a query using an `Interest` packet at a regular interval, say $r_I$. At $T2$, the consumer receives the data object back in the form of a `Data` packet from the producer for each `Interest` packet based on the $name$. Each time a `Data` packet matching the `Interest` packet $name$ is received, the pending interest is removed from the PIT (represented as ~~strikethrough~~ in the subfigure b indicating the PIT entries of broker node 1). Based on the regular interval $r_I$, a new entry in the PIT is created for each `Interest` packet (e.g., at time $T2$). This approach is not suitable because of the following problems: *(i)* the number of `Interest` packets needed to poll for a continuous query, *(ii)* the state in the PIT for each `Interest` packet sent by the consumer, *(iii)* very short polling interval is required for latency-sensitive applications, that further increases the overhead in terms of traffic. In contrast, a large interval would lead to high response times, which is alleviated even more with the number of brokers involved in forwarding or processing the data, as noted in figure (a) by red circles at time $T3$ and $T4$.

*High overhead in terms of interest packets*




(a) Periodic request mechanism (mimic push by pull).

T1 – T2: Broker 1

| Pending Interest Table | |
|---|---|
| Name | Face(s) |
| /node/AggrData | 0 |

Polling Interest <name>

T2 – T3: Broker 1

| Pending Interest Table | |
|---|---|
| Name | Face(s) |
| ~~/node/AggrData~~ | ~~0~~ |
| /node/AggrData | 0 |

Renew with periodic Interests

T3 – T4: Broker 1

| Pending Interest Table | |
|---|---|
| Name | Face(s) |
| ~~/node/AggrData~~ | ~~0~~ |
| ~~/node/AggrData~~ | ~~0~~ |
| /node/AggrData | 0 |
| ... | .. |

(b) Pending Interest Table entries of Broker 1.

Figure 33: **First limitation** stands for enormous overhead using the periodic request mechanism in terms of number of packets and the state in PIT.

*Stale data in CS*

Another problem with this approach is the stale data response (cf. Figure 34) from the Content Store as seen in the CS of consumer, broker 1, and broker 2 at time $T2$ for the communication in Figure 33 (a). The **second limitation** is that an old data is served from the CS of broker 1 at time $T2$, while a new data object is sent by the producer as seen in the CS of broker 2. A solution to this problem could be to send an `Interest` packet with an indication to fetch a new data object and expire the old data object in the PIT entry. This can be done, for instance, by appending sequence numbers similar to TCP packets [228] to the `Data` packet. Still, this mechanism would require additional synchronization, which results in further management overhead in addition to the overhead caused by multiple `Interest` packets.

Another possibility is to use ICN architectures that provide only the producer-initiated communication mechanism [145] while supporting the pull-based communication mechanism. Ahmed et al. [135] proposed to support what we call “mimic pull by push” by using a three-way message exchange of what amounts to a one-way message in a pull-based communication mechanism, as illustrated in Figure 35. In their approach, the authors use so-called *beacon* packets, as shown in Figure 35, that announces a $name$ served by the data objects generated by the producer. The beacon packet initiates a virtual PIT entry so that later `Data` packets can be forwarded to the consumer using the same path. This serves as a callback to the con-

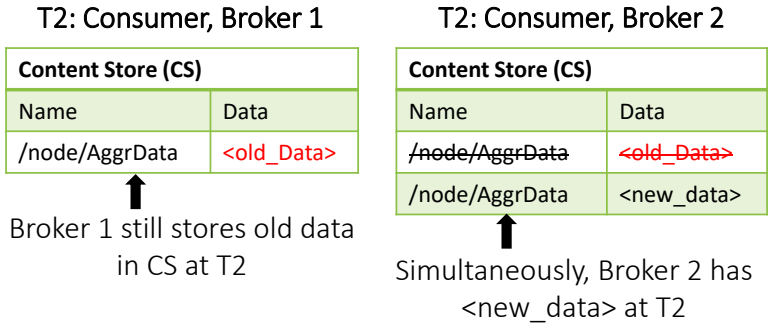



Figure 34: CS entries of Consumer and Broker 1 and 2 at T2 illustrating **Second limitation** of the old entries returned from the Content Store of Broker 1.

sumer. If the consumer is interested in the data object with name prefix $name$ announced in the beacon packet, the consumer expresses an interest in the data object using an `Interest` packet, which is sent by the producer in the third step. **Third limitation**: Apart from the overhead of three-way exchange, this approach suffers from other major limitations such as the double state in the PIT; one of the virtual PIT entry related to the beacon packet ($/node/AggrData$) and another of the PIT entry corresponding to the `Interest` packet ($/node/AggrDataI$) as illustrated in Figure 35 (b). Moreover, the network architecture has to support handling two kinds of `Interest` packets: one from the producer (so-called beacon packet) is not supposed to fetch data and another that is supposed to fetch the data.

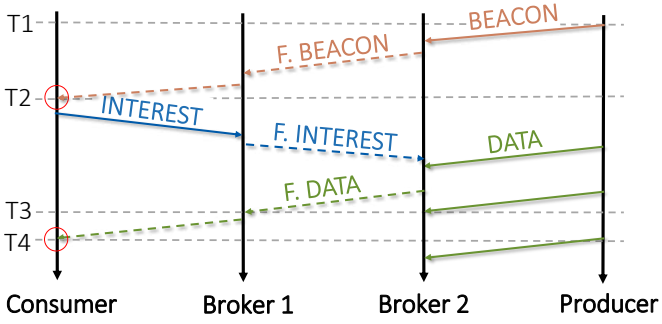
*Three-way message exchange*

One more plausible way to support continuous queries is using long-lived `Interest` packets [146], which also has multiple side effects. The approach is to create long-lived PIT entries that are not removed after a `Data` packet is received. **Fourth limitation**: However, this accounts for a larger state in PIT that stays as long as the PIT entry remains, which may become unnecessary when the consumer is no more interested in the complex event. Another overhead is to renew the PIT entry each time so that it does not expire and predicting the time of expiry for each PIT entry is not a practical solution.

*PIT entry never removed*

### *Hierarchical Naming Schemes of ICN*

The existing naming schemes of ICN architectures are restricted in terms of expressing higher-level interests in the data items from the producer. It assumes a hierarchical naming scheme to address named data,
e.g., $/cardTerminalA/cardAmount$, to fetch data items from card terminal, e.g., $\$450$ that acts as a producer. A simple way to achieve a naming scheme to specify higher-level interests for CEP operators would be to include operator identifiers as follows: $/cardTerminalA/max/cardAmount$. However, this has several problems as follows. **Fifth limitation:** *(i)* It is restricted to data items




(a) Beacon mechanism (mimic pull by push).

T1 – T2: Broker 1

| Pending Interest Table | |
|---|---|
| Name | Face(s) |
| /node/vAggrData | 0 |

Virtual PIT entry

T2 – T3: Broker 1

| Pending Interest Table | |
|---|---|
| Name | Face(s) |
| /node/vAggrData | 0 |
| /node/AggrDataI | 0 |

T3 – T4: Broker 1

| Pending Interest Table | |
|---|---|
| Name | Face(s) |
| /node/vAggrData | 0 |
| ~~/node/AggrDataI~~ | ~~0~~ |
| ... | .. |

(b) Pending Interest Table entries of Broker 1.

Figure 35: **Third limitation**: Three-way exchange of packets and double state in PIT is required.

*Single node of failure and lack of expressiveness*

from a single producer and data transformation from multiple producers is not possible. *(ii)* Conventionally in ICN, the processing function is executed at the consumer, which can be inefficient for queries involving multiple operators or scenarios with multiple queries. *(iii)* This approach is not extensible and expressive as queries with multiple operators would mean appending them in the naming scheme that further reduces the readability. Therefore, we need an expressive query language that complements the naming scheme of existing ICN architectures, such that higher-level operations can be specified on the data while performing standard network-centric functions.

*Low efficiency of overlay networks*

**Sixth limitation:** Another problem is how to enable efficient query execution in the ICN network. The standard ICN resolution engine can only resolve `Interest` and `Data` packets based on the name prefix *name* and is unable to express a query. Moreover, engine of Named Function Networking [61] suffers from limitations related to name prefix as detailed above. A way to deal with this problem is to offload to a separate CEP engine node. However, centralized processing is not a solution since this would unnecessarily overload the network with unwanted data, which could have been transformed before being sent to the central CEP engine. Moreover, a central CEP engine node is susceptible to a single node of failure for large-scale IoT applications. Another way is to offload processing to a distributed CEP system in an overlay network, as conventionally done in CEP systems. However, similar to a centralized solution, using this solution also leaves the network with unwanted

| Approach | Limitations |
|---|---|
| Periodic Request mechanism [142] (mimic push by pull) | *(i)* This approach demands a very high number of `Interest` packets which eventually leads to a large PIT state. Also, very short polling interval is required for latency-sensitive applications that would unnecessarily congest the network.<br>*(ii)* A separate mechanism to deal with stale data is required. |
| Beacon mechanism [135] (mimic pull by push) | *(iii)* This method requires three-way message exchange of what amounts to two-way exchange that results into unnecessary overhead. |
| Long-lived Interests [146] | *(iv)* This approach results into larger PIT state amounting to as long as the interest stays. Moreover, a prediction on the time until the long lived interest remains in PIT is infeasible. |
| Hierarchical Naming Scheme [129] for continuous queries | *(v)* Transformation of data arriving from multiple producers is not possible. It requires that the data is processed only by the consumer, which can lead to single node of failure. This approach is not extensible as CEP may require multiple nodes for processing operator graphs. |
| Query Resolution Engine of NFN [61] | *(vi)* Besides the limitations in naming scheme (as stated in fifth limitation) the query resolution engine of NFN is incapable of resolving CEP queries because of its original design. Moreover, it only supports resolving so-called named-functions as part of the application layer and not inside the network layer like we do. |

Table 13: Summary of six key limitations derived from the analysis of alternative approaches.

data. Moreover, overlay networks are inefficient, and need several middleboxes where the data has to be shipped and, hence, unable to meet the challenging QoS demands (e.g., latency). To deal with this limitation, we propose centralized and distributed query processing algorithms in the data plane of ICN that efficiently parse, place, and execute queries.

Overall, we derive six key limitations of the alternative approaches summarized in Table 13 that are mitigated by the proposed unified communication and query execution mechanisms.

## 5.2 The INetCEP Architecture

This section presents the overall design of the proposed networking architecture. First, Section 5.2.1 presents the core ideas to solve the research problems introduced in Section 5.1. Later, Section 5.2.2 presents the unified communication mechanism and Section 5.2.3 presents the meta query language as well as the CEP query engine.

### 5.2.1 *Core Ideas*

In the following, we present the core ideas of this chapter and how we deal with challenges pertaining to the research questions discussed previously *(i)* unified communication and flow control and *(ii)* processing of continuous data stream and query processing on top of ICN.

#### *RQ2.1: A Unified Communication Model*

*Novel data stream and query packets*

We address the limitations enabling co-existing pull- and push-based communication mechanisms as part of a novel unified communication model. To accomplish this, we enhance the data plane mechanisms of ICN with additional packets and their behavior for the unified communication. Table 14 summarises the key differences to the current ICN architectures, and INETCEP architecture with the proposed unified model. One of the major difference is in the packet types and their handling at the ICN node. We propose three new packet types `Data Stream`, `Add Continuous Interest` and `Remove Continuous Interest`, as well as changes to the data structures to handle them. The handling of the new packets is proposed in the data plane in conjunction with the existing `Interest` and `Data` packets. For instance, we trigger query processing on the reception of a new `Data Stream` packet and we store `Continuous Interest` in PIT and the (transformed) `Data` packets in CS so that the complex event can be retrieved efficiently. To cope with the flow imbalance problem, we propose a lossless rate-based flow control mechanism based on [229], which balances the input event rate using active feedback to the producer based on the available resources on the links and flows. We advance the work [229] by proposing a distributed algorithm for a large network, removing the overhead of management packets, and incorporating the approach for ICN architectures. The proposed flow control mechanism aims to reduce the response times further and prevent any event loss.

*Flow control mechanism to deal with event loss*

#### *RQ2.2: Reactive and Asynchronous Handling of Data Streams*

*Highly expressive meta query language*

We propose network-centric query execution algorithms that reactively handles the continuous flow of data streams and plan execution of queries on the data plane of ICN (cf. Table 14). The algorithms ensure an efficient distribution of operators on the available infrastructure to deliver the required QoS requirements. As noted in the system model (cf. Section 5.1.1), the query $Q$ includes the QoS specification that has to be fulfilled by the INETCEP architecture. Accordingly, the algorithm decides to either resolve the query centrally or in a distributed manner.

| **Feature** | **Classical ICN architecture** | | **INETCEP architecture** | |
|---|---|---|---|---|
| | ICN | Description | INETCEP | Description |
| Packet Types (cf. §5.2.2) | `Interest` | Interest packet is used to express a pull-based request on a data object | `Interest` | Interest packet is used to express a pull-based request on a (transformed) data object |
| | `Data` | Data packet returns a response on the pull-based request including the data object of interest | `Data` | Data packet returns a response on a pull-(`Interest`) or push-based request (`Continuous Interest`) including the data object |
| | - | - | `Data Stream` | Data Stream packet encapsulates a single event tuple from the continuous event stream from the producer |
| | - | - | `Add Continuous Interest` | `Add Continuous Interest` packet expresses a push-based (continuous) request on fetching a (transformed) data object |
| | - | - | `Remove Continuous Interest` | Remove Continuous Interest packet deregisters the push-based request specified by `Continuous Interest` on a (transformed) data object |
| Data Structures (cf. §5.2.2) | PIT | Stores pull-based requests in a table form including the Interest name prefix and the face information | PIT | Stores pull- and push-based requests in a table form including the Interest name prefix and the face information |
| | CS | Stores data objects already served by pull-based request for other consumers in a table form including the Interest name prefix and the data object | CS | Stores (transformed) data objects already served by pull- or push-based request for other consumers. Also acts as an intermediate buffer for stateful operators like windows |
| | FIB | Stores routing information towards producers for the incoming requests in a table form including the data object name prefix and face information | FIB | Stores routing information towards producers for the incoming pull- or push-based requests in a table form including the data object name prefix and face information |
| Data Processing (cf. §5.2.3) | - | - | CEP engine | A meta query language to express CEP queries, a query parser, and query deployment module to process and derive complex events |

Table 14: Key differences of INETCEP network architecture to traditional ICN architectures ("-" means no support). Here, PIT is Pending Interest Table, CS is Content Store, FIB is Forwarding Information Base, and CEP engine is the Complex Event Processing engine [139, 26].

*Network-centric operator placement*

An operator placement mechanism is proposed to distribute the query on to a distributed set of fog-cloud infrastructure comprising of ICN nodes. The operator placement is complemented with the rate information from the flow control mechanism that ensures the correctness of delivered events. Using the operator graph data structure, we decompose the query into primitives, so-called *sub* `Continuous Interest` (s) that are executed on the ICN data plane. Furthermore, parallel processing of sub `Continuous Interest` ensures a short response time and deals with higher event rates requirements of IoT applications.

### 5.2.2 *Unified Communication Model*

This section provides the description of the unified communication model providing a solution to RQ2.1 (how to enable continuous data stream processing?). We identified four limitations of the alternative approach that we mitigate with our design (cf. Table 13). In the following, first, we provide an overview on the unified communication model. Later, we explain the key components of the model.

#### *Conceptual Overview*

The proposed unified communication model supports both push- and pull-based communication mechanisms. This is accomplished by integrating three novel packet types in ICN architecture, `Data Stream` packet that encapsulates the continuous event stream, `Add Continuous Interest` packet that initiates a continuous query or simple interest, and `Remove Continuous Interest` packet that initiates the removal of the continuous query or interest. In the following, we explain how both the communication mechanisms are enabled in our approach.

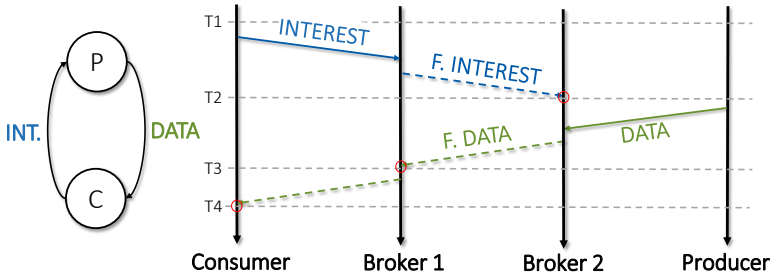
*Interest can encapsulate both request and query*

In the pull-based communication mechanism, as illustrated in Figure 36 a, the consumer initiates a request on a data object by sending an `Interest` packet. The `Interest` packet is forwarded by the brokers towards the producer based on the FIB entries, much like the routing table in IP protocol. On the way to the producer, the PIT entries are updated so that the data object can follow the path back to the consumer. As soon as the producer receives the `Interest` packet, the data object in the `Data` packet matching the $name$ is sent back to the consumer. Recall, that $name$ represents the content of the data objects corresponding to the `Data` packet in ICN.

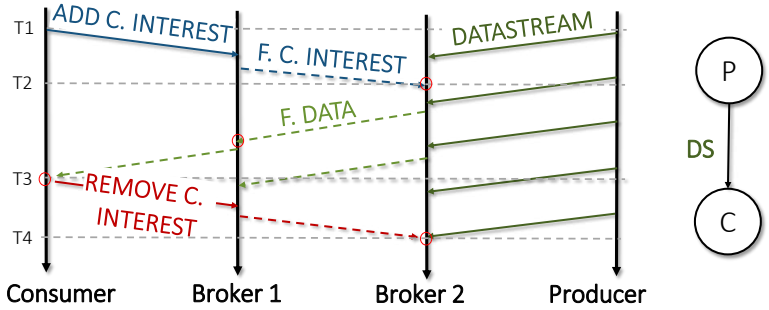
We propose the push-based communication mechanism as illustrated in Figure 36 b. Here, the producer initiates communiciation by sending a con-

tinuous data stream packet `Data Stream` towards the broker network (as seen in the figure comprising of two brokers). The producer uses multicast for the `Data Stream` packet, in order to disseminate it to all the brokers, so that the same can be forwarded towards the consumers. The consumer expresses interest in a higher-level event or a complex event using a Query $Q = \{q, qname, QoS, \tau_{qos}\}$ encapsulated in an `Interest` or `Add Continuous Interest` packet (Add C. Interest in Figure 36 b). Here, $qname$ is used to identify the query in PIT similar to the name prefix $name$ is the `Interest` packet in consistent to ICN architectures. In contrast to the `Interest` packet, the `Add Continuous Interest` registers a continuous query such that the results are delivered to the consumer unless it is removed. The consumer sends a `Remove Continuous Interest` packet to remove the query identified using the $qname$ by removing the entry from the PIT (Remove C. Interest Figure 36 b).

*Continuous interest for push-based mechanism*




(a) Consumer-initiated communication mechanism.




(b) Producer-initiated communication mechanism.

Figure 36: The unified communication mechanisms supported by INETCEP. The subfigure (a) shows the pull-based mechanism, while (b) shows the push-based mechanism [139].

In this way, a basic pull- and push-based communication is established in the proposed unified communication model. In the next sections, first, we describe the proposed packet types and its structure in detail. Second, the handling of the new packet types in the node components, namely, PIT, FIB, CS and the novel CEP engine are discussed. Finally, the flow control mechanism that enables efficient processing of continuous data streams in ICN is presented.

### *Packet Types and Structure*

To better understand how we resolve the limitations summarized in Table 13, we first explain the newly introduced packets and the existing packets suported by INETCEP. Table 15 shows the packets supported by INETCEP, including the basic description. Each packet in ICN includes a fixed-size header depending on the packet type, a variable-length data object name prefix, optional type length value format representing the packet encoding, and the payload that includes the data object (cf. Chapter 2: Figure 7) [139]. In addition to the newly introduced packets, we modify the `Interest` and `Data` packets as well to include data transformation function, which is explained as follows.

*Discussion of new packets*

*(i)* The `Interest` packet is used to express a *pull-based* request on a simple name prefix that fetches a data object or a CEP query. It can transform the data object to deliver higher-level information to the consumer, however, not in a continuous form. In this way, the `Interest` packet fulfills the basic functionality required by classical ICN architectures, as well as support transformation of data for CEP. The consumer can use an `Interest` packet to issue what we call a *pull-based* query that is encapsulated in the variable-length name field in the packet. In contrast to `Continuous Interest`, the function of this query is to fetch the aggregated data only once and not continuously. The packet structure comprises a common fixed-size header including the msg type and msg length, the variable lenght name and TLV fields.

*(ii)* The `Data` or *reply* packet includes the data object requested by the consumer that satisfies an `Interest` packet or the `Add Continuous Interest` (cf. point *(iv)* ). The packet structure is similar to an `Interest` with an addition of the payload that contains the data object.

*(iii)* The newly introduced `Data Stream` packet encapsulates the continuous data stream arriving from the producers. Recall, a data stream is an unbounded stream comprising a set of event tuples $e = \{(k_1, v_1), \ldots, (k_{max^e}, v_{max^e})\}$, where each event tuple is timestamped (cf. Section 3.2.1). Here, each packet encapsulates a single event tuple e.g., $(k_1, v_1)$. However, based on the size of the packet in the respective ICN hardware, this is reconfigurable and can be adapted.

*(iv)* The `Add Continuous Interest` packet represents a `Continuous Interest` encapsulating a CEP query $Q$. In contrast to an `Interest` packet, when an `Add Continuous Interest` packet is received at an ICN node, a persistent PIT entry is created that remains until the consumer explicitly removes the `Continuous Interest` using a `Remove Continuous Interest` (cf. point *(v)* ). Therefore, using

| Packet Type Code | Packet Type | Packet Description |
|---|---|---|
| 0x00 | `Interest` | Interest packet to express a request or a query on a (transformed) data object. |
| 0x01 | `Data` | Data packet comprising a (transformed) data object to satisfy a request expressed by an Interest. |
| 0x02 | `Add Continuous Interest` | Add Query Interest packet to express a continuous query $Q$ or an interest. |
| 0x03 | `Remove Continuous Interest` | Remove Query Interest packet to remove a (continuous) query or an interest. |
| 0x04 | `Data Stream` | Data Stream packet representing continuous event stream comprising event tuples. |

Table 15: Extended packet type codes (0x02 – 0x04) in comparison to standard packet types in ICN architectures (0x00 – 0x01) (extension to Chapter 2: Table 3).

a single packet, the query result is continuously received by the consumer because of so-called persistent PIT entries to resolve the **first limitation** (cf. Table 13). The structure of this packet is similar to an `Interest` packet with a distinction in the persistence of the PIT entry.

*(v)* The `Remove Continuous Interest` packet is used to remove the persistent entry of `Continuous Interest` from the PIT. It indicates that the consumer is no more interested in getting notifications on the previously registered query given by `Add Continuous Interest`. The handling of the packets is done in the forwarding plane of ICN architecture, as later explained in Section 5.2.2.

In the following section, we detail on the INETCEP node components and how we deal with the limitations of existing approaches.

### *Node Components*

In this section, we introduce the components of an ICN node of INETCEP architecture. Each node maintains a CS or a cache, a PIT, a FIB table and a CEP Query Engine. The cache or CS acts as a store to the (transformed) data objects fulfilling the request from consumer expressed using an `Interest` or `Add Continuous Interest` packet. *Example:* A CS entry for an `Interest` packet with $name$ $/node/temperature$ stores a temperature value, e.g., 45F, while for an `Add Continuous Interest` packet with $qname$ expressing transformation on the temperature readings such as $min$, $max$ or $avg$ stores a minimum, maximum or average from the set of temperature values given by the query $Q$ of `Add Continuous Interest` packet. Basically, it stores the $name$ or $qname$ with the result of each respectively. Therefore, if multiple consumers specify inter-

est in the same (transformed) data object simultaneously, the result can be retrieved from the CS and the query does not have to be reprocessed.

*Query sent once, processed continuously*

In the periodic request mechanism (**first limitation** in Table 13), the producer is continuously polled and would need to reprocess data each time. In contrast, in our proposed unified communication mechanism, the request is only received *once* and processed periodically for each new `Data Stream` packet. To ensure that no stale data objects are returned from the CS and handle the **second limitation**, a reactive mechanism is proposed in the CEP Query Engine that handles query processing each time a new `Data Stream` packet is received, as explained in the next subsection. The challenge here lies in the resolution, decomposition, ution of the query into sub interests while complying with the strict QoS demands of the consumers. The reactive mechanism employed in the Query Engine ensures that the always up to date result of the query is stored in the CS by discarding the old entries.

*Always fresh data in Content Store*

The PIT acts as a store to the requests expressed by `Interest` and `Add Continuous Interest` packets that are not yet fulfilled. Basically, it stores the requests from the consumers, such that the (transformed) data objects can be returned to the interested consumers. The $face$ and $name$ or $qname$ of the incoming `Interest` or `Add Continuous Interest` packet, respectively, is stored in the PIT. Here, $face$ stands for the incoming interface of the downstream node, e.g., a consumer, $name$ stands for a name prefix of the incoming `Interest` packet and $qname$ stands for the name prefix of the query. We distinguish between the `Add Continuous Interest` and `Interest` packets in terms of persistent and transient PIT entries, respectively[37]. Meaning that when an `Add Continuous Interest` packet is received, a PIT is persistent, such that it is not removed when the transformed data object satisfying the query is received, but the data objects are continuously sent to the consumer. The PIT entry for an `Add Continuous Interest` is removed on the reception of a `Remove Continuous Interest` packet.

*PIT stores persistent query requests, which...*

In contrast, a transient PIT entry for an `Interest` is removed once the data object satisfying the request is received at the node. Therefore, the reactive mechanism and persistent PIT entries handle the **third limitation** (cf. Table 13) of a three-way message exchange. Furthermore, to deal with the **fourth limitation**, `Remove Continuous Interest` is introduced that removes the PIT entry once the consumer is no more interested in getting query updates.

*... are removed if consumer is not interested*

The FIB table maintains the routing information towards the producer as the `Data Stream` is received at the broker nodes. This information is used to forward the `Interest` or the `Add Continuous Interest` packets towards the pro-

[37] Each PIT entry has a flag indicating that the entry is related to an `Add Continuous Interest` or an `Interest` packet.

ducer. The FIB table also stores the $face$ and $name$ information. Finally, each node maintains a CEP query engine responsible for *(i)* query specification using the meta query language, as well as *(ii)* parsing, placement, and execution using the proposed query execution algorithms (cf. Section 5.2.3). In the following, we explain how the newly introduced packet types are handled in the data plane of ICN to achieve network-centric event procesing.

### *Data Plane Handling*

Algorithm 3 (lines 1-30) and Figure 37 together define the packet handling for `Add Continuous Interest` and `Data Stream` packets on an ICN node. In the algorithm, we define *events*, *packets*, and their *handlers* using the widely used definition given by the asynchronous event-based composition model in [210]. Therefore, events, packets and their attributes are denoted as: ⟨*Event or Packet Type*|*Attributes,...*⟩.

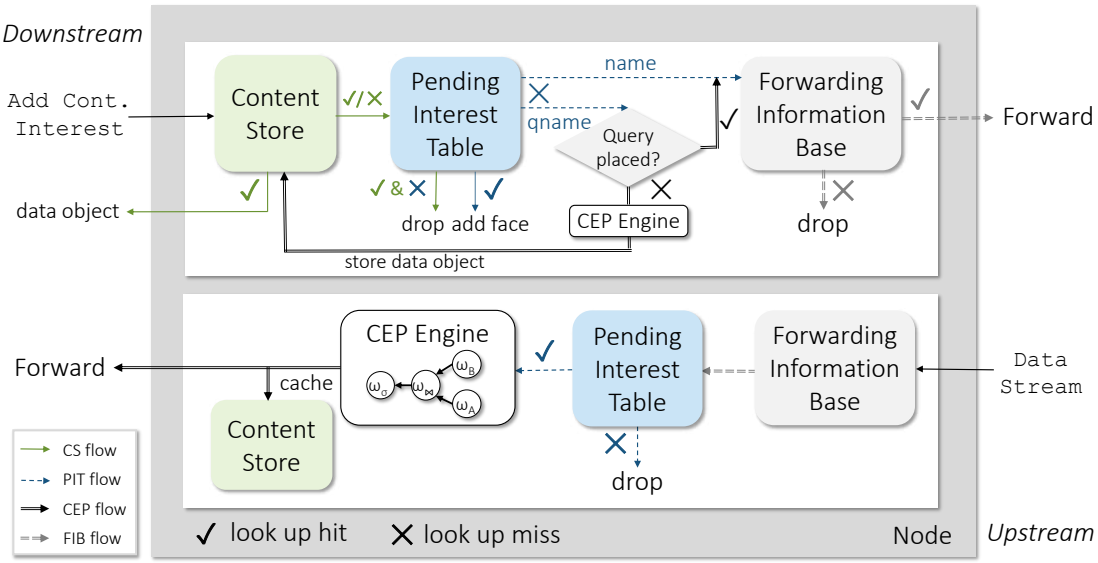



Figure 37: Data plane handling of newly introduced packets `Add Continuous Interest` and `Data Stream`.

*Handling of continuous interests with query in ICN*

On the arrival of an `Add Continuous Interest` packet, an ICN node looks up the CS table for the latest transformed data object that matches the $qname$ (i.e., the name prefix of the query). If the latest matching entry exists and no `Data Stream` packet arrived after reception of the `Add Continuous Interest` packet, the node forwards the transformed data object to the downstream node (consumer or another broker). Additionally, if an entry in the PIT is found, the face of the downstream node is updated in the PIT. This is different than the handling of an `Interest` packet because of two reasons. *(i)* For `Interest` packet, a transient PIT entry is created, while for `Add Continuous Interest` a persistent PIT entry is created. The match found earlier in the CS

entry may correspond to the `Interest` packet instead of the `Add Continuous Interest` packet. Hence, the PIT entry for the `Add Continuous Interest` packet has to be updated. *(ii)* The face information needs to be updated since the `Add Continuous Interest` may originate from another node other than the one already existing in the PIT table. Also, since the consumer has to be notified continuously, the persistent PIT entry must be updated. In contrast, if a PIT entry is not existing and a CS entry is existing, the packet is discarded since the node is only used for forwarding (lines 2-7).

**Algorithm 3 :** Packet handling for `Add Continuous Interest` and `Data Stream` packets [139].

**Variables :**

| | | |
|---|---|---|
| $CS$ | $\leftarrow$ | Content Store table of the current node |
| $PIT$ | $\leftarrow$ | Pending Interest Table of the current node |
| $FIB$ | $\leftarrow$ | Forwarding Information Base table of the current node |
| $qname$ | $\leftarrow$ | Name prefix representing a CEP query |
| $facelist$ | $\leftarrow$ | List comprising face information for matching qname in PIT |
| $eventTuple$ | $\leftarrow$ | Event tuple contained in `Data Stream` packet's payload |
| $data$ | $\leftarrow$ | (Transformed) data object found in the Content Store |
| $ts$ | $\leftarrow$ | Timestamp of current event tuple in the `Data Stream` packet |

```
1  upon receipt ⟨ Add Continuous Interest |qname⟩ packet do
2      if qname is found in CS.LOOKUP() then
3          data ← CS.FETCHCONTENT(qname);
4          if qname is found in PIT.LOOKUP() then
5              PROCESSPIT(qname, Add Continuous Interest);
6          return data;
7          (Discard Add Continuous Interest packet)
8      else if qname is found in PIT.LOOKUP() then
9          PROCESSPIT(qname, Add Continuous Interest);
10     else if qname is found in FIB.LOOKUP() then
11         PIT.ADDFACE(qname);
12         trigger ⟨deployAndProcess|qname⟩ (Refer Algo. 5);
13         Forward Add Continuous Interest;
14     else
15         Discard Add Continuous Interest packet ;
16 upon receipt ⟨ Data Stream |ds⟩ packet do
17     for each qname ∈ PIT do
18         if eventTuple satisfies qname then
19             PROCESSPIT(qname, Data Stream) ;
20         else
21             Discard DataStream packet;
22 function PROCESSPIT(qname, packet)
23     if packet is DataStream and packet.ts > qname.ts then
24         trigger ⟨deployAndProcess|qname⟩ (Refer Algo. 5);
25         Forward packet;
26     else
27         facelist ← PIT.GETFACES(qname);
28         if qname.face is not found in facelist then
29             PIT.ADDFACE(qname);
30         Discard packet;
```

If a CS entry is not found, the node continues to look into the PIT for the $qname$ (lines 8-9). If found, the new face is added to the PIT (lines 27-29), which means that the current query is in execution. As soon as a result is generated, it is forwarded to the respective consumers registered in the PIT (lines 28-30). Otherwise, if the PIT entry is not found, a new PIT entry is created and the `Add Continuous Interest` packet is forwarded towards the producer based on the FIB entry (lines 10-15). In this step, we distinguish between the conventional request given by name prefix $name$ or a push-based query given by name prefix $qname$. If it is a simple request (with $name$), it is directly forwarded towards the producer based on the FIB entry. Otherwise (if $qname$) and the query given by $qname$ was not earlier placed, the push-based query is placed by the CEP query engine, and the sub query interests (`Continuous Interest`) are forwarded to other brokers for placement and processing (as defined later in Section 5.2.3). Similar to the `Add Continuous Interest` packet, the `Interest` packet with a standard request $name$ is handled conventionally similar to ICN architectures, while a pull-based query $qname$ is handled equally as `Add Continuous Interest` packet with a distinction of a transient PIT entry and that the query result is retrieved only once but not continuously (pull-based). Furthermore, if a data object satisfying a $qname$ is found in CS then the PIT entry is no more checked for `Interest` packet, because it is of no more interest to the consumer (as it has received the data object once).

*Difference between plain requests and queries*

A `Data Stream` packet is handled in a similar way like a `Data` packet (lines 16-21), with the exception of query execution on the reception of each new packet (lines 23-25). First, the FIB table is populated to keep track of the path to the producers to forward the `Interest` and `Add Continuous Interest` packets. Second, if an entry in the PIT exists, the event tuples are processed by the CEP engine, as explained in the later section. Essentially, it executes the operator dictated by the $qname$ on the event tuples given by the `Data Stream` packet. Otherwise, if the PIT entry is not found, the `Data Stream` packet is discarded. This is because there are no downstream nodes registered for the `Data Stream` packet (as the PIT entry is missing).

*Handling of data stream packet*

**Algorithm 4 :** Remove continuous interest handling.

**Variables** : $PIT$ ← pending interest table of current node
$qname$ ← qname to be removed from PIT

1 **upon** ***receipt*** ⟨`Remove Continuous Interest` |$qname$⟩ *packet* **do**
2  **if** $qname$ *found in* $PIT$.LOOKUP() **then**
3  $PIT$.REMOVE($qname$);
4  Forward `Remove Continuous Interest`;
5  **else**
6  Discard `RmQInterest`;

*Handling of remove continuous interest*

Algorithm 4 explains the handling of a `Remove Continuous Interest` packet. On the reception of a `Remove Continuous Interest` packet, the node checks if a persistent PIT entry indicated by the flag for the $qname$ given by the `Remove Continuous Interest` packet exists. If an entry is existing, it is removed, and the packet is forwarded towards the next node based on the FIB entries. Otherwise, the packet is discarded since this node did not process the query in the first place (lines 1–6).

In summary, we show that coexisting pull- and push-based communication mechanisms can be enabled in the data plane of ICN architecture while complying with the basic principles of ICN architectures involving pull-based communication. In the next section, we detail how flow balance is achieved while processing continuous data streams in the data plane of ICN.

### *Rate-based Flow Control*

*How to handle flow imbalance*

As noted in the research challenges, it is important to deal with the flow imbalance problem, which could result in heavy inconsistencies in the delivery of complex events and hence also affect the accuracy of the former. It is a known problem in IP-based push-based approaches. For instance, current CEP systems such as Apache Flink deals with this problem using a credit-based flow control mechanism[38] with many similarities to the rate-based flow control, still it experiences many event loss [229]. In contrast, the proposed flow control mechanism is lossless, as it avoids overflowing the input buffer of events by keeping the event rate always below or equal (optimal) to the network capacity [230]. We advance the previous work by introducing a distributed algorithm and reducing the overhead of management packets by efficiently utilizing standard packet types in ICN architecture.

*Adapt flow based on rate information*

Each producer maintains an estimate of an optimal event rate based on the capacity of the interconnected network. In the beginning, the producer issues event tuples (encapsulated in `Data Stream` packet) based on its desired event rate, which is updated based on the estimates provided by the intermediate packets. As introduced above, each `Data Stream` and `Data` packet are appended with two new fields, *underloading* bit or *u-bit* and the event rate estimate known as *stamped rate*. Each time the producer sends the next `Data Stream` or `Data` packet, it puts the current rate estimate in the *stamped rate* field and clears the *u-bit*. By monitoring the event traffic, each ICN node, including the producer, calculates the flow capacity of its incoming and outgoing links. This quantity is called the *advertised rate*. Therefore, before sending the next `Data Stream` or `Data` packet, the producer puts its current estimate

[38] A deep dive into Flink´s network stack. `https://flink.apache.org/2019/06/05/flink-network-stack.html` [Accessed in May 2021].

in the *stamped rate* field and sets the u-bit of outgoing `Data Stream` or `Data` packet to 1. When this packet reaches the next ICN node, it compares the received stamped rate to its advertised rate. In case the stamped rate is greater than the advertised rate, then the stamped rate is set to the advertised rate, and the u-bit is set in the `Data Stream` or `Data` packet. Otherwise, the fields remain unchanged in the respective packet.

When the consumer receives the `Data Stream` or the `Data` packet, the stamped rate is the minimum of the rate estimate that the flow allows between the broker nodes and the producer. The consumer acknowledges the reception of the `Data Stream` or `Data` packet to the producer. After a complete round trip of the network, the u-bit indicates if the flow is constrained along the path to the consumer or not. In other words, if the rate is limited by the flow, then the producer adjusts its event rate accordingly. Otherwise, the event rate is increased until the advertised rate threshold is reached. The flows where the advertised rate matches the stamped rate are the preferred links and are used for the operator placement (as discussed in Section 5.2.3).

Theoretically, the advertised rate can be computed as follows. Each ICN node maintains a list of all seen stamped rates which is referred to as recorded rates. The set of flows where the recorded rates are higher than the advertised rates are referred to as unrestricted flows, $\mathcal{U}$. In contrast, the set of flows with recorded rates lower than the advertised rates are referred to as restricted flows $\mathcal{R}$. The flows in the restricted set $\mathcal{R}$ are the bottleneck, while the flows in the unrestricted set $\mathcal{U}$ are continuously monitored to recompute the advertised rates. Given this information, the advertised rate $\mu$ can be calculated as follows:

$$\mu = \frac{\mathcal{C} - \mathcal{C}_{\mathcal{R}}}{\mathcal{N} - \mathcal{N}_{\mathcal{R}}}. \tag{12}$$

In the above equation, $\mathcal{C}$ is the overall capacity of the link, $\mathcal{C}_{\mathcal{R}}$ is the overall capacity of the restricted set, $\mathcal{N} = f + kb$ and $\mathcal{N}_{\mathcal{R}} = f_{\mathcal{R}} + kb_{\mathcal{R}}$, where $f, b, f_{\mathcal{R}}$ and $b_{\mathcal{R}}$ are the total and restricted outgoing and incoming flows of the respective links. When $k = 0$ and $k = 1$, $\mathcal{N}$ and $\mathcal{N}_{\mathcal{R}}$ are total and the restricted number of flows traversing from the link, respectively.

Therefore, by controlling the event rate based on the estimates on the set of flows, we prevent overflowing the input buffers with events and hence event loss. Furthermore, by preferring nodes connected with unrestricted flows for placement, we maintain high throughput of events.

### 5.2.3 *CEP Query Engine*

This section provides the description of the CEP query engine that operates in the data plane of ICN (solution to RQ2.2: how to improve efficiency in executing CEP queries?). In this section, we solve the limitations in terms of expressing queries to be resolved in the data plane of ICN. Particularly, fifth and sixth limitations discussed in Section 5.1.2: Table 13. In the following, first the meta query language is explained and then the query deployment component, including query parsing, placement and execution is explained.

#### *Meta Query Language*

*An expressive CEP meta language for ICN*

This section presents a meta query language to specify CEP queries on the ICN data plane solving the **fifth limitation** i.e., lack of query specification abstractions in current ICN architectures (cf. Table 13). The *meta* query language acts as an abstraction on top of the standard naming scheme of ICN, such that the language can be applied to a variety of ICN architectures following hierarchical naming schemes such as Named Data Networking [137], Content-Centric Networking [129] and Named Function Networking [132]. The query language aims towards three main design goals: *(i)* specification of both push- and pull-based queries, *(ii)* interpreting the query to an equivalent naming prefix for ICN architectures, and *(iii)* support for standard relational algebraic operators while ensuring extensibility to additional operators and even custom operators for novel IoT applications.

*Both pull- and push-based queries can be processed*

We differentiate between push- and pull-based queries depending if the incoming data is continuous or not. Particularly if a single `Data` packet is received or a continuous `Data Stream` is received at the node. This heavily changes the way operators function in a CEP engine, especially for the stateful operators. In the following, we will show the difference using an example of a push- and pull-based window operator.
*Example:* Recall that a window operator accumulates event tuples for a given window size indicated by either a tuple- or a time-based window, such that those event tuples can be transformed using an operator succeeding the window operator in the operator graph. A key difference in the handling is that for every event tuple, a new packet is fetched by the pull-based window until the window is complete. Whereas a push-based window operator accumulates the event tuples until a window size is reached and increments the window slice for the next window while purging the old state.

Similarly, other operators either dependent on a window or independent, fetch new event tuples explicitly from the producer in the pull-based implementation. In contrast, in the push-based implementation, the incoming

tuples are advanced automatically by the incoming `Data Stream` packets. The query result is encapsulated in a `Data` or a `Data Stream` packet depending on the kind of result. For instance, a join on two continuous data streams is stored into a `Data Stream` packet, while an aggregation on a window of the combined data stream is stored in a `Data` packet. The query parser, placement and execution are defined in the next subsection. In the following, we show the example specification of standard CEP operators using the meta query language.

*Specification in meta language*

```
1 WINDOW(GPS_S1, 4s)
```

Listing 2: A window size of 4 seconds from a gps source one data stream [26].

```
1 FILTER(WINDOW(GPS_S1, 4s),'latitude'<50)
```

Listing 3: Filter on the window of size 4 seconds with latitude value of less than 50 pts [26].

```
1 JOIN(
2   FILTER(WINDOW(GPS_S1, 4s), 'latitude'<50),
3   FILTER(WINDOW(GPS_S2, 4s), 'latitude'<50),
4   GPS_S1.'ts' = GPS_S2.'ts'
5 )
```

Listing 4: A join on the window size of 4 seconds for the two data streams gps source one and two where the event time matches [26].

```
1 SUM('speed', Window(GPS_S1, 4s))
```

Listing 5: A sum aggregate over a window of size 4 seconds arriving from a gps source one data stream [26].

```
1 SEQUENCE(
2   FILTER(WINDOW(GPS_S1, 1s), 'latitude'=50) →
3   FILTER(WINDOW(GPS_S2, 1s), 'latitude'=50)
4 )
```

Listing 6: Detects if gps source one arrives before gps source two at a certain location given by the latitude value [26].

Section A.2 presents the query language grammar based on [231] and extensibility aspects based on the scenarios. In the next sections, we detail on query parser, placement, and its execution.

### *Query Deployment*

Algorithm 5 presents the query deployment process, including parsing, placement and execution of a query. This process solves the **sixth limitation** by proposing a distributed, parallel and asynchronous algorithm for query deployment. As noted in the data plane handling, the `Add Continuous`

`Interest` and `Data Stream` triggers the query deployment (cf. Section 5.2.2). The consumer issues a `Continuous Interest`, including a query using an `Add Continuous Interest` packet that triggers the query placement using the query engine. If the query result is not found in the CS and a PIT entry with $qname$ is also not found, the query placement is initiated by the $deployAndProcess$ event (cf. Algorithm 3: Line 12).

If the query given by the $qname$ is new, first, the query is parsed. In the later steps, the query parser transforms the query into an operator graph (lines 1–4 and 12–21). Next, the interpreter constructs an equivalent name prefix (lines 12-15) used to resolve the query in the given ICN architecture. Afterwards, the query is placed on the path that fulfills the given QoS objectives of the query (lines 5–9). Finally, the query is executed on the nodes given by the path (lines 10–11).

*Network-centric query parsing, placement and execution*

**Algorithm 5 :** Operator tree creation, placement, and processing [139].

**Variables :**

| | | |
|---|---|---|
| $query$ | ← | Input CEP query |
| $\tau curList$ | ← | A top-down list of three $\omega$ of tuple $\tau$ |
| $\omega_{cur}$ | ← | The current operator |
| $planNode$ | ← | A single $\omega$ node from the operator graph |
| $curNode$ | ← | The current node |
| $allNodes$ | ← | A list of all nodes |
| $paths$ | ← | The possible paths that can be used for operator placement |
| $opPath$ | ← | The chosen (optimal) path for placement |

```
1  upon event ⟨deployAndProcess|qname⟩ do
       // Query parsing
2      if query ∈ qname is new then
3          τcurList ← GETCURLIST(query);
4          curNode ← PARSEQUERY(τcurList);
       // Query placement
5      if query ∈ qname is not placed or query must be replaced then
6          allNodes ← GETNODESTATUS(curNode);
7          paths ← BUILDPATHS(curNode, allNodes);
8          opPath ← FINDOPTIMALPATH(paths, allNodes);
9          DEPLOYOPERATORS(optimalPath, allNodes);
       // Query processing
10     for all ω_cur ∈ τcurList do in parallel
11         PROCESSOPERATOR(ω_cur, optimalPath);
12 function PARSEQUERY(τcurList)
13     ω_cur ←GETOPERATOR(τcurList);
14     nfnExp ← CONSTRUCTNFNQUERY(ω_cur);
15     planNode ← new PLANNODE(nfnExp);
16     if size(τcurList) == 1 then
17         return node;
18     else if size(τcurList) > 1 then
19         PARSEQUERY(τcurList.left);
20         PARSEQUERY(τcurList.right);
21         return planNode;
```

In detail, the parser transforms the query into a binary operator graph $G$ given by a tuple $\mathcal{T} = (L, S, R)$, where $L$ and $R$ are binary sub trees comprising operators, and $S$ is a singleton set representing a single operator ($\omega$). The operator graph is formed top-down while specifying the dependencies in the data flow between the operators. Typically, in an operator graph, the dependencies flow is defined in ascending order with the lowest level of a leaf operator dependent on all its successor operators until the root operator $\omega_{cur}$ (line 4), where the subscript $cur = root$ in the first step. The new operators are allocated by traversing the operator graph in depth first pre-order manner[39] by visiting the successor and then left (L) and right (R) subtrees (lines 12-21).

*Operator graph formation*

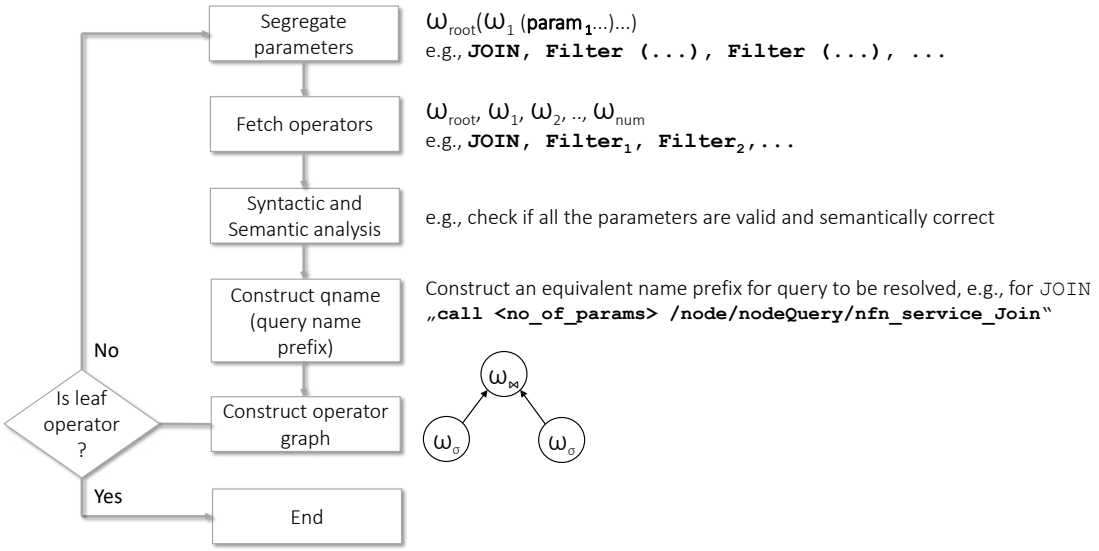



Figure 38: An example of join query parsing as explained in Algorithm 5 [139].

The parser simultaneously maps the subtrees or intermediate subqueries to the corresponding name prefix given by so-called lambda expressions to execute on ICN nodes while obtaining the operator graph.

*Query parsing example*

*Example:* We explain this using an example of the join query defined in Query 4, as seen in Figure 38. The query of the form given in Query 4 is received as an input to the parser, which in the first step, segregates the parameters to perform syntactic and semantic checks on the operators. The operators are identified and mapped to their respective functionalities in the system, followed by syntactic and semantic checks on them. Afterwards, each operator node in the operator graph is transformed to an equivalent name prefix $qname$ or sub `Continuous Interest` as previously introduced. This process continues recursively until leaf nodes in the operator graph are reached. Those sub `Continuous Interest` (s) are then resolved in a centralized or distributed manner on multiple ICN nodes. Overall, the meta query language

[39]Worse case traversal time is linear in the number of operators ($\Omega$).

provides an abstraction on top of lambda expressions representing name prefix for queries in ICN architecture so that CEP developers can easily create queries on top of ICN architectures without any domain knowledge. Specifically, the interpreted lambda expression for a `JOIN` query is given as follows.

*Interpreter representation*

```
1 (call <no_of_params> /node/nodeQuery/nfn_service_Join
2 (call <no_of_params> /node/nodeQuery/nfn_service_Filter
3 (call <no_of_params> /node/nodeQuery/nfn_service_Window
4  4s), 'latitude' < 50) (call <no_of_params>
5 /node/nodeQuery/nfn_service_Filter
6 (call <no_of_params> /node/nodeQuery/nfn_service_Window
7  4s), 'latitude' < 50) GPS_S1.'ts' = GPS_S2.'ts')
```

As noted in the listing above, it is not even complex but also not easily readable, which is why we need our meta query language. In the above expression, `<no_of_params>` represents the number of parameters in the respective lambda function, `nfn_service_Join` states the name of an operator, i.e., a join operator followed by filter and window operator of $4s$ window size. The name prefix of each operator is preceded by `/node/nodeQuery/..` that is dynamically replaced by the name of the ICN node that processes the query.

*Network-centric monitoring of QoS*

Each placement algorithm requires a means to share monitoring information about the ICN node and the network to determine the ICN node that fulfills the QoS demands of the application. For this purpose, we make use of `Data` packets with a flag indicating a management packet. In our design, a placement module maintains information on the nodes and the network connecting them. The information includes runtime statistics on the node and the network about the QoS metrics specified by the query, such as end-to-end delay, load, network usage, etc., but also the rate information estimated by the flow-control mechanism (cf. Line 6). The monitored information is encapsulated in the payload of the management `Data` packet in the form of a tuple: $start|b_i|end = lat|NU|\ldots|\mu_p$. Here, on the left side of tuple, $start$ denotes the initial node given by the path, $b_i$ denotes the intermediate ICN nodes or brokers on the path, and $end$ denotes the last node on the path (whose metrics are included in the packet). On the right side, $lat$ is the end-to-end latency of the path, $NU$ is the network usage, $\ldots$ denotes other QoS metrics that can be included, and the last parameter $\mu_p$ is the rate estimate of this flow on the respective path.

*Rate information is used for placement*

In particular, as an `Add Continuous Interest` packet is received at a node, it receives the node status information denoted by prefix $node/nodeStatus$. The initial ICN node acts as a placement coordinator and determines the path where the query given by $qname$ is placed (Line 5–9). The placement is determined only initially when the query is first received or when a placement update request is received to keep the placement overhead bare minimum.

So far, we have defined the formulation of the operator graph and its placement on the distributed ICN network.The sub `Continuous Interest` given by the query, e.g., filter and join operators in the join query, are deployed on the nodes determined by the placement. Furthermore, by sending the sub `Continuous Interest`, the PIT is populated so that the respective ICN nodes deployed with a query receive the intermediate flow of events, which are required to process the sub `Continuous Interest`. Depending on the flow of the operator graph, the intermediate `Continuous Interest` are processed in a parallel manner, e.g., a join of two windows, where windows are processed in parallel. In this way, the operator graph is processed in a distributed asynchronous and parallel manner in the ICN data plane (line 10–11).

## 5.3 Evaluation

We evaluate INETCEP in a two-fold manner:

*Key evaluation questions*

**EQ1:** Does the unified communication mechanism enable both pull- and push-based communication mechanisms, and can it compete with the existing CEP system, and ICN architecture?

**EQ2:** How is the query performance for a centralized and distributed setup in an edge-cloud topology?

In the following, we answer the above evaluation questions. Before presenting the evaluation results, Section 5.3.1 presents the evaluation environment. Afterwards, Section 5.3.2 presents the performance analysis of the unified communication mechanism, and Section 5.3.3 presents the performance evaluation of the INETCEP query engine.

### 5.3.1 *Evaluation Environment*

This section describes the INETCEP implementation, evaluation platform, datasets and queries, metrics used for evaluation and finally, baselines used for comparison.

#### ***Implementation***

*Implementation on standard ICN architecture*

We built the INETCEP system on top of widely used ICN architecture Named Function Networking [132], [61]. The CEP operators are embedded into the

so-called *named functions* used in the name prefix as lambda expressions as explained in the design. Since there is no way to support coexisting push- and pull-based communication mechanisms currently in Named Function Networking, we implemented the unified communication mechanism on top of the ICN architecture. It works together with CCN-lite [62] (written in C++), which provides a lightweight implementation of CCNx and Named Data Networking protocols that are standard protocols used in ICN. We implemented the unified communication mechanism (cf. Section 5.2.2) in CCN-lite (v2.0.1) in 13,454 LOCs that provides the ICN data structures and its handling. In Named Function Networking architecture (v0.2.1), we implemented the CEP query engine, particularly, the parser, placement and processing algorithms as described in Section 5.2.3 in 19,491 LOCs. INETCEP is available publicly for use here[40].

### *Evaluation Platform*

*CORE and MACI as evaluation platforms*

We used CCN-lite and *Common Open Research Emulator* (CORE) [232] for our evaluations. To repeat the evaluations for different configurations as specified in Table 21, we used MACI [233]. In contrast to simulation-based approaches such as NS-3 [234], the CORE emulator uses Linux namespaces to execute binaries and scripts natively.

| | |
|---|---|
| Experiment time $t_s$ | 20 mins |
| Warmup time $t_w$ | 60 secs |
| Number of runs | 30 |
| Topologies | Manhattan graph (WSN), line and tree-based topologies |
| Queries | Window, Filter, Join, Heatmap, Predict (cf. Query 2– 4, 7, 8) |
| Cloud instances used for distributed setup | GC c2-standard-8 and AWS c5d.2xlarge instances |
| Input event rate | 1000, 10000 and 50000 (Poisson distribution) |
| Our System and Baselines | INetCEP_UCL (ours), Apache Flink [54], Periodic Request (PR) [142] |

Table 16: Configuration parameters used for the evaluation. *Default or commonly used parameters are underlined* [139].

In CORE, services for the individual components of INETCEP such as Query Engine and CCN stack are created and initialized on each emulated

[40] INETCEP webpage: `https://luthramanisha.github.io/INetCEP/` [Accessed in May 2021].

node accordingly. The emulator instantiates a topology provided by the user and exports topology information used to setup CCN-lite and the NFN. It is used to keep track of the experiment configurations, hence all variables to be evaluated, such as the topology and query processing. When running an experiment, MACI creates the cross product of the configuration options and distributes the experiment instances to multiple workers (cloud machines) via the network. The workers execute a single experiment using CORE and return the results, such as log files and performance measurements, to the MACI instance. For a realistic evaluation environment, we used cloud instances from Google Cloud and Amazon Web Services (AWS). The configuration of the machines used on AWS type c5d.2xlarge with 8vCPUs, the processing speed of up to 3.5 GHz, and 16 GiB memory, while in Google Cloud type c2-standard-8 with 8vCPUs, the processing speed of up to 3.8 GHz, and 32 GiB memory. Each machine acts as an ICN node in our evaluation that can host CEP queries. We used real-world topologies for sensor networks, the well known WSN topology [235] called *Manhattan graph*, and a tree topology that resembles edge and data-center networks using a different number of nodes (cf. Figure 39). Each node communicates over ICN protocols, namely CCNx, and NDN instead of IP.

*Cloud resources for deployment*

### *Datasets and Queries*

We use two real-world IoT datasets for the scenarios previously described in Chapter 3: Section 3.1: ② post-accident management and ③ smart plug load prediction application.

*First Dataset.* As a first application, we consider heat map query using a real-world field test mimicking a post-accident situation [64]. The dataset comprises sensor information, including location coordinates. The sensor data stream of the first dataset is given in the following schema.

*Real world workload and queries*

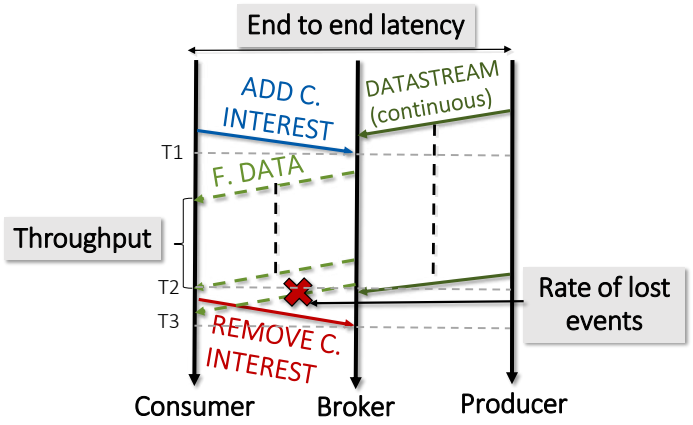
$< ts, s\_id, latitude, longitude, altitude, accuracy, distance, speed >$

*Scenario Query.* We take a heat map query that aids in rescue operations after an accident situation. It takes as an input the sensor data stream and uses the location information to derive an area where survivors are densely located in an accident location. A reference algorithm from the literature [236] is used for this purpose[41].

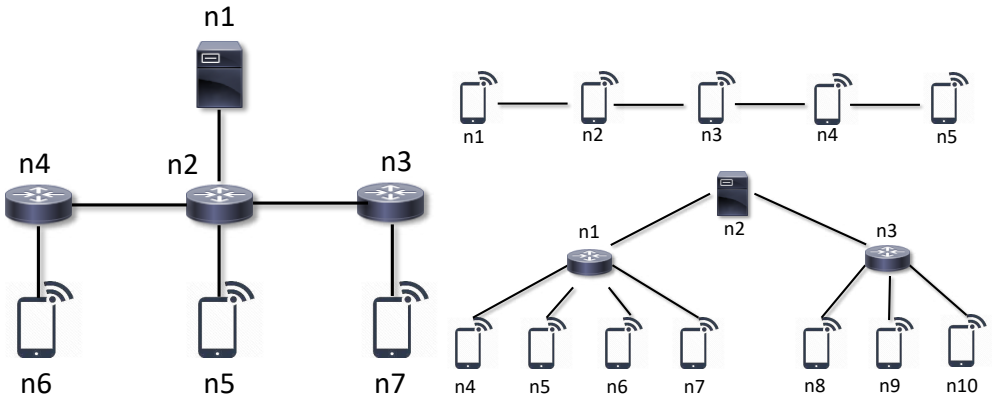
```
1 HEATMAP(
2   'cell_size', 'area',
3   WINDOW(GPS_S1, 4s)
4 )
```

Listing 7: Display the heat map distribution of the first gps source in the given area with a given cell size [26].

[41] In Section A.2, we explain the reference algorithm in details.




(a) End-to-end latency, throughput and loss rate visualized in a simple setup comprising a producer, consumer and a broker.




(b) Topologies used for evaluation: Manhattan graph, line, and tree topology from left to right.

Figure 39: Centralized and Distributed processing setup [139].

*Second Dataset.* In the second application, we consider a load prediction query that predicts when a smart grid could be overloaded. The dataset is based on the Grand Challenge of a premier distributed systems conference DEBS [63]. It is based on the real-world information collected from smart home installations. The dataset represents load measurements from 2001 unique smart plugs as given by the following schema.

$< ts, id, value, property, plug\_id, household\_id, house\_id >$

*Scenario Query.* We take the input queries from the Grand Challenge and provide an existing solution [237] on top of INETCEP architecture[41]. The query applies predictions on a given window size of smart plug load and provides predictions in the future for the given time interval (in the below example, 30 seconds).

```
1 PREDICT(30s, WINDOW(PLUG_S1, 4s))
```

Listing 8: Predict the load on the smart plug source one every 30 seconds for one minute into the future [26].

### *Evaluation Configuration and Plots*

*Configuration parameters for evaluation*

Table 21 presents the overall parameters used for the evaluations. Each evaluation runs for $20$ minutes with a no measurement phase of $60$s. All experiments on the defined metrics are repeated 30 times if not specified otherwise. We evaluate the performance of INETCEP against Apache Flink [54], a widely used publicly available CEP system that has so far the best performance in terms of latency and throughput metrics. Another ICN based approach we evaluate is the periodic request mechanism (PR) (cf. Section 5.1.2), which is also used extensively [142] to enable push-based communication while still using consumer initiated interaction.

*Metrics.* Although INETCEP is capable of delivering specified QoS metric by using the power of programmable networks, we focus on two major metrics that are highly important for CEP systems for our evaluations: end-to-end latency and throughput. Recall, latency is measured as the time interval between the generation of a low-level event and the reception of the complex event at the consumer. As a second metric, we focus on throughput, or precise sustainable throughput (cf. Definition 11) measured as the total number of events received at the consumer per time unit. To evaluate the performance of the lossless flow-control mechanism, we refer to the metric event loss rate (%) given as $\frac{(\text{total events} - \text{processed events})}{\text{total events}} \times 100$ (cf. Definition 13). Lastly, we are interested in measuring the loss in terms of complex events using the accuracy metric (cf. Definition 12). Figure 39a illustrates the evaluated metrics in a single node setup.

*Plots.* We use the Cumulative Distribution Function (CDF), point plots, bar plots and box plots to illustrate the results. CDF shows the cumulative distribution of the investigated metric. It depicts the probability distribution function as $P(X \leq x)$ such that the value of the investigated metric is the probability that $X$ takes a value less than or equal to $x$. The point plot shows a point that depicts the mean value while the error bars report the percentiles including the investigated metric values between the $5$th and the $95$th percentile. The line that joins each point in the plot between the given discrete values in x-axis allows for comparison between the heights of the different groups represented by the x-axis. The bar plot shows the mean of the investigated metric and the error bar on the top, denoting the values between the $5$th and the $95$th percentile. Finally, the box plot represents a five-number summary of the latency measurements, minimum, first quartile (Q1), median,

third quartile (Q3), and maximum. The vertical line seen in the box near Q1 is the median, which is also written explicitly in the figure. The left bound of the box represents the minimum value observed for the investigated metric, Q1 represents the starting value, Q3 represents the ending value, and the right bound of the box represents the maximum value observed for the investigated metric. The outliers are not reported in the boxplot.

### 5.3.2 *Evaluation of Unified Communication*

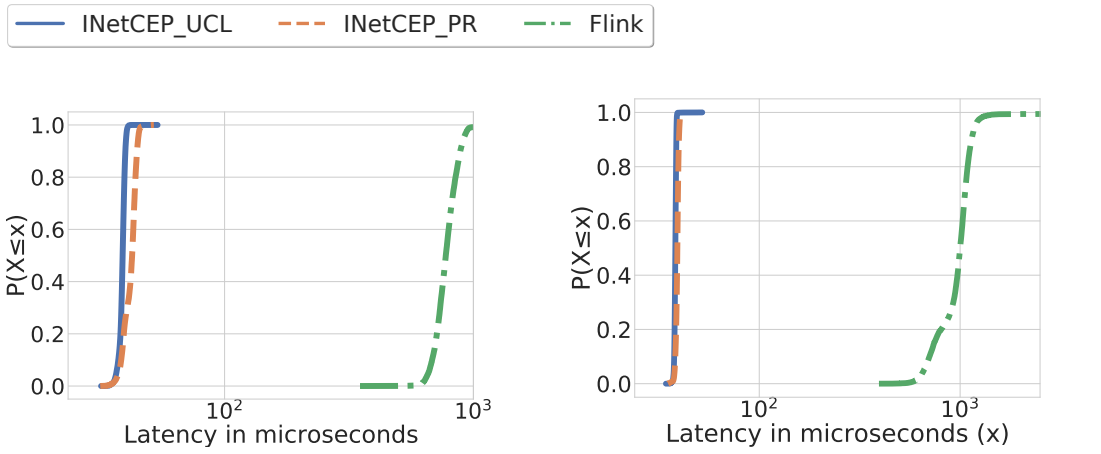
The following presents an evaluation of the unified communication mechanism proposed in Section 5.2.2. We measure the performance using the following metrics: end-to-end latency, throughput, and loss rate in forwarding the low-level events. For the evaluation setup, we used a Google Cloud c2 instance, where a producer, a consumer and a broker runs simultaneously. Figure 39a illustrates how the measurements for the above metrics is obtained. The producer sends a continuous data stream comprising event tuples using `Data Stream` packets given by the first dataset. The consumer sends a `Continuous Interest` using the `Add Continuous Interest` packet to the broker, which creates a persistent PIT entry for the consumer in its PIT table. Afterwards, it starts receiving the GPS sensor data stream. The evaluations are repeated 30 times to collect enough samples for the confidence interval (12,029 data points are collected). The evaluation runs for 20 minutes, and afterwards, the consumer deregisters the `Continuous Interest` using the `Remove Continuous Interest` packet. In the following, we analyze the results for each metric.

#### ***End-to-end Latency***

*Very low latency using unified communication of around* $38\mu s$

Figure 40 shows the cumulative distribution function (CDF) of the observed end-to-end latency in forwarding the data stream. As anticipated in our hypothesis, we observe very low latency when the data stream is forwarded using the unified communication mechanism. Particularly, we observe a mean delay of $38.77\mu s$, $38.2\mu s$ and $73.57\mu s$ for an event workload of 1000, 10000 and 50000 events per second, respectively (cf. Figure 40a, b and c left to right). In essense, we are $20$ (1000 ev/s), $25$ (10000 ev/s) and $25\times$ (50000 ev/s) better in terms of mean delay than the widely used CEP system, Apache Flink. This is because of several reasons: *(i)* Flink inherently relies on Apache Kafka as a data streaming system or event producer that uses a *pull-based* approach to capture and process data streams. *(ii)* Flink is designed to operate as a middleware while we offload the CEP functionality to the network and

leverage from the line speed of ICN substrate. Moreover, we compared the performance of the unified communication mechanism with the pull-based reference approach on top of the CCN-lite called periodic request mechanism (PR). Although the periodic request mechanism inspired by the work [142] performs equally well in terms of latency, it suffers from severe packet loss, as explained in the following paragraphs.




(a) $\bar{x}(Flink) = 792.19\mu s$, $\bar{x}(PR) = 41.82\mu s$, $\bar{x}(UCL) = 38.77\mu s$.

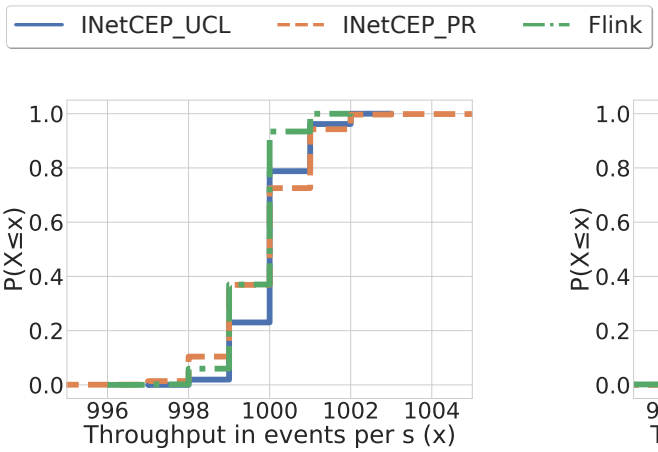
(b) $\bar{x}(Flink) = 982.16\mu s$, $\bar{x}(PR) = 39.06\mu s$, $\bar{x}(UCL) = 38.20\mu s$.

(c) $\bar{x}(Flink) = 1162.53\mu s$, $\bar{x}(PR) = 50.38\mu s$, $\bar{x}(UCL) = 73.57\mu s$.

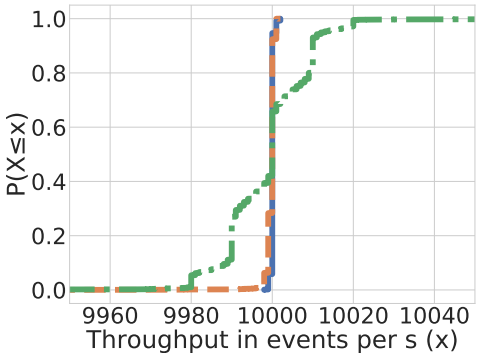
Figure 40: End-to-end latency of unified communication mechanism in INETCEP, Flink and periodic request mechanism using cumulative distribution function and different event rates. The results show that our approach is $20, 25$ and $15\times$ better than Flink for input rate of 1000, 10000 and 50000 events/s (from left to right), respectively. Moreover, the periodic request mechanism performs at par with our approach but on the cost of event loss [139].

### *Forwarding Throughput*

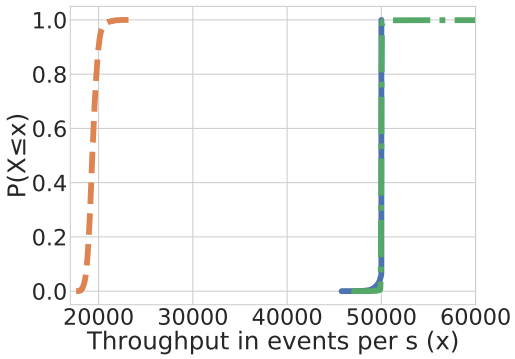
Figure 41 shows the CDF of the observed throughput while forwarding 1000, 10000; and 50000 events per second (left to right). Similar to the performance evaluations for latency, the setup used is illustrated in Figure 39a.




(a) $\bar{x}(Flink) = 999.86ev/s$, $\bar{x}(PR) = 999.63ev/s$, $\bar{x}(UCL) = 1000.0008ev/s$




(b) $\bar{x}(Flink) = 9997.99ev/s$, $\bar{x}(PR) = 9999.65ev/s$, $\bar{x}(UCL) = 10000.0008ev/s$




(c) $\bar{x}(Flink) = 50035.28ev/s$, $\bar{x}(PR) = 19372.90ev/s$, $\bar{x}(UCL) = 49972.34ev/s$

Figure 41: Throughput of unified communication mechanism in INETCEP, Flink and periodic request mechanism using cumulative distribution function and different event rates. The results show that our approach attains an optimal throughput (maximum possible) without incurring any event loss for input rate of 1000, 10000 and 50000 events/s (from left to right). For high event rates of 50000 events, our approach is superseding the periodic request mechanism by maintaining integrity in the reception of complex events [139].

The first plot (a) for a smaller event rate of 1000 events per second shows that all three approaches can perform equally well. Also, a step-wise increment of the throughput values is observed because of the cardinality of

the number, meaning either 999 or 1000 events are received and not 999.5 events. The second plot (b) shows the performance of the approach while receiving 10000 events per second. In this plot, we observe that the unified communication mechanism (UCL) and periodic request mechanism (PR) are near the optimal throughput of 10000 events per second, while for Flink, we observe that it diverges from the optimal throughput in 80% of the values. The reason is that the pull-based batching mechanism of Flink and Kafka that forwards events to the consumer in batches instead of a push-based continuous data stream. Due to this, we observe that events are missing in the beginning (values less than 10000 events), which are then received in the middle and towards the end of the experiment (values more than 10000 events). The third plot (c) shows the performance when the input rate is 50000 events per second.

*High throughput of up to 50000 events/s*

An important observation is that the periodic request mechanism (PR) suffers from heavy event loss; the maximum throughput for this approach is around 20,000 events per second. This is because the network becomes congested due to the number of packets required to retrieve 50000 events per second. The periodic request mechanism sends 50000 `Interest` packets to retrieve 50000 `Data` packets, and hence the total number of packets sent is twice the number of events that are to be received by the consumer, which results in network congestion. We observe packet loss when the UDP buffer is full for the periodic request mechanism. This is not seen in Flink because it is TCP-based, and it incorporates a credit-based flow control mechanism to prevent buffer overflow. Most importantly, our unified communication mechanism does not suffer from packet loss even though it is UDP-based, thanks to the lossless rate-based flow control mechanism.

### *Event Loss during Forwarding*

To measure the event/packet loss during the transmission, we use the loss rate metric. Table 17 presents the observed loss rate (mean, min, max and percentiles (90, 95, 99) values) for unified communication, periodic request mechanism and Flink. For smaller event rates, we observe only a small amount of event loss in the periodic request aproach due to a lower amount of packet exchange; for a higher event rate of 50000 events per second, a mean loss rate of $61.18\%$ is seen. This correlates to the throughput evaluations seen above, where only around $40\%$ of events were received at the consumer due to the network congestion and the UDP buffer overflow.

*No event loss because of flow control*

An even interesting observation is that Flink, with Kafka as an event producer, also suffers from event loss. Although Kafka is assumed to be highly resilient and performant, it depends on multiple configuration parameters

to avoid event loss that is conflicting at times and hardly optimizable[42]. Using the default configuration parameters, we saw an event loss using Apache Flink as a CEP query engine, which can be problematic for many applications that need 100% accuracy.

In contrast, the unified communication mechanism *does not* suffer from any event loss thanks to the proposed flow control mechanism.

### 5.3.3 *Evaluation of* INETCEP *Query Engine*

*Query engine evaluation with known WSN topologies*

This section provides a performance analysis of the INETCEP query engine as proposed in Section 5.2.3. We provided a two-fold analysis, centralized (cf. Figure 39a) and distributed (cf. Figure 39b) with different topologies and queries specified in Table 21. In centralized evaluation, the query engine runs on a single node; we show the benefit of operator placement in the distributed setup. We use end-to-end latency as a metric to analyze the performance because, for most IoT applications, low latency is very important, especially for safety-critical applications. To analyze if optimal throughput is reached, we refer to an accuracy metric that implicitly includes event loss depicted by low accuracy since the output events delivered is incorrect.

#### ***Centralized Processing***

We perform this evaluation on a Google Cloud c2 instance, where a consumer issues query 2–8 by encapsulating them into the `Add Continuous Interest` packet. For comparison, we implemented the same queries on Apache Flink and measured the performance using the Google Cloud c2 instance. In the periodic request mechanism, we use the same implementation as INETCEP except for the underlying communication strategy, which is pull-based and the placement of the queries. Particularly, an `Interest` packet is sent continuously to retrieve the data item, and hence the operator placement is performed each time as if a new query is placed. In the experiment, we set the rate at which the `Interest` packets are sent to be equal to the input rate of events at the producer. However, intuitively this rate would differ for complex events where the output rate of events is not equal to the input event because of the event transformation (filter, join, etc.), which accounts for another problem with the periodic request mechanism. In the following, we analyze the latency metric for centralized processing.

[42]Data loss in Flink with Kafka. `https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/kafka.html` [Accessed in May 2021]

| | 1000 events per second | | | | 10000 events per second | | | | 50000 events per second | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **System** | *mean* | *max* | *min* | *percentiles (90, 95, 99)* | *mean* | *max* | *min* | *percentiles (90, 95, 99)* | *mean* | *max* | *min* | *percentiles (90, 95, 99)* |
| **Flink** | 0.013 | 0.014 | 0.013 | 0.013, 0.014, 0.014 | 0.135 | 0.271 | 0.069 | 0.211, 0.241, 0.265 | 0.0135 | 0.020 | 0.003 | 0.016, 0.018, 0.019 |
| **INetCEP_PR** | 0.038 | 0.044 | 0.036 | 0.04, 0.042, 0.043 | 0.003 | 0.004 | 0.001 | 0.004, 0.004, 0.004 | 61.185 | 61.226 | 61.153 | 61.216, 61.221, 61.225 |

Table 17: Event loss rate (in %) of Flink and periodic request mechanism in INETCEP for event rates 1000, 10000 and 50000 events per second. Both the baselines suffer from event loss. However, the unified communication mechanism in INETCEP does not suffer from any loss, thanks to the flow-control mechanism [139].

| | window | | filter | | join | | heatmap | | predict | |
|---|---|---|---|---|---|---|---|---|---|---|
| **System** | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* |
| **Flink** | 2.97 | 0.15, 0.16, 0.18 | 0.329 | 0.155, 0.17, 0.191 | 10.627 | 20.64, 41.52, 41.53 | 2.098 | 4.09, 4.128, 4.167 | 12.44 | 0.34, 0.36, 0.39 |
| **INetCEP_PR** | 1.51 | 1.9, 1.96, 2.01 | 1.41 | 1.85, 1.92, 1.97 | 5.54 | 5.94, 5.99, 6.03 | 1.46 | 1.75, 1.85, 1.91 | 1.49 | 1.88, 1.95, 1.99 |
| **INetCEP_UCL** | **0.007** | **0.009, 0.010, 0.014** | **0.010** | **0.014, 0.017, 0.020** | **0.011** | **0.017, 0.02, 0.025** | **0.010** | **0.013, 0.015, 0.019** | **0.014** | **0.019, 0.02, 0.024** |

Table 18: Mean and percentiles of end-to-end latency (in seconds) observed in Figure 42 for Flink, unified communication mechanism and query engine in INETCEP, and periodic request mechanism using the standard and application queries. Our approach outperforms both significantly, in particular, $888\times$ (best case: predict query) and $32\times$ (average case: filter query) in comparison to Flink; and $503\times$ (best case: join query) and $106\times$ (average case: predict query) in comparison to periodic request mechanism [139].

| | window | | filter | | join | | heatmap | | predict | |
|---|---|---|---|---|---|---|---|---|---|---|
| **System** | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* |
| **INetCEP_PR_M** | 1.634 | 1.951, 1.98, 6.31 | 1.883 | 2.559, 2.702, 5.354 | 7.424 | 9.646, 9.652, 10.678 | 2.061 | 2.985, 2.991, 6.926 | 2.503 | 3.702, 5.38, 6.939 |
| **INetCEP_UCL_M** | **0.006** | **0.008, 0.009, 0.012** | **0.011** | **0.014, 0.015, 0.021** | **0.011** | **0.015, 0.017, 0.023** | **0.011** | **0.014, 0.016, 0.020** | **0.018** | **0.027, 0.029, 0.035** |
| **INetCEP_PR_T** | 1.59 | 1.764, 1.789, 1.886 | 1.542 | 1.664, 1.698, 1.797 | 6.955 | 9.527, 9.56, 9.594 | 1.599 | 1.768, 1.82, 1.943 | 1.556 | 1.704, 1.725, 1.784 |
| **INetCEP_UCL_T** | **0.007** | **0.009, 0.010, 0.012** | **0.012** | **0.015, 0.017, 0.021** | **0.013** | **0.019, 0.022, 0.030** | **0.012** | **0.015, 0.016, 0.021** | **0.016** | **0.021, 0.023, 0.027** |

Table 19: Mean and percentiles of end-to-end latency (in seconds) observed in Figure 43 for unified communication mechanism and query engine in INETCEP, and periodic request mechanism using the standard and application queries for the different topologies. Here, INetCEP_PR_M refers to the periodic request mechanism with the manhattan graph, and INetCEP_PR_T refers to the results on tree topology. Similar to centralized evaluations, our approach supersedes again substantially by a factor of $674\times$ (best case for join query in manhattan graph) and $97\times$ (average case for predict query in tree topology) [139].

*Impact on Latency*

Figure 42 shows the CDF and Table 18 summarizes the mean and percentiles (90, 90, 99) of the observed end-to-end latency in the three approaches for the given application and standard CEP queries. The performance of the query engine is as good as the forwarding results using the unified communication mechanism. In particular, it is around $32\times$ better in latency (mean value) than query processing using Apache Flink (cf. Table 18). For instance, a filter query is processed in a mean latency of $10.72$ ms using the unified communication mechanism in the INETCEP query engine, while in Flink, it takes around $329.47$ms for the same query. We see a substantial gain for application-related queries (heat map and predict) involving complex operations, typically required by IoT applications. Our approach performs around $966\times$ and $888\times$ better than the Flink baseline for the join and predict query, respectively (cf. Table 18). Although Flink is known to have a pretty good performance in terms of latency that is the best amongst the available state-of-the-art CEP systems, it lags behind our approach due to multiple reasons. One of the major causes is the serialization and deserialization overhead of data stream objects in Flink and the standard pull-based implementation of the Kafka event producer in Flink. However, using a batching mechanism, it supersedes the performance of periodic request mechanism for simple queries like filter. But most of the times periodic request mechanism supersedes the performance of Apache Flink due to the network-centric execution similar to our approach.

*Very low latency in query processing of around* $10ms$

Most importantly, the unified communication mechanism with the query engine realization outperforms the periodic request mechanism substantially, specifically $503\times$ for the join query and $106\times$ for the predict query (cf. Table 18). This is because for more complex operations like join, heatmap and predict queries, the periodic request mechanism needs a significant amount of packets to retrieve the complex event. It is clear from the above evaluations that Flink suffers highly because of the pull-based implementation of Kafka event producer. Therefore, we compare with a single pull-based reference implementation based on ICN, i.e., periodic request mechanism for the remaining evaluations, since it is more comparable to our approach due to the similar network stack.

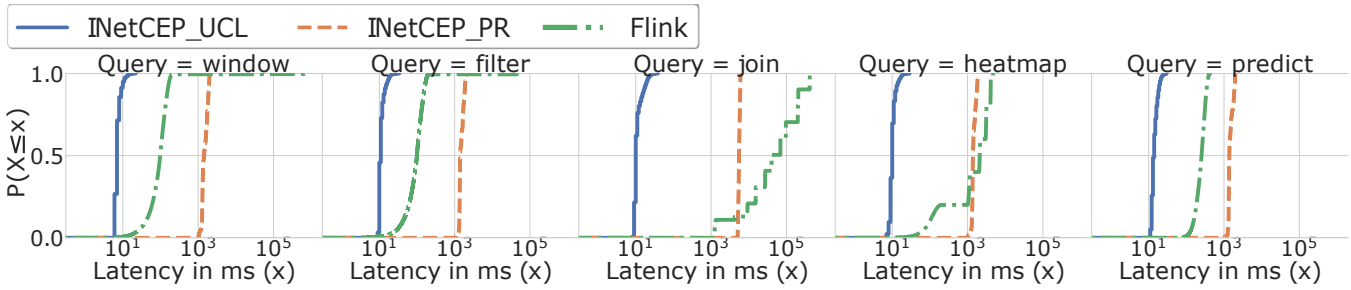



Figure 42: Centralized evaluation of end-to-end latency for unified communication mechanism and query engine in INETCEP, Flink, and periodic request mechanism using cumulative distribution function and different queries. The results show that our approach outperforms both Flink and periodic request mechanism in all the queries (lowest mean time for the window query $\bar{x}(UCL) = 7.3ms$ and highest for the predict query of $\bar{x}(UCL) = 14.48ms$) [139].

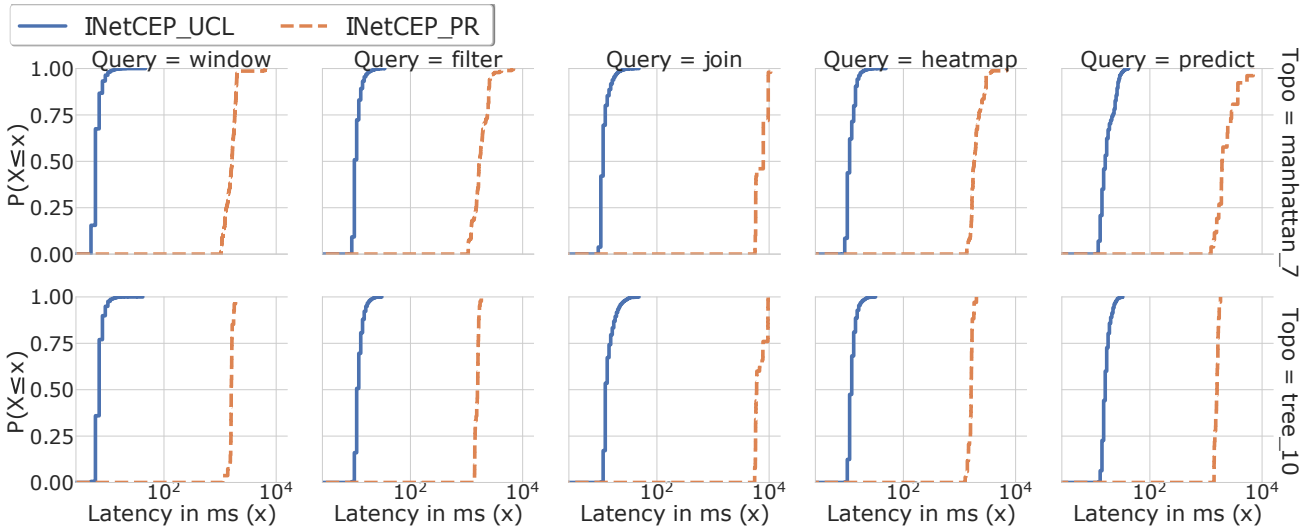



Figure 43: Distributed node evaluation of end-to-end latency for unified communication mechanism and query engine in INETCEP compared to periodic request mechanism using cumulative distribution function, different queries, and topologies. Our approach outperforms against periodic request mechanism again consistently to centralized evaluations in all the queries and the manhattan graph (top) and tree topology (bottom) with 7 and 10 nodes. Lowest $\bar{x}(UCL) = 6.46ms$ for window query and highest $\bar{x}(UCL) = 18.08ms$ for predict query [139].

### *Distributed Processing*

We use different topologies for the distributed node evaluations as given in Figure 39b: line, manhattan graph, and tree topology. We use the application and CEP standard queries for the evaluation, similar to the centralized processing. In these evaluations, we aim to understand the performance of operator placement, deployment and query execution using multiple nodes for processing. For deployment, we consider a cloud server, edge switches and end nodes using the AWS instances as given in Table 21. Each node executes the unified communication mechanism, CEP query engine, operator placement and the nodes communicate using the ICN protocol. In the following, we present the results per topology.

#### *Manhattan Graph*

For the distributed evaluation, we used the manhattan graph, which is a widely used topology in wireless sensor networks [235] (cf. Figure 39b). Here node $n6$ acts as an event producer, $n5$ as an event consumer and all the other nodes $n1, \ldots, n4, n7$ acts as event brokers. Figure 43 shows the CDF and Table 19 presents the mean and percentiles (90, 90, 99) of the observed end-to-end latency measurements. We compare our approach: INETCEP using a unified communication mechanism and query engine including operator placement to the periodic request mechanism and query engine including operator placement for each new request. In Figure 43, the first row shows the results of the Manhattan graph, where the lower the value (towards left), the better it is. Our approach (INETCEP_UCL_M) outperforms the periodic request mechanism substantially by a factor of $674\times$ for the join query (best case) and around $139\times$ for the predict query (average case). The improvement is even better than centralized processing because the periodic mechanism suffers from large delays in disseminating the request each time to the brokers and have an additional placement overhead. Our approach does not have such an overhead, thanks to the push-based approach in unified communication mechanism as well as the reactive and concurrent query deployment algorithm that places a query and triggers query processing each time it encounters a new `Data Stream` packet. Table 19 lists the mean and percentile values of latency for the two approaches using the manhattan and tree topology. In summary, we encounter the lowest mean delay of $\bar{x}(UCL) = 6.46ms$ for the window query, while the highest $\bar{x}(UCL) = 18.08ms$ for the predict query, which is still $139\times$ better than the periodic request mechanism.

*Consistent with other scenarios latency of only around 6 ms*

*Tree Topology*

We performed a similar distributed query evaluation for the tree and line topologies (cf. Figure 39b (top and bottom on the right)) using the AWS instances. Here, a lower number of nodes are required and we use CORE and MACI for a higher number of nodes. Since the results for the line and tree topologies are identical, therefore, we report the results for tree topology in Figure 43 (second plot). In consistent to the evaluations using the manhattan graph, we observe that our approach substantially supersedes the periodic request mechanism by a factor of $535\times$ in the join query and about $97\times$ in the predict query (cf. Figure 43, refer INETCEP_UCL_T and INETCEP_PR_T). The minimum and maximum mean dealy are also again for the window query $\bar{x}(UCL) = 7.07ms$ and predict query $\bar{x}(UCL) = 16.65ms$, respectively.

| | **heatmap** | | **predict** | |
|---|---|---|---|---|
| **System** | *mean* | *percentiles (90,95,99)* | *mean* | *percentiles (90,95,99)* |
| **Centralized, INetCEP_PR** | 54.36 | 69.80, 70.90, 71.78 | 48.70 | 46.62, 51.88, 56.09 |
| **Distributed, INetCEP_PR** | 66.66 | 66.66, 66.66, 66.66 | 66.66 | 66.66, 66.66, 66.66 |

Table 20: Accuracy evaluation for the application queries: heatmap and predict queries in the periodic request mechanism. The approach suffers from severe event loss and a significant decrease in the accuracy of up to 66%, while INETCEP consistently delivers 100% accuracy in detecting the complex events [139].

### *Impact on Accuracy*

*100% accuracy because of no event loss*

This section evaluates the event loss and correctness of the delivered complex events using the queries. As indicated in the scenarios, many applications need to be highly robust and cannot tolerate any event loss; for example, a missed fraud detection event might lead to heavy monetary loss. Similarly, in a prediction query, a miss prediction of high load might lead to smart grid breakdown. A known cause of false negatives or false positives in event detection is due to event loss. We represent this using accuracy previously defined in Chapter 3: Definition 12. We use the F1-score metric that precisely measures the accuracy given by $\frac{TP}{TP+\frac{1}{2}(FP+FN)}$. Here, $TP$, $FP$ and $FN$ are true positive, false positive and negative, respectively. Table 20 presents the F1-scores for the centralized and distributed query processing using the periodic request mechanism for heatmap and predict queries, where event loss can lead to hard implications. We observe a significant drop in F1-scores for this approach as a consequence of event losses due to network congestion, around $45\%$ and $33\%$ in centralized and distributed processing, respectively. Similar observations were seen in Apache Flink due to event loss. Contrarily,

the unified communication mechanism in the INETCEP query engine does not suffer from any event loss and hence delivers 100% F1-score, because of the responsive and parallel query processing along with the lossless flow control mechanism.

## 5.4 Summary

This chapter proposes the second contribution of this thesis solving the efficiency problem in the face of dynamic environmental conditions. Efficiency in Internet of Things applications, for instance, in terms of latency, is of extreme importance, which if not delivered would cause heavy monetary loss (fraud detection application) can be even life threatening (accident detection application for autonomous cars). In order to react in such situations, a CEP system has to perform event processing in a very efficient manner, for example, using in-network resources that can deliver the results very quickly. We tackle this challenge using Information-centric Networking paradigm and propose to perform event processing over ICN resources. To this end, we contribute *(i)* a unified communication model for ICN architectures to enable event processing in the programmable data plane of ICN substrate, *(ii)* a meta query language to express CEP queries that can be resolved on the ICN substrate, *(iii)* query execution algorithms that reactively handle events and process them in a parallel manner to derive the complex events in an efficient and robust manner while ensuring the correctness, and *(iv)* a networking architecture called INETCEP that integrates the above models, abstractions and algorithms into an in-network processing architecture for their evaluation. Our evaluation using INETCEP in the context of two IoT case studies and standard CEP queries shows that *(i)* event forwarding using the unified communication model is about 15× faster than than the state-of-the-art CEP system Flink, *(ii)* IoT queries can be easily expressed using the proposed meta query language, and *(iii)* using the proposed query execution algorithms, the complex events are delivered about 32× faster than *Flink* and about 100× faster than a pull-based reference approach.

# 6

# Unified Serverless CEP Middleware

This chapter provides a solution to the *interoperability problem* in the context of Complex Event Processing, aiming to enhance the reusability of mechanisms across multiple CEP execution environments. Current CEP systems both in academia [21, 22, 23, 24, 25, 26, 27] and industry [35, 36, 52, 28, 9] are restricted in terms of reusability (cf. Chapter 2: Section 2.3.3). The main reason for this restriction is that usually in CEP systems, the specification language used to express the complex events, and the execution environment used to detect events, are tightly coupled. Although, this tight coupling have been regarded as a key to high performance in CEP, this dependency is problematic because of the following arguments. *(i)* The foremost problem is that the application developers need to have an extensive knowledge on the execution environment, which is highly complex by nature. *(ii)* Another problem is that due to the tight dependency there is little support to update operators on the fly that is often required by applications. For instance, a fraud detection application (right side of Figure 44) requires to dynamically update or even change machine learning models to deal with the newly observed fraud patterns. Due to the monetary cost associated with the fraud detection application, it becomes critical that the system is updated in a very short time (in few milliseconds) and ideally without any downtime [13].

*Key research gap: lack of reusability*

To overcome these open problems, this chapter proposes a programming interface and mechanisms that unify heterogeneous CEP execution environments while allowing dynamic changes in the operator specification. We propose CEP middleware, which is a serverless, named CEPless. It enables developers to write applications without depending on or knowing the underlying execution environment, and therefore, benefit from multiple CEP systems simultaneously. CEPless supports dynamic changes in the operator specification, allowing to switch functional components of CEP operators at runtime that often becomes necessary, as noted in Chapter 3 for applications like fraud detection. However, such unification of multiple CEP systems and dynamic changes in the operator specification is not trivial. The reasons are manifold, and let us explain them using the overall architecture seen in Figure 44 with CEPless-centric components. On the top-most tier, we see the numerous CEP programming models, including those proposed in this thesis, Tcep (in Chapter 4) and INetCEP (in Chapter 5), and oth-

*Solution: serverless CEP middleware*

*Challenge 1: tight coupling*

ers from academia [54] and industry [35]. Each of these CEP systems offers very distinct mechanisms and concepts for specific infrastructures in mind. For instance, INETCEP is designed to provide network-centric execution in the ICN routers, while Apache Flink [54] is designed specifically for cloud infrastructures. Comprehending each of these systems is a tedious process and without a detailed understanding on how each of these work, it is not only difficult but a never-ending process to unify them. Moreover, there is no means to combine or to reuse mechanisms because of the tight coupling of the language and execution environments as mentioned before.

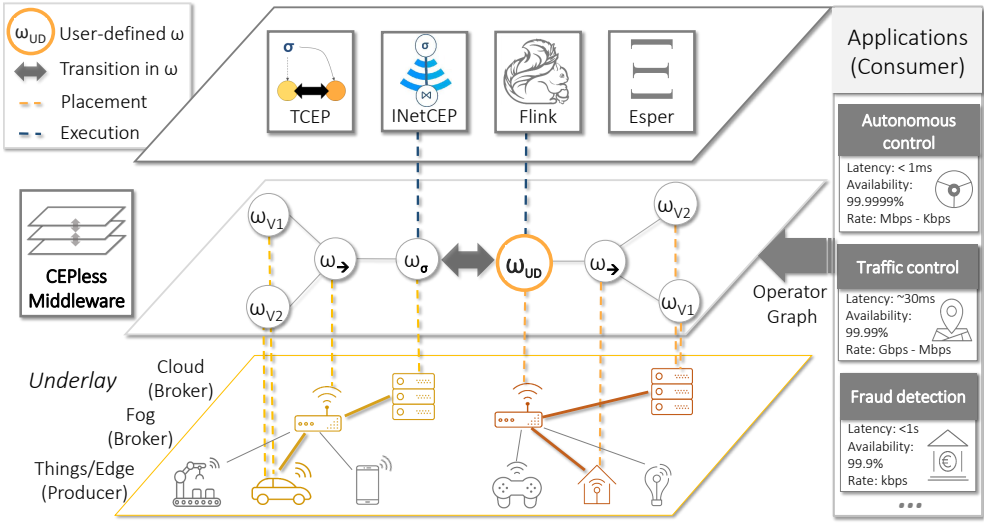



Figure 44: Overall system architecture focusing on CEPLESS that enables interoperability in the CEP execution environment and the operator deployment.

*Challenge 2: State management*

A novel paradigm that is currently used to provide decoupling in huge cloud computing environments such as Amazon [164] and Google [165] is using *Serverless Computing* (cf. Chapter 2: Section 2.3.2). Basically, it provides a programming abstraction to the developers so that the system-specific components are hidden and the developers can focus on the application-specific components. Although existing serverless platforms provide important concepts that can accelerate scalable execution of operators, these platforms are missing crucial semantics needed for CEP such as state management and operator placement as discussed in Chapter 2: Section 2.3.3. Moreover, current CEP systems provides many mechanisms such as query optimization, placement and execution, which are not easily extendable to the existing serverless platforms.

These findings lead to our last research question, which is solved in this chapter:

*RQ3: How to achieve interoperability across different CEP programming models?*

*Final Research Question and its sub RQs*

*RQ3.1* How to express operators without dependency on the underlying CEP system's programming model and execution environment?

*RQ3.2* How to enable reusability of distinct CEP execution environments while ensuring dynamic operator updates?

To enable reuse and interoperability across distinct CEP programming models, we solve two crucial challenges corresponding to the above sub research questions. *(RQ3.1)* Because of the dependency between the specification language and execution environment as detailed above, simply rewriting the query specification from one to another CEP programming model does not work, especially because the query execution semantics and the language is known to diverge a lot. Therefore, we answer this sub research question by providing a common programming interface inspired by the concept of *serverless computing* in cloud infrastructures [181, 174]. Similar to serverless computing, we propose so-called user-defined operators $\omega_{UD}$ (seen in the figure on the middleware tier) that can be implemented using our programming abstraction. Using $\omega_{UD}$ operators, we combine the benefit of multiple CEP execution environments by interacting with them using the middleware. Most importantly, the programming interface allows the CEP application developers to specify operators using the language of their preference without imposing any side-effects on the system-dependent operators (inbuilt CEP operators).

*(RQ3.2)* Another limitation of the above dependency is that there is limited support in updating the operators at runtime. This becomes highly essential for applications such as fraud detection, where the business logic has to be updated at runtime while ensuring system availability throughout the deployment. Furthermore, the statefulness of some CEP operators raises additional challenges for the CEPless middleware towards state management. Therefore, we answer this sub research question by proposing mechanisms that allow dynamic operator updates: in-memory queue management and batching mechanisms. These mechanisms handle the state and continuously delivers output to the consumers, respectively. Thus, current CEP systems can quickly adapt their event detection logic based on the application context and the underlying infrastructure.

The findings presented in this chapter are based on the author's previous publications in [238], [239]. The structure of this chapter is explained as follows. Section 6.1 presents the extended system model and the problem

*Key publications and structure*

statement. Section 6.2 presents the overall design of the novel CEP serverless middleware CEPLESS that introduces the individual contributions: *(i)* the programming interfaces and the *(ii)* in-memory queue management and batching mechanisms. Finally, Section 6.3 presents the extensive evaluation and implementation of CEPLESS on two CEP systems Apache Flink [54] and TCEP [196] (cf. Chapter 4). CEPLESS with the programming interfaces and the extensions to the CEP systems are publicly available for use here[43].

*Open source, try it out!*

## 6.1 Analysis of Flexibility in Operator Specification

This section extends the common system model presented in Chapter 3 to define the flexibility problem (cf. in Section 6.1.1). Moreover, we describe the flexibility problem using the fraud detection scenario (cf. in Section 6.1.2).

### 6.1.1 *Extended System Model*

Figure 44 illustrates the common system model used in this chapter with a focus on the presented CEPLESS middleware components as described below.
In CEPLESS, operator graphs can be deployed on the infrastructure comprising fog-cloud and things producing continuous data streams to be processed by the higher tiers. A query $Q = \{q, QoS, T_{QoS}\}$, including the QoS requirements such as latency, etc., can be specified by the distinct CEP execution environment from the execution tier, Apache Flink [54], Esper [35], and novel CEP systems presented in this thesis, including TCEP (cf. Chapter 4) and INETCEP (cf. Chapter 5). The CEP systems can specify so-called user-defined operators that are designed to be exchanged and reused across the different execution environments. These operators are processed by the CEPLESS middleware (seen in the middle tier), instead of the execution environments directly but they can interact with the other standard CEP operators in the operator graph (so-called system defined operators) using event queues. The user-defined operators can be developed using the CEPLESS programming interface that allows for language and platform independence to the application developers. These are maintained in a repository which is meant to be reused across different CEP platforms. The CEPLESS middleware presented in this chapter that *(i)* benefits from the distinct execution environments of the top tier, *(ii)* allows placement on the heterogeneous infrastructure of the bottom tier, and *(iii)* can replace operators at runtime. In the following, we elaborate

[43] CEPLESS webpage `https://luthramanisha.github.io/CEPless/` [Accessed in May 2021].

on the two extensions to the common system model: the operator model and the queue model, which help us model the aforementioned contributions.

#### *Operator Model*

As seen represented in the CEPLESS middleware in Figure 44, the operator graph mainly comprises two kinds of operators: $\Omega = \{\Omega_S, \Omega_{UD}\}$. Here $\omega_S \in \Omega_S$ are referred to as system-defined operators and $\omega_{UD} \in \Omega_{UD}$ are referred to as user-defined operators. The definitions are given as follows.

**Definition 16.** *System-defined operators* ($\Omega_S$) are standard operators already defined in a CEP system, such as single-item operators like *selection*, logical operators like *conjunction*, window operators like *sliding window*, and flow management operators like *join* [6].

**Definition 17.** *User-defined operators* ($\Omega_{UD}$) can contain custom business logic, which typically could not be expressed by system-defined operators, such as machine learning models.

The functionality of $\omega_{UD}$ operators is defined in such a way that they can be dynamically changed. Each node hosting the user-defined operator $\omega_{UD}$ is managed by a node manager $NM_n$ (recall, $n$ stands for node), which handles all the requests related to the $\omega_{UD}$. Operator containers $C_\omega$ are used as defined in the common system model in Definition 5 to provide an execution environment for all operators. Especially, for user-defined operators $\omega_{UD}$, the operator container serves as an interface to deliver the incoming events processed by the given processing function $f_\omega$. The user-defined operators $\omega_{UD}$ interact with the underlying CEP systems using the above interface by sending one or multiple events for further processing, e.g., using system-defined operators. We provide in-memory queues that handle the transfer of events to and from the $\omega_{UD}$ and act as a store for the stateful operators. In the following, we detail the queue model.

#### *Queue Model*

We use a queuing system as a messaging interface between the CEPLESS middleware and the execution tier. With the queuing model, we aim for three main goals *(i)* statefulness, *(ii)* in-order processing, and *(iii)* high performance in event processing. The queue is not restricted in terms of size[44] and it can

[44]The model can support different sizes based on the implementation of the in-memory queue, e.g., using off-the-shelf implementations like Redis [240].

hold as many events as to be processed. In the following, we define the related concepts of the queuing model, *(i)* the event queues that stores the incoming and outgoing events, *(ii)* the event batch that determines the subset of the emitted events, *(iii)* the back-off interval that waits for remaining events to fill the batch, and *(iv)* the interface responsible for interaction between the CEP system and the queues.

**Definition 18.** *Event queue* ($eq_{\omega_{UD}}$) acts as a store for event transfer between the CEPLESS middleware and the execution tier for better performance.

Each node contains two event queues: *Input* ($eq\_in_{\omega_{UD}}$) and *Output* ($eq\_out_{\omega_{UD}}$). The former is for the incoming events from the CEP system to the user-defined operator $\omega_{UD}$ and the latter is for the outgoing events from the $\omega_{UD}$ to the CEP system. The user-defined operators $\omega_{UD}$ and the underlying CEP systems can communicate only using these queues placed on their respective node. Since we assume that the queue resides on the same node, there is no networking overhead to ship the events to the queue (at a remote location) and send them back. Therefore, the communication between the queue and the CEP system is local.

**Definition 19.** *Event batches* ($b_s$ and $b_r$) represent a subset of event tuples sent to ($b_s$) and from ($b_r$) the CEPLESS middleware via the event queues (defined in Definition 18).

Instead of sending the events one by one, we use event batches to communicate with the underlying CEP systems. This avoids the unnecessary overhead of opening and closing connections for requests each time a new event is received.

**Definition 20.** *Back-off interval* ($t_{backoff}$) is the time to backoff from polling the event queue $eq_{\omega_{UD}}$ when no events are received.

**Definition 21.** *User-defined operator (UDO) interface*. The event queues interact with the execution tier using the UDO interface. Each CEP system implements this interface, a minimalistic realization of opening and closing connections to and from event queues $eq_{\omega_{UD}}$.

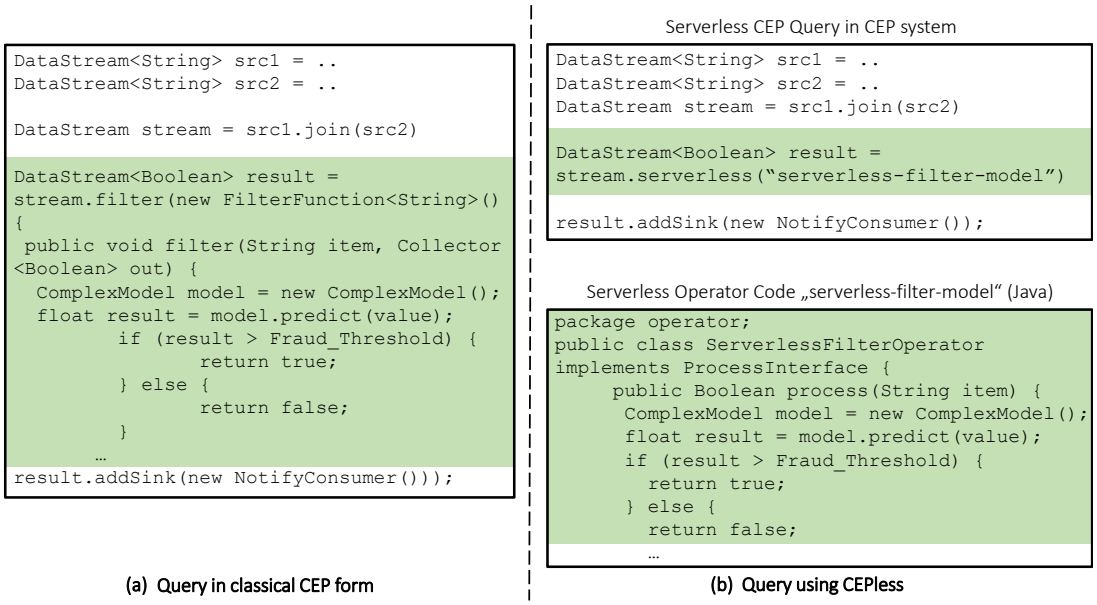



Figure 45: A simple fraud detection logic using Filter operator, where `highlighted code` refers to the operator business logic. We represent Filter operator, in the left figure (a) using the specification language of a CEP system, Flink and (b) using Flink (top) and CEPless programming interface for Java (bottom). In (b) we have provided a serverless abstraction for `serverless-filter-model` to decouple operator logic from the CEP runtime APIs like `DataStream`.

### 6.1.2 *Problem Statement*

This section analyzes the ability of modern CEP systems to flexibly specify operators and signify their importance using the fraud detection scenario previously introduced in Chapter 3: Section 3.1 ④.

Typically, a fraud detection algorithm is expected to detect complex fraud patterns encapsulated in business logic comprising machine learning models. Consider a financial institution that uses an exclusive machine learning library that is implemented in the fraud department of the institute. The fraud department wants to use a CEP system to model the fraud and detect it in real-time and at a very large scale of events. Moreover, since frauds are detected in real-time and the fraud patterns vary over time, the department wants the system to specify complex business logic and exchange it at runtime based on the observed patterns.

It is important to note that existing CEP systems fail to fulfill these requirements, which we exmplarily show in the following using a simple specification of fraud detection in a highly performant CEP system Apache Flink [54] (cf. Figure 45 (a)).

In the figure, the `highlighted` code represents the complex business logic or a pre-trained machine learning model used to detect fraud. In this example, we hide the complexity of the prediction logic behind `model.predict` for better understandability. It can be seen that the specification of filter operator in Figure 45 (a) is highly dependent on Flink's APIs such as `DataStream` and `FilterFunction`. Moreover, the `filter` operator is ossified and dynamically updating it without redeployment is not possible. Also, the language of preference of the financial institution, in this example, Rust, cannot be used due to the missing APIs. This dependency on the CEP execution environment and the programming language is problematic for multiple reasons. *(i)* The fraud detection algorithm might benefit from the libraries defined in more than one CEP system, which can be achieved by integrating two or more CEP systems that are not currently possible. For instance, integrating the machine learning models by extending the runtime and external modules of the CEP system can be very cumbersome, if not impossible. *(ii)* Existing CEP systems are highly complex (e.g., Flink in total comprises more than 900,000 LOC), and the respective department responsible for developing the fraud detection system might not have the expertise to deal with the complexity. *(iii)* Finally, the financial institution might have a programming language of preference for developing the fraud detection algorithm, e.g., Rust in the above example. This is not easily possible using existing CEP systems without completely extending the APIs of the CEP system or a complete rewrite of all the operators in the CEP system

*Problems: language and runtime dependence*

In Figure 45(b), on the right side, we show our proposal of the programming model that mitigates all the limitations stated above. As shown in the figure, we segregate the business logic of the operator `serverless-filter-model` using serverless abstractions, making them independent of the CEP runtime. This means `ServerlessFilterOperator` can be executed independent of the underlying CEP execution environment and can be written in any programming language of preference.

## 6.2 The CEPless System Design

CEPless proposes the following mechanisms and programming interfaces targetting key research questions discussed previously: *(i) (RQ3.1)* Programming interfaces to express user-defined operators $\omega_{UD}$ that interacts with the middleware and the CEP systems to execute them. Here, User-defined Operator Interface acts as a communication channel between the given CEP system and the $\omega_{UD}$, and the User-defined Programming Interface facilitates an easy submission of $\omega_{UD}$ to the CEPless middleware. *(ii) (RQ3.2)* Mechanisms for in-memory queue management and batching enable stateful processing of events, in-order, and efficiency in delivering complex events. Moreover, the

mechanisms aid in dynamic operator updates while consistently delivering complex events.

In the following, we introduce the CEPLESS system design conceptually before presenting the main contributions of this work: the CEP serverLESS middleware in Section 6.2.1 and the programming interfaces in Section 6.2.1.

*Conceptual Overview.*

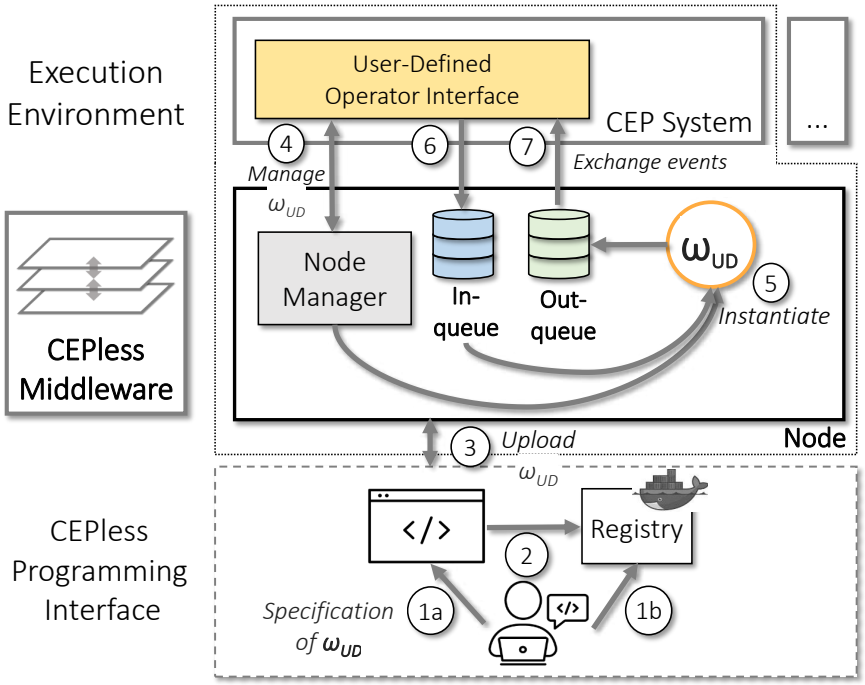



Figure 46: The conceptual view of the CEPLESS system. *Here, the dotted line shows the components encapsulated in each node.*

*Query workflow with UD operators*

Figure 46 presents the three key components of the CEPLESS System. *(i)* The *Execution layer* comprises distinct CEP systems providing the execution environment for the queries. *(ii)* The CEPLESS *middleware* allows reusability of different execution environments from the Execution layer while ensuring dynamic operator updates. *(iii)* The *Programming interfaces* enable the specification of user-defined operators in the preferred programming language and integrate distinct CEP execution environments to the CEPLESS middleware. In the following, we present the workflow of specifying, instantiating, deploying, and processing user-defined operators $\omega_{UD}$ using the execution environment. First, as seen in the CEPLESS programming interface layer, the CEP application developers can specify user-defined operators $\omega_{UD}$ either using the (1a) interface or using the (1b) centralized operator registry. The programming interface provides a means to define and deploy user-defined

operators $\omega_{UD}$ independent from language and execution environment. The centralized registry maintains the executable operator containers $C_\omega$, such that multiple applications can reuse them. ② After submitting the code for $\omega_{UD}$ using the programming interface, the system wraps the code into a container and pushes it to the common registry. ③ Once the code is pushed into the registry; any node can use the $\omega_{UD}$ with a deployed CEPLESS middleware instance. The node in question can simply download the $\omega_{UD}$ and interact with other CEP systems in the Execution layer. ④ The deployment of $\omega_{UD}$ is triggered by the *Node Manager* by invoking the *User-Defined Operator (UDO) interface* (cf. Definition 21) of a specific CEP system. Whenever interested, the UDO Interface initiates the deployment of the required $\omega_{UD}$ by requesting the Node Manager $NM_n$ instance, which is responsible for downloading the operator container $C_{\omega_{UD}}$ on the node. ⑤ Once the operator $\omega_{UD}$ is available; it is instantiated on the respective node for execution on a specific CEP system. The communication between the CEP system and the $\omega_{UD}$ that operates on the CEPLESS middleware happens using the event queues (cf. Definition 18). ⑥ On the one hand, using the input queue $eq\_in_{\omega_{UD}}$, the incoming events from the CEP system are sent to the CEPLESS middleware for processing of the $\omega_{UD}$. ⑦ On the other hand, the complex events are emitted from the middleware via the output queue $eq\_out_{\omega_{UD}}$ to the CEP system for further processing by the system-defined operators or forwarding to the consumer.

### 6.2.1 *Serverless Middleware*

*In-memory event processing*

The CEPLESS middleware communicates with the CEP systems in the Execution layer using input and output in-memory queues $eq_{\omega_{UD}}$. The in-memory queues aim for two main design goals *(i)* high performance in the delivery of events and *(ii)* in-order and stateful event processing while ensuring system independent transfer of events from the user-defined operator $\omega_{UD}$ to the CEP system. As seen in Figure 46, CEPLESS middleware and the CEP system resides on the same node and avoid any network latency. In a distributed setting where multiple instances of $\omega_{UD}$ interact with the CEP system, we ensure that operators with dependency on the predecessors are residing on the same node. For instance, if a $\omega_S$ is dependent on a $\omega_{UD}$ operator, they are enforced to reside on the same node.

#### *In-memory Queue Management*

We provide in-memory queue management and batching mechanisms to ensure efficiency in delivering events from CEPLESS middleware. These mechanisms aim to process multiple events per time unit to ensure high throughput

in forwarding events. The forwarding of events from the middleware to the Execution layer is done using a client-server model. Instead of opening a new request for each event, we use batching to send the events such that the number of requests and the socket connections are reduced. These mechanisms aim to *(i)* maintain high performance in terms of throughput and latency and *(ii)* provide in-order and stateful processing of events. In the following, we elaborate on the above two design goals.

*Guarantee High Performance in the Delivery of Events*

To guarantee high performance in delivery events, we use a well-known control mechanism of in-memory queues called *command flushing* [240]. The command flushing mechanism, essentially, manages the time when a new command for a request is issued. Existing queueing systems [240, 241] uses automatic command flushing, which issues commands on each invocation by the client. This is very inefficient as it results in many commands written to the network sequentially, i.e., every time a new event is received. In contrast, we aim to issue concurrent command flushes at specified intervals, which works independently of the CEP query. In the context of CEPLESS middleware, each command refers to the request to receive or send an event.

Algorithm 6 presents the manual command flushing mechanism triggered when events are received from the CEP system and sent to the CEP system. In the algorithm, we define *events* and their *handlers* using the widely used definition given by the asynchronous event-based composition model in [210]. Therefore, events, modules, and attributes are denoted as $\langle$*Name of Module*, $Event$|*Attributes,...*$\rangle$.

In the main thread, the events are collected from the Execution layer, i.e., the primary events arriving from a producer via a specific CEP system or complex events arriving from the system-defined operator ($\omega_S$) of the CEP system (Line 1–2). Simultaneously, another background thread continuously processes the primary and complex events received from the CEP system (Line 3–17). We use two batches to store events for sending ($b_s$) and receiving ($b_r$). We maintain a threshold for sending (outBatchSize) and receiving (inBatchSize) events for each batch. The batching mechanism collects the events and sends the batch to the in-memory event queue ($eq\_in_{\omega_{UD}}$) after the threshold is reached (Line 9). Once the buffer is full, the background thread issues command flushes (Line 14–16). The background thread then waits until the events are sent and processed, as well as there are no new events in the batch (Line 5). The waiting time or the backoff time $t_{backoff}$ is determined based on new events in the batch $b_s$. In other words, we linearly increase the total backoff time $t_{sleep}$ every time we encounter an empty batch $b_s$.

*Timely dispatch of events allow high performance*

**Algorithm 6 :** Event queue handling and command flushing mechanism.

**Variables :**

| | | |
|---|---|---|
| $b_s$ | $\leftarrow$ | Sent and processed batch of events at $\omega_{UD}$ |
| $b_r$ | $\leftarrow$ | Batch of events received from $\omega_{UD}$ |
| $t_{backoff}$ | $\leftarrow$ | Increment in the back-off interval |
| $t_{sleep}$ | $\leftarrow$ | Total current back-off time |
| $eq\_in_{\omega_{UD}}$ | $\leftarrow$ | Input event queue for the $\omega_{UD}$ |
| $eq\_out_{\omega_{UD}}$ | $\leftarrow$ | Output event queue for the $\omega_{UD}$ |
| $e_{UD}, e_S$ | $\leftarrow$ | Events to and from user-defined operator (UD) and system-defined operator (S) |
| cmds | $\leftarrow$ | Event queue client commands |
| inBatchSize | $\leftarrow$ | Batch size of commands to be received |
| outBatchSize | $\leftarrow$ | Batch size of commands to be sent |

```
// actions when receive event is triggered by the Execution layer (ω_S)
upon receiveEvent ⟨ω_S, e_S⟩ do
    b_s.push(event);
// actions when send event is triggered by the ω_UD
upon sendEvent ⟨ω_UD, e_UD⟩ do
    t_sleep ← 0;
    if b_s.size = 0 then
        t_sleep ← t_sleep + t_backoff ;
        sleep(t_sleep);
    else
        if b_s.size > outBatchSize then
            eq_in_ω_UD ← b_s.pop(0, outBatchSize);
        else
            eq_in_ω_UD ← b_s;
            b_s.clear();
        if eq_in_ω_UD.size > 0 then
            cmds ← eq_in_ω_UD.compileCommands();
            cmds.flush();
        t_sleep ← 0;
// actions when receive events is triggered by the ω_UD
upon receiveEvent ⟨ω_UD, e_UD⟩ do
    t_sleep ← 0;
    t_sleep ← 0;
    b_r ← range(0, inBatchSize) ;
    if b_r.size = 0 then
        t_sleep ← t_sleep + t_backoff ;
        sleep(t_sleep);
    eq_out_ω_UD ← b_r ;
    trigger forwardEvent ⟨ω_S, Events in b_r⟩ ;
    t_sleep ← 0;
```

Furthermore, $t_{sleep}$ ensures that no event keep waiting for long until the batch is full, for example, that aids in keeping the latency in delivery of events low. In this way, we avoid wastage of resources since there is no need of

sending a request for an empty batch. We reset the total backoff time $t_{sleep}$ once there are events processed in the batch $b_s$ (Line 17).

After being processed by the $\omega_{UD}$ operator, the events are retrieved by another background thread (Line 18–27). To fetch all the events at once, we utilize a range query (Line 21). Like database range queries, in CEPLESS, the in-memory queue defines a start index and the length to fetch events from the queue. Using a range query, we again reduce the number of requests to retrieve an event and fast-forward the process. This thread also uses a linearly increasing backoff interval to avoid filling the queue with commands or requests, much like sendEvent background thread (Line 23).

We provide a universal format[45] to encapsulate events that are passed to the queue that provides runtime independence. This allows the middleware to interact with the different CEP systems without worrying about their semantics. Also, the format allows to keep the event size low that is necessary to avoid high latencies. Therefore, a combination of manual command flushing and batching mechanism aids in delivering events with efficiency, i.e., high throughput, while maintaining low latency.

*In Order and Stateful Processing of Events*

The in-memory queues use a FIFO ordering in forwarding and reception of the events. This means if a client at the CEPLESS middleware creates a request to add a new event, the same is appended to the tail of the specified operator event queue $eq_{\omega_{UD}}$. Note, each user-defined operator $\omega_{UD}$ maintains its unique event queues for incoming and outgoing events. The communication to the Execution layer is managed using the commands `push` and `pop` for sending and receiving the events, respectively, which inherently ensures FIFO ordering (cf. Algorithm 6). Therefore, the FIFO mechanism, combined with the communication to the Execution layer, ensures in-order processing of events. Furthermore, CEPLESS relies on the Execution layer ordering mechanisms, e.g., Flink ensures ordering using watermarks, such that the output events are produced in a correct manner[46]. For stateful processing, we make use of in-memory queues, as explained in the above paragraph. The queues store the history of events without having to persist the events on the external storage that could induce high I/O latency.

[45]Specifically, JSON data format is used in our implementation. `https://www.json.org/json-en.html` [Accessed in May 2021].

[46]The assumption is that the underlying CEP system is able to handle changes in the order of events, e.g., using on mechanisms for out-of-order arrival [242]

### 6.2.2 *Programming Interfaces*

We provide a set of interfaces for the application developers to develop CEP applications and the domain experts to extend existing CEP systems for compatibility with the CEPLESS middleware. The goal for these interfaces is to provide runtime and language independence to the application developers. In the following, before detailing the interfaces, we first explain the *Node Manager* component introduced in Figure 46 that is responsible for interaction between the user-defined operator interface in the specific CEP systems at the *Execution layer*. Afterwards, we detail the *User-defined Operator Interface* used to extend existing CEP systems to interact with CEPLESS middleware for runtime independence and the CEPLESS *Programming Interface* used to develop runtime- and language-independent operators in CEPLESS.

#### ***Node Manager***

Each node holding a $\omega_{UD}$ operator is managed by a Node Manager, as seen in Figure 46. Hence, each node has a running $NM_n$ instance. The node manager is colocated with the CEPLESS middleware (cf. Section 6.2.1) on the same node. The main function is to manage the deployment and processing of the $\omega_{UD}$ once requested by the CEPLESS programming interface. The node manager can handle multiple incoming requests for operators distinguished using a unique operator identifier. Thus, the manager can handle multiple $\omega_{UD}$ operators at the same node simultaneously by multiple CEP system instances.

The CEP system initiates a request through the *UDO Interface* implemented in each CEP system of the Execution layer. If the requested $\omega_{UD}$ operator exists in the registry at the programming interface, the operator is fetched and deployed at the middleware. Once deployed, the events are exchanged between the $\omega_{UD}$ operator via the event queues. Here, each operator remembers the queue address with which it communicates to the CEP system instance. In the following, we elaborate on the functionality of the User-defined Operator Interface. Another important functionality of the node manager is to support dynamic updates in the $\omega_{UD}$ operator specification while maintaining the state. If an update of an operator is trigerred from the programming interface, the node manager deploys the new operator, empties the old event queue, and redirects the “new” events to the “new” successor operator in the operator graph. If the old operator is not required anymore, it terminates the same or transfers it to a new node location.

### *User-Defined Operator (UDO) Interface*

We propose a *UDO Interface* that handles communication to the user-defined operator $\omega_{UD}$ running in the CEPLESS middleware. The interface needs to implement common functionality that handles events to and from the $\omega_{UD}$ and the CEP system. Motivated by the serverless computing principles, the interface provides a minimalistic abstraction for CEP systems to support *operators-as-a-service* in the CEPLESS middleware. Depending on the underlying mainstream programming language of the CEP system, the interface differs slightly. However, it provides a minimal overhead to the existing CEP system codebase.

*UDO interface provides platform independence*

Listing 9: User-Defined Operator Interface.

```
1 trait UserDefinedOperatorInterface {
2   def requestOperator(operatorName: String, cb: OperatorAddress => Unit): Unit
3   def sendEvent(e: Event, address: OperatorAddress): Unit
4   def addListener(address: OperatorAddress, cb: Any => Unit): Boolean
5   def removeListener(address: OperatorAddress): Boolean
6 }
```

In Listing 9, we show the major functionality of this interface required to be implemented by a CEP system to interact with the proposed CEPLESS middleware. First, the CEP system issues a request for a specific $\omega_{UD}$ operator at the CEPLESS middleware (Line 1). The first parameter `operatorName` gives the unique operator identifier for the $\omega_{UD}$ operator. The second parameter is a callback `cb` that is invoked when the $\omega_{UD}$ is deployed and returns a unique address at which it is reachable through the interface. The operator address is used to transfer the events between the CEP system and the CEPLESS middleware via the interface. `sendEvent()` method is used to send an event to a $\omega_{UD}$ operator at the middleware. The `Event` parameter is specific to a CEP runtime and has to be serialized to a readable format for the $\omega_{UD}$ operator (Line 3). To be able to receive events from the deployed $\omega_{UD}$ operator, the respective CEP system can add or remove listener using the methods in Line 4 and 5, respectively. The listener uses the operator address and expects a callback function, which is invoked every time new events arrive at the CEP system from $\omega_{UD}$ at the CEPLESS middleware.

### *UDO Programming Interface*

This module enables application developers to program $\omega_{UD}$ in runtime- and language-independent way. We allow user-defined operators $\omega_{UD}$ to be used across different CEP systems without having to change the operator design based on the execution semantics. Hence, we abstract away the processing

*Programming interface enables operators as a service*

function $f_\omega$ encapsulated inside $\omega_{UD}$ from the CEP system. The $\omega_{UD}$ operators can be developed, utilized, and updated using the preferred programming language supported by the virtualization environment[47]. This design choice gives the application developers the ability to use effectively any programming language of choice that is executable in a virtualization environment of the container by doing only minor modifications.

The applications can submit new $\omega_{UD}$ operators in two different ways to the CEPLESS middleware: *(i)* submitting only the function using our programming interface (CLI and web) and *(ii)* submitting a pre-built $\omega_{UD}$ operator container into the registry. By this, motivated by the *serverless computing* principles [164], we provide an equivalent *operator-as-a-service* for CEP systems. While providing the predefined $\omega_{UD}$ operator container, we allow application developers to use *any* programming language to implement operators. In contrast to the traditional serverless platforms like AWS Lambda [164] that supports a handful of programming languages, we provide essentially support *any* language, such as introduced in the example, to be integrated into CEPLESS. Besides, we provide the advantage of using multiple CEP systems simultaneously, facilitating the reuse of enormous functionalities provided by the CEP systems such as backpressure, query optimization, and query placement mechanisms.

Furthermore, developers do not have to know the underlying handling mechanism of forwarding events to the specific CEP system running in the Execution layer. The only assumption towards the implementation of $\omega_{UD}$ operators we have is that they can handle the transformations of input and output events. In CEPLESS, we do this using event queues as explained in the previous Section 6.2.1. CEPLESS manages all the registered $\omega_{UD}$ operators' containers $C_\omega$ and the privileges of the developers in a centralized container registry[48] that provides a central database for all the containers.

Once the application developer submits a $\omega_{UD}$, the programming interface prepares a container along with the communication interface, namely the UDO Interface, in a single container $C_\omega$ to have a fully functioning $\omega_{UD}$ operator. The UDO then handles the communication to and from the CEP system at the Execution layer along with the serialization and deserialization of event objects for the interaction with the middleware. Using the CEPLESS programming and UDO interfaces, the application developers need not know the underlying communication methodology of the used CEP system in the Execution layer and can seamlessly use $\omega_{UD}$ to execute a query. In this way, CEPLESS provides application developers with the freedom to develop $\omega_{UD}$ operators using only "operator" as a starting point.

[47] In this work, we base our implementation on Docker virtualization environment, which can support a variety of programming languages.

[48] Much alike docker registry. `https://docs.docker.com/registry/` [Accessed in May 2021].

## 6.3 Evaluation

This section evaluates CEPLESS in a three-fold manner aligned to our research contributions presented in Sections 6.2.1 and 6.2.2.

*Key evaluation questions*

1. Is it possible to replace operators dynamically, and how long does it take? (Section 6.3.2)
2. How much is the overhead of CEPLESS in terms of forwarding events and query performance compared to event processing in CEP systems? (Section 6.3.3)
3. Is CEPLESS able to provide runtime and programming language independence? (Section 6.3.4)

### *Section Structure*

Section 6.3.1 explains the evaluation environment: implementation, platform, dataset, queries, configuration, and the plots used for representation. Section 6.3.2 provides the performance evaluation of dynamically updating user-defined operators $\omega_{UD}$ compared to Apache Flink [54], a highly performant widely used CEP system. Section 6.3.3 provides the performance evaluation of CEPLESS middleware in terms of latency and throughput compared to Apache Flink and TCEP (cf. Chapter 4). Section 6.3.4 shows that user-defined operators $\omega_{UD}$ can be executed in CEPLESS middleware without any prior knowledge using two case studies of Flink and TCEP. Moreover, by using CEPLESS for $\omega_{UD}$ operators in five different programming languages, we show that $\omega_{UD}$ operators can be developed in a preferred programming language with minor modifications.

| | |
|---|---|
| Simulation time $t_s$ | 20 min |
| Warmup time $t_w$ | 60 s |
| Number of runs | 30 |
| Back-off interval increment $t_{backoff}$ | 1 ns |
| Output batch size $outBatchSize$ | 1, 10, <u>100</u> events |
| Input batch size $inBatchSize$ | 1, 10, <u>100</u>, 10000 events |
| Input event rate | <u>1000</u>, 10000, 100000 events per second |
| CEP systems | <u>Apache Flink</u> [28], TCEP [25] |
| Queries | Fraud detection (Listing 10), <u>Forward</u>, and K-means [243] |

Table 21: Configuration parameters for the evaluation. *Default or mostly used parameters are underlined* [239].

### 6.3.1 *Evaluation Environment and Methodology*

This section describes the CEPLESS implementation, evaluation platform, dataset, queries and the plots used for evaluation.

#### *Implementation*

CEPLESS *is extended with Flink and* TCEP

CEPLESS system follows a modular design and implementation written majorly in Golang and Java. In the middleware, the node manager is written in Golang that offers integration of in-memory queues like Redis [240] and Infinispan [241] with the proposed functionalities in Section 6.2.1. The CEPLESS programming interface provides a means to develop user-defined $\omega_{UD}$ operators, which is written majorly in Java, and the web interface is provided using the Angular framework written in TypeScript as proposed in Section 6.2.2. Finally, the UDO interface for the respective CEP system is written in the given programming language of the CEP system, example, for Apache Flink, it is written in Java, while for TCEP it is written in Scala. CEPLESS implementation with all its components and integration to TCEP and Flink is available open-source for use[49].

#### *Evaluation Platform*

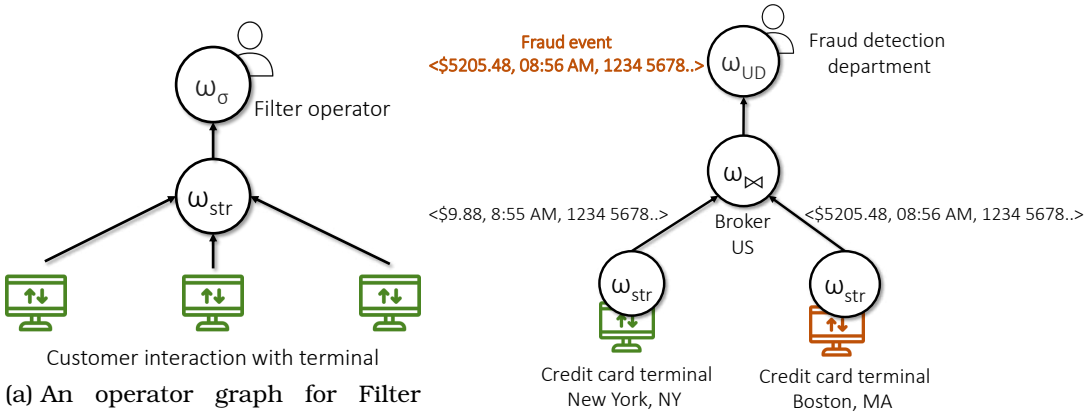
We use a machine with Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz processors and Ubuntu 18.04 operating system for evaluation. The machine is equipped with 128 GiB of memory and 24 cores. CEPLESS is a stand-alone CEP system with current extensions on Flink and TCEP. For comparison, we use two modes for evaluation *(i)* direct implementation of queries on the respective CEP systems (baseline) and *(ii)* implementation of $\omega_{UD}$ operator using CEPLESS extension on respective CEP systems. In the former mode, the query containing the $\omega_{UD}$ residing in a docker container executes inside a CEP system, while in the latter the $\omega_{UD}$ container and the query with system-defined operators $\omega_S$ run as separate containers on the CEP system[50].

#### *Dataset*

We use a dataset with financial transactions [65] specially dedicated for a fraud detection application. The dataset is taken from the real world and

[49] CEPLESS webpage: `https://luthramanisha.github.io/CEPless` [Accessed in May 2021].

[50] The docker containers are not restricted in terms of resources they can use but can use all the resources of the available machine.




(a) An operator graph for Filter query.

(b) An operator graph for a simple fraud detection query.

Figure 47: Queries used for evaluating CEPLESS: (a) Filter and (b) Fraud detection query [239].

is anonymized for research purpose. It contains 284,807 event tuples comprising credit card transactions. Each event tuple is of the following form: <*timestamp*, *amount*, *cardId*, *terminalId*>.

*Real world workload*

### *Queries*

Figure 47 shows the queries used for the evaluation. For the performance evaluation of CEPLESS (cf. Section 6.3.3), we use a Forward operator that forwards the event tuples to the consumer and a Filter operator (a). To evaluate for dynamic operator updates (cf. Section 6.3.2) and runtime independence (Section 6.3.4), we replace the Filter operator with a user-defined $\omega_{UD}$ operator (b) provided in Listing 10. As shown in Figure 47, we consider credit card terminals as event producers, which generate continuous data stream as transactions from the terminal. The event tuples act as an input to the CEP query (a) and (b) that transforms the events to detect fraud. Both queries (a) and (b) joins event tuples from multiple card terminals using a Join operator ($\omega_{\bowtie}$) that joins the event tuples within a sliding window of 30 seconds. The resulting data stream is sent for further processing to (a) a system-defined $\omega_S$ Filter operator or (b) a user-defined Filter $\omega_S$ operator. The Filter operator depicts if the processed transaction is fraudulent or not and this information is forwarded to the credit card fraud department.

### *Evaluation Configuration and Plots*

Table 21 presents a summary of the parameters used for the evaluation. *Output* and *Input Batch Size* describes the number of sent and fetched events

from the respective event queue in use as presented in Section 6.2.1. *Back-off time interval increment* is the value incremented on top of the back-off time. It is time until when the event queue waits for the new events to arrive (cf. Algorithm 6).

We use line plots, bar plots, and point plots for comparing our evaluation results. The line plot shows a thin line that depicts the mean, while the shaded areas report the percentiles, including the values of the investigated metric between the $5$th and the $95$th percentile. The bar plot shows the mean of the investigated metric and the error bar on the top, denoting the values between the $5$th and the $95$th percentile. The point plot, similar to the line plot, shows a point that depicts the mean value while the error bars report the percentiles, including the investigated metric values between the $5$th and the $95$th percentile. In the next sections, we present the performance evaluation of CEPless.

### 6.3.2 *Evaluation of Dynamic Operator Update*

This section aims to understand the performance of dynamic operator updates using CEPless middleware. We repeated the experiment 30 times to measure a 95% confidence interval on the investigated metrics. We utilize three evaluation metrics for measuring the performance: *(i)* downtime, *(ii)* update time, and *(iii)* throughput or (output) events. Here, downtime refers to the time interval when no events were received at the consumer and update time refers to the time interval between the termination of the old operator and initiation of the new $\omega_{UD}$ operator. Throughput is defined as the the maximum amount of event traffic that a CEP system can handle at a given time. As earlier motivated in Section 6.1.2, such updates in the business logic of custom operators are crucial for applications like fraud detection. In such examples, small downtimes and update times are considered good because system availability is critically important. In the experiment, we exchange the $\omega_{UD}$ operator with a new pattern $\omega_{UD}$ operator capable of detecting even more fraud patterns, and hence, we see an increase in the throughput after the exchange at time $t = 300$. There should be no throughput decline or spike in an ideal case and it should remain constant as it was before the update in the operator.

CEPless *promisingly updates the operator in a few ms*

Figure 48 shows the observed throughput measurements and operator update at $t = 290$s. Here, the experiment time is shown on the x-axis and the y-axis shows the throughput. We compare dynamic operator updates with Apache Flink, which do not have support for updating operators at runtime. We extended Flink to support this using an automated shell script. However, even when using a script in Flink, the query has to be redeployed, which

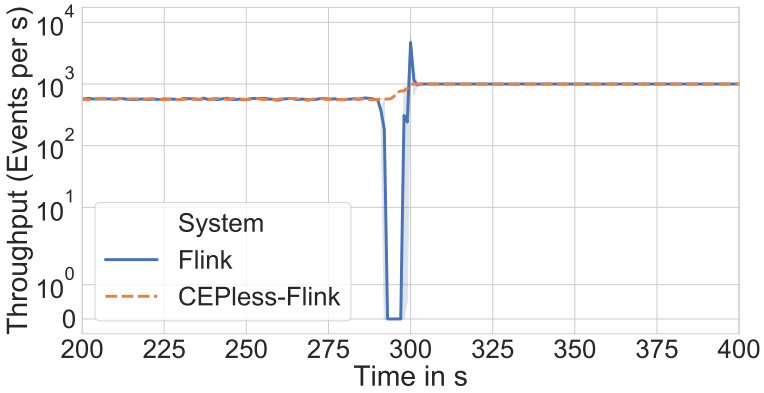



Figure 48: Throughput of Apache Flink with and without CEPLESS over the simulation time. CEPLESS quickly updates the operator while guaranteeing delivery of events. Whereas Flink experiences a downtime of around 8.6 seconds [239].

causes a mean downtime of around 8.6 s as seen in the plot at $t = 298$. The spike at $t = 300$ is seen due to the buffered events that were not processed due to the downtime of the operator. However, as soon as the query is deployed again, it is processed from the saved checkpoint where it stopped.

In contrast, CEPLESS, we see no downtime. We observe a small mean update time of around 238 ms (~97% reduction) while ensuring continuous delivery of output events. In fact the throughput of the new $\omega_{UD}$ operator is observed quite steadily in the system, as seen at $t = 300$. Therefore, we observe a very small update time using our implementation while inducing no disruption in the output, thanks to the quick deployment of the new operator by the node manager in contrast to bringing down the complete query. It is also important to note that while the update is taking place, the events are stored in the event queues earlier explained in Section 6.2.1, thus, none of the events are lost.

### 6.3.3 *Evaluation of* CEPLESS *Middleware*

This section aims to understand the performance of CEPLESS middleware in terms of event processing. We consider two metrics that are extremely important for CEP applications, but also in the context of the fraud detection application: *end-to-end latency* (cf. Definition 6) and *throughput* (cf. Definition 11). In this evaluation, we use a Forward operator introduced in Section 6.3.1 to observe all the output events emitted from the CEP system. We use two metrics in this evaluation, *(i)* the latency is measured as the time taken to forward the events from the producer to the consumer and *(ii)* the throughput is

measured as the number of output events received at the consumer per time unit. We consider the following system configurations:

1. Forward operator executed on the native CEP system without CEPLESS (baseline).

2. Forward operator executed on the CEP system as a $\omega_{UD}$ using our CEPLESS extension, Redis [240] as an in-memory queue, with and without batching.

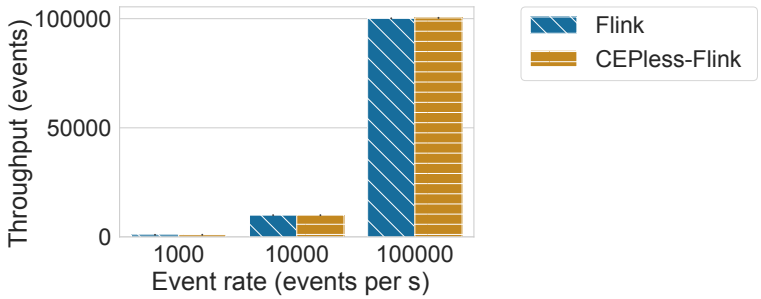
In the following, we show impact on the performance in terms of *(i)* throughput, *(ii)* latency and *(iii)* different batch sizes using our reference implementation of CEPLESS on Flink and TCEP. By showing the applicability of two CEP systems, we emphasize that CEPLESS can be extended with diverse CEP systems.

### *Impact on Throughput*

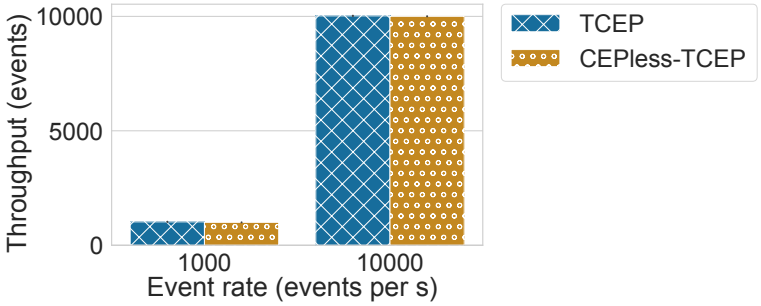
This section aims to understand the performance of CEPLESS by measuring the throughput. We refer to the throughput being *optimal* when the output events match the input event rate, i.e., all the forwarded events are received at the consumer. Again, we collected the measurements over 20 minutes and repeated the evaluations 30 times (cf. Table 21). We report the 95% confidence interval of throughput measurements in the barplot with different input event rates. The error bar on the top shows the interval bound. Figure 49 (a) presents the mean throughput values observed for Flink with and without CEPLESS over an event rate of 1000, 10000, and 100000 events per second, respectively (left to right). Consistent with our hypothesis, we do not see any drop in the throughput. In fact, we can match the *optimal* throughput in the output events as done by Flink. Table 23 in Section A.3 elaborates on the statistics presented in Figure 49 (a) with *mean*, *minimum*, *maximum*, and percentile (90, 95, and 99) of the throughput observations. Here also, we see that CEPLESS with Flink matches the baseline implementation under all the event rates.

*Zero overhead in terms of throughput compared to native CEP*

Figure 49 (b) shows similar observations with the underlying CEP system as TCEP. Again, with CEPLESS with TCEP, we can match the optimal throughput as done by TCEP. The reason why we do not report the observations for 100000 events is due to the inability of the TCEP baseline to deal with this amount of events. This is due to the absence of backpressure and flow-control mechanisms.




(a) Throughput of Apache Flink with CEPLESS and without CEPLESS (baseline) and with CEPLESS.




(b) Throughput of TCEP with CEPLESS and without CEPLESS (baseline).

Figure 49: Throughput evaluation of CEPLESS compared to native (without) CEP system Flink and TCEP. We show that the respective extensions with CEPLESS is equally performant to the baseline thanks to the efficient queue management and batching mechanisms.

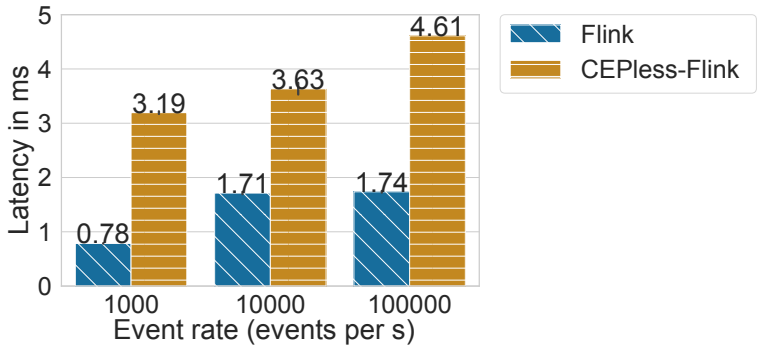
Nevertheless, we show that using CEPLESS on top of Flink and TCEP can match the throughput performance of the baseline even for very high event rates of 100000 events per seconds. Another important observation is the batch size setting to achieve the optimal throughput. Especially for a higher event rate of 100000 events per second, we had to set the input batch size to 10000 events. The empirical analysis on this is presented in Section 6.2.1. However, the main reason for achieving high throughput is that while the size of the range query has increased, there are fewer requests for event processing and eventually lower internal network round-trips. However, at times, larger batch sizes cost higher latency, which we evaluate in the next subsection. One more interesting observation is that we see at times a higher throughput than the input event rate for the CEPLESS extension. After further analysis, we found out that due to the batching mechanism proposed in Algorithm 6, the events stay in the batch for some time (backpressured) instead of getting processed immediately. These events are emitted as soon as the batch size threshold is reached, which means that, initially, a lower number of events than the input rate could be received, while later, a higher number of events could be received (see max value in Table 23).

| | 1000 events/s | | | | 10000 events/s | | | | 100000 events/s | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **System** | *mean* | *min* | *max* | *percentiles (90, 95, 99)* | *mean* | *min* | *max* | *percentiles (90, 95, 99)* | *mean* | *min* | *max* | *percentiles (90, 95, 99)* |
| **Flink** | 0.78 | 1.0 | 31.46 | 0.98, 1.06, 1.55 | 1.71 | 1.0 | 341.61 | 2.09, 2.22, 3.11 | 1.74 | 1.0 | 87.91 | 2.29, 2.45, 2.88 |
| **CEPless-Flink** | **3.19** | **1.0** | **26.19** | **4.10, 4.35, 4.9** | **3.63** | **1.15** | **503.50** | **4.21, 4.48, 5.15** | **4.61** | **1.82** | **956.31** | **5.71, 6.11, 7.80** |
| **Tcep** | 1.14 | 1.0 | 27.89 | 1.45, 1.57, 1.82 | 4.21 | 1.0 | 496.63 | 2.54, 2.74, 24.32 | - | - | - | - |
| **CEPless-Tcep** | **4.94** | **0.72** | **21.27** | **6.10, 6.37, 6.86** | **5.21** | **1.97** | **488.84** | **6.25, 6.75, 10.38** | - | - | - | - |

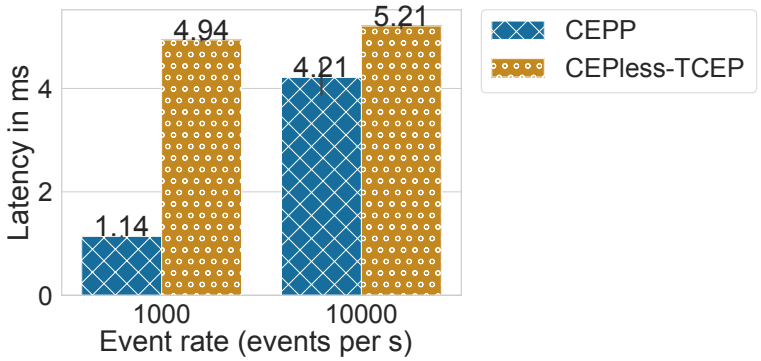
Table 22: Latency measurements: mean, min, max, and percentiles (90, 95, 99) for the Forward operator (in ms).

| | 1000 events/s | | | | 10000 events/s | | | | 100000 events/s | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **System** | *mean* | *min* | *max* | *percentiles (90, 95, 99)* | *mean* | *min* | *max* | *percentiles (90, 95, 99)* | *mean* | *min* | *max* | *percentiles (90, 95, 99)* |
| **Flink** | 1047 | 108 | 1093 | 1001, 1002, 1002 | 10475 | 2180 | 19990 | 10010, 10011, 10020 | 100123 | 16770 | 200168 | 100098,100136,100797 |
| **CEPless-Flink** | **1070** | **378** | **1190** | **1003, 1004, 1004** | **10477** | **1174** | **20007** | **10019, 10027, 12081** | **100353** | **18353** | **249689** | **100623,100815,101806** |
| **Tcep** | 1021 | 396 | 2292 | 1001, 1002, 1002 | 10022 | 2660 | 45185 | 10010, 10015, 10110 | - | - | - | - |
| **CEPless-Tcep** | **1000** | **214** | **4927** | **1001, 1002, 1004** | **10002** | **580** | **32091** | **10015, 10025, 10163** | - | - | - | - |

Table 23: Throughput measurements: mean, min, max, and percentiles (90, 95, 99) for the Forward operator.




(a) Latency of Apache Flink with CEPLESS and without CEPLESS (baseline).




(b) Latency of TCEP with CEPLESS and without CEPLESS (baseline).

Figure 50: Latency evaluation of CEPLESS compared to native (without CEPLESS) CEP system Flink and TCEP. The results show that CEPLESS induces a very low overhead of around 2 ms for very high event rates of up to 100000 events per second.

### *Impact on Latency*

This section aims to understand the performance of CEPLESS in terms of end to end latency. Like the throughput evaluations, we measure the latency in receiving output events using the Forward operator with and without CEPLESS in Flink and TCEP, respectively. Figure 50 (a) shows the latency measurements of running a Forward operator on the Flink with and without CEPLESS for the different event rates from 1000 to 100000 events per second. We observe that CEPLESS induces only a minimal overhead on top of Flink and attains a good performance in terms of latency compared to the baseline.

In particular, we elaborate on the results in Table 22 and see that CEPLESS induces an overhead of only 1.92 ms for 10000 events per second and 2.87 ms for 100000 events per second (calculated by subtracting the mean latency with baseline with mean latency with CEPLESS). Similar observations were made for CEPLESS with TCEP as a CEP system. For TCEP a mean overhead of around 1 ms was observed over a load of 10000 events per second.

*Negligible overhead in latency*

We observe only a small overhead in latency because we have parameterized batch size for different event rates, as we elaborate in Section A.3.1. In essence, the overhead majorly comprises two factors *(i)* the serialization and deserialization of event objects to and from the universal format and *(ii)* the total round trip time to and from the event queues and the CEP systems.

### 6.3.4 *Runtime and Language Independence*

This section evaluates the runtime and language independence of CEPLESS middleware. We show this by a two-fold evaluation. *(i)* We show the language independence by showing that specification of user-defined operators can be performed in different languages: Go, Java, Scala, C++, and Python, irrespective of the mainstream language of the CEP system. *(ii)* We show the runtime independence by integrating previously introduced CEP systems into the Execution layer, which can be selected independently for executing operators. Besides, we show the analysis on the implementation overhead for user-defined operators using the proposed programming interface compared to native CEP systems in Section A.3.1. In the following, we present the realization of CEPLESS middleware using different programming languages and the respective CEP systems, hence validating the universal applicability of our approach.

*CEPLESS operator specification is available in five different languages*

#### ***Language Independence***

We implemented a fraud detection operator using the CEPLESS middleware specification interface introduced in Section 6.2.2. Listing 10 shows a basic implementation of the operator. First, the function receives a map as input with keys of string type and values of any type containing the previously introduced structure of events received from a terminal (Line 1). To know which events were already received, we define a cache that saves the previously received events (Line 3). When an event is received, the function iterates through the cache and checks the following requirements: Line 4-6: Checks if the card ID of the previously received event is equal to the current event, and if the card ID is equal, the transaction was received in a shorter timeframe than 10 minutes. If both requirements are met, the incoming event is

Listing 10: A Java example for a user-defined fraud detection operator as described in the scenario.

```
1  public void handleEvent(Map<String, Any> input) {
2      for (int idx : events) {
3          Map<String, Any> event = events[idx];
4          if (!input["cardId"].equals(event["cardId"]) && TimeUnit.MILLISECONDS.
               toMinutes(input["timestamp"] - event["timestamp"]) > 10) {
5              continue;
6          }
7          Map<String, Any> compositeEvent = new Map<String, Any>();
8          compositeEvent.put("event1", input);
9          compositeEvent.put("event2", event);
10         this.emit(compositeEvent);
11     }
12     this.events.add(input);
13 }
```

considered to be fraud. The operator emits a composite event consisting of two conflicting transactions.

In CEPLESS middleware, we provide templates containing the UDO interface (cf. Listing 9), in Golang, Java, Scala, C++ and Python made publicly available here[51]. Furthermore, the programming interfaces and the project structure are written to be easily reused for other programming languages in order to realize custom applications. Hence, we show that the CEPLESS middleware can be used to specify user-defined operators in the different programming languages, with easy possibilities of extension.

### *Runtime Independence*

We evaluate the applicability of CEPLESS on existing real-world CEP systems to show runtime independence. Therefore, we ported it on two open-source CEP systems: Apache Flink [28] and TCEP [25]. The ported versions are *not* simplified, but the respective components of CEPLESS are implemented to achieve a functionally equivalent system.

#### *Apache Flink*

To enable the usage of CEPLESS in a widely used streaming system, like Apache Flink [28], we first extended the respective query language. We implemented a new keyword named `.serverless()` (as seen in Figure 45) that enables end-users to specify a user-defined operator in a query. This function

[51] Operator templates in CEPLESS. `https://github.com/luthramanisha/CEPless/tree/main/operators` [Accessed in May 2021].

expects an operator name and returns an `AbstractStreamOperator` reference as expected by the Flink streaming engine. Besides this, we implemented the UDO interface to enable communication between Flink and UD operators. As soon as an event is received, we transform it into the universal format and submit it to the event queue, as seen in Algorithm 6. Furthermore, we start the receiving process to listen to the results of the operator. When an event is received by this process, the UDO interface converts the universal format back into the Flink event format and passes the event object down the query. We mainly see dependencies on the following three aspects: query language, operator architecture, and event architecture. These aspects have to be considered when implementing the extension of CEPless middleware into a new execution engine. The implementation of the UDO interface took 90 lines of code (LOC) in the Flink engine.

*CEPless can be easily extended with CEP systems*

### Tcep

As a second case study, to demonstrate the extensibility of our approach, we use Tcep [25] written in Scala. Coherently to the Flink implementation, we started by extending the used query language of the system by defining a new `.serverless()` function to define new operators. We implemented the UDO interface to send and receive events as well, equally to the Flink implementation. Furthermore, we extended Tcep to include a Kafka producer, which was already provided natively by Flink. In comparison to Flink, the UDO interface took 130 lines of code. The difference in LOC is mainly caused by the different communication semantics internally in Tcep.

## 6.4 Summary

This chapter proposes the third and the final contribution of this thesis solving the interoperability problem in the context of Complex Event Processing systems. So far, it is hard to share or reuse functionalities across the diverse CEP programming models because of the tight dependencies between the specification language and the runtime environment of operators. Such dependencies can be problematic for applications which require flexible change in operator specification, for instance, fraud detection application. Furthermore, reuse across CEP systems becomes necessary for IoT applications that often need very distinct but unifiable functionalities between heterogeneous systems. We aim to resolve these challenges by proposing a middleware inspired by serverless computing principles called CEPless. With CEPless we contribute a *operators-as-a-service* model providing the following programming interfaces and mechanisms. *(i)* Novel programming interfaces to develop user-defined operators that can be independently programmed without de-

pendency to the CEP runtime environment and still be the part of the query pipeline. *(ii)* In-memory queue management and batching mechanisms to process events with high performance as well as to manage state for stateful operators. At the same time, by changing the operator specification at runtime, we enable dynamic operator updates that are highly desirable for many applications like fraud detection. Our evaluation using CEPLESS demonstrates that it integrates well into existing CEP systems Apache Flink and TCEP proposed in this thesis. Furthermore, for the ingested workload of up to 100,000 events per second, it can attain equivalent throughput as the native CEP system (Flink and TCEP) and dynamic operator updates can be accommodated in around 238 ms.

# 7

# Conclusion and Outlook

> "*One accurate measurement is worth a thousand expert opinions.*"
>
> – Grace Murray Hopper

This thesis presents three key contributions to solve the problems of *adaptivity (RQ1)*, *efficiency (RQ2)* and *interoperability (RQ3)* in the context of Complex Event Processing (cf. Figure 51).

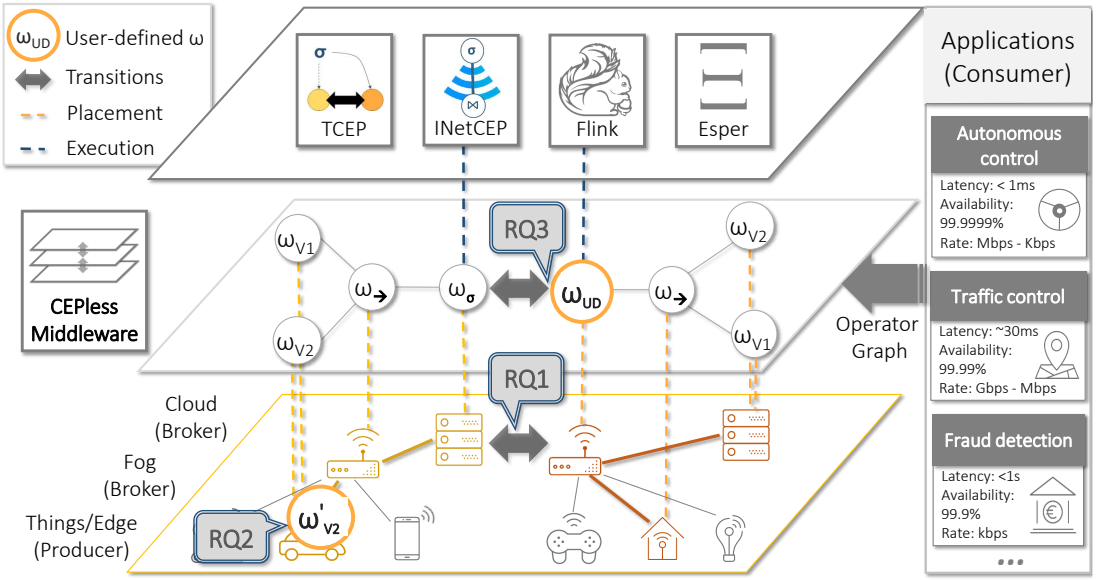



Figure 51: Summary of all the contributions as a single architecture. Here, TCEP contributes a novel transition-capable programming model for OP mechanisms, INETCEP contributes a unified communication model for network-centric query execution, and CEPLESS proposes a novel unified serverless middleware for query specification that is independent from the CEP execution model.

*Overall contributions*

In particular, we proposed *(i)* a programming model and a methodology for transitions in CEP systems, *(ii)* a unified communication model and effi-

cient algorithms to support adaptivity in operator placement and query execution mechanisms, and *(iii)* a serverless middleware with programming abstractions to allow reuse of multiple and diverse CEP execution models. We implemented and evaluated the findings of the aforementioned contributions in three novel concepts, named TCEP, INETCEP, and CEPLESS, all of which are publicly available[52].

This chapter summarises the contributions of this thesis and thereby the answers to the research questions in Section 7.1. After that, we provide conclusions based on the contribution results in Section 7.2 and finally highlight a few directions for potential future work in Section 7.3.

## 7.1 Contributions Revisited

*Key research gaps and summary*

This thesis solves three fundamental problems related to *adaptivity*, *efficiency*, and *interoperability* in the mechanisms as well as programming models of CEP as identified in Chapter 1. We identify three critical research gaps in Chapter 2, namely, *(i)* lack of adaptivity in mechanisms of CEP failing to fulfill changing quality requirements in the presence of dynamic environmental conditions, *(ii)* missing network-centric abstractions in expressing event processing operations and executing them in the underlay, and *(iii)* missing programming abstractions that allow reuse of diverse CEP execution environments and allow changes in the query specification.

To this end, in Chapter 4, we proposed a novel transition-capable programming model, named TCEP, and methods for transitions between OP mechanisms to meet the changing quality requirements of applications. In TCEP, we propose concepts to express adaptable OP mechanisms and cost-efficient transition execution strategies.In Chapter 5, we proposed a unified communication model, named INETCEP, that enabled processing of continuous data streams in the network layer. We proposed network-centric query execution on top of Information-centric Networking architecture to increase the efficiency in event processing. Finally, in Chapter 6, we presented a unified serverless middleware, named CEPLESS, that provides a programming abstraction for specification of queries independent of the underlying CEP execution model. Using CEPLESS, application developers can combine the benefit of the proposed and other CEP systems in a unified way, as well as the queries can be updated at runtime as and when required by the applications.

[52]TCEP https://luthramanisha.github.io/TCEP/ [Accessed in May 2021].
INETCEP https://luthramanisha.github.io/INetCEP/ [Accessed in May 2021].
CEPLESS https://luthramanisha.github.io/CEPless/ [Accessed in May 2021].

## 7.2 Key Results

We show the benefits of the proposed contributions by performing extensive real-world system evaluations for each of our three contributions (cf. Figure 52).

*Key answers on the contribution results*

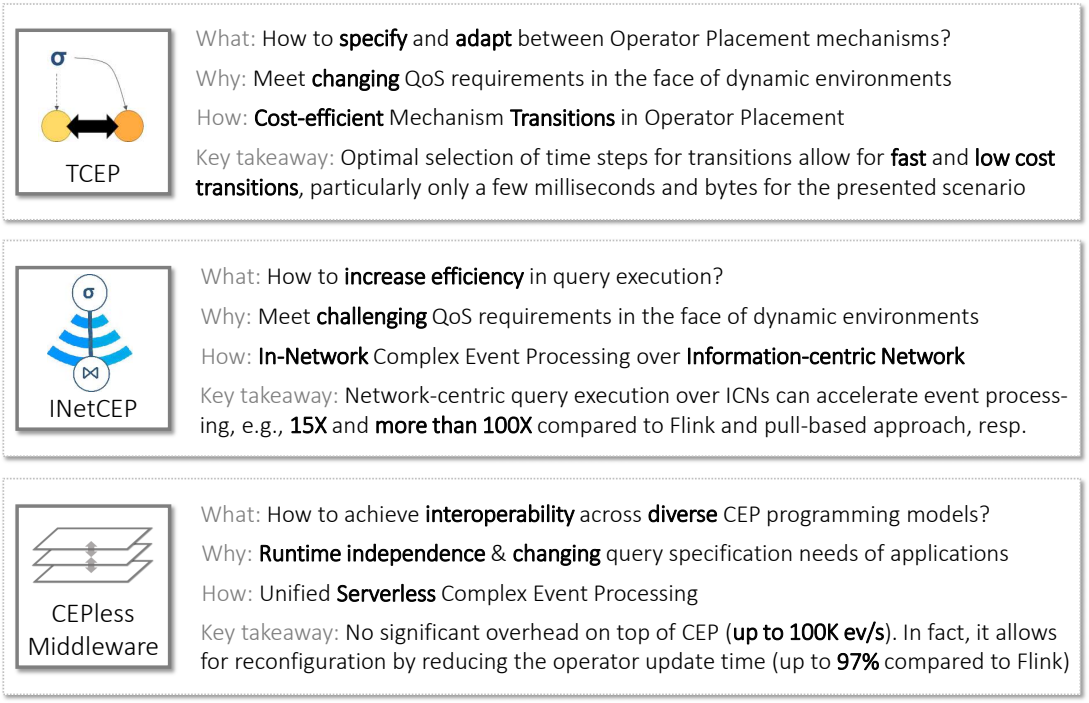



Figure 52: Key insights into the research contributions based on the motivation, the methods and the evaluation results.

First, we showed, using the proposed programming model of TCEP, that transition-capable OP mechanisms can be specified considering a broad-spectrum of QoS requirements and algorithm design. By implementing six state-of-the-art OP mechanism, we showed that TCEP enables the utilisation of distinct mechanisms and provides a good understanding of their performance and cost characteristics. We showed that TCEP enables *(i)* quality-based selection of OP mechanism and *(ii)* transitions between OP mechanisms to meet the changing QoS requirements of applications.

*TCEP reduced the transition costs*

We analyzed the ability to deliver events while performing a transition and showed that a transition can be performed in a live and seamless manner. We performed a detailed analysis of the costs involved in terms of transitions and the learning overhead for a transition. An important observation is that the transition algorithms induce only a negligible overhead while performing a transition, thanks to the optimal selection of time steps to perform a transi-

tion when the state to be transferred is minimal. Similarly, for learning, overhead is negligible because of zero offling training costs. Another reason is the appropriate balance between exploration and exploitation that is achieved by applying an appropriate selection pressure.

Second, using the unified communication model and network-centric query execution algorithms proposed in INETCEP, we improved significantly the efficiency of event detection using in-network resources by applying the concepts of Information-centric Networking. By proposing the unified communication model for ICN, we enabled a broad-spectrum of applications to benefit from the network-centric resources. The concept can be applied to the applications that need continuous processing of data as well as to the applications that need data on request.

*INETCEP improved the efficiency by reducing event delivery time and loss*

Our evaluations showed that INETCEP enables specification of both kinds of applications: push- and pull-based. Notably, for the push-based applications we observed a tremendous improvement in latency while forwarding events compared to a widely used open-source CEP system Flink ($15\times$ better for the event rate of 50K events per second). Morover, we observed a 0% loss rate compared to $61.18\%$ seen in the ICN pull-based reference approach. Similar observations were made for throughput, where our methods achieve a high equivalent throughput as Flink. We performed an extensive performance evaluation of query execution algorithms using two IoT case studies: disaster and smart plug prediction scenarios. Using these queries and the standard CEP queries, we again observed a tremendous improvement in the performance in terms of latency compared to Flink that processes a query in an overlay network in contrast to the underlay. Like Flink, our approach superseded the performance of the pull-based reference approach due to the unified communication model that induces no overhead on the network in retrieving the query results continuously. Similarly, our approach outperformed for distributed evaluation with distinct topologies, by benefiting from the query execution algorithms that pragmatically places operators on the in-network resources while considering the QoS requirements and link capacity.

Finally, by proposing a serverless CEP middleware, named CEPLESS, we provided an abstraction for application developers to specify queries independent of a specific CEP execution environment. This platform independence hides away the complexity of the CEP system[53] from the application developers that aids in the fast development of application queries. Furthermore, the ability to update operators at runtime in a very quick and seamless manner allows to deal with the dynamic needs of an application.

*CEPLESS abstracts the complexity and enables reuse*

[53]For instance, Flink comprises more than 900,000 LOC, which can be highly complex to understand by application developers.

Our results show that using the CEPless middleware, queries can be specified in the programming language of choice and can run independently of the CEP execution model. Our evaluation shows runtime and language independence by the specification of user-defined operators in *five* different programming languages and two distinct CEP systems: Flink and TCEP. We show that CEPless is equally performant in throughput as the native CEP systems Flink and TCEP. Moreover, we reduced the reconfiguration time of an operator by 97% compared to Flink while ensuring no downtime and steady throughput of complex events.

## 7.3 Future Outlook

We present the future work for the respective contributions in the following.

### TCEP

While the proposed learning algorithm in TCEP aims to provide less overhead for resource-constrained IoT applications, other applications with no resource constraints might benefit from exhaustive machine learning models like neural networks. Deep learning models can provide better estimations on the OP mechanism. Furthermore, estimations could incorporate environmental conditions. An optimal selection method and performance modelling of the OP mechanism can better fulfill the QoS requirements.

*Deep learning methods for transitions*

TCEP proposes a generic transition-capable CEP system that enables mechanism transitions between OP mechanisms. The system can benefit from transitions between other query mechanisms such as query optimisation and elasticity in conjunction with OP mechanisms. Moreover, transitions can be extended to other state-of-the-art CEP systems that are originally non-transition-capable. Coordination between transition- and non-transition-capable systems is a core research topic in the third funding phase of Collaborative Research Centre “MAKI”.

*Coordinating transition- and non-transition capable mechanisms*

### INETCEP

INETCEP proposes to meet the efficiency requirements of IoT applications by network-centric execution of queries. Probabilistic guarantees in this realm can be investigated so that by using in-network resources guarantees on the fulfillment of the QoS requirements can be provided. To provide such guarantees, an estimation of the performance related to the in-network resources are required. Guaranteeing such mission-critical requirements is another core

*Mission-critical guarantees*

research topic in the third phase of Collaborative Research Centre "MAKI". Although INETCEP shows the benefit of hybrid processing in the underlay and the overlay network, larger benefits can be uncovered by offloading processing to the underlay pragmatically.

*P4 programmable CEP*

Other networking paradigms such as Software-defined Networks provide control over the in-network resources and the ability to process operators inside the programmable hardware, for instance, P4 [244] is a programming abstraction for data plane of SDN. In our early work [66], we show that simple filter operations can be supported using such hardware to uncover tremendous performance improvement.

### CEPLESS

*Serverless Operator Placement*

CEPLESS provides an important programming abstraction for CEP middleware and serves as a crucial initial work in the realm of serverless CEP. While we have enabled independent specification for queries, other CEP mechanisms can benefit from this independence as well, such as OP mechanisms. By doing this, a common programming model for OP mechanisms can be established, which do not rely on a single CEP execution model. Such an abstraction is highly beneficial for the development of novel OP mechanisms, that is crucial for distributed query processing in CEP.

*Auto-scaling and just-in-time billing operators*

Finally, CEPLESS presents a foundation to develop CEP systems with advanced serverless features such as just-in-time billing and auto-scaling in the cloud environment. By doing this, the applications do not have to depend on the CEP specific scaling strategies [245]. For instance, the proposed user-defined operator programming interface can provide information required for billing and the virtualisation using containers, which is at the core of CEPLESS can be utilised to support auto-scaling [73].

### Acknowledgements

This work has been co-funded by the German Research Foundation (DFG) as part of the project C2 within the Collaborative Research Center (CRC) 1053 – MAKI.

# Bibliography

*Internet resources have been accessed in May 2021.*


[1] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions." In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660 (cit. on pp. 1, 43).

[2] Mohammad Nasajpour, Seyedamin Pouriyeh, Reza M. Parizi, Mohsen Dorodchi, Maria Valero, and Hamid R. Arabnia. "Internet of Things for Current COVID-19 and Future Pandemics: an Exploratory Study." In: *Journal of Healthcare Informatics Research* 4 (2020), pp. 325–364 (cit. on p. 1).

[3] Krishna Kumar, Narendra Kumar, and Rachna Shah. "Role of IoT to avoid spreading of COVID-19." In: *International Journal of Intelligent Networks* 1 (2020), pp. 32–35 (cit. on p. 1).

[4] Kevin Ashton et al. "That 'internet of things' thing." In: *RFID journal* 22.7 (2009), pp. 97–114 (cit. on p. 1).

[5] Cisco. *Internet of Things at a glance*. `http://www.audentia-gestion.fr/cisco/pdf/at-a-glance-c45-731471.pdf` (cit. on p. 1).

[6] Gianpaolo Cugola and Alessandro Margara. "Processing flows of information: From data stream to complex event processing." In: *ACM Computing Surveys (CSUR)* 44.3 (2012), pp. 1–62 (cit. on pp. 2, 14–16, 159).

[7] David Luckham. "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems." In: *Proceedings of Rule Representation, Interchange and Reasoning on the Web (RuleML)*. 2008, pp. 3–3 (cit. on pp. 1, 16).

[8] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. 1st. Springer Publishing Company, Incorporated, 2009 (cit. on pp. 1, 25).

[9] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter heron: Stream processing at scale." In: *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. 2015, pp. 239–250 (cit. on pp. 1, 3, 5, 6, 20, 22, 24, 43, 62, 88, 155).

[10] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing." In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1792–1803 (cit. on pp. 1, 22, 55).

[11] Apache. *Apache Kafka: A distributed streaming platform*. http://kafka.apache.org/ (cit. on pp. 1, 41, 43).

[12] Manisha Luthra, Boris Koldehofe, and Ralf Steinmetz. “Transitions for Increased Flexibility in Fog Computing: A Case Study on Complex Event Processing.” In: *Informatik Spektrum* 42.4 (2019), pp. 244–255 (cit. on pp. 2, 63).

[13] Kelly Herrel. *Defeating latency is at the heart of the AI, fraud and data challenges at banks*. `https://www.bai.org/banking-strategies/article-detail/defeating-latency-is-at-the-heart-of-the-ai-challenge-at-banks/`. 2020 (cit. on pp. 2, 5, 114, 155).

[14] Boris Koldehofe. “Principles of Building Scalable and Robust Event-Based Systems.” habilitation. Technical University of Darmstadt, 2019 (cit. on pp. 3, 17, 18).

[15] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. “Network-Aware Operator Placement for Stream-Processing Systems.” In: *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*. 2006, pp. 49–61 (cit. on pp. 3, 5, 17–19, 23, 57, 65, 66, 89–91, 94, 99).

[16] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. “Dynamic Plan Migration for Continuous Queries over Data Streams.” In: *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. 2004, pp. 431–442 (cit. on pp. 3, 5, 80, 87).

[17] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. “Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources.” In: *Procedings of the VLDB Endowment* 10.7 (2017), pp. 733–744 (cit. on p. 3).

[18] Shengchao Liu, Jianping Weng, Jessie Hui Wang, Changqing An, Yipeng Zhou, and Jilong Wang. “An adaptive online scheme for scheduling and resource enforcement in Storm.” In: *IEEE/ACM Transactions on Networking* 27.4 (2019), pp. 1373–1386 (cit. on pp. 3, 5, 20, 22).

[19] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. “Optimal operator placement for distributed stream processing applications.” In: *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems (DEBS)*. 2016, pp. 69–80 (cit. on pp. 3, 5, 17–19, 21, 23, 89).

[20] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. “Efficient Operator Placement for Distributed Data Stream Processing Applications.” In: *IEEE Transactions on Parallel and Distributed Systems* 30.8 (2019), pp. 1753–1767 (cit. on pp. 3, 5, 19, 23, 88, 103).

[21] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. “Aurora: A New Model and Architecture for Data Stream Management.” In: *The VLDB Journal* 12.2 (2003), pp. 120–139 (cit. on pp. 3, 6, 155).

[22] Elke A. Rundensteiner, Luping Ding, Timothy M. Sutherland, Yali Zhu, Bradford Pielech, and Nishant K. Mehta. “CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity.” In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. 2004, pp. 1353–1356 (cit. on pp. 3, 6, 113, 155).

[23] Ying Xing, Stan Zdonik, and J-H Hwang. “Dynamic load distribution in the Borealis stream processor.” In: *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. 2005, pp. 791–802 (cit. on pp. 3, 5, 6, 21, 57, 113, 155).

[24] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, and Patrick Valduriez. “StreamCloud: A Large Scale Data Streaming System.” In: *Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS)*. 2010, pp. 126–137 (cit. on pp. 3, 5, 6, 52, 80, 155).

[25] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. "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 (DEBS)*. 2018, pp. 136–147 (cit. on pp. 3, 6, 11, 63, 66, 155, 171, 181, 182).

[26] Manisha Luthra, Boris Koldehofe, Jonas Höchst, Patrick Lampe, Ali Haider Rizvi, Ralf Kundel, and Bernd Freisleben. "INetCEP: In-Network Complex Event Processing for Information-Centric Networking." In: *Proceedings of the 15th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–13. URL: https://luthramanisha.github.io/INetCEP/ (cit. on pp. 3, 6, 115, 123, 135, 141, 143, 155).

[27] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, Kirak Hong, David Lillethun, and Umakishore Ramachandran. "MCEP: A Mobility-Aware Complex Event Processing System." In: *ACM Transactions on Internet Technology (TOIT)* 14.1 (2014), pp. 1–24 (cit. on pp. 3, 5, 6, 19, 21, 23, 155).

[28] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine." In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (cit. on pp. 3, 5, 6, 11, 16, 22, 24, 43, 62, 155, 171, 181).

[29] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. "Benchmarking Distributed Stream Data Processing Systems." In: *Proceedings of the 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1507–1518 (cit. on pp. 3, 6, 22, 55, 58).

[30] Fabrice Starks, Vera Goebel, Stein Kristiansen, and Thomas Plagemann. "Mobile distributed complex event processing—Ubi Sumus? Quo vadimus?" In: *Mobile Big Data*. 2018, pp. 147–180 (cit. on pp. 3, 6, 22, 24, 88, 89, 94).

[31] Alibaba Cloud. *Double 11 Real-Time Monitoring System with Time Series Database*. https://www.alibabacloud.com/blog/594855. 2019 (cit. on p. 4).

[32] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. "Adaptive Online Scheduling in Storm." In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS)*. 2013, pp. 207–218 (cit. on pp. 5, 19, 22, 23, 62).

[33] T. M. Sutherland, B. Pielech, Yali Zhu, Luping Ding, and E. A. Rundensteiner. "An adaptive multi-objective scheduling selection framework for continuous query processing." In: *Proceedings of the 9th International Database Engineering Application Symposium (IDEAS)*. 2005, pp. 445–454 (cit. on pp. 5, 19, 22).

[34] Rahul Dwarakanath, Boris Koldehofe, and Ralf Steinmetz. "Operator Migration for Distributed Complex Event Processing in Device-to-Device Based Networks." In: *Proceedings of the 3rd ACM Workshop on Middleware for Context-Aware Applications in the IoT (M4IoT)*. 2016, pp. 13–18 (cit. on pp. 5, 53, 82).

[35] *EsperTech – Esper*. http://www.espertech.com/esper/. 2006 (cit. on pp. 5–7, 16, 20, 24, 94, 155, 156, 158).

[36] *Next-generation technologies to future-proof your business*. https://wso2.com/. 2015 (cit. on pp. 5, 6, 20, 24, 155).

[37] Yanif Ahmad and Uğur Çetintemel. "Network-aware Query Processing for Stream-based Applications." In: *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. 2004, pp. 456–467 (cit. on pp. 5, 19, 23).

[38] Dan O'Keeffe, Theodoros Salonidis, and Peter Pietzuch. "Network-aware stream query processing in mobile ad-hoc networks." In: *Proceedings of IEEE Military Communications Conference (MILCOM)*. 2015, pp. 1335–1340 (cit. on pp. 5, 19, 23).

[39] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. "Operator Placement for In-Network Stream Query Processing." In: *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. 2005, pp. 250–258 (cit. on pp. 5, 19, 89).

[40] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. "Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks." In: *Proceedings of the 19th International Conference on Computer Communications and Networks (ICCCN)*. 2010, pp. 1–6 (cit. on pp. 5, 17, 19, 23, 57, 89–91, 94, 99).

[41] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. "R-Storm: Resource-Aware Scheduling in Storm." In: *Proceedings of the 16th Annual Middleware Conference (Middleware)*. 2015, pp. 149–161 (cit. on pp. 5, 19, 20, 24).

[42] Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, and Yves Robert. "Resource Allocation for Multiple Concurrent In-network Stream-Processing Applications." In: *Parallel Processing Workshops (Euro-Par)*. Springer Berlin Heidelberg, 2010, pp. 81–90 (cit. on pp. 5, 18, 19).

[43] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. "Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System." In: *Proceedings of the Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I (OTM)*. 2006, pp. 54–71 (cit. on pp. 5, 19, 23, 62).

[44] Raphael Eidenbenz and Thomas Locher. "Task allocation for distributed stream processing." In: *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*. 2016, pp. 1–9 (cit. on pp. 5, 19, 23, 62).

[45] Rahul Dwarakanath, Boris Koldehofe, Yashas Bharadwaj, The An Binh Nguyen, David M. Eyers, and Ralf Steinmetz. "TrustCEP: Adopting a Trust-Based Approach for Distributed Complex Event Processing." In: *Proceedings of IEEE International Conference on Mobile Data Management (MDM)*. 2017, pp. 30–39 (cit. on pp. 5, 19, 21, 23).

[46] Matteo Nardelli, Gabriele Russo Russo, Valeria Cardellini, and Francesco Lo Presti. "A Multi-level Elasticity Framework for Distributed Data Stream Processing." In: *Proceedings of Parallel Processing Workshops (Euro-Par)*. 2019, pp. 53–64 (cit. on p. 5).

[47] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. "EmuFog: Extensible and scalable emulation of large-scale fog computing infrastructures." In: *Proceedings of IEEE Fog World Congress (FWC)*. 2017, pp. 1–6 (cit. on p. 5).

[48] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. "Mobile Fog: A Programming Model for Large-Scale Applications on the Internet of Things." In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Mobile Cloud Computing (MCC)*. 2013, pp. 15–20 (cit. on pp. 5, 7, 34).

[49] Takashi Takenaka, Hiroaki Inoue, Takeo Hosomi, and Yuichi Nakamura. "FPGA-accelerated complex event processing." In: *Proceedings of Symposium on VLSI Circuits (VLSI Circuits)*. 2015, pp. C126–C127 (cit. on p. 5).

[50] Gianpaolo Cugola and Alessandro Margara. “Low latency complex event processing on parallel hardware.” In: *Journal of Parallel and Distributed Computing* 72.2 (2012), pp. 205–218 (cit. on p. 5).

[51] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. “P4CEP: Towards In-Network Complex Event Processing.” In: *Proceedings of the Morning Workshop on In-Network Computing (SIGCOMM)*. 2018, pp. 33–38 (cit. on pp. 5, 25, 114).

[52] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Spark: Cluster Computing with Working Sets.” In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. 2010, p. 10 (cit. on pp. 6, 80, 88, 155).

[53] Google. *Apache Beam: An advanced unified prgoramming model*. `https://beam.apache.org/` (cit. on pp. 6, 22).

[54] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. “Apache Flink: Stream and Batch Processing in a Single Engine.” In: *IEEE Data Engineering Bulletin* 38.4 (2015), pp. 28–38 (cit. on pp. 7, 10, 20, 24, 52, 88, 113, 140, 143, 156, 158, 161, 171).

[55] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. “GENI: A federated testbed for innovative network experiments.” In: *Computer Networks* 61 (2014), pp. 5–23 (cit. on pp. 9, 95).

[56] CloudLab. *CloudLab network infrastructure*. `https://www.cloudlab.us/` (cit. on pp. 9, 95).

[57] Björn Richerzhagen, Boris Koldehofe, and Ralf Steinmetz. “Immense Dynamism.” In: *German Research* 37 (2015), pp. 24–27 (cit. on pp. 9, 22, 61, 68, 95).

[58] Rehmat Ullah, Syed Hassan Ahmed, and Byung-Seo Kim. “Information-Centric Networking With Edge Computing for IoT: Research Challenges and Future Directions.” In: *IEEE Access* 6 (2018), pp. 73465–73488 (cit. on p. 9).

[59] Nikos Fotiou, A. Vasilos Siris, George Xylomenos, C. George Polyzos, Konstantinos Katsaros, and George Petropoulos. “Edge-ICN and its application to the Internet of Things.” In: *Proceedings of IFIP Networking Conference and Workshops (IFIP)*. 2017, pp. 1–6 (cit. on p. 9).

[60] *INetCEP Github*. https://github.com/luthramanisha/INetCEP. 2020 (cit. on p. 10).

[61] Christian Tschudin and Manolis Sifalakis. “Named Functions and Cached Computations.” In: *Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC)*. 2014, pp. 851–857 (cit. on pp. 10, 28–30, 114, 115, 120, 121, 139).

[62] *CCN-lite github.* https://github.com/cn-uofbasel/ccn-lite. 2019 (cit. on pp. 10, 140).

[63] *2014 DEBS grand challenge*. https://debs.org/grand-challenges/2014/. 2014 (cit. on pp. 10, 142).

[64] Flor Álvarez, Lars Almon, Patrick Lieser, Tobias Meuser, Yannick Dylla, Björn Richerzhagen, Matthias Hollick, and Ralf Steinmetz. “Conducting a Large-scale Field Test of a Smartphone-based Communication Network for Emergency Response.” In: *Proceedings of the 13th Workshop on Challenged Networks*. 2018, pp. 3–10 (cit. on pp. 10, 141).

[65] ULB Machine Learning Group. *Credit Card Fraud Detection: Anonymized credit card transactions labeled as fraudulent or genuine*. https://www.kaggle.com/mlg-ulb/creditcardfraud. 2017 (cit. on pp. 11, 172).

[66] Ralf Kundel, Christoph Gärtner, Manisha Luthra, Sukanya Bhowmik, and Boris Koldehofe. "Flexible Content-based Publish/Subscribe over Programmable Data Planes." In: *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*. 2020, pp. 1–5 (cit. on pp. 11, 25, 114, 190).

[67] Martin Pfannemüller, Markus Weckesser, Roland Kluge, Janick Edinger, Manisha Luthra, Robin Klose, Christian Becker, and Andy Schürr. "CoalaViz: Supporting Traceability of Adaptation Decisions in Pervasive Communication Systems." In: *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 590–595 (cit. on p. 11).

[68] Martin Pfannemüller, Janick Edinger, Markus Weckesser, Roland Kluge, Manisha Luthra, Robin Klose, Christian Becker, and Andy Schürr. "Demo: Visualizing Adaptation Decisions in Pervasive Communication Systems." In: *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 335–337 (cit. on p. 11).

[69] Pratyush Agnihotri, Manisha Luthra, and Sascha Peters. "UrbanPulse: Adaptable Middleware to Offer City and User Centric Smart City Solution." In: *Proceedings of the 20th International Middleware Conference Demos and Posters (Middleware)*. 2019, pp. 29–30 (cit. on p. 11).

[70] Yassin Alkhalili, Manisha Luthra, Amr Rizk, and Boris Koldehofe. "3-D Urban Objects Detection and Classification From Point Clouds." In: *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems (DEBS)*. 2019, pp. 209–213 (cit. on p. 11).

[71] Stefan Wilk, Manisha Luthra, and Wolfgang Effelsberg. "One Sensor is Not Enough: Adapting and Fusing Sensors for the Quality Assessment of User Generated Video." In: *Proceedings of the 24th ACM International Conference on Multimedia (ACM MM)*. 2016, pp. 626–630 (cit. on p. 11).

[72] Emmanouil Vasilomanolakis, Jörg Daubert, Manisha Luthra, Vangelis Gazis, Alex Wiesmaier, and Panayotis Kikiras. "On the Security and Privacy of Internet of Things Architectures and Systems." In: *Proceedings of the International Workshop on Secure Internet of Things (SIoT)*. 2015, pp. 49–57 (cit. on p. 11).

[73] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. "Emu: Elastic Multi-stage Deep Learning Inference with SLA Guarantees." In: *Submitted*. 2021, pp. 1–12 (cit. on pp. 11, 190).

[74] Opher Etzion and Peter Niblett. *Event Processing in Action*. 1st. Manning Publications Co., 2010 (cit. on p. 14).

[75] Annika Hinze, Kai Sachs, and Alejandro Buchmann. "Event-Based Applications and Enabling Technologies." In: *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2009, pp. 1–15 (cit. on p. 15).

[76] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@twitter." In: *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. 2014, pp. 147–156 (cit. on pp. 16, 20, 22–25, 62).

[77] Björn Schilling, Boris Koldehofe, and Kurt Rothermel. "Efficient and distributed rule placement in heavy constraint-driven event systems." In: *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications (HPCC)*. 2011, pp. 355–364 (cit. on pp. 17, 18, 23).

[78] Ioana Stanoi, George Mihaila, Themis Palpanas, and Christian Lang. "WhiteWater: Distributed Processing of Fast Streams." In: *IEEE Transactions on Knowledge and Data Engineering* 19.9 (2007), pp. 1214–1226 (cit. on pp. 18, 19).

[79] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. "I-Scheduler: Iterative scheduling for distributed stream processing systems." In: *Future Generation Computer Systems* 117 (2021), pp. 219–233 (cit. on pp. 18, 19, 23, 88).

[80] Fabrice Starks and Thomas Peter Plagemann. "Operator placement for efficient distributed complex event processing in manets." In: *Proceedings of IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2015, pp. 83–90 (cit. on pp. 18, 19, 23, 56, 57, 62, 66, 91, 100).

[81] Ralf Steinmetz and Klara Nahrstedt. "Quality of Service." In: *Multimedia Systems*. Springer Berlin Heidelberg, 2004, pp. 9–76 (cit. on pp. 18, 58).

[82] Geetika T. Lakshmanan, Ying Li, and Rob Strom. "Placement strategies for internet-scale data stream systems." In: *IEEE Internet Computing* 12.6 (2008), pp. 50–60 (cit. on pp. 18, 41, 57, 88, 89).

[83] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. "T-storm: Traffic-aware online scheduling in storm." In: *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*. 2014, pp. 535–544 (cit. on pp. 19, 24).

[84] Felipe Rodrigo de Souza, Marcos Dias de Assunçao, Eddy Caron, and Alexandre da Silva Veith. "An Optimal Model for Optimizing the Placement and Parallelism of Data Stream Processing Applications on Cloud-Edge Computing." In: *Proceedings of IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2020, pp. 59–66 (cit. on pp. 19, 23, 62, 88).

[85] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. "Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems." In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2014, pp. 13–22 (cit. on pp. 19, 21, 23, 62).

[86] Tiziano De Matteis and Gabriele Mencagli. "Proactive elasticity and energy awareness in data stream processing." In: *Journal of Systems and Software* 127 (2017), pp. 302–319 (cit. on pp. 20, 22, 23, 62).

[87] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. "Quality-aware Runtime Adaptation in Complex Event Processing." In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2017, pp. 140–151 (cit. on pp. 20, 21, 23, 24, 53, 62).

[88] Gabriele Russo Russo, Matteo Nardelli, Valeria Cardellini, and Francesco Lo Presti. "Multi-level elasticity for wide-area Data Streaming Systems: A reinforcement learning approach." In: *Algorithms* 11.9 (2018), pp. 1–27 (cit. on p. 20).

[89] Albert Jonathan, Abhishek Chandra, and Jon Weissman. "WASP: Wide-Area Adaptive Stream Processing." In: *Proceedings of the 21st International Middleware Conference (Middleware)*. 2020, pp. 221–235 (cit. on pp. 20, 22, 23, 62).

[90] Adriano Vogel, Dalvan Griebler, and Gustavo Luiz Fernandes. “Providing high-level self-adaptive abstractions for stream parallelism on multicores.” In: *Journal of Software: Practice and Experience* (2021). Early View (cit. on p. 20).

[91] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. “Apache spark: a unified engine for big data processing.” In: *Communications of the ACM* 59.11 (2016), pp. 56–65 (cit. on pp. 20, 24).

[92] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. “The Stratosphere Platform for Big Data Analytics.” In: *The VLDB Journal* 23.6 (2014), pp. 939–964 (cit. on pp. 20, 24, 57).

[93] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. “Auto-scaling techniques for elastic data stream processing.” In: *Proceedings of the 30th International Conference on Data Engineering Workshops (ICDEW)*. 2014, pp. 296–302 (cit. on p. 21).

[94] Thomas Heinze, Mariam Zia, Robert Krahn, Zbigniew Jerzak, and Christof Fetzer. “An Adaptive Replication Scheme for Elastic Data Stream Processing Systems.” In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2015, pp. 150–161 (cit. on p. 21).

[95] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. “Online Parameter Optimization for Elastic Data Stream Processing.” In: *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*. 2015, pp. 276–287 (cit. on p. 21).

[96] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments.” In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592 (cit. on p. 21).

[97] André Martin, Andrey Brito, and Christof Fetzer. “StreamMine3G: Elastic and Fault Tolerant Large Scale Stream Processing.” In: *Encyclopedia of Big Data Technologies*. Springer International Publishing, 2018, pp. 1–10 (cit. on p. 21).

[98] Henriette Röger and Ruben Mayer. “A Comprehensive Survey on Parallelization and Elasticity in Stream Processing.” In: *ACM Computing Survey* 52.2 (2019) (cit. on pp. 21, 22).

[99] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. “Distributed data stream processing and edge computing: A survey on resource elasticity and future directions.” In: *Journal of Network and Computer Applications* 103 (2018), pp. 1–17 (cit. on p. 21).

[100] Eugene Wu, Yanlei Diao, and Shariq Rizvi. “High-Performance Complex Event Processing over Streams.” In: *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. 2006, pp. 407–418 (cit. on pp. 22, 113).

[101] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. “Towards Expressive Publish/Subscribe Systems.” In: *Proceedings of Advances in Database Technology (EDBT)*. 2006, pp. 627–644 (cit. on p. 22).

[102] Arvind Arasu, Shivnath Babu, and Jennifer Widom. “The CQL Continuous Query Language: Semantic Foundations and Query Execution.” In: *The VLDB Journal* 15.2 (2006), pp. 121–142 (cit. on pp. 22, 25).

[103] Gianpaolo Cugola and Alessandro Margara. "TESLA: A Formally Defined Event Specification Language." In: *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2010, pp. 50–61 (cit. on pp. 22, 24, 62).

[104] DFG SFB MAKI. *MAKI – Multi Mechanism Adaptation for the Future Internet*. `https://www.maki.tu-darmstadt.de/` (cit. on pp. 22, 68).

[105] Alexander Frömmgen, Björn Richerzhagen, Rückert Julius, David Hausheer, Ralf Steinmetz, and Alejandro Buchmann. "Towards the Description and Execution of Transitions in Networked Systems." In: *Proceedings of Intelligent Mechanisms for Network Configuration and Security (IFIP)*. 2015, pp. 17–29 (cit. on pp. 22, 23, 68, 79).

[106] Bastian Alt, Markus Weckesser, Christian Becker, Matthias Hollick, Sounak Kar, Anja Klein, Robin Klose, Roland Kluge, Heinz Koeppl, Boris Koldehofe, Wasiur R. Khudabukhsh, Manisha Luthra, Mahdi Mousavi, Max Mühlhäuser, Martin Pfannemüller, Amr Rizk, Andy Schürr, and Ralf Steinmetz. "Transitions: A Protocol-Independent View of the Future Internet." In: *Proceedings of the IEEE* 107.4 (2019), pp. 835–846 (cit. on pp. 22, 61, 68).

[107] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003), pp. 41–50 (cit. on pp. 22, 73).

[108] Matthias Wichtlhuber, Bjoern Richerzhagen, Julius Rueckert, and David Hausheer. "TRANSIT: Supporting transitions in Peer-to-Peer live video streaming." In: *Proceedings of IFIP Networking Conference (IFIP)*. 2014, pp. 1–9 (cit. on pp. 22, 23, 61).

[109] Stefan Wilk, Roger Zimmermann, and Wolfgang Effelsberg. "Leveraging Transitions for the Upload of User-Generated Mobile Video." In: *Proceedings of the 8th International Workshop on Mobile Video (MoViD)*. 2016, pp. 1–6 (cit. on pp. 22–24, 61).

[110] Björn Richerzhagen, Nils Richerzhagen, Julian Zobel, Sophie Schönherr, Boris Koldehofe, and Ralf Steinmetz. "Seamless Transitions between Filter Schemes for Location-Based Mobile Applications." In: *Proceedings of the 41st Conference on Local Computer Networks (LCN)*. 2016, pp. 348–356 (cit. on pp. 23, 24, 61).

[111] Björn Richerzhagen, Marc Schiller, Max Lehn, Denis Lapiner, and Ralf Steinmetz. "Transition-enabled event dissemination for pervasive mobile multiplayer games." In: *Proceedings of the 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. 2015, pp. 1–3 (cit. on pp. 23, 24, 61).

[112] Björn Richerzhagen, Stefan Wilk, Julius Rückert, Denny Stohr, and Wolfgang Effelsberg. "Transitions in Live Video Streaming Services." In: *Proceedings of the Workshop on Design, Quality and Deployment of Adaptive Video Streaming (VideoNext)*. 2014, pp. 37–38 (cit. on pp. 23, 61).

[113] Alexander Frömmgen, Stefan Haas, Michael Stein, Robert Rehner, Alejandro Buchmann, and Max Mühlhäuser. "Always the Best: Executing Transitions Between Search Overlays." In: *Proceedings of the European Conference on Software Architecture Workshops (ECSA)*. 2015, pp. 1–4 (cit. on pp. 23, 61).

[114] Nils Richerzhagen, Patrick Lieser, Björn Richerzhagen, Boris Koldehofe, Ioannis Stavrakakis, and Ralf Steinmetz. "Change as Chance: Transition-enabled Monitoring for Dynamic Networks and Environments." In: *Proceedings of 14th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*. 2018, pp. 51–58 (cit. on pp. 23, 24, 61).

[115] Julius Rückert, Björn Richerzhagen, Eduardo Lidanski, Ralf Steinmetz, and David Hausheer. "TOPT: Supporting flash crowd events in hybrid overlay-based live streaming." In: *Proceedings of IFIP Networking Conference (IFIP)*. 2015, pp. 1–9 (cit. on p. 23).

[116] Stefan Wilk. “Quality-aware Content Adaptation in Digital Video Streaming.” PhD thesis. Technical University of Darmstadt, 2016 (cit. on pp. 23, 24).

[117] Björn Richerzhagen. “Mechanism Transitions in Publish/Subscribe Systems.” PhD thesis. Technical University of Darmstadt, 2017 (cit. on pp. 23, 24).

[118] Alexander Frömmgen. “Programming Models and Extensive Evaluation Support for MPTCP Scheduling, Adaptation Decisions, and Dash Video Streaming.” PhD thesis. Technical University of Darmstadt, 2018 (cit. on p. 23).

[119] Nils Richerzhagen. “Transition in Monitoring and Network Offloading-Handling Dynamic Mobile Applications and Environments.” PhD thesis. Technical University of Darmstadt, 2019 (cit. on pp. 23, 24).

[120] Rhaban Hark. “Monitoring Federated Softwarized Networks.” PhD thesis. Technical University of Darmstadt, 2019 (cit. on pp. 23, 24).

[121] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. “Optimal Operator Replication and Placement for Distributed Stream Processing Systems.” In: *SIGMETRICS Performance Evaluation Review* 44.4 (2017), pp. 11–22 (cit. on pp. 23, 88, 89).

[122] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. “Rollback-recovery without checkpoints in distributed event processing systems.” In: *Proceedings of the 7th ACM international conference on Distributed event-based systems (DEBS)*. 2013, pp. 27–38 (cit. on p. 23).

[123] Gerald G. Koch, Boris Koldehofe, and Kurt Rothermel. “Cordies: expressive event correlation in distributed systems.” In: *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2010, pp. 26–37 (cit. on p. 23).

[124] Michael Stein, Alexander Froemmgen, Roland Kluge, Frank Loeffler, Andy Schuerr, Alejandro Buchmann, and Max Muehlhaeuser. “TARL: Modeling Topology Adaptations for Networking Applications.” In: *Proceedings of IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2016, pp. 57–63 (cit. on p. 24).

[125] Yong Yao and Johannes Gehrke. “The Cougar Approach to In-Network Query Processing in Sensor Networks.” In: *ACM Sigmod record* 31.3 (2002), pp. 9–18 (cit. on p. 25).

[126] Sharma Chakravarthy and Deepak Mishra. “Snoop: An expressive event specification language for active databases.” In: *Data and Knowledge Engineering* 14.1 (1994), pp. 1–26 (cit. on pp. 25, 52).

[127] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. “Adaptive Query Processing.” In: *Foundations and Trends in Databases* 1.1 (2007), pp. 1–140 (cit. on p. 25).

[128] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. “Software-Defined Networking: A Comprehensive Survey.” In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76 (cit. on p. 25).

[129] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. “Networking Named Content.” In: *Proceedings of the 5th ACM International Conference on Emerging Networking Experiments and Technologies (CONEXT)*. 2009, pp. 1–12 (cit. on pp. 25, 27, 53, 117, 121, 134).

[130] Sukanya Bhowmik, Mohammad Adnan Tariq, Boris Koldehofe, Frank Dürr, Thomas Kohler, and Kurt Rothermel. “High Performance Publish/Subscribe Middleware in Software-Defined Networks.” In: *IEEE/ACM Transactions on Networking* 25.3 (2017), pp. 1501–1516 (cit. on p. 25).

[131] Alessandra Fais, Giuseppe Lettieri, Gregorio Procissi, Stefano Giordano, and Francesco Oppedisano. “Data Stream Processing for Packet-Level Analytics.” In: *Sensors* 21.5 (2021) (cit. on pp. 25, 114).

[132] Christian Tschudin and Manolis Sifalakis. “Named Functions for Media Delivery Orchestration.” In: *Proceedings of the 20th International Packet Video Workshop (PV)*. 2013, pp. 1–8 (cit. on pp. 25, 29, 30, 117, 134, 139).

[133] Alejandro Buchmann, Christof Bornhövd, Mariano Cilia, Ludger Fiege, Felix Gärtner, Christoph Liebig, Matthias Meixner, and Gero Mühl. “DREAM: Distributed Reliable Event-Based Application Management.” In: *Web Dynamics: Adapting to Change in Content, Size, Topology and Use*. Springer Berlin Heidelberg, 2004, pp. 319–349 (cit. on p. 26).

[134] Sebastian Frischbier. “Runtime Support for Quality of Information Requirements in Event-based Systems.” PhD thesis. Technical University of Darmstadt, 2016 (cit. on pp. 27, 31, 32).

[135] Syed Hassan Ahmed and Dongkyun Kim. “Named data networking-based smart home.” In: *ICT Express* 2.3 (2016). Special Issue on ICT Convergence in the Internet of Things (IoT), pp. 130–134 (cit. on pp. 27, 29–31, 117, 118, 121).

[136] Alcatel-Lucent Bell Labs. *Content-Centric Networking Packet Header Format*. `https://tools.ietf.org/html/draft-ccn-packet-header-00`. Internet Research Task Force (IRTF), Internet Draft, draft-ccn-packet-header-00. 2015 (cit. on p. 28).

[137] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, Kc Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. “Named data networking.” In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 66–73 (cit. on pp. 28–30, 134).

[138] Jean-Louis Krivine. “A Call-by-Name Lambda-Calculus Machine.” In: *Higher Order Symbolic Computation* 20.3 (2007), pp. 199–207 (cit. on p. 28).

[139] Manisha Luthra, Johannes Pfannmüller, Boris Koldehofe, Jonas Höchst, Artur Sterz, Rhaban Hark, and Bernd Freisleben. “Efficient Complex Event Processing in Information-centric Networking at the Edge (under submission).” In: (2021), pp. 1–17. URL: `https://arxiv.org/pdf/2012.05070.pdf` (cit. on pp. 29, 58, 115, 123, 125, 126, 130, 136, 137, 140, 142, 145, 146, 149, 151, 153).

[140] Manolis Sifalakis, Basil Kohler, Christopher Scherb, and Christian Tschudin. “An Information Centric Network for Computing the Distribution of Computations.” In: *Proceedings of the 1st ACM Conference on Information-Centric Networking (ICN)*. 2014, pp. 137–146 (cit. on pp. 29, 30, 114).

[141] Ioannis Król Michałand Psaras. “NFaaS: Named Function As a Service.” In: *Proceedings of the 4th ACM Conference on Information-Centric Networking (ICN)*. 2017, pp. 134–144 (cit. on pp. 29, 30, 114).

[142] Wentao Shang, Adeola Bannis, Teng Liang, Zhehao Wang, Yingdi Yu, Alexander Afanasyez, Jeff Thompson, Jeff Burke, Beichuan Zhang, and Lixia Zhang. “Named Data Networking of Things.” In: *Proceedings of the 1st IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2016, pp. 117–128 (cit. on pp. 29, 30, 114, 117, 121, 140, 143, 145).

[143] Christopher Scherb, Claudio Marxer, Urs Schnurrenberger, and Christian Tschudin. "In-network live stream processing with named functions." In: *Proceedings of the IFIP Networking Conference and Workshops (IFIP Networking)*. 2017, pp. 1–6 (cit. on p. 29).

[144] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. "A data-oriented (and beyond) network architecture." In: *SIGCOMM Computer Communication Review* 37.4 (2007), pp. 181–192 (cit. on pp. 29, 30).

[145] Jiachen Chen, Mayutan Arumaithurai, Lei Jiao, Xiaoming Fu, and K. K. Ramakrishnan. "COPSS: An Efficient Content Oriented Publish/Subscribe System." In: *ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. 2011, pp. 99–110 (cit. on pp. 29, 30, 118).

[146] Cenk Guendogan, Peter Kietzmann, Thomas C. Schmidt, and Matthias Waehlisch. "HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things." In: *Proceedings of the 43rd IEEE Conference on Local Computer Networks (LCN)*. 2018, pp. 331–334 (cit. on pp. 29–31, 119, 121).

[147] Nicola Blefari Melazzi. "CONVERGENCE: Extending the Media Concept to Include Representations of Real World Objects." In: *The Internet of Things* (2010), pp. 129–140 (cit. on p. 30).

[148] *FP7 Green ICN project*. http://www.greenicn.org/. 2013 (cit. on pp. 30, 31).

[149] Antonio Carzaniga, Michele Papalini, and Alexander L. Wolf. "Content-based Publish/Subscribe Networking and Information-centric Networking." In: *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking (ICN)*. 2011, pp. 56–61 (cit. on p. 30).

[150] Antonio Carzaniga, Koorosh Khazaei, Michele Papalini, and Alexander L. Wolf. "Is Information-centric Multi-tree Routing Feasible?" In: *SIGCOMM Computer Communication Review* 43.4 (2013), pp. 3–8 (cit. on p. 30).

[151] Ouassim Karrakchou, Nancy Samaan, and Ahmed Karmouch. "ENDN: An Enhanced NDN Architecture with a P4-ProgrammabIe Data Plane." In: *Proceedings of the 7th ACM Conference on Information-Centric Networking*. 2020, pp. 1–11 (cit. on pp. 30, 31, 114).

[152] Michele Papalini, Antonio Carzaniga, Koorosh Khazaei, and Alexander L. Wolf. "Scalable Routing for Tag-Based Information-Centric Networking." In: *Proceedings of the 1st ACM Conference on Information-Centric Networking (ICN)*. 2014, pp. 17–26 (cit. on p. 31).

[153] Lizhe Wang, Gregor von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. "Cloud Computing: a Perspective Study." In: *New Generation Computing* 28 (2010), pp. 137–146 (cit. on p. 31).

[154] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing. Technical Report*. Tech. rep. National Institute of Standards and Technology, Information Technology Laboratory, 2011 (cit. on p. 32).

[155] S. Kachele, C. Spann, F. J. Hauck, and J. Domaschka. "Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking." In: *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC)*. 2013, pp. 75–82 (cit. on p. 32).

[156] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. "Serverless Computing: Current Trends and Open Problems." In: *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20 (cit. on pp. 33, 34).

[157] Amazon. *Amazon fog location service: AWS Cloud Front*. https://aws.amazon.com/cloudfront/ (cit. on p. 33).

[158] Google. *Google fog location service: Google Cloud CDN*. https://cloud.google.com/cdn. 2020 (cit. on p. 33).

[159] Enrique Saurez, Bharath Balasubramanian, Richard Schlichting, Brendan Tschaen, Zhe Huang, Shankaranarayanan Puzhavakath Narayanan, and Umakishore Ramachandran. "METRIC: A Middleware for Entry Transactional Database Clustering at the Edge." In: *Proceedings of the 3rd Workshop on Middleware for Edge Clouds and Cloudlets*. 2018, pp. 2–7 (cit. on p. 33).

[160] *Cisco: Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*. https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf (cit. on p. 33).

[161] *OpenFog Consortium*. https://www.openfogconsortium.org/what-we-do/ (cit. on p. 33).

[162] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "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 (cit. on p. 33).

[163] M. Amir Rahmani, L.J.-S. Pasi Preden, and Axel Jantsch. "Fog Computing in the Internet of Things." In: *Intelligence at the Edge*. Springer International Publishing, 2018 (cit. on p. 33).

[164] Amazon Web Services. *AWS Lambda*. https://aws.amazon.com/lambda/. 2019 (cit. on pp. 34, 36, 38–40, 156, 170).

[165] Google. *Google Cloud Functions*. https://cloud.google.com/functions/. 2019 (cit. on pp. 34, 39, 40, 156).

[166] Microsoft. *Microsoft Azure Functions*. https://azure.microsoft.com/en-us/services/functions/. 2019 (cit. on pp. 34, 39, 40).

[167] IBM. *IBM OpenWhisk*. https://developer.ibm.com/open/projects/openwhisk/. 2019 (cit. on pp. 34, 39–41).

[168] Joseph M. Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. "Serverless Computing: One Step Forward, Two Steps Back." In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 2020 (cit. on p. 34).

[169] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-Performance Complex Event Processing over Streams." In: *Proceedings of ACM International Conference on Management of Data (SIGMOD)*. 2006, pp. 407–418 (cit. on p. 35).

[170] Sebastian Hennig. "Virtualizing CEP Execution Environment Using Serverless Paradigm for Latency Sensitive Applications." KOM-B-0649. Bachelor Thesis. Technical University of Darmstadt, 2019 (cit. on p. 37).

[171] *Netflix: A Video Streaming Platform*. https://www.netflix.com/ (cit. on p. 37).

[172] Alex Ellis. *OpenFaaS*. https://www.openfaas.com/. 2019 (cit. on p. 38).

[173] The Linux Foundation. *Kubernetes*. https://kubernetes.io/. 2019 (cit. on p. 38).

[174] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. “A serverless real-time data analytics platform for edge computing.” In: *IEEE Internet Computing* 21.4 (2017), pp. 64–71 (cit. on pp. 38, 157).

[175] Kubeless. *Kubeless*. https://kubeless.io/. 2019 (cit. on pp. 38, 39).

[176] IBM. *IBM Cloud Functions*. https://cloud.ibm.com/functions/ (cit. on pp. 39, 40).

[177] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. “Occupy the Cloud: Distributed Computing for the 99%.” In: *Proceedings of the 2017 Symposium on Cloud Computing (SOCC)*. Association for Computing Machinery, 2017, pp. 445–451 (cit. on p. 39).

[178] Amazon Web Services. *Amazon S3*. https://aws.amazon.com/de/s3/. 2020 (cit. on p. 39).

[179] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. “Comparison of FaaS Orchestration Systems.” In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 148–153 (cit. on p. 39).

[180] Amazon Web Services. *Amazon Step Functions*. https://aws.amazon.com/de/step-functions/ (cit. on p. 39).

[181] Austin Aske and Xinghui Zhao. “Supporting multi-provider serverless computing on the edge.” In: *Proceedings of the 47th International Conference on Parallel Processing Companion (ICPP)*. 2018, 20 pages (cit. on pp. 41, 42, 157).

[182] Amazon Web Services. *AWS IoT Greengrass*. https://aws.amazon.com/greengrass/ (cit. on pp. 40–42).

[183] Microsoft Azure. *Azure IoT Edge*. https://azure.microsoft.com/en-us/services/iot-edge/ (cit. on pp. 40, 41).

[184] Amazon Web Services. *AWS Kinesis*. https://aws.amazon.com/kinesis/. 2019 (cit. on p. 41).

[185] Google IoT. *Cloud IoT Core*. https://cloud.google.com/iot-core (cit. on p. 41).

[186] Alex Glikson. *Serverless Edge-to-Cloud computing: the open source way*. https://medium.com/openwhisk/serverless-edge-to-cloud-computing-the-open-source-way-28ea33f60bf6. 2017 (cit. on p. 41).

[187] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. *Vision and Challenges for Realising the Internet of Things*. European Commission. Directorate-General for the Information Society and Media, 2010 (cit. on p. 43).

[188] Forrester Research J. Belissent. *Getting clever about smart cities: new opportunities require new business models*. 2010 (cit. on p. 43).

[189] Donato Toppeta. *Smart Cities, not only new Research Papers but also exciting forecast!* http://ict4green.wordpress.com/2012/02/11/smart-cities-not-only-newresearch-papers-but-also-exiting-forecast/. 2012 (cit. on p. 45).

[190] Digitales Hessen. *Digital Stadt Darmstadt*. https://www.digitalstadt-darmstadt.de/ (cit. on p. 45).

[191] “IEEE Guide for Wireless Access in Vehicular Environments (WAVE) - Architecture.” In: *IEEE Std 1609.0-2013* (2014), pp. 1–78 (cit. on pp. 46, 53, 64).

[192] Kees Wevers and Meng Lu. “V2X Communication for ITS - from IEEE 802.11p Towards 5G.” In: *IEEE 5G Tech Focus* 1.2 (2017), pp. 5–10 (cit. on p. 47).

[193] M. Amadeo, C. Campolo, and A. Molinaro. “Information-centric networking for connected vehicles: a survey and future perspectives.” In: *IEEE Communications Magazine* 54.2 (2016), pp. 98–104 (cit. on p. 47).

[194] Sara Castellanos. *The Wall Street Journal: Visa to Test Advanced AI to Prevent Fraud*. https://www.wsj.com/articles/visa-to-test-advanced-ai-to-prevent-fraud-11565205158. 2019 (cit. on p. 48).

[195] Software AG. *Apama CEP*. http://www.apamacommunity.com/. 2018 (cit. on p. 49).

[196] Manisha Luthra, Boris Koldehofe, Niels Danger, Pascal Weisenburger, Guido Salvaneschi, and Ioannis Stavrakakis. “TCEP: Transitions in Operator Placement to Adapt to Dynamic Network Environments.” In: *Journal of Computer and Systems Sciences (JCSS), Special Issue on Algorithmic Theory of Dynamic Networks and its Applications (accepted for publication)* (2020), pp. 1–76. URL: https://luthramanisha.github.io/TCEP/ (cit. on pp. 51, 63, 66, 72, 81, 82, 85, 86, 158).

[197] David L Mills. “Internet time synchronization: the network time protocol.” In: *IEEE Transactions on communications* 39.10 (1991), pp. 1482–1493 (cit. on pp. 52, 86).

[198] Sérgio Esteves, Gianmarco De Francisci Morales, Rodrigo Rodrigues, Marco Serafini, and Luís Veiga. *Aion: Better Late than Never in Event-Time Streams*. 2020. eprint: 2003.03604 (cs.DC) (cit. on p. 52).

[199] Scarlet Schwiderski-Grosche and Ken Moody. “The SpaTeC Composite Event Language for Spatio-Temporal Reasoning in Mobile Systems.” In: *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2009, pp. 1–12 (cit. on p. 52).

[200] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. “MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing.” In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. DEBS ’13. 2013, pp. 183–194 (cit. on p. 52).

[201] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. “SPECTRE: Supporting Consumption Policies in Window-based Parallel Complex Event Processing.” In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 161–173 (cit. on p. 55).

[202] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. “Vivaldi.” In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), p. 15 (cit. on pp. 56, 66, 89).

[203] Zheng Chu, Jiong Yu, and Askar Hamdull. “Maximum Sustainable Throughput Evaluation Using an Adaptive Method for Stream Processing Platforms.” In: *IEEE Access* 8 (2020), pp. 40977–40988 (cit. on p. 58).

[204] Ying Loong Lee, Jonathan Loo, and Teong Chee Chuah. “Chapter 24 - Modeling and performance evaluation of resource allocation for LTE femtocell networks.” In: *Modeling and Simulation of Computer Networks and Systems*. Ed. by Mohammad S. Obaidat, Petros Nicopolitidis, and Faouzi Zarai. Morgan Kaufmann, 2015, pp. 683–716 (cit. on p. 58).

[205] Manisha Luthra and Boris Koldehofe. “ProgCEP: A Programming Model for Complex Event Processing over Fog Infrastructure.” In: *Proceedings of the 2nd International Workshop on Distributed Fog Services Design (DFSD@Middleware)*. 2019, pp. 7–12 (cit. on pp. 63, 89, 91).

[206] Manisha Luthra. “Adapting to Dynamic User Environments in Complex Event Processing System Using Transitions.” In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS), Doctorial Symposium*. 2018, pp. 274–277 (cit. on p. 63).

[207] Manisha Luthra, Sebastian Hennig, and Boris Koldehofe. “Understanding the Behavior of Operator Placement Mechanisms on Large-Scale Networks.” In: *Proceedings of the 19th ACM/IFIP Middleware Conference: Posters and Demos (Middleware)*. 2018, pp. 19–20 (cit. on p. 63).

[208] Pascal Weisenburger, Manisha Luthra, Boris Koldehofe, and Guido Salvaneschi. “Quality-aware Runtime Adaptation in Complex Event Processing.” In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017, pp. 140–151 (cit. on pp. 64, 91–94).

[209] Shanzhi Chen, Jinling Hu, Yan Shi, Ying Peng, Jiayi Fang, Rui Zhao, and Li Zhao. “Vehicle-to-Everything (v2x) Services Supported by LTE-Based Systems and 5G.” In: *IEEE Communications Standards Magazine* 1.2 (2017), pp. 70–76 (cit. on p. 64).

[210] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer Publishing Company, Incorporated, 2011 (cit. on pp. 69, 129, 165).

[211] Niels Danger. “Specification of Transitions in Complex Event Processing Systems using Context Feature Models.” KOM-B-0609. Bachelor Thesis (*In collaboration with Real Time Systems Lab, Technical University of Darmstadt*). Technical University of Darmstadt, 2018 (cit. on p. 74).

[212] Haibo He, Sheng Chen, Kang Li, and Xin Xu. “Incremental Learning From Stream Data.” In: *IEEE Transactions on Neural Networks* 22.12 (2011), pp. 1901–1914 (cit. on p. 74).

[213] Benedikt Lins. “Optimal decisions of transitions based on online learning methods.” KOM-M-0689. Master Thesis. Technical University of Darmstadt, 2019 (cit. on p. 74).

[214] Darrell Whitley. “The GENITOR Algorithm and Selection Pressure: Why Rank-based Allocation of Reproductive Trials is Best.” In: *Proceedings of the 3rd International Conference on Genetic Algorithms (GA)*. 1989, pp. 116–121 (cit. on pp. 75, 78).

[215] Tobias Blickle and Lothar Thiele. “A Comparison of Selection Schemes Used in Evolutionary Algorithms.” In: *Evolutionary Computation* 4.4 (1996), pp. 361–394 (cit. on pp. 78, 79).

[216] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. “Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems.” In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1303–1316 (cit. on p. 80).

[217] Rahul Wermund. “Privacy-Aware and Reliable Complex Event Processing in the Internet of Things.” PhD thesis. Technical University of Darmstadt, 2018 (cit. on p. 82).

[218] Jochen Van den Bercken and Bernhard Seeger. “Query Processing Techniques for Multiversion Access Methods.” In: *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*. 1996, pp. 168–179 (cit. on p. 87).

[219] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. “Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines.” In: *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 2020, pp. 2471–2486 (cit. on p. 87).

[220] Xunyun Liu and Rajkumar Buyya. “Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions.” In: *ACM Computing Surveys* 53.3 (2020) (cit. on p. 89).

[221] *Akka*. `http://akka.io/`. 2009 (cit. on p. 94).

[222] *Docker – Community edition*. `https://www.docker.com/community-edition`. 2013 (cit. on p. 94).

[223] GENI. *Geni network infrastructure*. `https://www.geni.net/` (cit. on p. 95).

[224] Alejandro Cuadrado Torre, Marco Fiore, Claudio Casetti, Marco Gramaglia, and Maria Calderón. “Bidirectional highway traffic for network simulation.” In: *2017 IEEE Vehicular Networking Conference (VNC)*. 2017, pp. 77–80 (cit. on p. 95).

[225] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. “Reproducible Network Experiments Using Container-Based Emulation.” In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. 2012, pp. 253–264 (cit. on p. 97).

[226] *FP7 PSIRP project*. http://psirp.org/. 2008 (cit. on p. 117).

[227] *FP7 SAIL project*. http://www.sail-project.eu. 2010 (cit. on p. 117).

[228] F. Gont and S. Bellovin. *Defending against Sequence Number Attacks*. RFC 6528, Internet Engineering Task Force (IETF). 2012 (cit. on p. 118).

[229] H.T. Kung. *Traffic Management for High-Speed Networks*. The National Academies Press, 1997 (cit. on pp. 122, 132).

[230] A. Charny, D. D. Clark, and R. Jain. “Congestion control with explicit rate indication.” In: *Proceedings IEEE International Conference on Communications (ICC)*. Vol. 3. 1995, pp. 1954–1963 (cit. on p. 132).

[231] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006 (cit. on p. 135).

[232] Jeff Ahrenholz. “Comparison of CORE network emulation platforms.” In: *IEEE Milcom Military Communications Conference*. 2010, pp. 166–171 (cit. on p. 140).

[233] Alexander Frömmgen, Denny Stohr, Boris Koldehofe, and Amr Rizk. “Don’t repeat yourself: seamless execution and analysis of extensive network experiments.” In: *Proceedings of the 14th ACM International Conference on emerging Networking Experiments and Technologies (CONEXT)*. 2018, pp. 20–26 (cit. on p. 140).

[234] Gustavo Carneiro. “NS-3: Network simulator 3.” In: *UTM Lab Meeting April*. Vol. 20. 2010, pp. 4–5 (cit. on p. 140).

[235] Boris Jan Bonfils and Philippe Bonnet. “Adaptive and Decentralized Operator Placement for In-Network Query Processing.” In: *Telecommunication Systems* 26.2 (2004), pp. 389–409 (cit. on pp. 141, 152).

[236] Cornelius Köpp, Hans-Jörg von Mettenheim, and Michael H. Breitner. "Decision Analytics with Heatmap Visualization for Multi-step Ensemble Data." In: *Business & Information Systems Engineering* 6.3 (2014), pp. 131–140 (cit. on p. 141).

[237] André Martin, Rodolfo Marinho, Andrey Brito, and Christof Fetzer. "Predicting energy consumption with streammine3g." In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS)*. 2014, pp. 270–275 (cit. on p. 142).

[238] Manisha Luthra and Boris Koldehofe. "Highly Flexible Server Agnostic Complex Event Processing Operators." In: *Proceedings of the 20th ACM/IFIP International Middleware Conference: Posters and Demos (Middleware)*. 2019, pp. 11–12 (cit. on p. 157).

[239] Manisha Luthra, Sebastian Hennig, Kamran Razavi, Lin Wang, and Boris Koldehofe. "Operator as a Service: Stateful Serverless Complex Event Processing." In: *Proceedings of the IEEE International Conference on Big Data Workshops (BigData Workshop)*. 2020, pp. 1–10. URL: https://luthramanisha.github.io/CEPless (cit. on pp. 157, 171, 173, 175).

[240] Salvatore Sanfilippo. *Redis: In-memory database*. https://redis.io/. 2019 (cit. on pp. 159, 165, 172, 176).

[241] RedHat Inc. *Infinispan: In-memory database*. https://infinispan.org/. 2019 (cit. on pp. 165, 172).

[242] Jonas Traub, Philipp Marian Grulich, Alejandro Rodriguez Cuellar, Sebastian Bress, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. "Scotty: Efficient Window Aggregation for Out-of-Order Stream Processing." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1300–1303 (cit. on p. 167).

[243] Android Open Source Project. *Android k-Means Implementation*. https://android.googlesource.com/platform/frameworks/base.git/+/master/core/java/com/android/internal/ml/clustering/KMeans.java. 2017 (cit. on p. 171).

[244] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-independent Packet Processors." In: *SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95 (cit. on p. 190).

[245] Vincenzi Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. "StreamCloud: An Elastic and Scalable Data Streaming System." In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012), pp. 2351–2365 (cit. on p. 190).

# A

# Appendix

## A.1 Supplementary Material to Chapter 4

In the following, Section A.1.1 provides additional insights into the performance evaluation for the proposed programming model and the transition strategies. Next, we present additional proof for the proposed selection algorithm in the Chapter 4: Section 4.3.1.

### A.1.1 *Additional Insights on the Performance Evaluation*

#### *OP Mechanism*

In this section, we report additional insights into the performance of OP mechanisms analyzed in Section 4.4.2: Figure 23. Table 24 summarizes the mean, minimum, maximum, and quantiles (90, 95, 99%) of the metrics latency and message overhead for the different OP mechanisms. The table presents the results for Q1, Q4, and Q5 (cf. Table 10) execution using the different OP mechanisms. An observation here is that the mechanisms behaves different for distinct queries, for instance Relaxation and Global Optimal mechanisms perform well for Stream and Conjunction queries, while Producer Consumer supersedes them for Filter and Join queries. This is because the former has an objective function involving latency while the latter can achieve this performance because of its close proximity to the producers and consumers. Similar observations are for other metrics, thus there is no single mechanism that performs the best in all queries or all scenarios. Furthermore, it can be derived from Figure 23 and Table 24 that Relaxation and MDCEP mechanisms stand representatives for the metrics latency and message overhead, respectively.

| | | Latency (ms) | | | | Message Overhead (MB) | | | |
|---|---|---|---|---|---|---|---|---|---|
| OP mechanism | Query | mean | min | max | percentiles (90, 95, 99) | mean | min | max | percentiles (90, 95, 99) |
| Relaxation | Stream | **4.52** | **2** | **24** | **6, 6, 10** | 11.03 | 10.3 | 10.3 | 10.3, 10.3, 10.3 |
| | Join | **8.98** | **4** | **34** | **12, 14, 19.86** | 21.38 | 19.12 | 22.83 | 22.83, 22.83, 22.83 |
| | Congestion Detection | **9.23** | **6** | **34** | **12, 13, 19.96** | 249.52 | 145.74 | 330.16 | 329.77, 330.16, 330.16 |
| MOPA | Stream | 6.19 | 3.0 | 24 | 7, 9, 12 | 11.03 | 10.3 | 11.8 | 11.8, 11.8, 11.8 |
| | Join | 12.33 | 7 | 49 | 16, 19, 33.04 | 23.84 | 20.48 | 27.34 | 27.34, 27.34, 27.34 |
| | Congestion Detection | 19.42 | 12 | 64 | 26, 29.15, 50.32 | 246.23 | 150.6 | 364.15 | 354.23, 263.27, 364,15 |
| Global Optimal | Stream | 4.62 | 3 | 10 | 5, 6, 8.29 | 7.21 | 7.16 | 7.31 | 7.31, 7.31, 7.31 |
| | Join | 9.05 | 5 | 23 | 12.3, 14, 16.86 | 14.31 | 14.24 | 14.49 | 14.49, 14.49, 14.49 |
| | Congestion Detection | 11.17 | 6 | 51 | 15, 17, 20.68 | 130.5 | 127.45 | 132.48 | 132.48, 132.48,132.48 |
| MDCEP | Stream | 6.07 | 4 | 38 | 8, 8, 12 | **1.08** | **1.08** | **1.08** | **1.08, 1.08, 1.08** |
| | Join | 11.07 | 6 | 45 | 15, 17, 24 | **3.13** | **1.92** | **4.96** | **4.96, 4.96, 4.96** |
| | Congestion Detection | 15.68 | 10 | 53 | 21, 25.25, 41.10 | **17.97** | **6.22** | **25.04** | **23.19, 25.04, 25.04** |
| Producer Consumer | Stream | 4.82 | 3 | 14 | 6, 7, 11 | - | - | - | - |
| | Join | 7.7 | 3 | 24 | 11, 12, 17 | - | - | - | - |
| | Congestion Detection | 10.22 | 5 | 47 | 15, 15, 19 | - | - | - | - |
| Random | Stream | 5.22 | 3 | 23 | 7, 8, 12 | - | - | - | - |
| | Join | 12.41 | 6 | 36 | 17, 20, 26.03 | - | - | - | - |
| | Congestion Detection | 34.54 | 13 | 1036 | 25.6, 33, 1022.3 | - | - | - | - |

Table 24: Performance results of OP mechanisms Relaxation and MDCEP are among the best compared (marked in **bold**) to their alternative mechanisms for the conflicting metrics latency and message overhead.

*Transition Cost for the Proposed Transition Strategies*

Table 25 elaborates on the statistics of transition cost for different transitions strategies as presented in Figure 29. It summarizes the mean transition time and overhead required by the different strategies. Clearly, the SMS strategies supersede both in terms of cost in time and overhead.

| | Operator | Mean transition time (in ms) | Mean transition overhead (in MB) |
|---|---|---|---|
| MFGS Sequential | Average | 105.13 | 20 |
| | Conjunction | 607.49 | 24.74 |
| | Sequence | 287.50 | 60.12 |
| | Stream | 676.50 | 6.67 |
| MFGS Concurrent | Average | 37.06 | 0.05 |
| | Conjunction | 445.98 | 0.05 |
| | Sequence | 210.30 | 0.107 |
| | Stream | 163.78 | 0.005 |
| SMS Sequential | Average | 66.66 | 0.007 |
| | Conjunction | 219.56 | 0.009 |
| | Sequence | 243.7 | 0.023 |
| | Stream | 352.6 | 0.002 |
| SMS Concurrent | Average | **41.1** | **23.74 Bytes** |
| | Conjunction | **158.2** | **21.13 Bytes** |
| | Sequence | **158.3** | **41.86 Bytes** |
| | Stream | **85.1** | **2.41 Bytes** |

Table 25: Mean values for transition time and overhead for MFGS and SMS strategies. SMS strategies clearly supersedes in both time and overhead required to transfer the operator.

### A.1.2 *Illustration of* TCEP *System*

Figure 54 and Figure 53 demonstrates the proposed transition strategies and selection algorithm while placing a traffic congestion query on a set of GENI cloud nodes. In the following, we show the demonstration in four key steps utilized to acquire cloud nodes to place queries and perform transitions.

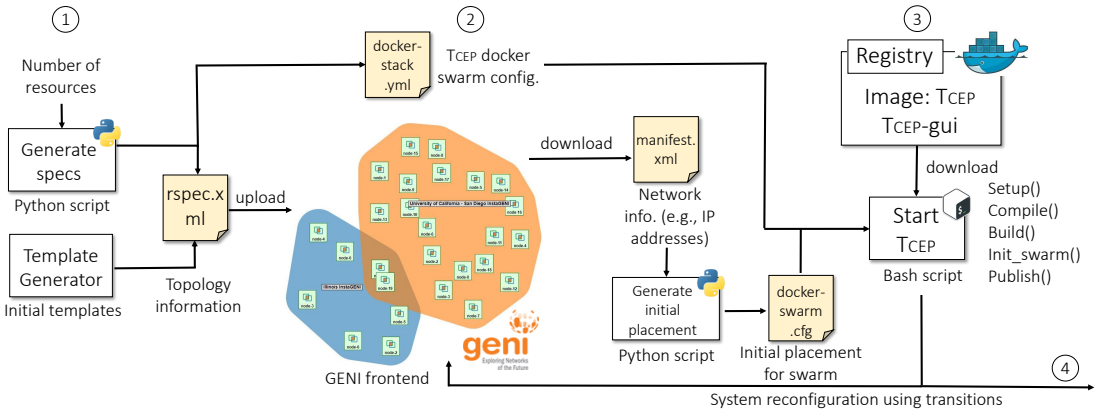



Figure 53: The transition interface that enables the application developers to program Operator Placement mechanisms and use the proposed transition strategies to execute transitions between them [207].

The TCEP demonstrator is based on Akka distributed toolkit that employs an actor programming model. In TCEP, each Akka actors and Docker containers together enables a large-scale and efficient communication on top of fog-cloud infrastructure as shown in teh demonstration. Although, we show the workflow for acquiring GENI cloud nodes, TCEP facilitates integration of heterogeneous infrastructures with CloudLab and MAKI compute instances.

The workflow is divided into four main steps as seen in Figure 54. ① First, the *topology generator* requests for the specified number of resources required to place operators. Although, the demonstrator initially accepts request for a fixed number of nodes, this is later made flexible using dynamic scaling of nodes. For the initial configuration, the *generator* provides templates to acquire the mentioned resources. ② We make use of Docker swarm that manages the allocated resources. The nodes communicate using a Docker-based overlay network, even though the underlying physical topology is different. ③ Next, the TCEP and TCEP Gui applications are downloaded which are readily available in the Docker registry[54]. ④ For visualization of the transitions, OP mechanisms and its performance characteristics we provide a web-based interface as seen in Figure 54 based on d3js[55]. The interface allows selection of various queries, OP mechanisms, the proposed transition strategies and the selection algorithm. Furthermore, TCEP allows easy integration of different queries and OP mechanisms in future that are automatically made available in this interface. In summary, in this demonstration we have shown the advantage of TCEP using real-world fog-cloud infrastructures of GENI, CloudLab and MAKI.

[54]TCEP in the docker registry. `https://hub.docker.com/repository/docker/mluthra/tcep` [Accessed in May 2021]

[55]d3js library. `https://github.com/d3/d3` [Accessed in May 2021]

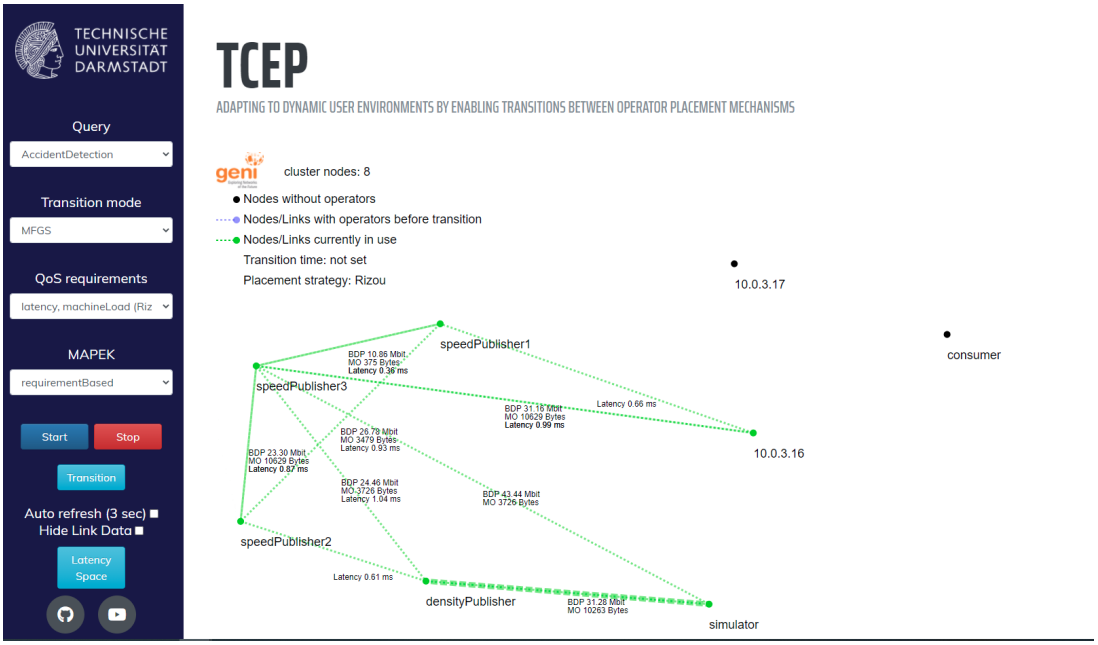



Figure 54: The visualization of TCEP system with real-time query execution on GENI cloud infrastructure and a live topology seen on the right side.

### A.1.3 *Selection Method for OP mechanism*

In Section 4.3.1, we noted that using an appropriate selection pressure, we can have a balance between exploration and exploitation. Blickle et al. [215] derived to what they call "selection intensity" presented as selection pressure in the following.

**Definition 22.** Selection pressure $(\mathcal{S})$. It is used to characterize the strong or high respectively weaker or small emphasis of selection on the best OP mechanisms. The selection pressure $\mathcal{S}$ for the fitness disrtribution $\overline{s}(f)$ is defined as follows.

$$\mathcal{S} = \frac{\overline{\mathcal{M}^*} - \overline{\mathcal{M}}}{\overline{\sigma}} \tag{13}$$

In Equation (13), the selection pressure depends on the fitness distribution of the population. Therefore, for different fitness distributions will generally lead to different selection pressure even for the same selection method. In order to define it specifically, we assume that the fitness distribution follows a Gaussian distribution $\mathcal{G}(0,1)$. In our evaluation, we have empirically validated this fact that the fitness distribution of all OP mechanisms follows a Guassian distribution, which leads to the following definition.

**Definition 23.** Standardized Selection Pressure ($\mathcal{S}_{\mathcal{R}}$). The standardized selection pressure $\mathcal{S}_{\mathcal{R}}$ is the expected average fitness value of the OP mechanism distribution after applying the linear ranking based selection method to the normalized Guassion distribution $\mathcal{G}(0,1)(f) = \frac{1}{\sqrt{2\pi}} e^{-\frac{f^2}{2}}$

$$\mathcal{S}_{\mathcal{R}} = \int_{-\infty}^{\infty} f(\overline{R}^*)(\mathcal{G}(0,1))(f)df \tag{14}$$

The effective and average fitness value of a Gaussian distribution with mean $\mu$ and variance $\sigma^2$ can be easily derived as $\mathcal{M}^* = \sigma\mathcal{S}_{\mathcal{R}} + \mu$.

**Theorem A.1.1.** The selection pressure using a linear ranking method can be derived as follows.

$$\mathcal{S}_R(\eta^-) = (1-\eta^-)\frac{1}{\sqrt{\pi}} \tag{15}$$

*Proof.* Using the definition of standardized selection pressure in Definition 23 and the Gaussian function for the initial fitness distribution, one can obtain

$$\mathcal{S}_R(\eta^-) = \int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \left(\eta^- + 2(1-\eta^-)\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy\right) dx$$

$$= \frac{\eta^-}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x \exp\left(-\frac{x^2}{2}\right) dx + \frac{1-\eta^-}{\pi} \int_{-\infty}^{\infty} x \exp\left(-\frac{x^2}{2}\right) \int_{-\infty}^{x} \exp\left(-\frac{y^2}{2}\right) dydx$$

Using

$$\int_{-\infty}^{\infty} x \exp\left(-\frac{x^2}{2}\right) = 0$$

and

$$\int_{-\infty}^{\infty} x \exp\left(-\frac{x^2}{2}\right) \left(\int_{-\infty}^{x} \exp\left(-\frac{y^2}{2}\right) dy\right)^2 dx = \sqrt{2}\pi$$

Equation (15) follows. □

## A.2 Supplementary Material to Chapter 5

This section defines the query grammar for the proposed meta query language in Chapter 5: Section 5.2.3 in Section A.2.1. Furthermore, Section A.2.2 shows the extensibility of the concept using two case studies. Finally, Section A.2.3 gives additional insights into the evaluation of INETCEP.

### A.2.1 *Query Grammar*

Typically, a programming language grammar consists of four main components defined as follows. *(i)* A set of terminals or tokens symbols occurring in a language. *(ii)* A set of non terminals or syntactic variables represented by a set of strings. Usually, these are defined the way they are used. *(iii)* A set of production rules defining the replacement of non-terminals with terminals or non-terminals or a mixture of both. Here, the terminal represents the *head* or the left side of the production rule, while the replaced part is the *body* or the right side of the production rule. *(iv)* A non-terminal can act as a *start symbol* for a production. Given the above definition and the Chomsky-Hierarchy [231], we select a type 2 grammar for the proposed meta query language of INETCEP. There are two reasons for this. *(i)* The type 2 grammar or the context-free grammar enables to combine the production rules. Furthermore, the head of a producer comprise of one non-terminal and the body is not to one terminal or non-terminal. *(ii)* We need a way to embed the operators in parentheses complying the rules of data plane language of NFN. This ability is provided by context-free grammars where paranthesis can be memorized.

We present the grammar of INETCEP meta query language using the BNF (Backus-Naur form) in Table 26, considering the above arguments. We make use of regular expressions as seen in the table, $REG(\ldots)$. Here, $(\ldots)$ can be literals or numbers. As per the guidelines of Backus-Naur form, the plus sign $+$ in $REG([a-z]+)$ denotes that at least one lowercase letter must appear. The relational algebra operators referred to as $comparison$ introduces a binary relation between two elements. For instance, this can be a key,value pair of an event tuple, and using the grammar rules a $comparison$ symbol can filter values from an event tuple.

### A.2.2 *Extensibility*

To facilitate extensions in the query language we follow a widely used design principle of object-oriented programming known as *Abstract Factory*. The pre-

| | |
|---|---|
| $\omega$ | ::= $\bowtie \mid \sigma \mid win \mid agg \mid seq$ |
| $\bowtie$ | ::= `JOIN( format , ` $\omega$ `, ` $\omega$ `, boolExp )` |
| $\sigma$ | ::= `FILTER( format , ` $\omega$ `)` |
| win | ::= `WINDOW( latinNumber , number )` |
| seq | ::= `SEQUENCE( format , ` $\omega \rightarrow \omega$`)` |
| agg | ::= `AGGFUN(format, latinNumber, win)` |
| AGGFUN | ::= `SUM \| MIN \| MAX \| AVG \| COUNT` |
| number | ::= *REG([0-9]+)* |
| latinNumber | ::= *REG([a-zA-Z0-9]+)* |
| boolExp | ::= latinNumber comparison latinNumber \| boolExp concat boolExp |
| comparison | ::= < \| > \| = \| <= \| >= |
| concat | ::= & \| "\|" |
| time | ::= *nn : nn : nn . nnn* |
| n | ::= *REG([0-9]{1})* |
| format | ::= `Data Stream \| Data` |

Table 26: Query Grammar for the meta langauge proposed in Section 5.2.3 [26].

viously presented Algorithm 5 starts the creation of an operator using our query parser.

To develop a new operator, the operator implementation has to override existing predefined methods of an abstact class `Operator`. The correctness rules for the parameters has to be defined for each new operator. The abstract pattern facilitates integration of new operators with minimum changes and serves as an important foundation for future work, for example, to integrate user-defined operators as done in Chapter 6.

In the remaining subsections, we define how the query language is used to implement applications which were used as a basis to evaluate INETCEP (cf. Section 5.3).

### *Prediction Algorithm*

One of the applications used to evaluate INETCEP is the prediction on smart plug query taken from the DEBS Grand Challenge in 2014. We reimplemented an existing solution on the problem [237] with the following design requirements. *(i)* Collect historical data to perform better predictions on the future energy consumption. *(ii)* Implement a lightweight solution for resource constrained IoT infrastructure.

The prediction formula was given in the original challenge, where $i$ refers to the current time, $s_i$ is the current recorded energy value at time $i$, $s_j$ gives

the past values at time $j = i + 2$. Thus, the prediction done two steps in the future comprise of the current and the average electricity consumption.

$$predicted\_load\,(s_{i+2}) = (avgLoad\,(s_i) + median\,(\{avgLoads\,(s_j)\})) \tag{16}$$

As per the above requirements and the Query 8, we define the prediction algorithm. It is determined whether or not for each time window of 1 minute, the prediction time is reached, that is given as another input to the query. If not, then the value of the prediction is stored as a state object to be used later. In contrast, if the prediction time is reached, the load prediction of the following form is emitted. $< ts, plug\_id; household\_id; house\_id; predicted\_load >$ Here, $ts$ is the current time, $plug\_id$ identifies the smart plug, $household\_id$ identifies the household, $house\_id$ identifies the house within a household and $predicted\_load$ gives the prediction calculated in Equation (16).

***Heatmap Algorithm***

In this section, we explain the second scenario used for the evaluation of INETCEP in Section 5.3. A heatmap is a widely used tool used for visualization in disaster scenarios [64]. We use an existing algorithm based on the work [236] and use the INETCEP meta query language to realize the query as given in Query 7. The algorithm takes as an input from the query, the window size, area of the location and the cell size that indicates how finely the output heat map is meshed. Algorithm 7 defines the process of heat map creation and visualization using the disaster field dataset. First, the horizontal and the vertical cells are computed based on the input area (Lines 1–2). Afterwards, for every window of the data stream of location tuples, the absolute longitude and latitude values are retrieved (Lines 3–5). Finally, the position in the heatmap is obtained by dividing the absolute values by the cell size (Line 6).

### A.2.3 *Additional Insights on the Performance Evaluation*

This section provides additional insights into the evaluation of the concepts introduced for network-centric query execution in Chapter 5. Specifically, we elaborate on *(i)* the placement time for the different queries considered for the evaluation in Section 5.3.1 and *(ii)* the hybrid processing of queries in the underlay and overlay.

**Algorithm 7 :** Algorithm used to compute heat map in the Query 7 [26].

**Variables :**

| | | |
|---|---|---|
| $loc$ | $\leftarrow$ | Window of location $< lat, long >$ tuple of the survivor data |
| $Lat_{min}$ | $\leftarrow$ | minimum latitude value |
| $Lat_{max}$ | $\leftarrow$ | maximum latitude value |
| $Long_{min}$ | $\leftarrow$ | minimum longitude value |
| $Long_{min}$ | $\leftarrow$ | maximum longitude value |
| $HC$ | $\leftarrow$ | number of horizontal cells needed to map the values |
| $VC$ | $\leftarrow$ | number of vertical cells needed to map the values |
| $cell\_size$ | $\leftarrow$ | granularity |
| $Grid$ | $\leftarrow$ | two dimensional array |

1 $HC$ = $\lfloor \frac{Long_{max} - Long_{min}}{cell\_size} \rfloor$ ;
2 $VC$ = $\lfloor \frac{Lat_{max} - Lat_{min}}{cell\_size} \rfloor$ ;
3 **for each** *line in* $S_D$ **do**
4 absLatVal = $loc$[lat] - $Lat_{min}$;
5 absLongVal = $loc$[long] - $Long_{min}$;
6 $Grid[\lfloor \frac{absLatVal}{cell\_size} \rfloor][\lfloor \frac{absLongVal}{cell\_size} \rfloor]$ += 1;
7 **return** $Grid$

### *Placement Time*

We investigated the difference in query processing in terms of the number of nodes, where we found that the placement time differs for the queries with more operators, which is intuitive. In the following paragraph, we present those results.

We analyzed the time taken for placement where the number of nodes and the topologies is different. The placement time is defined as the time taken to determine a path where the operators are placed. Quite intuitively, the placement time should be higher when the number of nodes is higher. However, we also analyze how the same differs between topologies. We use the manhattan topology and tree topology with seven and ten nodes, respectively. Figure 55 presents the result in the form of a point plot. As expected, the higher the number of nodes considered for placement, the higher is the time taken to find the placement and vice versa.

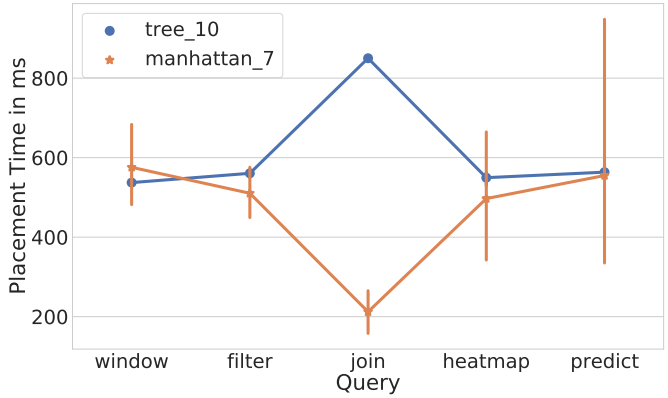



Figure 55: Analysis of the time taken to place operators for different topologies and number of nodes. The key take away is that it depends on multiple factors like operator depth. Still, it takes only a few milliseconds to place multiple operators thanks to parallel processing [139].

Interestingly, for all queries besides join, there is no large difference in the required placement time. This is because of the high number of operators (6 in total) required to place while executing the join query. In the tree topology case, it takes almost double the time with ten nodes than in the manhattan graph. Another reason operator placement in the manhattan graph is faster is that the number of paths possible for placement is quite less than in the tree topology. Therefore, multiple factors besides the number of nodes contribute to the metric placement time, such as the number of paths possible in a topology, operator graph depth and the number of operators.

Another interesting observation is that the error bar representing the $95\%$ confidence intervals is larger for the manhattan graph than for the tree topology. This can be explained by the evaluation setup used to run the experiments for a different number of nodes. Due to the unavailability of sufficient cloud resources for tree topology, the experiments were executed using CORE, where the results are not affected by real network latencies. In contrast, in the manhattan graph experiments, due to the execution of real-world cloud resources, network latencies are incorporated in the results. More importantly, using this evaluation, we understand the involvement of other factors in the placement time and the difference between them.

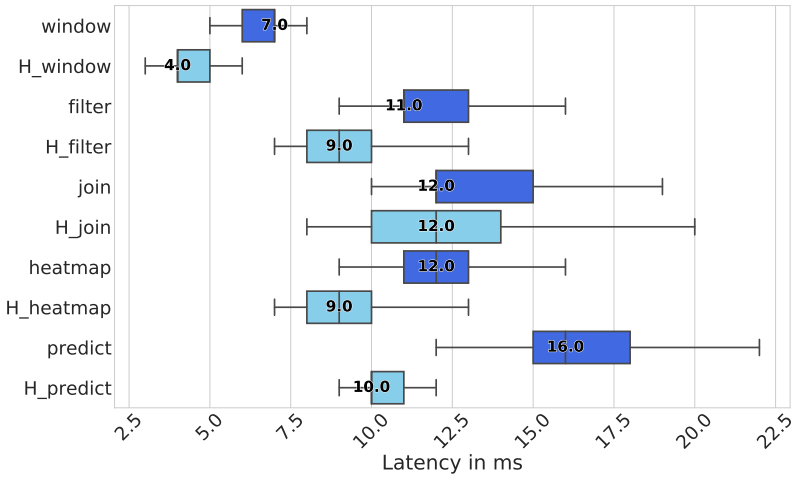



Figure 56: End-to-end latency evaluation for hybrid processing of operator graphs with the implementation of window operator in unified communication mechanism. Here, hybrid is denoted by H_Q, where $Q \in \{window, filter, join, heatmap, predict\}$). We note that the hybrid query processing is always faster than the standard processing when a single operator is moved to the network layer implementation [139].

### *Hybrid Processing*

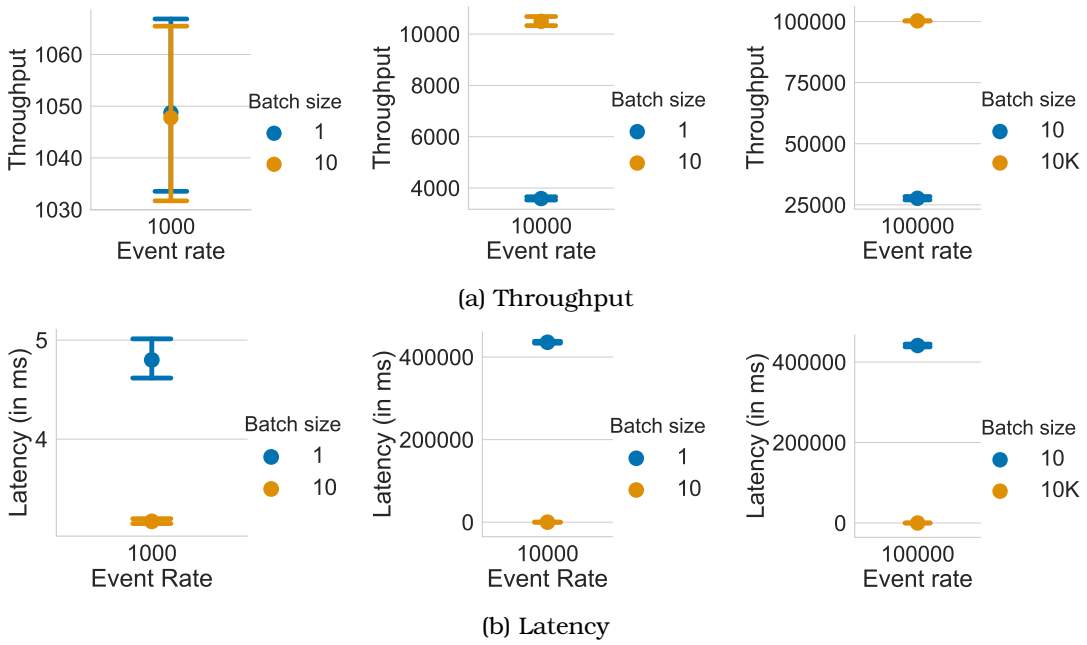
We take our evaluations one step further to understand if the operator execution can take advantage of the in-network line speed by directly implementing CEP operators in the ICN data plane. We call this implementation hybrid query processing since part of the operator is implemented directly in the network, while the other operators are executed isolated from the network inside the query engine. The implementation with standard INETCEP query engine differs in how the window operator is realized in the CCN-lite implementation alongwith the unified communication mechanism, while the other operators are implemented in Named Function Networking compute server implementation. Figure 56 presents the evaluation of hybrid query processing in the form of a box plot, where the value given by the middle line represents the median. We represent the hybrid implementation results using the keyword H_Q. We observe a benefit of offloading the implementation to the network, which is clear from the box plot. Although moving only a single operator opens such benefits, more operators will unfold substantial gains in terms of latency and the multiple open questions like coordination of query execution with network-centric operations.

## A.3 Supplementary Material to Chapter 6

In this section we provide additional insights into the performance evaluation of CEPLESS middleware in Section A.3.1 and the programming interface of CEPLESS middleware in Section A.3.2.

### A.3.1 *Additional Insights on the Performance Evaluation*

We review the batch size effect on the performance of CEPLESS event queues based on the algorithm proposed in Section 6.2.1. Afterwards, we discuss additional insights into the implementation overhead using the proposed programming interface (cf. Section 6.2.2).




(a) Throughput

(b) Latency

Figure 57: Impact of different batch sizes on the performance of Flink with CEPLESS.

***Impact of batching on performance***

To understand the impact on the performance of the different batch sizes where one corresponds to *no batching*, whereas $10$, $100$, and $10,000$ corresponds to the respective input batch sizes ($inBatchSize$) for the incoming events ingested in the Redis event queue as described earlier in Algorithm 6. Both performance metrics, throughput as seen in Figure 57a, and latency in Figure 57b are affected. With no batching, i.e., batch size of 1, we observe

that the throughput is declined and latency is increased at higher event rates which match our hypothesis as follows. There are more TCP requests per batch with a smaller batch size, which blocks I/O for higher event rates and results in higher latency and low throughput, as observed in Figure 57. Using this empirical result, we determined the batch size of 100 events as suitable for 1K, and 10K for 100K event rate.

### *Implementation Overhead of Operators*

In addition to improving the flexibility, we also simplify the integration of new operators into the CEP system using CEPLESS. In this section, we evaluate the complexity of integrating new operators into CEPLESS in terms of LOC (lines of code). We integrate the three queries as introduced in Section 6.3.1 to compare the process of implementing these directly into the CEP system or using CEPLESS.

| | Forward | Fraud-detection | K-means |
|---|---|---|---|
| Operator LOC | 1 | 14 | 94 |
| Total LOC (Flink) | 25 | 39 | 132 |
| Total LOC (TCEP) | 25 | 39 | 155 |
| **Total LOC (CEPLESS)** | **7** | **20** | **115** |
| LOC Overhead (Flink) | 96% | 64% | 28% |
| LOC Overhead (TCEP) | 96% | 39% | 39% |
| **LOC Overhead (CEPLESS)** | **85%** | **30%** | **18%** |

Table 27: LOC comparison of Apache Flink, TCEP, CEPLESS.

Table 27 shows the comparison of the implemented operators in the different systems. The first row shows the operator LOC, which serves as a comparison towards the overhead added by different systems.

We calculate the LOC overhead in % as $\frac{\text{Total LOC} - \text{Operator LOC}}{\text{Total LOC}} * 100$. Here, Total LOC includes the operator logic; hence, the remaining lines is the overhead associated with respect to the CEP systems. As an example operator that could be integrated into a CEP system, we chose to use an unsupervised ML algorithm named *k-means*. To get realistic evaluation results, we utilized the k-means implementation of the open-source Android codebase that encapsulates the algorithm in its machine learning library [243]. This implementation requires 94 LOC in order to give a result. The same operator implemention into Apache Flink requires 132 total LOC and in TCEP even 155 total LOC, resulting in an overhead of 28% and 39% compared to the baseline version of the algorithm. However, implementing the same algorithm code using CEP-

LESS is done in a total LOC of only 115, an overhead of only 18%, an effective reduction of 21% and 10% for Flink and TCEP, respectively.

While we understand that LOC is a flexible measure heavily dependent on language semantics and syntactic sugar, we want to see how the *same* business logic compares in different environments. Therefore, we also performed a user study in the Multimedia Communications Lab with the students enrolled in Computer Science and Electrical Engineering and Information Technology. Ten students participated in the user study, with 80% of them having some CEP experience. Every student used the CEPLESS platform web interface and evaluated it based on the usability and simplicity of the programming interface. Around 90% of the students found the interface usable and simple to use in the lab. In the Appendix A.3, we provide screenshots on the programming interface as well as the user study that was performed. A large scale user study with detailed feedback on the usability of the interface is visioned for the future work.

### A.3.2 *Programming Interface of* CEPLESS

Figure 58 shows the programming interface of CEPLESS. It uses Monaco Editor[56], a web-based code editor by Microsoft. It inherently supports many programming languages such as Java, Golang, Python, C++, JavaScript, etc., in syntax highlighting, keyword suggestion and syntax validation for frontend languages. In addition, bootstrap templates for decorative purpose are used.

The main section of the interface allows developers to select the programming language and an initial operator template for developing the user-defined $\omega_{UD}$ operator. When the code is ready for deployment, the user must click on *Check Operator* to validate the operator in terms of syntax. Afterwards, the respective compiler builds the code for the next step. In case the build fails for some reason, the feedback is returned to the *Output Console* related to the errors. Otherwise, the operator is successfully built. This enables the *Submit Operator* button that enables the application developer to build the operator container that is maintained in the centralized Operator registry, as discussed in Chapter 6: Section 6.2. The developer can also locally save the operator using *Save Operator* that creates a local copy of the operator source. On the other hand, *Delete Operator* deletes the same.

[56]Monaco Editor by Microsoft. https://microsoft.github.io/monaco-editor/ [Accessed in May 2021]

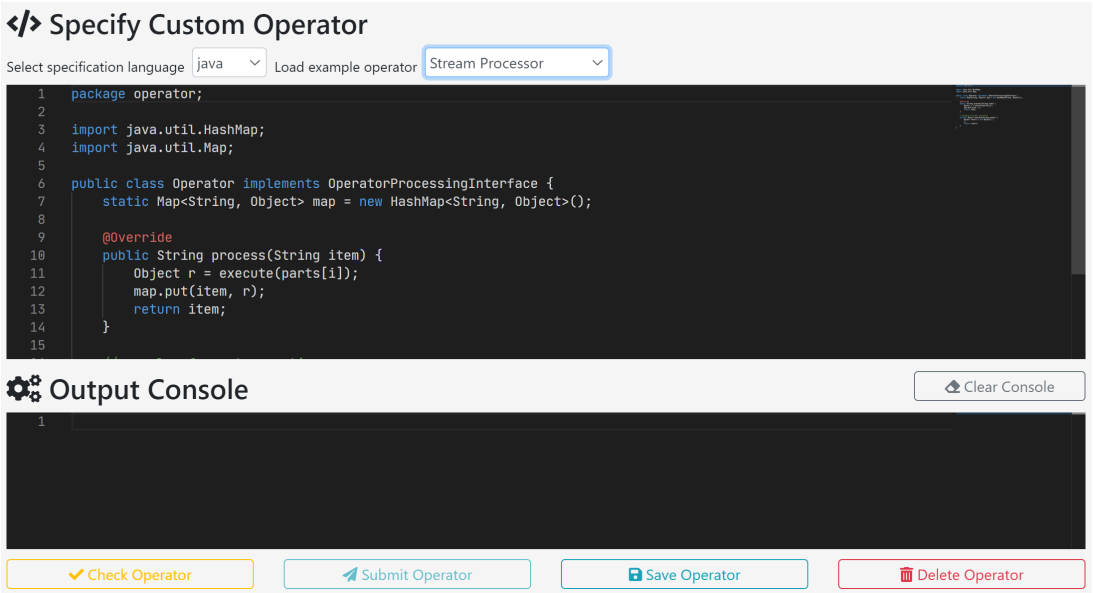



Figure 58: The programming interface of CEPLESS that enables application developers to program operators in the range of programming languages and use operator templates deployed before.

At the back-end, for each new programming language for operator specification, specific handlers have to be implemented for the operators and a Dockerfile to build the container. The implementation of the handlers is kept minimum corresponding to the UDO interface presented in Chapter 6: Section 6.2.2.

## A.4 Supervised Student Theses

[1] Sebastian Hennig. "Virtualizing CEP Execution Environment Using Serverless Paradigm for Latency Sensitive Applications." KOM-B-0649. Bachelor Thesis. Technical University of Darmstadt, 2019 (cit. on pp. xvi, xix).

[2] Niels Danger. "Specification of Transitions in Complex Event Processing Systems using Context Feature Models." KOM-B-0609. Bachelor Thesis (*In collaboration with Real Time Systems Lab, Technical University of Darmstadt*). Technical University of Darmstadt, 2018 (cit. on p. xvii).

[3] Raheel Arif. "Transition between Placement Strategies for Operator Networks." KOM-M-0596. Master Thesis. Technical University of Darmstadt, 2017 (cit. on p. xviii).

[4] Ali Haider Rizvi. "Situation-Aware Complex Event Processing over Information-Centric Networks." KOM-M-0620. Master Thesis. Technical University of Darmstadt, 2018 (cit. on p. xviii).

[5] Johannes Karl Pfannmüller. "Unified Communication Layer for In-Network Complex Event Processing." KOM-M-0701. Master Thesis. Technical University of Darmstadt, 2020 (cit. on p. xix).

[6] Lukas Fey. "A Migration Cost-Aware Evaluation Framework for VNF Placement Strategies." Master Thesis(*co-supervised*). Technical University of Darmstadt, 2017.

[7] Benedikt Lins. "Optimal decisions of transitions based on online learning methods." KOM-M-0689. Master Thesis. Technical University of Darmstadt, 2019.

[8] Niels Danger. "Optimal selection of performance models for operator placement." KOM-M-0726. Master Thesis *currently running*. Technical University of Darmstadt, 2021.

## A.5 Supervised Student Labs and Seminars

[1] Sebastian Hennig. "Large Scale Evaluation Experiments of Complex Event Processing on GENI testbed." AOC-6. Lab Project. Technical University of Darmstadt, 2018 (cit. on p. xviii).

[2] Johannes Karl Pfannmüller and Ahmed Zukic. "Evaluating In-Network Complex Event Processing over Named-Data Networks." AOC-2. Lab Project. Technical University of Darmstadt, 2018 (cit. on p. xviii).

[3] Sebastian Hennig. "Unified API for Complex Event Processing Operators using Serverless Paradigm." AOC-1a. Seminar. Technical University of Darmstadt, 2018 (cit. on p. xix).

[4] Moritz Fischer and Rupert Leimbach. "Point Cloud Stream Processing." AOC-1. Lab Project. Technical University of Darmstadt, 2019.

[5] Matheus Vieira, The-Khang Nguyen, and Minh Tran. "Serverless Programming for Streaming Systems." ACS-7. Lab Project. Technical University of Darmstadt, 2020.

[6] Pedro Matsumoto and Berk Namal. “An Analysis of Placement Strategies for Data Stream Processing Systems.” AOC-5. Seminar. Technical University of Darmstadt, 2017.

[7] Benedikt Lins. “Analysis of Performance Influence Model for Operator Placement.” Proseminar. Technical University of Darmstadt, 2018.

[8] Ahmed Zukic and Pascal Dornfeld. “Study on Function as a Service for Stateful Events.” AOC-1b. Seminar. Technical University of Darmstadt, 2018.

[9] Sebastian Sadkowiak and Siyuan Ye. “Research Advancements of Data Processing in Modern Networks.” ACS-6. Seminar. Technical University of Darmstadt, 2020.

[10] Azeem Ishola. “Efficient Data Processing on Modern Hardware.” ACS-6. Seminar. Technical University of Darmstadt, 2020.

# B

# Erklärung laut Promotionsordnung

**§ 8 Abs. 1 lit. c PromO**

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

**§ 8 Abs. 1 lit. d PromO**

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

**§ 9 Abs. 1 PromO**

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

**§ 9 Abs. 2 PromO**

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

*Darmstadt, 11th May 2021*

Manisha Luthra

## Colophon

This document is based on a typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`