TCEP: Transitions in Operator Placement to Adapt to Dynamic Network Environments

Manisha Luthra[a], Boris Koldehofe[a,b], Niels Danger[a], Pascal Weisenburger[a], Guido Salvaneschi[a,c], Ioannis Stavrakakis[d]

[a] *Technical University of Darmstadt, Germany*
[b] *University of Groningen, Netherlands*
[c] *University of St Gallen, Switzerland*
[d] *National and Kapodistrian University of Athens, Greece*

Abstract

Distributed Complex Event Processing (DCEP) is a commonly used paradigm to detect and act on situational changes of many applications, including the Internet of Things (IoT). DCEP achieves this using a simple specification of analytical tasks on data streams called operators and their distributed execution on a set of infrastructure. The adaptivity of DCEP to the dynamics of IoT applications is essential and very challenging in the face of changing demands concerning Quality of Service. In our previous work, we addressed this issue by enabling transitions, which allow for the adaptive use of multiple operator placement mechanisms. In this article, we extend the transition methodology by optimizing the costs of transition and analyzing the behaviour using multiple operator placement mechanisms. Furthermore, we provide an extensive evaluation on the costs of transition imposed by operator migrations and learning, as it can inflict overhead on the performance if operated uncoordinatedly.

1. Introduction

The unprecedented growth in IoT devices has enabled multiple applications in connected vehicles, financial trading, and industrial manufacturing. Cisco predicts that there will be 29.3 billion IoT devices by 2023, and among those, connected vehicles will be the fastest-growing application type [1]. IoT applications, especially involving highly mobile components such as connected vehicles, incorporate inherent dynamics in the environment and the required Quality of Service (QoS) demands. Such applications need to continuously adapt their system's components to meet specific QoS demands related to environmental conditions. An essential aspect in the adaptation cycle of IoT applications is detecting situational changes that trigger actions at the distributed application components– for instance, detecting and reacting to the change in the density of the vehicles depending on the time of the day, such as rush hours vs regular hours.

DCEP is a prevalent and frequently applied paradigm to detect and act on such situational changes. DCEP analyzes data streams from many distinct data sources and detects event patterns, named *complex events*, corresponding to the situational changes to which IoT applications need to adapt. The logic to detect such situational changes is modelled using a data flow graph, commonly referred to as an *operator graph*. An operator graph represents the computational units that help detect complex events, named *operators*, which are interconnected by data streams. DCEP needs to ensure that events of interest or complex events are delivered while meeting the specified QoS demands of the IoT application. For instance, a connected car application that shares contextual information between multiple vehicles for time-critical and safety-critical decisions has a latency demand of delivering information in less than 30ms [2]. A central mechanism of a DCEP system towards fulfilling such QoS demands is an Operator Placement (OP) mechanism that dictates the assignment of the operators on the resources of the IoT infrastructure. The placement of operators on resources, such as on the things (IoT devices) at the edge, or resources at the fog [3], or inside data centers, helps accomplish the specified QoS demands, such as low latency, bandwidth efficiency, or reliable delivery.

Typically, DCEP systems rely on a single OP mechanism optimized for one QoS demand or combining multiple demands. For instance, OP mechanisms have been widely researched to minimize latency [4], to reduce load [5, 6, 7], to minimize network usage (bandwidth-delay product) [8, 9, 10], and to preserve trust and privacy [11]. Some authors even combine multiple QoS demands in a multi-objective optimization formulation to find Pareto-optimal solutions for operator placement [12, 13]. However, under changing QoS demands, which are unknown beforehand, current DCEP systems fail to find a suitable OP mechanism because they are restricted to a single OP mechanism. Furthermore, the OP mechanisms are specialized for given environmental conditions, such as the mobility of producers or consumers – stationary or highly mobile.

Current OP mechanisms are known to have trade-offs regarding supported QoS demands dependent on the given environmental conditions. The reasons are twofold, *(i)* the conflicting nature of QoS demands, such as minimizing latency but limiting the overhead in assigning operators, and *(ii)* because OP mechanisms favor specific environmental conditions, such as high mobility vs low mobility of connected vehicles. Instead of aiming for a single universal mechanism supporting all kinds of QoS demands and environmental conditions, we pursue in this article the idea of dynamically changing mechanisms at runtime by introducing and analyzing an adaptation technique named *transition* [14]. The *transition* facilitates dynamic change of mechanisms to benefit ideally from the best suitable mechanisms required under specific environmental conditions.

Introducing transitions in a seamless and non-disruptive manner, i.e., without any interruption in the output into DCEP, is a highly challenging task and requires careful choice of system mechanisms. In this article, we aim to solve this challenge in the context of OP mechanisms. A critical issue that we address is to efficiently migrate operator graphs while maintaining the correctness of the results and imposing minimum costs into the DCEP system. Naively approaching the problem will lead to high overhead in terms of state transfer for stateful operators and communication overhead, which eventually leads to a failure in terms of fulfilment of QoS demands. Therefore, a systematic selection of an operator placement mechanism is required to fulfil the QoS demands.

In this article, we extend our previous findings on TCEP [15][1] by *(i)* proposing a programming model that enables analysis of distinct OP mechanisms and their adaptation for various QoS demands, *(ii)* determining optimal discrete-time points when to perform operator migrations such that the cost is minimal as part of the cost-efficient algorithm, and *(iii)* adaptively selecting OP mechanisms while maintaining a low overhead using genetic learning methods.

In more detail, this article provides the following contributions:

1. We formalize the problem of *transitions* for operator placement problem in the DCEP system, considering distinct QoS demands of applications, and present the definition of the *cost* that needs to be considered in performing *transitions* in OP mechanisms.

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

3. We present and analyze the *genetic learning*-based method for adaptively planning *transitions* between OP mechanisms to meet dynamically changing QoS demands and changes in the network environment.

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

5. We present an extensive evaluation to analyze the behaviour of state-of-the-art OP mechanisms using distinct queries and analyze their performance on the distributed set of fog-cloud infrastructure, including GENI [16], CloudLab [17], and MAKI [18] resources. Furthermore, we analyze the performance of *(i) mechanism transitions* under dynamics of environmental conditions, *(ii)* proposed *transition* algorithm in terms of costs imposed, and *(iii)* costs incurred by genetic learning-based selection algorithm.

[1] TCEP and its programming model are made publicly available for use. `https://luthramanisha.github.io/TCEP/` [Accessed on 21.04.2021]

Our extensive evaluations of TCEP show in the context of presented traffic congestion detection queries that the *transitions* can be performed in the range between $0.85 - 2$ seconds while maintaining 100% throughput in detecting the complex event due to the minimal costs in terms of time and overhead.

The remainder of this article is structured as follows. We provide a brief introduction to DCEP using an example of the traffic control scenario and motivate *mechanism transitions* by a preliminary evaluation in Section 2. We introduce the TCEP system model in Section 3 and present the problem statement in Section 4. We present the design of TCEP in Section 5 and evaluate the TCEP system in Section 6. Finally, in Sections 7 and 8, we present the related work and conclude our paper, respectively.

2. The need for transition of OP mechanisms

To demonstrate and motivate the need for exchanging OP mechanisms through transitions, we first introduce a typical use-case of Complex Event Processing (CEP) in the context of a traffic control application that is consistently used in this article until later on as part of the evaluation. Furthermore, we show significant shortcomings of current CEP systems for the scenario by performing an initial evaluation study on state-of-the-art placement mechanisms.

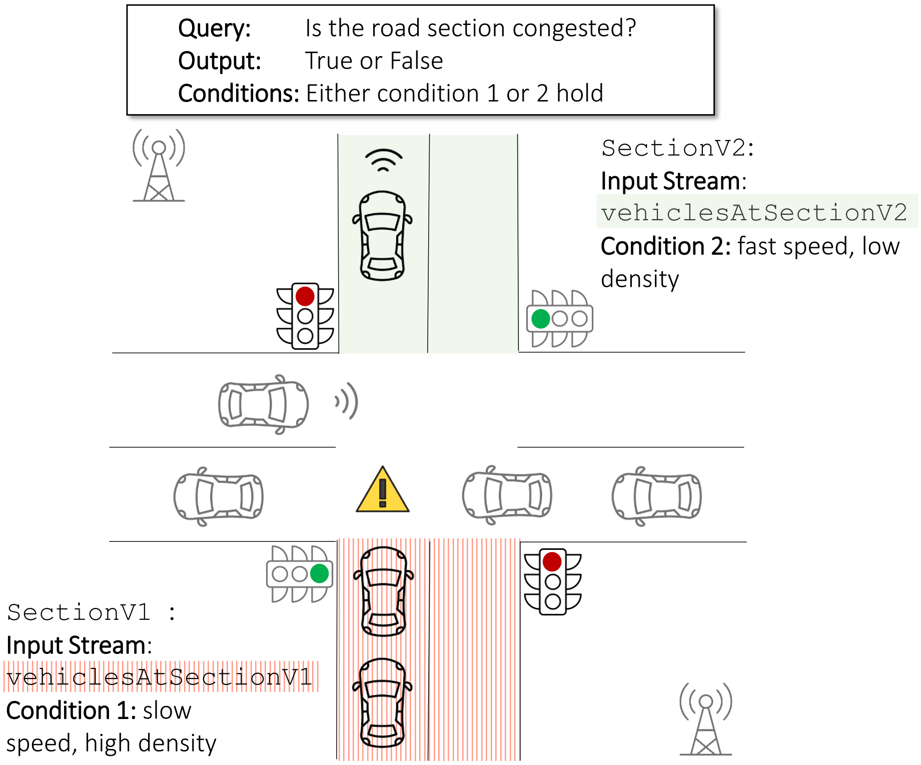
2.1. Complex Event Processing

CEP is a powerful paradigm that detects patterns in the incoming data streams to derive higher-level events such as traffic congestion. Consider a traffic control application in an IoT scenario that processes information from different producers such as *smart vehicles* and *radar sensors.* These producers generate continuous data streams comprising of event tuples of the following form– *vehicle sensor*: $< ts, section\_id, vehicle\_id, vehicle\_speed >$ and *radar sensor*: $< ts, section\_id, no\_vehicles, avg\_speed >$. CEP allows specification of the higher-level events such as traffic congestion in the form of a *query.* A query comprises computational units called *operators* such as *filter*, *join*, and *sequence* that can specify transformations on the data streams. CEP operators can be classified as *stateless* such as filter operator, and *stateful* such as window-join, window-aggregate and sequence operators. *Stateless* operators perform computation only on the current input tuples, while the *stateful* operators perform computation on the current and past input tuples depending on the semantics of the operator. The number of past tuples considered for computation in a stateful operator is typically formulated using a *window* based on time or tuple size. While there exist multiple window types, we consider a sliding window in our running example in this article[2]. Here, slide size refers to the number of event tuples shifted in a given window such that new event tuples from the data stream are included in the next window cycle. Moreover, the window size refers to the number of events tuples to be considered for the computation in the current window cycle.

In the running example, a stateful operation like joining of data streams observed at `SectionV1` (shaded in lines with red) and `SectionV2` (shaded in green), which are two road sections of a crossing, result in *composite data streams* seen in Figure 1b: `vehiclesAtSectionV1` and `vehiclesAtSectionV2`. Another example of a stateful operator used in detecting a traffic congestion event is when the sequence of Condition 1 followed by Condition 2 as described in Figure 1a takes place, which is detailed in the next section. We note that traffic detection in real applications is more complex than the provided example. However, for simplicity and better understandability, we refer to the above example.

CEP can be realized in two ways: *centralized* or *distributed*. While the processing at a single node (centralized) is beneficial for some scenarios, DCEP is particularly useful for large scale scenarios as in this work. In this work, we focus on DCEP that comprises multiple nodes, which collaboratively process the query. We further detail the problem using the traffic control application in the next section.

[2] Yet, the proposed system is not restricted to sliding windows and can be applied to other types of windows such as tumbling windows.


Query: Is the road section congested?
Output: True or False
Conditions: Either condition 1 or 2 hold
SectionV2:
Input Stream:
vehiclesAtSectionV2
Condition 2: fast speed, low density
SectionV1 :
Input Stream:
vehiclesAtSectionV1
Condition 1: slow speed, high density

(a) Congestion detection under dynamic environmental conditions performed by query in (b).

```
case class VehiclesAtSection(sectionId: Int, avgVehiclesDensity:
     Long, avgVehiclesSpeed Long, time: Long)
val vehiclesAtSectionV1: Stream[VehiclesAtSection] = ...
val vehiclesAtSectionV2: Stream[VehiclesAtSection] = ...
val congestedAdjacentRoadSections = Query[RoadSections]
  ((vehiclesAtSectionV1 where { v1 =>
     v1.avgVehiclesSpeed < NormalSpeedThreshold &&
     v1.avgVehiclesDensity > HighTrafficThreshold
   })
   ->
   (vehiclesAtSectionV2 where { v2 =>
     v2.avgVehiclesSpeed > NormalSpeedThreshold &&
     v2.avgVehiclesDensity < HighTrafficThreshold
   })
   within 1.min
   where { case (v1, v2) => v2.time > v1.time }
   demand QOS_DEMAND)
```

(b) Query to detect congestion at a road section.

Figure 1: Traffic congestion control scenario highlighting the change in environmental conditions at the two road sections SectionV1 and SectionV2 necessitates different OP mechanisms for dynamic user environments.

2.2. Case Study: IoT Traffic Control Application

In this section using the traffic control application introduced in the above Section 2.1, we show that under the dynamics of environmental conditions, state-of-the-art placement mechanisms [8, 6] fail to fulfil QoS demands while detecting a traffic congestion event under dynamically changing environmental conditions.

Let us consider a continuous query[3] to detect that a road section on a crossing is congested, as seen in Figure 1b. Any consumer can pose a query for a specific road section on the crossing, say `SectionV1`. Examples for a consumer could be emergency services, traffic lights, and all vehicles near `SectionV1`, which are interested in getting traffic updates. The query specifies conditions, such as high traffic density and low vehicle speed on `SectionV1` and its crossed road section, `SectionV2`. The query specifies a sequence (Line 9) of such conditions for `SectionV1` (Lines 5–7) and `SectionV2` (Lines 10–12). The composite data streams `vehiclesAtSectionV1` and `vehiclesAtSectionV2` are assumed to contain information on the average speed and density. This is done by employing transformation of data streams from heterogeneous sources such as sensor nodes in the IoT infrastructure, e.g., speed sensors, radar sensors, and road side units, as seen in the previous section. The complex event: "congestion of road `SectionV1`" is successfully detected when the sequence of conditions on `SectionV1` and `SectionV2` in a temporal timespan of one minute (Line 14) indicates *(i)* dense traffic and slow vehicles for `SectionV1` and *(ii)* sparse traffic and fast vehicles for `SectionV2`, respectively.

The execution of the query is performed in a distributed manner on the available resources in the IoT infrastructure, such as vehicles, that can directly communicate using techniques like V2X [20] and device-to-device communication [21]. The mapping of the operators to these resources is done through an OP mechanism, which must account for the `QoS_DEMAND` specified within the

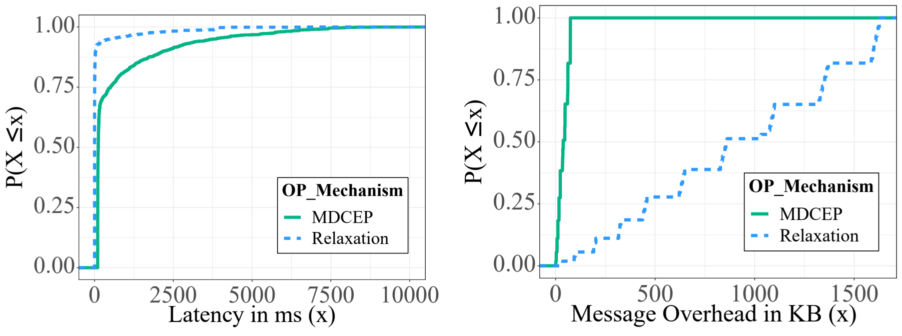
[3] in the AdaptiveCEP query language written in Scala [19].

query. As part of the query specification[3], these demands such as low latency can be specified according to the users' requirements.

A premise underlying our work is that the same OP mechanism cannot accommodate conflicting QoS demands. Therefore, we analyze the ability to fulfil specific QoS demands for the query in Figure 1b for two popular state-of-the-art OP placement mechanisms: *Relaxation* [8] and *Mobile DCEP* [6]. The key idea of the *Relaxation* mechanism is to place operators based on a model referred to as a latency space. The latency space allows determining communication delays between resources in the IoT environment, and the mechanism uses the relation to find a near-optimal embedding of an operator graph with respect to end-to-end latency. In contrast, *Mobile DCEP* avoids the overhead in maintaining any topological information, which needs to be updated frequently in a highly dynamic environment. Instead, the placement decisions are based on devices within the communication range capable of forming a device-to-device network closer to the data sources. In this way, the authors achieve a sub-optimal embedding of the operator graph at low control message overhead.

We analyzed the above two mechanisms in an IoT environment with mobile IoT resources (i.e., vehicles in this scenario) under the two crucial QoS demands *(i)* end-to-end latency defined as the total time required to detect events, and *(ii)* control message overhead needed to establish stable communication between the placed operators.

Figure 2 shows the measurements on end-to-end latency and control message overhead achieved by the OP mechanisms in a dynamic mobile environment for 50 incrementally deployed queries given in Figure 1b. The details on the evaluation configuration can be found later in Section 6. The cumulative distribution function (CDF) of latency under an increasing number of deployed queries confirms that *Relaxation* achieves consistently very low latency less than 100 ms for most of the queries, i.e., 80 % of the query workload, as seen in Figure 2a. This is consistent with the findings of Pietzuch et al. [8]. However, the control message overhead to coordinate the placement, in this case, to build the latency space, is increasing quickly with the number of deployed queries up to 1500 KB



(a) Relaxation achieves lower end-to-end latency than Mobile DCEP.

(b) Mobile DCEP achieves lower message overhead than Relaxation.

Figure 2: Performance comparison of Relaxation [8] and Mobile DCEP [6] OP mechanisms for 50 incrementally deployed queries.

on average, as seen in Figure 2b. In contrast, *Mobile DCEP* achieves little message overhead for all queries in the order of few bytes allowing for a very stable OP, but many queries suffer a long end-to-end latency of ∼7.5 s on average.

The above preliminary evaluation shows that different QoS demands require building on different OP mechanisms. Most importantly, depending on the changing environmental conditions – high or low mobility and high or low query workload – different mechanisms must fulfil the specific QoS demands. In a less dynamic environment concerning node mobility, such as with slow-moving vehicles, we measured a significantly lower control overhead for *Relaxation*, and hence it can be used to achieve low latency in condition 1. However, when changing from condition 1 (with lower dynamics) to condition 2 (with higher dynamics), a transition from *Relaxation* to *Mobile DCEP* is essential. Controlling the overhead improves the stability of the OP under the increased dynamics. In the presence of a dynamic environment and conflicting QoS demands, it becomes imperative to adapt OP mechanisms, which is the focus of this work.

3. System Model

In this section, we introduce the system model we use in describing the concepts of TCEP. In particular, we introduce the operator graph that models event processing to detect complex events, the IoT resource model that describes

Notation	Meaning
P	Set of event producers ($p \in P$)
C	Set of event consumers ($c \in C$)
B	Set of brokers ($b \in B$)
D	Continuous data stream
E	Set of event tuples ($e \in E$)
Ω	Set of CEP operators ($\omega \in \Omega$)
G	Operator graph
f_ω	Processing logic of an operator
B_I	Input buffer of an operator
B_O	Output buffer of an operator
$M_1 \dots M_N$	OP mechanisms
α	Mapping function of the operator ω
T	Transition function defining a dynamic change of mechanisms
$en(t)$	Environmental conditions dependent on time t

Table 1: Notations and their meaning.

the placement infrastructure, the node model that describes the entities participating in the processing of events, OP mechanisms and transition model for adapting DCEP, and QoS demand model which IoT applications use to express their requirements.

3.1. TCEP *Model*

TCEP 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 on the incoming data streams, and *(iii)* a set of event brokers (B), which host a set of operators (Ω) to process and forward events. Event consumers specify complex events that represent an event pattern by means of a continuous DCEP query. The query induces a directed acyclic *operator graph* $G = (\Omega \cup P \cup C,\ D)$, comprising of operators, producers, consumers and data streams, s.t., $D \subseteq (P \cup \Omega) \times (C \cup \Omega)$.

The operator graph dictates the execution plan specific to the query given by the event consumer. Figure 3 illustrates an operator graph for detecting traffic congestion at road sections corresponding to the query in Figure 1b. The data flow of the events in the operator graph is given from bottom to top of the graph. Here, the operators down the hierarchy are the predecessors (producers are at the last level), while the operators up the hierarchy are the successors (consumers are at the top level). Operators ω_{V1} and ω_{V2} correspond to the window-aggregate operators of the two input streams from the road sections $V1$ and $V2$. Operators $\omega_{\rightarrow}$ and ω_{σ} denote sequence and selection operators, respectively. Each operator ω dictates a processing logic f_{ω}. The data stream encapsulates a set of event tuples E, where each tuple is of the form $e = \{(k_1, v_1), \ldots, (k_{last^e}, v_{last^e})\}$. Here, k refers to the name of the tuple and v refers to the tuple value. In TCEP, we assume that the data streams arrive in the order indicated by the timestamp in the event tuple [22] and the system nodes are equipped with clocks that can be synchronized using a clock synchronization protocol such as Network Time Protocol [23].

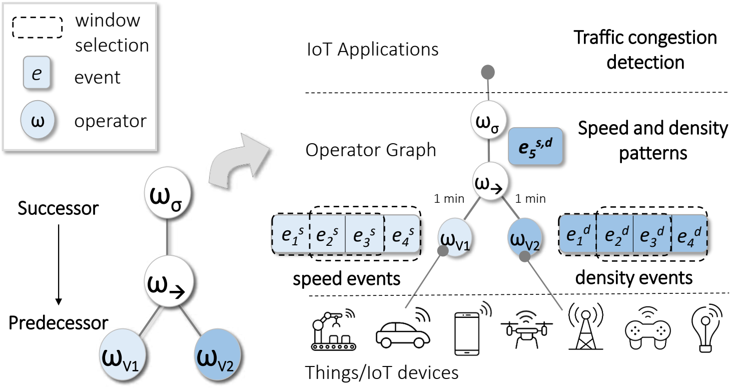


Figure 3: Operator graph for the query in Figure 1b.

Definition 3.1. *Operator Buffers and State.* The function f_{ω} processes ordered input data streams from the operator's input buffer B_I and produce output events stored in the operator's output buffer B_O. An operator either works based on the fixed computational parameters that are immutable (e.g., filter and stream operators) or it works on a dynamically changing computational state that is mutable (e.g., window and sequence operators), depending on the internal logic of the operator [24]. A mutable operator can dynamically change the selection of events determined by an operator-specific *selection policy* and *consumption policy* of window and sequence operators [25].

For instance, in Figure 3, the operator ω_{V1} specifies a selection policy for a sliding window size 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 ω_{V1} applies the processing function on the updated selection of events $\{e_2^s, e_3^s, e_4^s\}$ after slid-

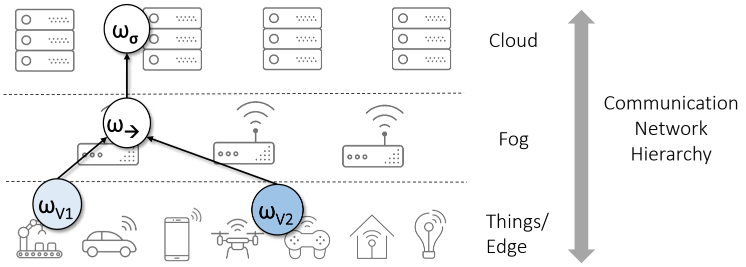


Figure 4: Example operator graph deployment and Tcep execution environment on the IoT network resources.

ing one event. 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*. In this example, the slide size defines the consumption policy, e.g., e_1^s is evicted when the subsequent transformation step with $\{e_2^s, e_3^s, e_4^s\}$ is performed.

3.2. IoT Resource Model

Although Tcep is not limited to a specific network topology and resource model, we will focus on the resources commonly considered in the context of IoT. Consider a hierarchical network infrastructure illustrated in Figure 4. The figure presents three layers: *(i)* (mobile) Things referring to IoT devices interconnected over wireless communication, *(ii)* a layer of resources at the fog that offer a low-latency link to the Things in physical proximity, *(iii)* and a fixed network layer comprising distributed resources in data centers or cloud. It is important to note that cloud and fog resources are assumed to communicate via a fixed IP infrastructure or novel ICN architectures [26]. In contrast, IoT devices and edge resources can form different wireless network topologies, including device-to-device communication [21] between IoT devices or V2X [20] between vehicles.

Things represent producers and consumers in the IoT scenario, while operators can be placed on any 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 processing power in the cloud. In contrast, IoT devices have resource constraints because they are battery-powered. Fog nodes are computationally more powerful than mobile nodes. For things, the availability of spatially nearby fog resources is restricted. For instance, IoT devices like Raspberry Pis and smartphones are resource-constrained and less powerful than computational resources at the fog locations such as micro data centers. Moreover, the availability of a fog location nearby an IoT device is not ascertained. Each operator ω is encapsulated in a container on the computational resources of the IoT infrastructure, as defined in Definition 3.2.

3.3. Node Model

A node acts as a host to the system entities producers, consumers, or brokers. Nodes refer to resources of the IoT resource model over which a producer, consumer, or broker can be executed. Note that the mapping of brokers on the node can change dynamically due to the dynamics in the environment. The nodes form an overlay network imposed by the interconnection of the operator graph on top of the IoT resource model. Figure 5 illustrates such an overlay network for the query and operator graph introduced in Section 2.2. Here, the rectangular boxes denote the nodes, and the operator graph is executed on them.

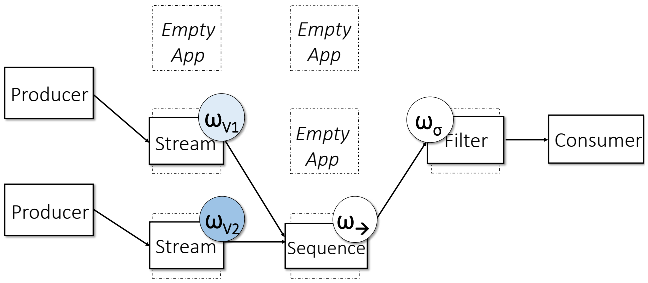


Figure 5: Tcep node model. The solid contour indicates pinned operators, while the dotted indicates unpinned operators.

Definition 3.2. *Containers.* A Tcep container enables the flexible movement of nodes in the IoT resource model.

Tcep differentiates between pinned entities, i.e., producers and consumers, from unpinned entities, i.e., DCEP operators. This is accomplished using *static* and *dynamic* containers. As the name suggests, the static containers are pinned to one node, while the dynamic containers are unpinned, meaning these support migration of operators on different nodes at runtime. An example of a static container is a producer and consumer, while a broker can be pinned or unpinned to a dynamic container named *Empty App* (cf. Figure 5). Although *EmptyApp* can hold more than one operator, they are free to move between other *EmptyApp* without harming the other operators being executed on the same node. In this way, we enable flexible operator deployment and operator migrations on the fog-cloud infrastructure.

3.4. OP Mechanism and Transition Model

The Tcep follows a modular design as a composition of multiple OP mechanisms $M_1, M_2, \ldots, M_N$.

Definition 3.3. *OP mechanism.* An OP mechanism determines *where* and *how* to map an operator graph G on a set of given brokers $B = \{b_1, b_2, \ldots, b_{last^B}\}$ in the IoT resource model. The mapped network of brokers is well known as an operator network. We define the mapping of the operator network as follows:

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

Definition 3.4. *Transition.* In this work, we define the concept of a transition for OP mechanisms, denoted as $T : M_A \rightarrow M_B$. A transition T performs a switch from a mechanism M_A to M_B, e.g., OP mechanisms at run time.

The goal is to perform a transition in a *seamless* or non-disruptive manner and to avoid oscillations during a transition. By *seamless* execution of transition, we mean no disruption in delivering complex events during the lifecycle of a continuous query (cf. Section 4). By oscillations, we mean that given the dynamics in the environmental conditions, the system may decide in a short interval to transit to a different mechanism multiple times. Tcep prevents oscillations by maintaining a balance in exploring multiple OP mechanism vs exploiting best OP mechanisms (cf. Section 5.2).

3.5. QoS Demand Model

An essential principle of an OP mechanism is to find a mapping of an operator graph to brokers that optimally satisfies an objective function of QoS demands, such as end-to-end latency, bandwidth, and control message overhead. TCEP allows specification of one or more QoS demands (QoS) and changing them at run time. The dynamics in the environmental conditions ($en_1, en_2, \ldots, en_{last^{en}}$), such as varying workload and mobility, influence the fulfilment of such QoS demands.

In this work, we consider two crucial performance metrics influencing the decision of operator placement in a dynamic environment: *end-to-end latency* and *control message overhead*.

Definition 3.5. *End-to-end latency.* It is the time taken to *(i)* receive the required primary events for the query at the placed nodes, *(ii)* process the query, *(iii)* emit a complex event, and *(iv)* transmit the complex event through the network path between the given event producers P to the given consumers C.

It is important to note that end-to-end latency can be time-varying due to the dynamic nature of the network and the placement update of the operators. In case multiple producers or consumers are involved, then latency is measured from the producer with the maximum network delay to the consumer, as explained in the example below.

To better understand, let us revisit the example scenario introduced in Section 2. We assume that two producers `vehiclesAtSectionV1` and `vehiclesAtSectionV2`, and a single consumer is interested in detecting congestion. Now, consider the path from p_1 : `vehiclesAtSectionV1` and p_2 : `vehiclesAtSectionV2` via some broker vehicles $b_1, b_2, \ldots, b_{last^B}$ to the consumer c. We assume the position of the consumer is at Section V1 when the query is triggered, and the OP was determined at the aforementioned producer and broker network path. In this case, the end-to-end latency is the sum of the network delay observed on the path $p_2, p_1, b_1, b_2, \ldots, b_{last^B}, c$ and the execution time of the query on these nodes in the path. In case multiple consumers, say c_1 and c_2, are interested in the same query, the end-to-end latency is given by the interval between the first primary event production at p_1 and the complex event reception at the consumer c_1 or c_2. Note, even when the query has been placed at the same set of brokers, the end-to-end latency for the different consumers will depend on the consumer's location, and hence it could be different for each consumer.

Definition 3.6. *Control message overhead* The number of control messages sent to assign all the operators $\omega \in \Omega$ of a query to the brokers $b \in B$. In essence, it is given by the overhead caused in exchanging messages to place the query on the IoT resource model.

Using the above definition of control message overhead and the assumptions on the traffic control scenario in Definition 3.5, let us demonstrate the meaning of control message overhead. To fulfil an objective, such as end-to-end latency, an OP mechanism such as Relaxation [8] maintains 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 operators on nodes are sent. We refer to the number of such control messages for OP as control message overhead. Some OP mechanisms like MDCEP [6] aim to minimize this metric on the cost of sub-optimal OP concerning metrics, like end-to-end latency, to prevent overhead on resource-constrained IoT nodes.

4. Problem Statement

Consider the availability of N-different OP mechanisms that can be selected to execute and place a query on the IoT network resources. Dependent on the environmental conditions $en(t)$ at time t, the QoS demands of consumers, say $QoS_{|en(t)}$ are changing. Furthermore, the ability and cost of an OP in terms of resource requirements to fulfil the QoS demands are changing over time.

The TCEP system aims to ensure that the QoS demands of queries are fulfilled despite changing environmental conditions using the IoT resource model. Therefore, we determine for changing environmental conditions $en(t)$ and corresponding $QoS_{|en(t)}$ demands a sequence of points in time, say $t_1, \ldots, t_n$ and a sequence of OP mechanisms $M(t_1), \ldots, M(t_n)$ on which a transition $T_i : M(t_i) \rightarrow$

$M(t_{i+1})$ is initiated at time t_i. It is important to note that while performing a transition, several operator migrations must take place. The operator migrations impose a high cost because of state migrations in terms of time and overhead. Moreover, the transition needs to be performed in a non-disruptive manner, i.e., even during the transition, the QoS demands of a query need to be satisfied. Consequently, state migrations have to take place in a cost-efficient manner.

We define the objective function of the *transition problem* considering two key cost factors, namely, the costs imposed in terms of transition time $C_{Time}(T_i)$ and transition overhead $C_{Overhead}(T_i)$. The transition time is defined as the time it takes to select a new target placement mechanism $M(t_{i+1})$ $(Time_{select})$, to find a placement α dependent on $M(t_{i+1})$ $(Time_{\alpha})$, and to migrate j operators $\omega_j \in \Omega$, $\forall j \in [1, num^j]$ to the target brokers $(Time_{mig.(\omega_j)})$ dependent on α. Thus, we define the cost in terms of transition time as:

$$C_{Time}(T_i) = Time_{select} + Time_{\alpha} + \sum_{j=1}^{num^j} Time_{mig.(\omega_j)} \ . \tag{2}$$

Similarly, the transition overhead is given by the overall number of messages exchanged in order to perform a transition, including the *(i)* selection of a placement mechanism $(Overhead_{select})$, *(ii)* the placement $(Overhead_{\alpha})$, *(iii)* and migration of the operators including their state $(Overhead_{mig.(\omega_j)})$. Formally, it is defined as follows:

$$C_{Overhead}(T_i) = Overhead_{select} + Overhead_{\alpha} + \sum_{j=1}^{num^j} Overhead_{mig.(\omega_j)} \ . \tag{3}$$

The transition problem in this paper, therefore, is to minimize a weighted sum of normalized values[4] transition time $(\hat{C}_{Time}(T_i))$ and transition overhead

[4] using mean normalization method.

$(\hat{C}_{Overhead}(T_i))$ in order to meet the QoS demands under the execution of transitions as stated below:

$$\begin{aligned}
&\min \Big[w_t * \hat{C}_{Time}(T_i) + w_o * \hat{C}_{Overhead}(T_i) \Big] \\
&s.t.\ \alpha(t) \text{ satisfies } QoS_{|en(t)} \text{ under the execution of } T_i \qquad (4) \\
&C_{Time}(T_i), C_{Overhead}(T_i), QoS_{|en(t)} \in \mathbb{R}^+ \ .
\end{aligned}$$

Here, $w_t,\ w_o \geq 0,\ w_t + w_o = 1$, denote weights for transition time and overhead, respectively.

5. The TCEP System Design

Conceptual Overview. The four key components of the Tcep system are represented in Figure 6. The *IoT resources* layer includes event consumers, which can pose queries with specific QoS demands; event producers, which generates continuous data streams that are to be processed; and event brokers, which process the data streams to derive results. The *TCEP engine* layer provides a programming environment to create and execute queries and OP mechanisms on the infrastructure of the IoT resources layer. Moreover, the Tcep engine provides mechanisms for monitoring the performance of the OP mechanism and the environmental conditions. The *TCEP control* layer utilizes and manages a library of state-of-the-art OP mechanisms in a so-called placement library. Here, the Transition engine decides when and how to perform a transition. It is also responsible for coordinating the transition, i.e., performing operator migrations building on the proposed transition execution algorithms.

Furthermore, the placement performance evaluator decides which placement mechanism to select for a transition. The deploy operator graph component performs the deployment of the operator graph on the infrastructure resources. Finally, *Managed Resources* represent the resources monitored and controlled by the Tcep system, such as environmental conditions, performance metrics, and OP mechanisms.

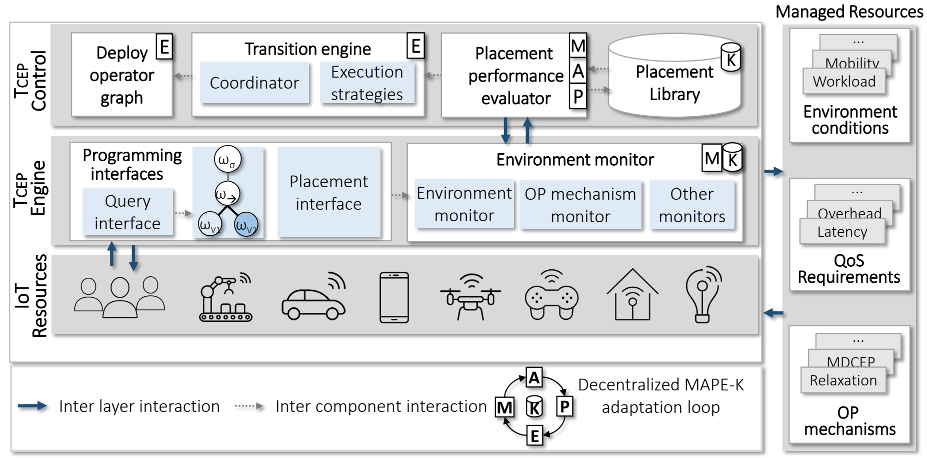


Figure 6: The TCEP system design.

A brief overview on the execution of a continuous query and its adaptation using transitions in TCEP is presented as follows. An event consumer poses a query using the query interface of the TCEP *engine* programming interfaces. The query is then transformed into an operator graph. The placement interface could be used to develop an OP mechanism. After transforming the query into an operator graph and selecting an OP mechanism, the placement performance evaluator deploys the query on the IoT resources based on the previously collected statistics on the query. If the environment monitor finds a change in the environmental conditions or the QoS requirements, a transition is triggered. The transition engine manages the adaptation, and the operator graph is redeployed by the deploy operator graph component.

Decentralized MAPE-K adaptation loop. TCEP follows the well-known *MAPE-K* [27] loop for adaptation. 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 6) 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 for the TCEP system.

Monitor (M): This function provides mechanisms to collect, aggregate, filter, and report details on the managed resources. Examples of monitoring information are environmental conditions such as mobility of cars and workload, performance metrics related to a query such as QoS metrics latency and bandwidth observed on the links, and performance metrics related to the transition of an OP mechanism: transition time and overhead. Hence, the decentralized monitoring components lie within the environment monitor and placement performance evaluator, responsible for collecting and aggregating the above monitoring information.

Analyze (A): This provides mechanisms that correlate and model complex situations. 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 operator placement for the respective environmental conditions.

Plan (P): It provides mechanisms that construct the actions needed to achieve the goals and objectives. For instance, the placement performance evaluator determines if a change to a new OP mechanism would help fulfil the QoS demands.

Execute (E): It provides mechanisms to manage the necessary changes required for the adaptation. It is responsible for carrying out the transition itself. For instance, the transition coordinator generates a plan on the operator graph transition, and the transition engine performs the transition.

Knowledge (K): The data shared across the above four functions are stored as shared knowledge. This includes OP mechanisms in the placement library, monitoring information on the performance, among others.

In the following sections, we will focus on four research questions, namely:

RQ 1 How to specify an operator placement and its performance characteristics?

RQ 2 How to adaptively select an OP mechanism for a transition?

RQ 3 How to realize transitions in a seamless manner?

RQ 4 How to decide when to perform a transition?

Structure. The following sections detail on the functionality of the aforementioned MAPE-K processes handled by the different components of TCEP in a decentralized manner. Section 5.1 presents (RQ 1) a programming model for specifying QoS demands in a query and OP mechanisms. Section 5.2 presents (RQ 2) the *genetic learning* algorithm for an adaptive selection of OP mechanism such that the QoS demands are satisfied. Section 5.3 addresses (RQ 3) and (RQ 4) by presenting seamless and concurrent execution of a transition while considering a minimal state for a cost-efficient transition.

In Figure 6, Section 5.1 is illustrated as the Programming interfaces component, Section 5.2 as Placement performance evaluator, and Section 5.3 as Transition engine.

5.1. Programming Model

The programming model provides a means for developers to implement novel OP mechanisms while utilizing IoT resources. Existing works [13, 28, 29] focus on proposing OP mechanisms for a diversity of QoS demands. However, none of them provides a common API for the development of novel OP mechanisms[5]. This section introduces the major components of the TCEP programming model: *(i)* QoS monitors that is an integral part of the programming model as each OP mechanism observes some QoS metrics (cf. Section 5.1.1), and *(ii)* OP interface that provides methods to develop unique OP mechanism.

5.1.1. QoS Monitors

As prominently discussed in the literature [29, 28, 30], our programming model characterizes the existing OP mechanisms based on the placement decision into two main categories: *(i)* centralized and *(ii)* decentralized. A centralized OP mechanism assumes global knowledge on the network and the nodes (specific QoS demands) to host an operator on a physical node. In contrast, a

[5] For a detailed discussion on related work, we refer the readers to Section 7.

Method	Description
`getPlacementMetrics()`	Determines the QoS demands that must be optimized
`configurePlacement()`	Resets placement parameters. It is called initially and on reconfiguration
`findPlacementNode()`	Finds placement node determined based on the QoS metrics
`findPossibleNodes()`	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 space

Table 2: Tcep placement API for developing OP mechanisms.

decentralized mechanism assumes only partial knowledge of the network, and hence the placement decision is decentralized. 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 [31]. Furthermore, the assignment varies with the QoS demands in consideration for the cost objective function. Hence, there exist many solutions and heuristics towards the OP problem.

Both kinds of placement heuristics assume monitoring knowledge on the network and host information. The Tcep programming model provides explicit extensible monitors for commonly used network and host 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 centralized OP mechanisms, the QoS monitors transfer the observed metric to a centralized node responsible for the placement decision. While for decentralized mechanisms, we provide decentralized monitoring solutions such as Vivaldi [32], which is prominently used in several OP mechanisms [8, 9, 33, 13] that handles the dissemination of monitoring information for placement decision.

5.1.2. OP Interface

Table 2 lists the foremost API of the TCEP programming model used to implement OP mechanisms in TCEP. `PlacementStrategy` API defines these methods for OP mechanisms in order *(i)* to formulate a single objective and multi-objective optimization function for centralized OP mechanisms, *(ii)* to define heuristics for decentralized OP mechanisms, and finally, *(iii)* to make OP mechanisms exchangeable at runtime to enable transitions.

An OP mechanism needs to represent a cost objective function dictating the QoS demands. An example of a cost objective function is to minimize the end-to-end latency from the producers to the consumers. Each mechanism, centralized and decentralized, must define a cost objective function for the QoS demands that need be optimized. The cost objective function can comprise a single or multiple QoS demands, 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 IoT resources. `getPlacementMetrics()` method is used (cf. Table 2) to fetch monitoring information related to the objective function. Consequently, this helps in formulating the cost function. The specific way of solving the placement problem (optimally or sub-optimally) using heuristics is defined in the specific implementations of the OP mechanisms.

In Table 3, we define the currently available implementations of OP in TCEP. We define the heuristic approaches used by the respective OP mechanism–for example, the Relaxation mechanism [8] uses a spring relaxation technique, while the MOPA mechanism [9] uses an approximation for the Weber problem, though both aim for the same QoS metric: bandwidth-delay product. Also, in optimal solutions of OP, the optimization problem can be solved using different methods. The heuristic used also varies based on the nature of the objective function (convex or concave) and the scenario at hand. Hence, the in TCEP programming model, we segregate the implementation of a specific optimization approach of the OP mechanism from the common interfaces.

OP Mechanism	Placement Decision	Optimization Goal	Approach	§ 6.1.1
Relaxation [8]	Centralized	bandwidth-delay^2 product (BDP)	Spring relaxation technique	(1)
MOPA [9]	Centralized	bandwidth-delay product	Approximation for Weber Problem	(2)
Global Optimal	Centralized	bandwidth-delay product	Optimally finds node with minimum BDP	(3)
MDCEP [6]	Decentralized	control message overhead, latency	Place on nearest neighbours unless producer or consumer	(4)
Producer-Consumer	Decentralized	hops	Always host on the producer or consumer	(5)
Random	Decentralized	-	Random allocation	(6)

Table 3: Design space of OP mechanisms.

The placement parameters are initialized using the `configurePlacement()` method, which is invoked in the beginning and each reconfiguration, e.g., during periodic updates of the same OP mechanism. `findPossibleNodes()` and `findPlacementNode()` methods determine the possible nodes where the operator can be deployed depending on the cost function and the optimal or sub-optimal (depending on the placement mechanism) node for the deployment, respectively. Some centralized mechanisms behave differently when performing OP initially and on reconfiguration, such as the Relaxation [8] mechanism. This mechanism places all operators of the query at once based on the virtual coordinate space using `initialVirtualOperatorPlacement()`, and the physical placement is performed using `findHost()` since no operator is physically deployed only using virtual placement. However, on reconfiguration, only the physical placement is changed. In contrast, decentralized mechanisms only implement the `findHost()` since their behaviour is the same during initial placement and transitions. Having understood the functionality of the programming model and the monitors of the TCEP *engine* layer, we detail the placement performance evaluator component in the following subsection.

5.2. Placement Performance Evaluator

This component measures the performance of the OP mechanisms continuously and analyze their behavior. A *lightweight online learning* algorithm is employed to statistically determine which mechanism best meets the QoS demands, building on a selection strategy of genetic algorithms [34]. *Lightweight* refers to the fact that learning does not rely on any training set but only uses statistics collected online during the execution. This component uses the online learned model to select an appropriate OP mechanism with the best performance based on the ranking provided by the learning algorithm. The environment monitor component keeps track of the performance behaviour (QoS demands and environmental conditions via QoS monitor and other monitors, respectively) and reports any changes to this component – e.g., if the QoS demand specified in the query is violated. When no empirical statistics are available during initialization, the target placement mechanism is determined by comparing the respective QoS demand with the specified optimization objective(s) of the placement mechanism. If more than one placement mechanism exists for the respective QoS demand, then the selection is performed in a round-robin fashion.

In the remaining section, we first define a heuristic fitness function to evaluate the performance of an OP mechanism during its execution. Then, we define an adaptive selection of an OP mechanism based on the observed statistics and the fitness function.

Heuristic Fitness Score for OP Mechanism.

We measure the performance of the current OP mechanism in execution for each continuous query at regular intervals. The collected performance statistics are then used for comparison between different OP mechanisms. To quantify the performance, we measure the fitness of each OP mechanism that is in execution per query. We define the heuristic fitness function with the objective to maximize the number of times an OP mechanism fulfils the current QoS demands. This means that if an OP mechanism fulfils QoS demands $x > max$ times between the time interval t_s (when the query was first submitted) and t_t (when the transition is triggered), then this mechanism is selected for the next execution. For each QoS demand, we update the fitness score at regular intervals until the next transition. The score provides information on how well the OP mechanism had performed over time, compared to the mechanisms that were in an execution before (when the query was first submitted). The goal is to find the best mechanism for the respective QoS demands by utilizing the collected statistical information. This goal is accomplished by maintaining the scores of the respective OP mechanisms $M_{i,qos_j}(t_t)$ for each QoS demand qos_j, and updating the score at the occurrence of a transition at time t_t. Since an OP mechanism can incorporate multiple QoS demands, for instance, in a multi-objective optimization function, the score is determined separately for each QoS demand. For each OP mechanism M_i, we maintain a score function $Score_{(M_{i,qos_j})}(t_t)$ obtained based on the evaluation of each QoS demand qos_j. The score $M_{i,qos_j}(t_t)$ is normalized for each OP mechanism M_i, based on the *mean normalization method* to make the scores comparable. We compute the fitness score based on the statistics collected from executing OP mechanism i (with subscript i), which is then compared to other mechanisms executed from time t_s (when the query was first submitted) until time t_t (when the transition is triggered), given as $t_{s,t}$:

$$\begin{aligned} M_{i,qos_j}(t_t) = \frac{\mu_{i,qos_j}(t_{s,t}) - \mu_{qos_j}(t_{s,t})}{max_{qos_j}(t_{s,t}) - min_{qos_j}(t_{s,t})} \cdot (1 - decay) + \\ M_{i,qos_j}(t_t - 1) \cdot decay \ . \end{aligned} \tag{5}$$

In Equation 5, $\mu_{qos_j}(t_{s,t})$, $max_{qos_j}(t_{s,t})$, and $min_{qos_j}(t_{s,t})$ denote the mean, maximum and minimum score values for *all* the OP mechanisms, respectively, that have been used until time t_t considering the QoS demand j. $\mu_{i,qos_j}(t_{s,t})$ represents the mean score value of OP mechanism M_i until time t_t considering the QoS demand qos_j. $M_{i,qos_j}(t_t - 1)$ is the last score of OP mechanism M_i, and a *decay* factor is used to exponentially reduce the effect of old statistics to prioritize the data that is recently collected. The decay factor ranges of $[0, 0.5]$, such that more preference is given to current statistics. The initial value of decay is set to 0 , and it 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 of an OP mechanism is computed based on all the statistics collected on the QoS demands fulfilled by the OP mechanism. The score is the sum of the normalized scores for each QoS demand $qos_j \in [qos_1, qos_2, \ldots, qos_k]$, where k is the maximum possible QoS demands considered by OP mechanism M_i:

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

Adaptive Selection of OP Mechanism.

The adaptive selection of an OP mechanism is performed once each OP mechanism has been defined with a fitness score. We adopt the *Linear Ranking Selection Strategy* [34], a selection method from Genetic Algorithms (GA). The ranking based method is suitable for our OP mechanism selection problem since it allows us *(i)* to perform a relative analysis suitable for the heuristic fitness function that indicates which OP mechanism is better, and *(ii)* by an appropriate selection pressure it favours exploitation over exploration avoiding selecting worse OP mechanisms. More specifically, by only using the fitness values of the OP mechanisms, the linear ranking method selects the best OP mechanism for the given QoS demands, which is a perfect choice since our goal is to compare OP mechanism relatively. The selection pressure defines the intensity of search focused towards the best OP mechanism. By reducing the selection pressure, the diversity of the OP mechanism increases, while 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 given fitness values $f_1, \ldots, f_F, (F \leq \mathcal{N})$ as follows:

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

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

$$\mathcal{S} = \frac{\overline{\mathcal{M}^*} - \overline{\mathcal{M}}}{\overline{\sigma}} \ . \tag{6}$$

Here, $\overline{s}(f)$ and $\overline{s^*}(f)$ are the fitness distribution and expected fitness distribution of the OP mechanisms, respectively. The notation $\mathcal{N}$ denotes the size of fitness distribution and $\overline{\sigma}$ denotes the standard deviation of the fitness distribution $\overline{s}(f)$. All functions assumed to be continuous are denoted with an overline, and the fitness values for the OP mechanisms are assumed to be sorted $(f_1 < f \leq f_F)$ [35].

After OP mechanisms are sorted according to their fitness values, the ranks are assigned to them. Rank R is assigned to the best OP mechanism, while rank 1 is assigned to the worst. The selection probability P_i is linearly assigned according to the rank as follows:

$$P_i = \frac{1}{R}\left(\eta^- + (\eta^+ - \eta^-)\frac{i-1}{R-1}\right); i \in [1, R] \ . \tag{7}$$

In Equation 7, $\frac{\eta^-}{R}$ is the probability that the worst OP mechanism is selected and $\frac{\eta^+}{R}$ the probability that the best OP mechanism is selected. Since OP mechanisms in the placement library are constant during runtime, the conditions $\eta^+ = 2-\eta^-$ and $\eta^- \geq 0$ must be fulfilled. Also, note that all the OP mechanisms are ranked differently, i.e., they have distinct selection probability – although they can have the same fitness score [35]. The probability of the OP mechanism to be selected is proportional to its fitness function score. The worst probability and the best probability are calculated as the minimum and maximum of the probability distribution function η:

$$\eta_i = \frac{Score_{(M_i)}}{\sum_i Score_{(M_i)}} \ . \tag{8}$$

The selection of the mechanism means the inclusion of it in the reduced search space, which gives well-performing OP mechanism a higher probability than the lower ones, i.e., we prefer OP mechanisms that were classified to perform better (exploitation of the learning algorithm). However, sometimes we also select worse OP mechanisms to update their score (exploration). Assuming that the fitness distribution follows a Gaussian distribution, and using Equation (10), it can be proved (cf. Proof in Appendix A) that the selection pressure for the ranking method can be computed as follows:

$$\mathcal{S}_{\mathcal{R}}(\eta^{-}) = (1-\eta^{-})\frac{1}{\sqrt{\pi}} \ . \tag{9}$$

Once all the OP mechanisms get assigned a rank based on their performance for a query, the TCEP system can decide whether the currently running mechanism M_i should be used again or changing to another OP mechanism yields better performance. We use a simple Radix sort to rank the OP mechanisms in linear time so that the comparison is cheap. The complexity of the sorting dominates the complexity of the selection algorithm, i.e., $\mathcal{O}(\mathcal{N})$, where $\mathcal{N}$ is the size of the fitness distribution function of OP mechanisms. Furthermore, the following challenges 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. Furthermore, 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.

5.3. Transition Engine

The TCEP transition engine coordinates how a transition is performed over the life cycle of a transition [36], i.e., from its invocation to its completion. The two transition algorithms define the life cycle of a transition. This component, therefore, is a core of the TCEP system.

We first provide a high-level view of the requirements for the transition phase. A transition from one OP mechanism to another involves several distributed entities of Tcep. The transition execution must be coordinated such that it is consistently performed across these entities. Thus, the *transition coordinator* maintains and orchestrates the transition life cycle. Tcep currently supports two transition algorithms (detailed below). The difference in the life cycle of the proposed transition algorithms lies in the *seamlessness*, i.e., how smooth the transition is performed and how much is the cost in terms of time and overhead ($C_{Time}(T)$ and $C_{Overhead}(T)$) as defined below.

During the execution of a transition, the target OP mechanism determines a set of target brokers for the new placement. As a result, all the operators have to migrate to the target brokers to comply with the new placement logic. While the coordinator performs operator migrations, it must continue satisfying the QoS demands by the event consumers, which is the primary goal. Operator migrations in this realm have been widely studied in the literature, such as stop and restart strategies [37, 38] as well as partial pause and resume strategies [22, 39]. 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. To do this, we specifically look into costs associated with performing a transition in terms of time and overhead. The transition execution algorithm dictates how *cost-efficient* operator migrations are performed while fulfilling the QoS demands. Considering these requirements, we present two transition execution algorithms that *(i)* coordinate the transition, *(ii)* perform operator migrations while ensuring the correctness and completeness of the delivered *complex* events to the consumers, and *(iii)* perform the *live* and *seamless* transition.

Moving Fine-Grained State (MFGS) Sequential Transition.

In this algorithm, the transition coordinator initiates operator migrations in a specific order, i.e., in a bottom-up fashion (cf. Algorithm 1: Lines 1-14). This means an operator is only migrated after all its predecessors were successfully migrated. Here, the dependency of operators follows a bottom-up fashion, where leaf operators are predecessors of their successors or dependent operators as we go level up in the operator graph. The operator migrations are performed in a sequential and breadth-first manner one at a time to the target brokers (Lines 2-3).

In the next step, the coordinator determines the target broker with the help of the newly selected OP mechanism (Line 5). It is important to note that the target OP mechanism is predetermined by the placement performance evaluator component (cf. Section 5.2). Consequently, an operator ω may need to be migrated to a new target broker (Line 6-7). For operator migrations, a minimum state is extracted, which corresponds to the intermediate state discussed in detail in the next paragraph (Line 8). Afterwards, this state is sent to the target broker to start executing the operator with the minimum migrated state.

The target broker subscribes to its producers or predecessors to receive data streams starting from the time of reception of the intermediate state (Line 9). When the migration is complete, the target broker will send an acknowledgement, including the sequence number of the first output event to the source broker and the coordinator (Line 10). After the source broker has been acknowledged, it will stop its execution, and the target OP mechanism will continue at the target broker (Line 11). We start the transition at time t_i, to sequentially perform m operator migrations until the transition is completed at time t_e. The recursive function performs the operator migration by traversing bottom-up the operator graph (Line 12). If the operator migration is not successful for some reason– the IoT resource becomes unavailable– and the acknowledgement is not received, the process is repeated until a new target broker is found and the operator is migrated. (Line 14). In Line 14, we assume a consumer specified parameter m that determines the maximum number of repetitions[6] of this loop and guarantees termination after m tries.

Algorithm 1: Moving Fine-Grained State Sequential Transition.

```
Variables  :  OList            ←  bottom-up list of set of operators
              $\omega$                ←  current operator to be migrated
              producers        ←  list of producers connected to $\omega$
              targetMechanism  ←  target OP mechanism
              targetBroker     ←  target broker host of $\omega$
              $\phi_{Int}$              ←  intermediate state of $\omega$

function Init-MFGS-SequentialTransition()
    OList ← bottomUpAsList($\Omega$);
    MFGS-Sequentialalgorithm(OList.head, targetMechanism)
function MFGS-SequentialTransition($\omega$, targetMechanism)
    targetBroker ← targetMechanism.findtargetBroker($\omega$);
    if  targetBroker $\neq$ $\omega$.sourceBroker then
        $\omega$.copyExecutionEnvironment(targetBroker);
        $\phi_{Int.}$ ← $\omega$.computeIntermediateState();
        targetBroker.StartExecutionWithData(producers,
          $\phi_{Int.}$);
        if $\omega$.next().receivedACK(timeout, retries) then
            StopExecution($\omega$.sourceBroker);
            MFGS-SequentialTransition($\omega$.next(),
              targetMechanism);
        else
            MFGS-SequentialTransition($\omega$,
              targetMechanism);
```

Cost-efficient Operator Migrations. The TCEP transition engine computes the fine-grained computational state of an operator for *cost-efficient* operator migrations. We improve on the operator state model introduced in [21, 40] by proposing cost-efficient and seamless operator migrations such that minimal state is transferred at discrete time steps, which are optimal for costs as we explain in the following subsection in seamless and minimal state concurrent transition. Furthermore, our migration model considers dependencies between operators during migration, hence providing a means to migrate an entire operator graph consistently. In the operator state model (cf. Figure 7), the input events are cached in the input buffer (B_I) selected by the *selector* to map the output events determined by the correlation function of the operator (f_ω). Next, the *selector* handles the removal of events from the input buffer B_i when the

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

same are either consumed or discarded by the correlation function f_ω. The resulting output or complex events are stamped with a sequence number (SN) by the *sequencer* and appended into the output buffer B_O which is then forwarded to the ω's successor. The events which the successor operators have already acknowledged are removed from the output buffer B_O. Although the state model is applicable to many modern CEP systems, such as Apache Flink[7], which assumes the presence of buffers, with a few adaptations in the internal structures, it can be applied to other CEP system, e.g., those that do not assume buffers [41].

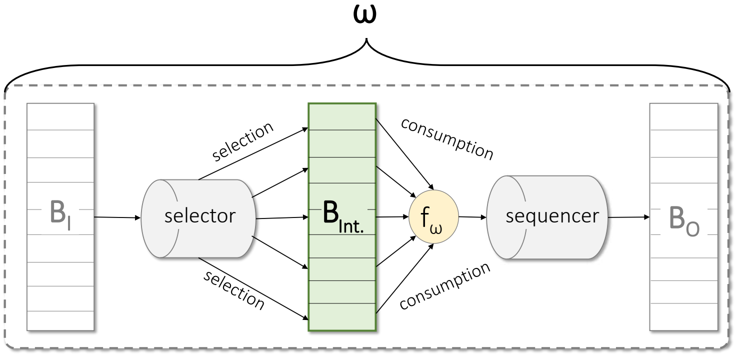


Figure 7: Intermediate buffer represented in the operator state model [40].

Conventionally, a CEP system transfers the internal state ϕ_ω that comprise the input buffer B_I, the *selector*, the correlation function f_ω, the *sequencer*, and the output buffer B_O. TCEP transfers the content of the intermediate buffer B_{Int} instead of the entire state ϕ_ω. The content of B_{Int} contains those events on which the correlation function f_ω is applied to obtain the complex events. For example, for a window-aggregate operator, the content of B_{Int} will be the events contained in the window, and those are selected to be aggregated by the correlation function (sum, min, or max). This set of events are updated each time the output events are generated, e.g., once the window slides (for a

[7] Apache Flink network stack. `https://flink.apache.org/2019/06/05/flink-network-stack.html` [Accessed on 18.4.2021]

sliding window operator) or the related event is either consumed (inserted into the output buffer B_O) or discarded by the correlation function.

The target broker must subscribe timely to the required incoming data streams to optimize the migration cost and completion time. Consider the intermediate state $\phi_\omega(t_i)$ of operator ω migrated at time t_i comprises B_{Int}, the correlation function f_ω, and the state of the sequencer (Line 9). Here, B_{Int} replays the events that were selected for correlation before the source broker went down (Line 9). At time $t_i - \delta_M$, the target broker subscribes to the input events from the producers or the predecessor operator. Here, δ_M is a small value to ensure that the target broker receives input events before the processing starts. All input events to the target broker until the source broker is executing are discarded (Line 10). It is important to note that a careful selection of δ_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.

Contrarily, if the δ_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 subsection of seamless minimal state concurrent transition.

Properties. 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 state ϕ_Ω and the time to iterate over all operators, which yields

$\mathcal{O}\left(|\Omega| + |\phi_{\Omega}|\right)$. Here, ϕ_{Ω} denotes the intermediate state of the set of operators Ω within the operator graph[8].

In this algorithm, we reduce the time required to perform an operator graph transition by transferring a minimum amount of state. However, the processing of an operator at the target broker does not occur unless the source broker is in execution. This means that while the selected state is being transferred (i.e., it is on the wire), some events sent to the target broker remains unprocessed. No output events are produced unless the intermediate state is transferred. Although with this transition algorithm, a minimum amount of state is achieved yet, state transfer involves costs in terms of time and resources. Another problem is the sequential transfer of operators. While sequential transfer does not consume much network resources, it is very time-consuming. To solve these issues, we propose a second transition algorithm.

Seamless Minimal State (SMS) Concurrent Transition.

In contrast to the above algorithm, this algorithm allows for more than one operator migrations simultaneously (cf. Algorithm 2: Lines 1-16). At each level $l = 0$ to m of the operator graph G, the coordinator triggers at most 2^l operator migrations (for binary operator graph) performed in a bottom-up fashion (Line 2). The benefit of concurrent operator migrations is perceived in the cost computation that is later analyzed in the properties of the algorithm. The operator migrations begin when the coordinator transfers the execution environment (Line 5). The coordinator determines an optimal time t_i for each operator ω when the operator state is minimal so that the transition consumes minimum resources (Line 7). For this, we assume the events follow a time order of arrival [37]. The selection of time t_i is such that for each operator ω, SMS algorithm waits until the operator ω is purged from its old state (Line 8), i.e., until B_{Int} and f_{ω} are purged from their old state.

[8] Ω 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.

Algorithm 2: Seamless Minimal State Concurrent Transition.

```
Variables : producers          ← list of the event producers
            OGlevel            ← operator graph level for migration
            targetMechanism    ← target OP mechanism
            targetBroker       ← target broker host of current operator
            $\phi_{sequencer}$ ← state of sequencer
            waitTime           ← time taken until current operator is purged
                                 from its state

function SMS-ConcurrentTransition(OGlevel, targetMechanism)
    for all $\omega \in$ OGlevel do in parallel
        targetBroker ← targetMechanism.findtargetBroker($\omega$);
        if targetBroker $\neq$ $\omega$.sourceBroker then
            $\omega$.copyExecutionEnvironment(targetBroker);
            NTPClockSynchronization,(targetBroker,
              $\omega$.sourceBroker);
            minimalStateTime ← $\omega$.determineMinimalStateTime();
            waitTime ← waitUntil(minimalStateTime));
            $\phi_{sequencer}$ ← $\omega$.lastSN;
            targetBroker.StartExecutionWithData(producers,
              $\phi_{sequencer}$);
            targetBroker.determineReferencePoint
              (minimalStateTime);
            if $\omega$.parent().receivedACK(timeout, retries) then
                StopExecution($\omega$.sourceBroker);
                SMS-ConcurrentTransition(OGlevel.next(),
                  targetMechanism);
            else
                SMS-ConcurrentTransition(OGlevel,
                  targetMechanism);
```

For example, in a window-aggregate operator, the target broker waits until the last event of the window is processed, $w + \delta_S$, where w is the window size, and δ_S is a small value to ensure that t_i is greater than any time instant of input events to the source broker. Time t_i is chosen as the *transition start time*. We call this time the *minimal state* time of an operator ($t_{imin}(\omega)$). The target broker starts its execution with the minimal state (the last SN) simultaneously at the *transition start time*, while the successor operators at the higher level are still under execution by the former OP mechanism. Thus, in this algorithm, the transition coordinator allows the execution of two OP mechanisms concurrently. This allows us to deal with the output disruption discussed as follows.

Seamless and Concurrent Operator Migrations. To explain the concurrent operator migrations, we refer to the operator graph from our example scenario in Figure 8. *Src* box refers to the placement of an operator at the source broker, and the *Trg* box refers to the placement at the target broker. Steps 3 and 4 show the B_{Int} buffer of the sequence operator with the event tuples being processed. The first step shows the initial placement, while the last one shows the final placement after migration. The concurrent execution of two OP mechanisms (cf. step 2 to 3 in Figure 8) enables seamless execution in this algorithm. However, migrations do not interfere with each other, while the operator network gradually transforms the placement (cf. step 4). The transition coordination is accomplished atomically in the TCEP transition engine.

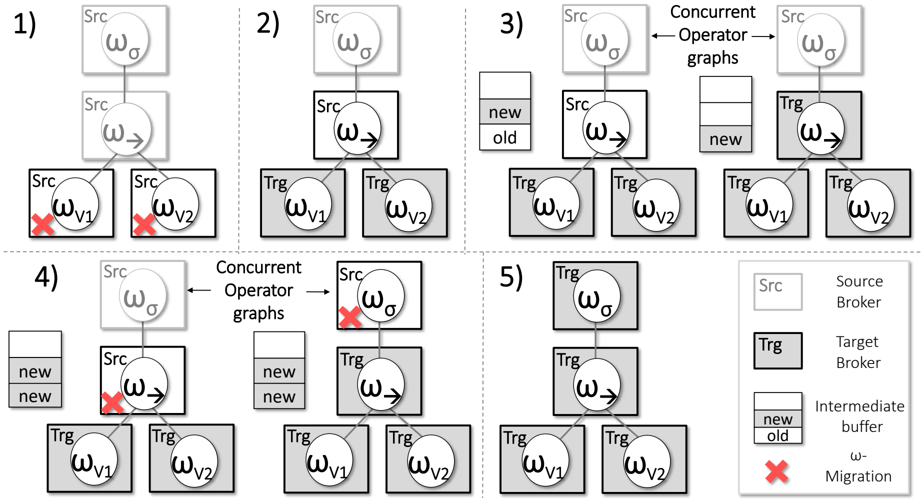


Figure 8: Sequence of operator migrations in the operator graph for SMS transition algorithm.

To better understand the cost of concurrent operator migrations, we analyze the reception of input events at both source and target brokers after transition start time t_i. For an operator ω, the state $\varphi_\omega(t_i)$ at transition time will only comprise the state of the sequencer (containing the SN of the first event to be produced at the target broker) (Line 9). The basic idea of this transition algorithm is that at the transition start time $t_{imin(\omega)}$, 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 ω run concurrently while 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 S events and slide size of 1 event, the old tuples still have to be retrieved until the target broker has received a full window size of S events. During this time, the output is continually produced by both the brokers, while duplicates are discarded using the reference point method later explained. 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, as illustrated in the figure with a red cross (✕).

To deal with the clock drift between the two clocks of the source and the target brokers, we perform distributed clock synchronization using standard Network Time Protocol (NTP) [23] at both ends (Line 6). To avoid duplicates in the output events due to concurrent processing, we use the reference point method [42] (Line 11). We treat the start timestamp of the results of the target broker as a reference point. Such timestamp is then compared to the *transition start time* t_i. If the reference point is larger than t_i, then the complex event is sent to the output buffer B_O.

Correctness of the results. We assess correctness on two aspects as widely done in the literature [37, 43]: the output is complete, and there are no duplicates in the output. Figure 8 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.

Properties. In this algorithm, we partition the transition at discrete time steps such that for each operator migration M_i, we determine the *minimal state time* as described before. This approach ensures a live and seamless transition without service disruption, thanks to minimal consumption of resources. Due to the concurrent transfer, the number of nodes in the new operator network increases exponentially over time with the increase of the size of the operator graph G. Therefore, the total transition time of this algorithm is within $\mathcal{O}\left(log(|\Omega|) + C\right)$, here $C = |\varphi_\Omega|$ that is constant (state of the sequencer) for a given set of operators Ω.

6. Evaluation

In the evaluation of TCEP, we aim to answer the following questions:

1. Is the programming model able to simply express existing operator placement mechanisms?

2. Does the mechanism transition concept satisfy changing QoS demands for dynamic environmental conditions?

3. Can a transition for the OP mechanism be performed in a live and seamless manner?

4. What is the cost involved in the execution of a transition, and is the cost acceptable?

To answer the above questions, we evaluate TCEP in four ways: *(i)* In Section 6.2, we evaluate the TCEP programming model in terms of the development

Number of producers	$1-9$
Number of brokers	$1-9$
Number of consumers	$1-2$
Number of queries	$1-50$
Type of queries	Stream (Q1), Filter (Q2), Conjunction (Q3), Join (Q4), Congestion detection (Q5) (Figure 10)
QOS_DEMANDS	latency, message overhead, network usage, hops
OP mechanisms	Relaxation [8], MOPA algorithm [9], MDCEP [28], Global Optimal, Producer-Consumer, Random
Transition execution algorithms	MFGS-Sequential, MFGS-Concurrent, SMS-Sequential, SMS-Concurrent
Placement selection algorithm	Genetic learning-based, Requirement-based

Table 4: Configuration parameters for the evaluation. *The default/mostly used parameters are underlined.*

of OP mechanisms and validating their performance. *(ii)* In Section 6.3, we evaluate the ability of Tcep to meet QoS demands with respect to latency and message overhead. *(iii)* In Section 6.4, we evaluate the stability of the system subject to transitions and the cost imposed by the distinct transition algorithms proposed in Section 5. *(iv)* In Section 6.5, we evaluate the costs of the genetic learning algorithm in terms of selection and performing a transition.

In the following sections, we first describe our evaluation execution environment, including details on the implementation of Tcep, the evaluation setup, and then present our evaluation findings.

6.1. Evaluation Environment and Setup

Implementation. The implementation of Tcep builds on an adaptive complex event processing system proposed in [19]. In particular, Tcep builds on the AdaptiveCEP programming model for specifying QoS demands at run time (cf. the query in Figure 1b). We provide the runtime environment based on the Akka actor system [44] and Akka Cluster to build a distributed network of containers for easy deployment in the edge-IoT scenario. The Docker container helps encapsulate a runtime environment to enable the deployment of operators on the IoT resources. Furthermore, we realized extensions in the form of a placement module that integrates state-of-the-art OP mechanisms [8, 6] and measure the resulting OP performance.

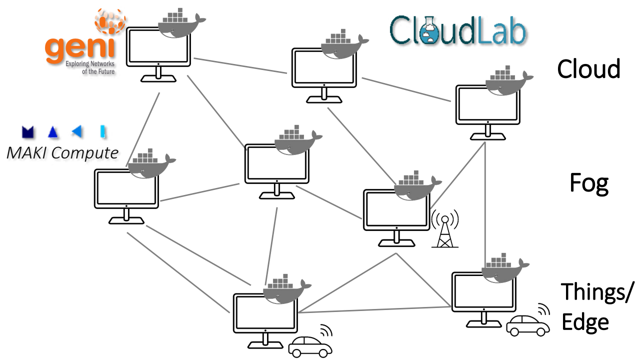


Figure 9: Our setup comprises 8 physical machines of publicly available network infrastructures running our virtualized DCEP system in docker containers.

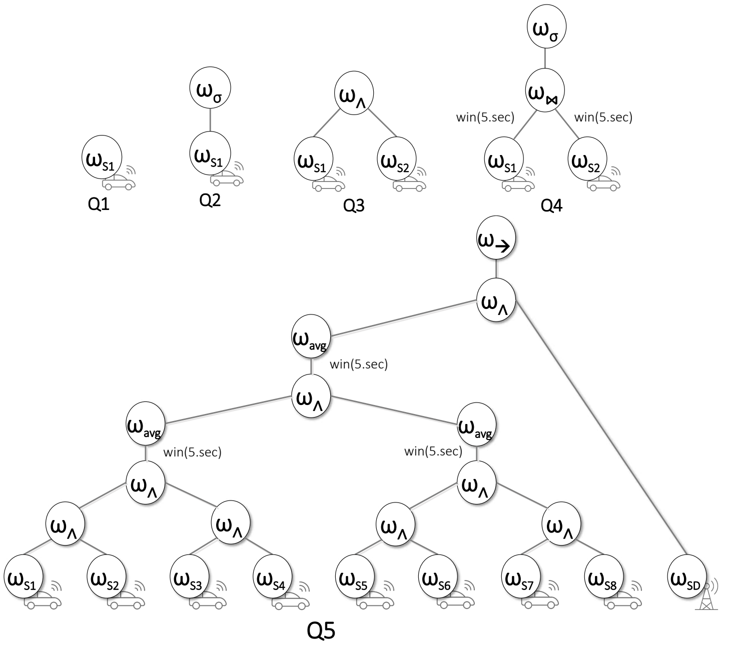


Figure 10: Operator graph for queries Q1 to Q5.

We build Tcep's Docker image upon the Alpine Linux distribution,[9] which is much smaller in size (base image size of only 5 MB) and lightweight than other

[9] Docker image upon Alpine Linux distribution. `https://github.com/gliderlabs/docker-alpine` [Accessed on 18.04.2021]

Linux based images. For instance, a standard Ubuntu docker image is 129 MB in size. Furthermore, the lightweight Docker-based execution environment is contained such that it does not exceed 2 GiB of allocated memory which is a reasonable assumption for small devices available nowadays as in the edge IoT scenario. The Docker containers communicate over an overlay network using TCP (Transmission Control Protocol) as an underlying transport protocol. We use Akka v. 2.6.0 [44], the Esper CEP engine v. 5.5.0 [45] and Docker v. 19.03.8-ce [46].

Platform and Setup. We deploy Docker services on 8 VMs with 8 GiB of memory and 8 processors per physical machine, as denoted in Figure 9. We consider different physical machines comprising of network infrastructures of Geni [16], CloudLab [17], and our onsite MAKI [47] compute machines. Together, these resources provide a realistic deployment environment similar to the IoT-fog-cloud infrastructure resource model introduced in Section 3.2 and hierarchically illustrated in Figure 9. With resources dispersed in North America (Ohio and UCLA) and Europe (Darmstadt), we have introduced geographical diversity and realistic network latencies, and packet loss environment for our experiments. The Docker network is setup based on the services that connect using an overlay network.

Queries. We use multiple standard CEP queries defined below[10] (cf. Table 4 and illustrated in Figure 10: Q1-Q4). Besides the standard CEP queries, we use a traffic congestion detection query presented in Section 2: Figure 1b. We elaborate on the query, such as the generation of complex data streams `vehiclesAtSectionV1` and `vehiclesAtSectionV2` for the average values related to speed and density. We illustrate the operator graph in Figure 10: Q5, comprising 8 publishers, each representing a Stream operator (ω_{S1} to ω_{S8}). In the operator graph, the speed information related to the vehicle from the Stream operators is analyzed to get the average speed of the two road sections. An-

[10] The queries are specified in AdaptiveCEP DSL in Scala programming language.

other Stream operator (ω_{SD}) contributes the density information related to the two road sections, which is combined to detect a sequence for the congestion detection using a Sequence-Filter operator ($\omega_{\rightarrow}$).

(i) `Stream` Operator

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

(ii) `Filter` Operator

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

(iii) `Conjunction` Operator

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

(iv) `Join` Operator

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

Dataset. We used a realistic dataset of the vehicular network scenario from Madrid [48] comprising the input data stream of the form $< time, position, lane, speed >$. This is used to generate complex data streams of `vehiclesAtSectionV1` and `vehiclesAtSectionV2` and evaluate further the congestion detection query. Similarly, for other queries as well the same dataset is used. We run each execution for 20 minutes and initiate the measurements after 2 minutes warm-up. Each measurement is taken at a regular interval of 5 seconds. For some evaluations, we incrementally increase the query workload for up to 50 queries. The evaluation metrics are influenced by multiple parameters such as the number of queries and the window size. To consider different environmental conditions, we perform a variability analysis on these parameters according to the ranges in Table 4.

6.1.1. Operator Placement Mechanisms

To understand how the performance of OP mechanisms, including those taken from the literature, differs in terms of QoS fulfilment, we implemented several OP mechanisms. In the following, we give a brief description of the design characteristics of the implemented mechanisms.

1. *Relaxation [8].* It is based on a so-called cost space that considers latency and bandwidth together as two dimensions. In the first step, the virtual operator placement is performed using the cost space, and in the second step, physical operator mapping is performed on the topology using KNN (K-nearest neighbours algorithm). The basic idea behind the first step, i.e., virtual operator 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 (Hooke's Law); the product of latency and bandwidth is the bandwidth-delay product (BDP). Note that Relaxation uses the squared latency 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, i.e., until the sum of forces inside the operator graph is zero. The operators are then mapped to the nodes closest to their respective virtual locations that are not overloaded. Through this heuristic, the overall BDP, i.e., the total amount of data in transit through the network at a given moment, is minimized, better known as network usage.

2. *MOPA Algorithm [9].* MOP is a variant of the Relaxation algorithm to minimize the bandwidth-delay product; hence instead of squared delay, this algorithm 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.

3. *Global Optimal.* Compared to the above two algorithms that find a suboptimal solution, we implemented a global optimal mechanism that chooses the best possible operator placement with minimum network usage (bandwidth-delay product) based on an exhaustive search of the possible placements. This OP mechanism requires global knowledge of the entire network.

4. *MDCEP [6].* The placement decision in MDCEP is made locally, and no cost information is shared among the nodes resulting in lower communication overhead and achieving a stable operator placement near the data sources. The authors consider a scenario of highly mobile nodes for operator placement. Hence, a decentralized mechanism with optimization criteria of minimizing message overhead and latency is considered.

5. *Producer-Consumer.* For comparison to the above approaches, we consider placement on the randomly chosen producer or consumer *only*. This is because the MDCEP mechanism considers stable operator placement that can be achieved by placing operators on producers or consumers where message loss can be minimal. Hence, this approach is also considered for comparison.

6. *Random.* This mechanism chooses a physical host for each operator randomly and serves as a naive comparison.

6.2. Performance of OP mechanisms

To understand the design space of OP mechanisms with distinct and conflicting optimization criteria, we evaluate them using the TCEP programming model presented in Section 5.1. We consider the QoS demands, queries, and OP mechanisms as stated in Table 4 for comparison. The performance metrics, including the QoS demands, are defined as follows: *(i)* Mean end-to-end latency or simply latency: It is the time taken from the query subscription was first received at the consumer end until the complex event was received back to the consumer (cf. Definition 3.5). *(ii)* Mean message control overhead or message overhead: The number of messages (in MB) exchanged to perform the operator placement. This includes establishing the broker network, exchanging network or node information for placement, and performing the placement (cf. Definition 3.6). *(iii)* Mean network usage: The amount of data in transit through the network given by the bandwidth-delay product as introduced in the Relaxation mechanism above. *(iv)* Mean number of hops: The number of hops or physical hosts used for an operator placement.

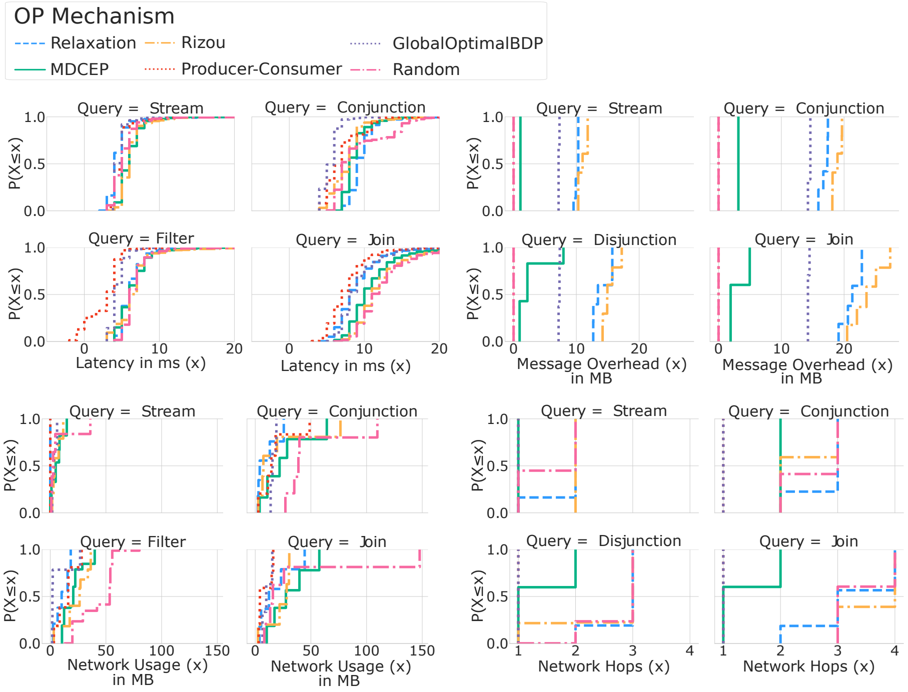


Figure 11: Performance evaluation of OP mechanisms (CDF) using programming model of TCEP in terms of latency, message overhead, network usage, and the number of hops for the standard CEP queries listed in Table 4. *Here, for all the metrics the more to the left is the distribution, the better it is.*

Figure 11 presents the cumulative distribution function (CDF) for the different QoS metrics using the given OP mechanisms (cf. Section 6.1.1) and standard CEP queries Q1 - Q4 (cf. Figure 10). It can be seen that each 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, 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 management 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 are focused 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 6.1) in the next paragraph.

Figure 12 presents the cumulative distribution function (CDF) for the different metrics using the given OP mechanisms while executing a traffic congestion query. 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. This further solidifies our belief that no mechanism can satisfy both the optimization criterion network usage and message overhead at the same time because these two are inherently conflicting. Table 6 in Appendix B summarizes the mean, minimum, maximum, and quantiles (90, 95, 99%) of the metrics latency and message overhead important for the considered scenario. The table presents the results for Q1, Q4, and Q5 execution using the different OP mechanisms. It can be derived from Figure 12 and Table 6 that Relaxation and MDCEP mechanisms stand representatives for the metrics latency and message overhead, respectively. Furthermore, as assumed in our hypothesis, there is no *one size fits all* mechanism [12].

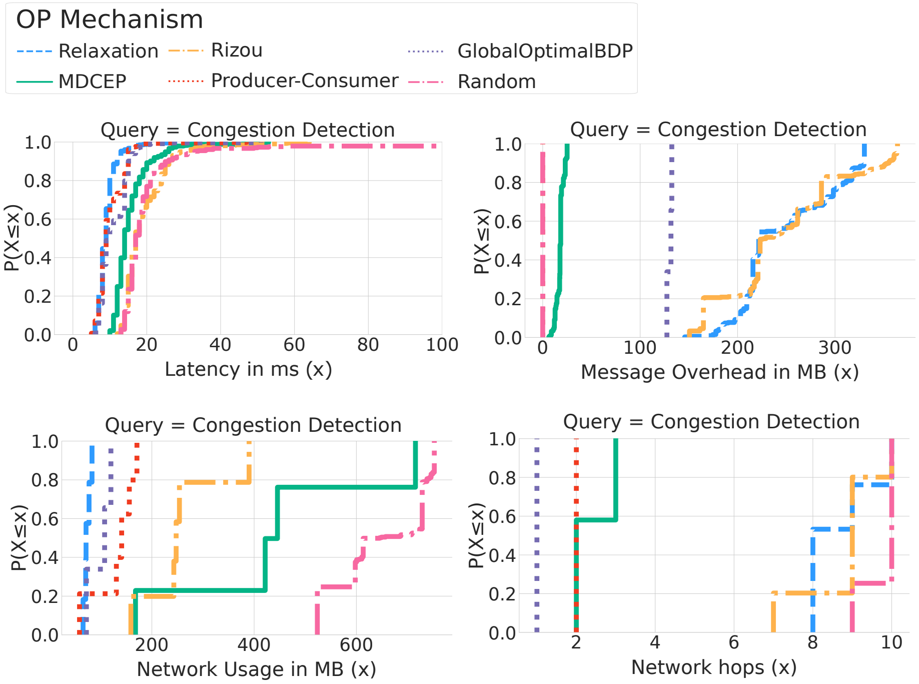


Figure 12: Performance evaluation of OP mechanisms (CDF) using programming model of TCEP in terms of latency, message overhead, network usage, and the number of hops for the congestion detection query. *Here, for all the metrics the more to the left is the distribution, the better it is.*

We have proposed mechanism transitions for such scenarios with dynamic environmental conditions and changing QoS demands. In the rest of the evaluation, we will focus on the two representative OP mechanisms, Relaxation and MDCEP and investigate the performance of mechanism transitions.

6.3. Performance of OP Mechanism Transitions

To understand whether the mechanism transition can fulfil changing QoS demands for dynamic environmental conditions, we evaluate the performance of

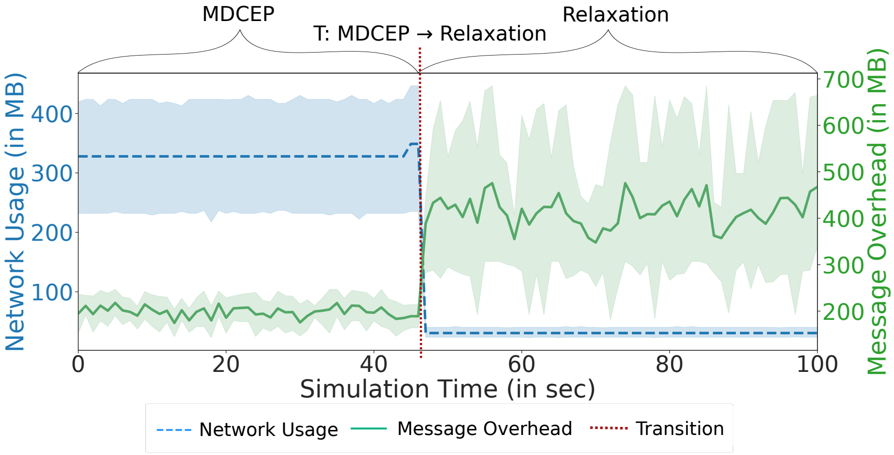


Figure 13: Network usage (y1-axis) and message overhead (y2-axis) measurement over a transition from MDCEP to Relaxation OP mechanism. TCEP system seamlessly transits to a fresh OP mechanism without incurring any overhead in terms of the specified QoS demands.

TCEP. We consider the following metrics: mean network usage (objective function for Relaxation), and mean control message overhead (objective function for MDCEP) as defined before. Furthermore, we consider a traffic congestion detection query for the rest of the evaluations because *(i)* it is representative of our scenario (Section 2), *(ii)* it captures the major standard CEP operators, *(iii)* and since for this query, we have observed significant variation in the performance of the OP mechanisms as shown in Figures 11 and 12. Figure 13 shows mean network usage on the first y-axis and control message overhead on the second for 5 runs in TCEP. At around 45 seconds (shown with an arrow), we observe a change in QoS demand from message overhead to network usage. TCEP handles this by executing a dynamic transition between MDCEP to Relaxation. TCEP triggers a transition automatically by selecting an appropriate placement mechanism that fulfils the QoS demand. It is noticeable that TCEP does not induce any interruption or costs in terms of optimized metrics and while performing a transition to a completely new OP mechanism. We further investigate the transition cost for the different algorithms for transition and selection of placement mechanism in the next sections.

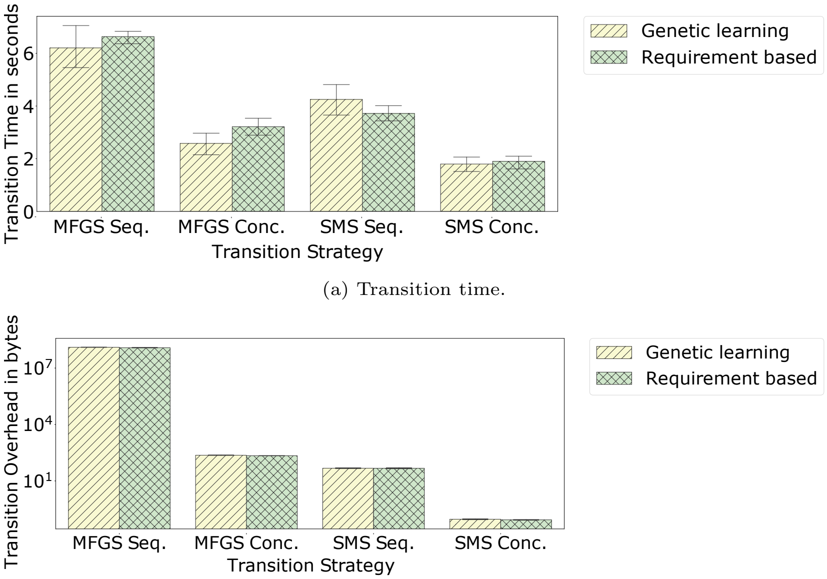
This section aims to understand how far the transitions are disruptive and the imposed cost in performing the transitions. In the evaluation, we consider: *(i)* the output event rate, *(ii)* the required time for the transition, and *(iii)* the transition overhead. To evaluate the transition execution algorithms reasonably, we extend Algorithm 1 to migrate the operators concurrently. Similarly, we extend Algorithm 2 to migrate the operators sequentially. The four approaches are enlisted in Table 4. Furthermore, we increase the query load to up to 10 queries to impose changes in the environmental conditions that trigger transitions in the TCEP system.

Cost of transition algorithms with learning-based selection

We analyze the cost of the different transition algorithms proposed in Section 5.3. The transition algorithm works together with the learning algorithm responsible for selecting the OP mechanism for a transition.

Besides the different transition algorithms, we implemented a requirement-based algorithm that selects a placement mechanism by matching the QoS demand with the optimization criteria for comparison with our learning algorithm. If there exists more than one mechanism matching the QoS demand, 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 the design section.

Figure 14 shows the transition time (a) and overhead (b) incurred by the transition algorithms using different selection algorithms. Here the costs of transition include the cost in time and overhead as represented earlier in Equation (2) and Equation (3), respectively, in Section 4. 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, while the SMS algorithms optimize for the minimal amount of state transfer (cf. Equation (4)). There is a substantial improvement in the transition time between



(a) Transition time.

(b) Transition overhead.

Figure 14: The plot shows the transition cost in terms of time and overhead for the proposed transition algorithms. SMS transition algorithms require minimal 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 algorithm. Moreover, the SMS Concurrent algorithm has only a negligible overhead of 0.72 bits, thanks to the cost-optimal algorithm (cf. Section 5.3).

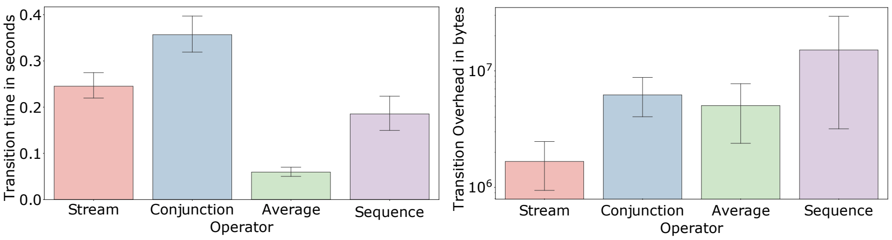
Sequential and Concurrent algorithms. This is because operators are migrated concurrently, which leads to lesser transition time. Finally, using the SMS Concurrent algorithm, we achieve an effective mean transition time of 1.82 seconds for the load of 10 complex congestion detection queries involving multiple stateful operators. We see an effective reduction of around 4 seconds compared to MFGS Sequential transition algorithm that takes 6.29 seconds to finish a transition with state transfer. The only cost parameter involved in SMS transition algorithms is in terms of selection of the OP mechanism and transition coordination costs in terms of communication between the distributed nodes. This is because there are no costs involved in terms of state migrations.

In the second plot, we observe the total transition overhead in terms of selecting an OP mechanism, transition coordination, and operator migrations due to transition (Equation (3) in Section 4). In consistent with the transition time, we observe a lower overhead of SMS algorithms due to the low amount of state involved in 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× better and the latter is 5× better than the MFGS Concurrent algorithm.

A second observation from these plots is that the genetic learning-based selection algorithm equally performs like a requirement-based algorithm because of no training and minimal learning costs involved. In Section 6.5, we elaborate on the learning costs of the algorithm. In conclusion, our results show that the SMS-Sequential and Concurrent algorithms perform better in both transition time and overhead, with the time within a range of $0.85 - 2.83$ seconds (for 10 queries) in comparison to 35 seconds (if the transition is performed naively using the stop and start migration algorithm) for the congestion detection query. We analyze costs per operator in the next section.

Cost of transition algorithms for different operators

In this section, we analyze the cost incurred by the transition algorithms in detail. The transition time comprises operator migration time and the time an operator has to wait for migration until the predecessor starts its operation at the target broker (cf. Section 5.3). For example, 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 *(i)* first and foremost 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 acknowledgements (cf. Section 5.3: Algorithm 1 and Algorithm 2). Stateful operators have costs in both dimensions, communication as well as migration costs depending on the transition algorithm – MFGS or SMS, while stateless operators like Filter and Stream do not have any state migration costs, but do have a small communication cost again depending on the transition algorithm – Sequential or Concurrent.



(a) Transition time for the different operators in congestion detection query.

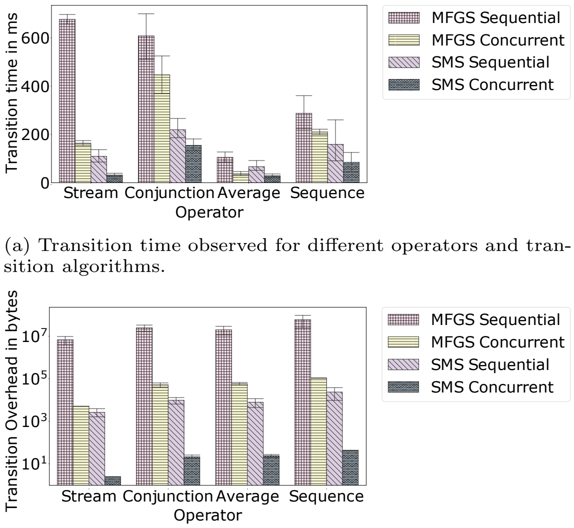
(b) Transition overhead for the different operators in congestion detection query.

Figure 15: Operator transitions are performed in the order of few milliseconds and with very low overhead using our transition algorithms.

Figure 15 shows the mean transition time (a) and overhead (b) using the transition algorithms for all the operators using 10 incrementally deployed congestion detection queries Q5 (cf. Figure 10). The total migration time correlates to the number of operators, and the transition state denoted as transition overhead in the second plot. It can be seen that the stateless operators like Stream, although high in number (90 operators), can be transited in 245.3 ms. 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. Table 7 in Appendix B summarizes the mean, minimum, and maximum values of the distribution.

Figure 16 classifies the transition costs further based on the transition algorithms. MFGS Sequential algorithm performs the worst clearly because of the high amount of transition overhead (mean value for Sequence operator 60.12 MB). In contrast, the SMS algorithms require a 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 5 shows the number of respective operators per congestion detection query and the total number of operators in a single run. The Conjunction op-



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

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

Figure 16: Transition time and overhead measurement for different operators for 10 incrementally deployed Q5 queries. SMS algorithms possess minimum migration time and overhead due to the minimal amount of transition overhead.

Operator	# pQ5.	# tot.
Stream	9	90
Conjunction	8	80
Average	3	30
Sequence	1	10

Table 5: Number of operators per congestion detection query (Q5 in Figure 10) and total in a single simulation run for 10 queries.

erator takes the highest amount of time to migrate, although the state involved is less due to a high number of operators involved. The same applies to the Stream operator. The number of Sequence operators to be migrated is less; however, it takes longer to transit due to the high amount of state (˜60 MB) transfer. Table 8 in Appendix B summarizes the mean transition time and overhead required by the different algorithms shown earlier in Figure 16.

In Figure 16, we analyze the cost per operator for the transition algorithms. In consistent with our findings in the previous section, with MFGS algorithms, operators take longer to migrate than SMS algorithms. The SMS Concurrent algorithm performs the best.

Seamless execution of transitions

To verify the seamless execution of transitions, we measured the throughput rate produced while TCEP's transition algorithms were executed (cf. Figure 17). A minor output disruption for MFGS-Sequential and Concurrent algorithms was observed in Figure 17 (around 0.02%). However, SMS-Sequential and Concurrent algorithms do not exhibit any disruption and continuously deliver output events with an output event rate of 100% for both the selection algorithms.

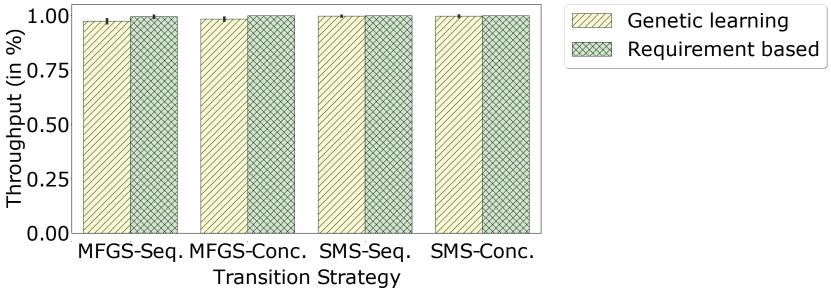
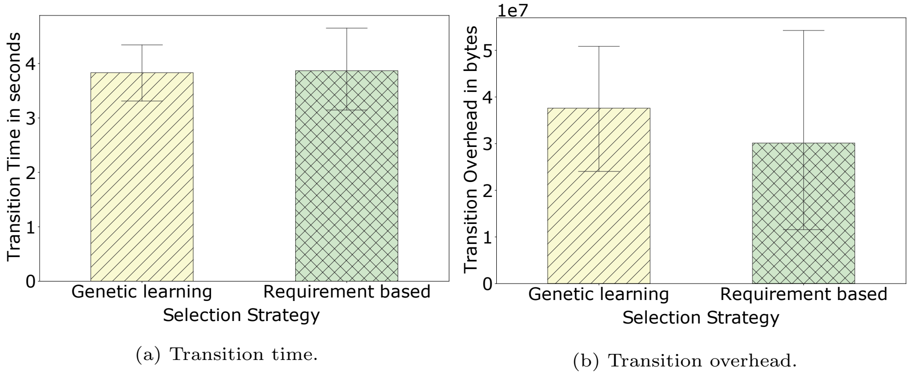


Figure 17: Throughput measurements using the different transition algorithms and selection algorithms for OP mechanisms. SMS transition algorithms consistently deliver output events enabling seamless execution of a transition.

6.5. Learning Costs of Placement Selection

This section aims to understand the learning costs of the adaptive placement selection algorithm introduced in Section 5.2. We consider the following metrics to determine 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 in 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 the local handling of operator placement selection.



(a) Transition time.

(b) Transition overhead.

Figure 18: Transition cost comparison with a requirement-based selection algorithm. The genetic learning-based algorithm does not impose high cost in terms of learning and is at par to the requirement-based algorithm.

Finally, to understand the influence of genetic learning-based selection algorithm on the performance mechanisms transitions, in Figure 18, we analyze the transition costs in time (a) and overhead (b). We compare the learning algorithm with a requirement-based algorithm where the selection of a mechanism is based on QoS demands.

From the figure, we observe that due to the negligible overhead of the genetic learning-based algorithm, the cost induced by it is comparable to the requirement-based algorithm. In fact, the transition time observed using the genetic learning-based is slightly less than the 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.

7. Related Work

It is highly important to fulfil QoS demands in a DCEP system for a wide range of application domains [49]. By enabling transitions, Tcep allows changing OP mechanisms, and in this way, fulfil QoS demands under dynamic environmental conditions. In this section, we analyze and compare related work in four key areas: programming models, operator placement and migration, adaptive event processing systems, and existing methods for mechanism transitions.

7.1. CEP Programming models

Many CEP languages have been developed in the past years, such as CQL [50], Cayuga [51], SASE [52], TESLA [53] for specifying complex events and detecting them by triggering notifications. Modern CEP programming models like Apache Flink [54], Heron [55], and Beam [56] provides extensive APIs to specify complex events for both batch and stream processing. Recently proposed benchmarking frameworks such as [57] and DCEP-Sim [28] unify CEP systems [58] and simulate CEP environment and operator placement, respectively. However, none of the above programming models has enabled the specification of distinct operator placement mechanisms in a heterogeneous environment of physical machines we do in this work. DCEP-Sim [28] has enabled the development of operator placement but only in a simulation environment. While we study the effect of distinct operator placement mechanisms, perform adaptations between them, and analyze the cost of adaptations in real-world network infrastructure and dynamic environment.

7.2. Operator Placement and Migration

OP mechanisms are widely studied to fulfil QoS demands while incurring minimum cost in performance [29, 28]. A wide range of OP mechanisms has been proposed considering different QoS demands, such as to achieve low latency [4], to minimize bandwidth [8, 9, 10], to lower message overhead [6], as well as to preserve trust and privacy [40].

The fulfilment of QoS demands, however, is only feasible under limited changes in environmental conditions. For instance, most existing work [12, 13, 4, 33, 9, 3] builds on stationary networks. Approaches considering dynamic changes, e.g., in the cause of mobility, introduces *(i)* redundancy by means of duplication [6] or checkpointing [59], *(ii)* placement update at regular intervals [8], or *(iii)* operator migrations when changing the placement [60, 61, 62, 40].

Overall, it is essential to note that current approaches for DCEP, so far, build on a *single* placement mechanism. In contrast, TCEP enables to benefit from adaptive use of *multiple* existing OP mechanisms by supporting transitions while increasing the range at which a DCEP system can adapt to meet a specific QoS demand.

Another critical mechanism that contributes to the cost in mechanism transitions is the operator migration mechanism. Operator migration has been extensively studied for data stream processing and complex event processing systems. Existing work can be characterized into the following three state migration strategies:
(i) Stop and restart. A naive way to proceed with state migration is to stop the execution of the source broker, safely transfer the state, and start the execution on the target source broker. Such strategies were used in early stream processing systems like CAPE [37] for dynamic query plan migration or runtime query optimization. Moreover, this method is most commonly used across fault-tolerance mechanisms, such as global state checkpoints. It is widely used by modern stream processing systems like Spark [38] and Flink [63].
(ii) Partial pause and resume. In the face of dynamic environmental conditions, streaming systems only have to migrate state for stateful operators, and hence stopping the entire streaming system is not necessary. This approach was introduced by a streaming system called Flux [64], which was later adopted and improved by multiple streaming systems, including StreamCloud [22], Chi [39], Seep [65], and FUGU [66] that only pauses the stateful operator in the operator graph.
(iii) Seamless migration. After our initial work on seamless operator migration for transitions [15], several other authors addressed similar concerns for different problems for state recovery [40], state migrations in streaming sys-

tems [67, 43, 68]. In contrast to the above mechanisms, we aim towards a cost-efficient transition capable system that integrates and benefits from multiple OP mechanisms through operator migrations.

7.3. Adaptive Event Processing Systems

In this section, we review approaches that have so far considered the adaptive exchange of mechanisms in the context of event processing systems. For example, Weisenburger et al. [19] proposed AdaptiveCEP, a programming model and CEP system that supports specifying QoS demands at run time. This work is complementary to Tcep since AdaptiveCEP does not focus on the adaptive selection and execution of transition strategies. However, in Tcep, the query language is used to specify changes in the QoS demands to instantiate a transition.

Heinze et al. proposed an elastic data stream processing system (DSPS) [66], where the number of active hosts can be scaled up and down, and operator migration is coordinated accordingly. Later, authors utilized an online learning approach [69] for auto-scaling. Based on this work, the same authors proposed an adaptive replication scheme for DSPS [70] that performs adaptation at runtime between active replication and upstream backup schemes for fault tolerance. Furthermore, the authors looked into the trade-off between monetary costs against the offered QoS [71]. Similar to the work of Heinze et al. [70], Martin et al. [72] also looked into the trade-off of active vs passive replication techniques for a fault-tolerant and elastic stream processing system. Proactivity in elasticity was proposed by Matteis et al. [73] using the *Model Predictive Control* method, which accounts for system behaviour over a future time horizon to predict the best reconfiguration to be executed.

Several surveys on elasticity [74, 72, 75, 76] highlight the importance of adaptivity of a streaming system towards the changing workload in terms of adding and removing resources on runtime. For instance, Lorido et al. [74] argue that the auto-scaling process in elastic streaming systems resembles the MAPE loop for autonomous systems similar to our work. Assunção et al. [76] discuss the advantages of the online approach over static approaches in adaptivity. Finally, Röger et al. [75] highlight the importance of distributed elasticity solutions using multiple operator approaches.

Furthermore, adaptivity in the OP mechanism has been investigated before. Aniello et al. [77] proposed an adaptive online scheduling algorithm for Apache Storm using two placement mechanisms. Sutherland et al. [78] developed an adaptive scheduling selection framework for continuous queries in DSPS. Liu et al. [79] advance the work on state migration to look into the problem of colocating stateful and stateless operators.

Although the aforementioned approaches benefit from integrating multiple mechanisms, the adaptation between the mechanisms is heavily dependent on the internals of the specific mechanisms in use. Therefore, integrating new alternative mechanisms is a complex task. By offering the abstraction of a transition, TCEP is highly extensible and can easily integrate new mechanisms. Furthermore, no previous work up today has studied the idea of adapting between distinct OP mechanisms.

7.4. Mechanism Transitions

The idea of mechanism transitions origins from the collaborative research centre MAKI, in which researchers investigate mechanism transitions for the Future Internet [18]. Within MAKI, mechanism transitions are investigated in the context of a wide range of communication mechanisms [80, 81, 82, 36, 83]. For example, in publish-subscribe systems, mechanism transitions between filtering schemes [80] and event dissemination mechanisms [81] are studied. Another line of work by Froemmgen et al. [84, 36] proposed transition strategies that always execute the best suitable search overlay. Richerzhagen et al. [83] recently proposed a transition-enabled monitoring service that executes transition on different monitoring mechanisms. Our work builds on and extends the concept of transitions proposed in prior work [80, 84]. By focusing on transitions for OP mechanism, our contribution is the design and understanding of transition strategies that can support highly dynamic and stateful mechanism transitions comprising many dependent distributed entities. The proposed strategies deal with the specific challenges for coordinated state migration as part of the SMS and MFGS transition strategies.

8. Conclusion and Future Work

In this work, we proposed TCEP, a transition-capable CEP system. TCEP is capable of dealing with changing QoS demands caused by dynamic network environment conditions. TCEP allows integration of state-of-the-art OP mechanisms using the programming model and dynamically executes the best matching OP mechanism to meet the QoS demands of IoT applications. To this end, we have explored how to perform transitions and analyzed their cost and performance. We proposed two transition execution algorithms for efficient migrations of operator state during a transition that is adaptively selected using the online learning algorithm. The learning algorithm possesses very low learning costs due to the online nature of performance analysis of operator placement mechanism. Our evaluation in the context of an IoT scenario and based on the state-of-the-art OP mechanisms shows that TCEP fulfils changing QoS demands by seamlessly performing transitions, i.e., without any output disruption.

The cost analysis shows that the transition execution time and overhead can be decreased to the range of $0.85-2$ seconds for the presented use case using our proposed transition strategies. Moreover, the learning cost of the lightweight selection algorithm proposed in this work is negligible. As future work, we consider trade-offs between different learning algorithms for the adaptive selection of an optimal learning algorithm.

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. We would like to express our gratitude to the anonymous reviewers for their detailed and very helpful feedback to improve this manuscript.

References

[1] Cisco, Cisco annual internet report (2018–2023) white paper, https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf, accessed 18.04.2021.

[2] TE-Connectivity, 6 key requirements of autonomous driving, https://spectrum.ieee.org/transportation/advanced-cars/6-key-connectivity-requirements-of-autonomous-driving, accessed 18.04.2021 (2020).

[3] M. Luthra, B. Koldehofe, R. Steinmetz, Transitions for Increased Flexibility in Fog Computing: A Case Study on Complex Event Processing, Informatik Spektrum 42 (4) (2019) 244–255.

[4] Y. Ahmad, U. 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.

[5] Y. Zhou, B. C. Ooi, K.-L. Tan, J. 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.

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

[7] R. Eidenbenz, T. Locher, Task allocation for distributed stream processing, in: Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM), 2016, pp. 1–9.

[8] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE), 2006, pp. 49–49.

[9] S. Rizou, F. Dürr, K. 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.

[10] B. Schilling, B. Koldehofe, K. 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.

[11] R. Dwarakanath, B. Koldehofe, Y. Bharadwaj, T. A. B. Nguyen, D. M. Eyers, R. Steinmetz, TrustCEP: Adopting a Trust-Based Approach for Distributed Complex Event Processing, in: Proceedings of the IEEE International Conference on Mobile Data Management (MDM), 2017, pp. 30–39.

[12] M. Nardelli, V. Cardellini, V. Grassi, F. L. Presti, Efficient operator placement for distributed data stream processing applications, IEEE Transactions on Parallel and Distributed Systems 30 (8) (2019) 1753–1767.

[13] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator replication and placement for distributed stream processing systems, SIGMETRICS Performance Evaluation Review 44 (4) (2017) 11–22.

[14] B. Alt, M. Weckesser, C. Becker, M. Hollick, S. Kar, A. Klein, R. Klose, R. Kluge, H. Koeppl, B. Koldehofe, W. R. Khudabukhsh, M. Luthra, M. Mousavi, M. Mühlhäuser, M. Pfannemüller, A. Rizk, A. Schürr, R. Steinmetz, Transitions: A protocol-independent view of the future internet, Proceedings of the IEEE 107 (4) (2019) 835–846.

[15] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, R. 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, p. 136–147.

[16] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, I. Seskar, GENI: A federated testbed for innovative network experiments, Computer Networks 61 (2014) 5–23.

[17] CloudLab, Cloudlab network infrastructure, `https://www.cloudlab.us/`, accessed 18.04.2021.

[18] B. Richerzhagen, B. Koldehofe, R. Steinmetz, Immense Dynamism, German Research 37 (2015) 24–27.

[19] P. Weisenburger, M. Luthra, B. Koldehofe, G. 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.

[20] S. Chen, J. Hu, Y. Shi, Y. Peng, J. Fang, R. Zhao, L. Zhao, Vehicle-to-everything (v2x) services supported by lte-based systems and 5g, IEEE Communications Standards Magazine 1 (2) (2017) 70–76.

[21] R. Dwarakanath, B. Koldehofe, R. 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.

[22] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, P. Valduriez, Streamcloud: A large scale data streaming system, in: Proceedings of the 30th International Conference on Distributed Computing Systems (ICDCS), 2010, pp. 126–137.

[23] D. L. Mills, Internet time synchronization: the network time protocol, IEEE Transactions on communications 39 (10) (1991) 1482–1493.

[24] B. Ottenwälder, B. Koldehofe, K. Rothermel, U. 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), 2013, p. 183–194.

[25] S. Chakravarthy, D. Mishra, Snoop: An expressive event specification language for active databases, Data and Knowledge Engineering 14 (1) (1994) 1–26.

[26] M. Luthra, B. Koldehofe, J. Höchst, P. Lampe, A. Rizvi, R. Kundel, B. 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, 2019, pp. 1–13.

[27] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[28] F. Starks, V. Goebel, S. Kristiansen, T. Plagemann, Mobile distributed complex event processing—ubi sumus? quo vadimus?, in: Mobile Big Data, 2018, pp. 147–180.

[29] G. T. Lakshmanan, Y. Li, R. Strom, Placement strategies for internet-scale data stream systems, IEEE Internet Computing 12 (6) (2008) 50–60.

[30] X. Liu, R. Buyya, Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions, ACM Computing Surveys 53 (3) (2020).

[31] U. Srivastava, K. Munagala, J. 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.

[32] F. Dabek, R. Cox, F. Kaashoek, R. Morris, Vivaldi, ACM SIGCOMM Computer Communication Review 34 (4) (2004) 15.

[33] V. Cardellini, V. Grassi, F. Lo Presti, M. 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.

[34] D. 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.

[35] T. Blickle, L. Thiele, A comparison of selection schemes used in evolutionary algorithms, Evolutionary Computation 4 (4) (1996) 361–394.

[36] A. Frömmgen, B. Richerzhagen, R. Julius, D. Hausheer, R. Steinmetz, A. Buchmann, Towards the Description and Execution of Transitions in Networked Systems, in: Proceedings of the Intelligent Mechanisms for Network Configuration and Security (IFIP), 2015, pp. 17–29.

[37] Y. Zhu, E. A. Rundensteiner, G. 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.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud), 2010, p. 10.

[39] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, S. Rao, Chi: A scalable and programmable control plane for distributed stream processing systems, Proceedings of the VLDB Endowment 11 (10) (2018) 1303–1316.

[40] R. Wermund, Privacy-aware and reliable complex event processing in the internet of things, Ph.D. thesis, Technical University of Darmstadt (2018).

[41] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso, I. Sourdis, Single window stream aggregation using reconfigurable hardware, in: Proceedings of the International Conference on Field Programmable Technology (ICFPT), 2017, pp. 112–119.

[42] J. V. d. Bercken, B. 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.

[43] B. Del Monte, S. Zeuch, T. Rabl, V. 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, p. 2471–2486.

[44] Akka, `http://akka.io/`, accessed 18.04.2021 (2009).

[45] EsperTech – Esper, `http://www.espertech.com/esper/`, accessed 18.04.2021 (2006).

[46] Docker – Community edition, `https://www.docker.com/community-edition`, accessed 18.04.2021 (2013).

[47] D. S. MAKI, Maki – multi mechanism adaptation for the future internet, `https://www.maki.tu-darmstadt.de/`.

[48] A. C. Torre, M. Fiore, C. Casetti, M. Gramaglia, M. Calderón, Bidirectional highway traffic for network simulation, in: IEEE Vehicular Networking Conference (VNC), 2017, pp. 77–80.

[49] M. A. Rahmani, L.-S. P. Preden, A. Jantsch, Fog Computing in the Internet of Things, Springer International Publishing, 2018.

[50] A. Arasu, S. Babu, J. Widom, The cql continuous query language: Semantic foundations and query execution, The VLDB Journal 15 (2) (2006) 121–142.

[51] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White, Towards expressive publish/subscribe systems, in: Proceedings of the Advances in Database Technology (EDBT), 2006, pp. 627–644.

[52] E. Wu, Y. Diao, S. Rizvi, High-performance complex event processing over streams, in: Proceedings of ACM International Conference on Management of Data (SIGMOD), 2006, pp. 407–418.

[53] G. Cugola, A. 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.

[54] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36 (4) (2015).

[55] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja, Twitter heron: Stream processing at scale, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2015, pp. 239–250.

[56] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al., The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proceedings of the VLDB Endowment 8 (12) (2015) 1792–1803.

[57] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, V. Markl, Benchmarking distributed stream data processing systems, in: Proceedings of the 34th International Conference on Data Engineering (ICDE), 2018, pp. 1507–1518.

[58] M. Luthra, B. 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, p. 7–12.

[59] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, M. 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.

[60] D. O'Keeffe, T. Salonidis, P. Pietzuch, Network-aware stream query processing in mobile ad-hoc networks, in: Proceedings of the IEEE Military Communications Conference (MILCOM), 2015, pp. 1335–1340.

[61] G. G. Koch, B. Koldehofe, K. 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.

[62] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, U. Ramachandran, MCEP: A Mobility-Aware Complex Event Processing System, ACM Transactions on Internet Technology 14 (1) (2014) 1–24.

[63] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, IEEE Data Engineering Bulletin 38 (4) (2015) 28–38.

[64] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, M. J. Franklin, Flux: an adaptive partitioning operator for continuous query systems, in: Proceedings of the 19th International Conference on Data Engineering (ICDE), 2003, pp. 25–36.

[65] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2013, pp. 725–736.

[66] T. Heinze, Z. Jerzak, G. Hackenbroich, C. 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.

[67] M. Hoffmann, A. Lattuada, F. McSherry, Megaphone: Latency-conscious state migration for distributed streaming dataflows, Proceedings of the VLDB Endowment 12 (9) (2019) 1002–1015.

[68] M. Luthra, S. Hennig, K. Razavi, L. Wang, B. Koldehofe, Operator as a service: Stateful serverless complex event processing, in: Proceedings of the IEEE International Conference on Big Data (Big Data), 2020, pp. 1964–1973.

[69] T. Heinze, V. Pappalardo, Z. Jerzak, C. 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.

[70] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, C. 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.

[71] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, C. Fetzer, Online parameter optimization for elastic data stream processing, in: Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC), 2015, pp. 276–287.

[72] A. Martin, A. Brito, C. Fetzer, StreamMine3G: Elastic and Fault Tolerant Large Scale Stream Processing, Springer International Publishing, 2018, pp. 1–10.

[73] T. D. Matteis, G. Mencagli, Proactive elasticity and energy awareness in data stream processing, Journal of Systems and Software 127 (2017) 302–319.

[74] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano, A Review of Autoscaling Techniques for Elastic Applications in Cloud Environments, Journal of Grid Computing 12 (4) (2014) 559–592.

[75] H. Röger, R. Mayer, A comprehensive survey on parallelization and elasticity in stream processing, ACM Computing Survey 52 (2) (2019).

[76] M. Dias de Assunção, A. da Silva Veith, R. Buyya, Distributed data stream processing and edge computing: A survey on resource elasticity and future directions, Journal of Network and Computer Applications 103 (2018) 1–17.

[77] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in storm, in: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS), 2013, pp. 207–218.

[78] T. M. Sutherland, B. Pielech, Y. Zhu, L. Ding, 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.

[79] S. Liu, J. Weng, J. H. Wang, C. An, Y. Zhou, J. Wang, An adaptive online scheme for scheduling and resource enforcement in storm, IEEE/ACM Transactions on Networking 27 (4) (2019) 1373–1386.

[80] B. Richerzhagen, N. Richerzhagen, J. Zobel, S. Schönherr, B. Koldehofe, R. 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.

[81] B. Richerzhagen, M. Schiller, M. Lehn, D. Lapiner, R. 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.

[82] B. Richerzhagen, S. Wilk, J. Rückert, D. Stohr, W. 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.

[83] N. Richerzhagen, P. Lieser, B. Richerzhagen, B. Koldehofe, I. Stavrakakis, R. Steinmetz, Change as chance: Transition-enabled monitoring for dynamic networks and environments, in: Proceedings of the 14th Annual Conference on Wireless On-demand Network Systems and Services (WONS), 2018, pp. 51–58.

[84] A. Frömmgen, S. Haas, M. Stein, R. Rehner, A. Buchmann, M. Mühlhäuser, Always the best: Executing transitions between search overlays, in: Proceedings of the European Conference on Software Architecture Workshops (ECSA), 2015, pp. 1–4.

Appendix A Selection Method for OP mechanism

Definition A.1. 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.

$$\mathcal{S} = \frac{\overline{\mathcal{M}^*} - \overline{\mathcal{M}}}{\overline{\sigma}} \tag{10}$$

In Equation (10), 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 A.2. 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 $\mathcal{G}(0, 1)(f) = \frac{1}{\sqrt{2\pi}}e^{-\frac{f^2}{2}}$

$$\mathcal{S}_{\mathcal{R}} = \int_{-\infty}^{\infty} f(\overline{R}^*)(\mathcal{G}(0, 1))(f)df \tag{11}$$

The effective and average fitness value of a Gaussian distribution with mean μ and variance σ^2 can be easily derived as $\mathcal{M}^* = \sigma\mathcal{S}_{\mathcal{R}} + \mu$.

Theorem A.1. The selection pressure using a linear ranking method can be derived as follows.

$$\mathcal{S}_{\mathcal{R}}(\eta^-) = (1 - \eta^-)\frac{1}{\sqrt{\pi}} \tag{12}$$

Proof. Using the definition of standardized selection pressure in Definition A.2 and the Gaussian function for the initial fitness distribution, one can obtain

$$\mathcal{S}_{\mathcal{R}}(\eta^-) = \int_{-\infty}^{\infty} x \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \left(\eta^- + 2(1 - \eta^-) \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right) dy\right) dx$$

$$= \frac{\eta^-}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x \exp\left(-\frac{x^2}{2}\right) dx + \frac{1 - \eta^-}{\pi} \int_{-\infty}^{\infty} x \exp\left(-\frac{x^2}{2}\right) \int_{-\infty}^{x} \exp\left(-\frac{y^2}{2}\right) dydx$$

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 (9) (and Equation (12)) follows. □

Appendix B Additional Insights into the Performance Evaluation

OP mechanism. In this section, we report additional insights into the performance of OP mechanisms analysed in Section 6.2: Figure 11. Table 6 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 4) execution using the different OP mechanisms.

It can be derived from Figure 12 and Table 6 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	quantiles (90, 95, 99)	mean	min	max	quantiles (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 6: Performance results of OP mechanisms Relaxation and MDCEP are among the best in comparison to its alternative mechanisms.

Transition cost per operator. Table 7 elaborates on the statistics of the transition cost for different operators presented in Section 6.4. It summarizes the mean, minimum and maximum values for the transition cost in time and overhead for the different operators in Q5: congestion detection query (cf. Figure 15). Intuitively, the stateful operators like Sequence require a higher amount of state to be migrated compared to stateless operators like Stream.

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 7: Mean, min and max values of transition time and overhead per operator for 10 incrementally deployed Q5 queries.

Transition cost for different transition algorithms. Table 8 elaborates on the statistics of transition cost for different transitions algorithms as presented in Figure 16. It summarizes the mean transition time and overhead required by the different algorithms. 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 8: Mean values for transition time and overhead for MFGS and SMS strategies. SMS strategies clearly supersede both time and overhead required to transfer the operator.