TCEP: Adapting to Dynamic User Environments by Enabling Transitions between Operator Placement Mechanisms

Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, Raheel Arif
Technical University of Darmstadt
Germany
{manisha.luthra,boris.koldehofe}@kom.tu-darmstadt.de
{weisenburger,salvaneschi}@cs.tu-darmstadt.de

ABSTRACT

Operator placement has a profound impact on the performance of a distributed complex event processing system (DCEP). Since the behavior of a placement mechanism strongly depends on its environment; a *single* placement mechanism is often not enough to fulfill stringent performance requirements under environmental changes. In this paper, we show how DCEP can benefit from the adaptive use of *multiple* placement mechanisms. We propose Tcep, a DCEP system to integrate multiple placement mechanisms. By enabling *transitions*, Tcep can *seamlessly* exchange distinct operator mechanisms at runtime. We make two main contributions that are highly important for a *cost-efficient* transition: i) a transition strategy for efficiently scheduling state migrations and ii) a lightweight learning algorithm to adaptively select an appropriate placement mechanism as a consequence of a transition. Our evaluations for important decentralized placement mechanisms in the context of an IoT scenario show that transitions can better fulfill QoS demands in a dynamic environment. Thereby, efficient scheduling of state migrations can help to faster complete transitions by up to 94 %.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Networks** → **Network dynamics**.

KEYWORDS

Complex Event Processing; Stream Processing; Operator Placement; Migration; Adaptation; Transitions; Internet of Things

ACM Reference Format:
Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, Raheel Arif. 2018. 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 '18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210284.3210292

DEBS '18, June 25–29, 2018, Hamilton, New Zealand

ACM ISBN 978-1-4503-5782-1/18/06…$15.00
https://doi.org/10.1145/3210284.3210292

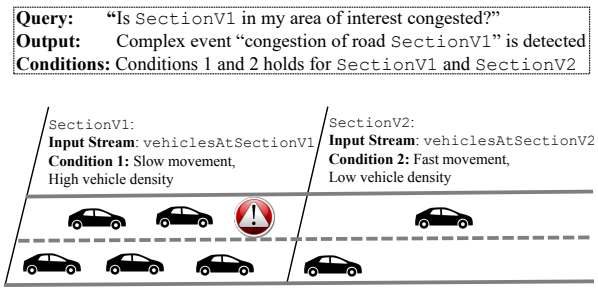
1 INTRODUCTION

Distributed Complex Event Processing (DCEP) is widely known as a key paradigm in the analysis of real-time data streams from distributed sources. Key factors for the success of DCEP in a wide spectrum of application domains [7] are (i) the possibility to easily *specify* event patterns of interest to applications and (ii) to efficiently *detect* such event patterns. The latter is of extreme importance for the applications in the era of the Internet of Things (IoT) where the number of interconnected devices is massively growing and expected to exceed 20 billion by 2020 [4].

To cope with a massive influx of IoT data and to fulfill stringent Quality of Service (QoS) demands for IoT applications, research efforts have already explored a wide spectrum of so-called *Operator Placement mechanisms* [12, 21, 34–36]. A mechanism for Operator Placement (OP) provides a mapping of the event detection logic in the form of an *operator graph* to available resources that are selected to execute event operators. These resources can be physically dispersed and may comprise of anything, e.g., physical servers in a data center, user devices and, as recently proposed, computational resources at the edge of the network [22, 28], e.g., Docker containers.

The placement decision significantly influences the ability of a DCEP system to meet QoS demands [19]. For example, to meet a low latency demand, an OP mechanism may favor resources at the edge of the network in order to enforce low communication delays between event producers, operators, and event consumers. However, in a mobile environment, high dynamics will impose frequent changes in both (i) the placement of operators and (ii) the availability of resources (such as mobile devices). Therefore, OP mechanisms as proposed by [37] favor high stability and low control message overhead in adapting the placement rather than just focusing on low latency communication.

In fact, the selection of an OP mechanism used for a DCEP system strongly depends on the desired QoS demands and the respective environmental conditions. Therefore, building on a single OP mechanism restricts the DCEP system in its ability to (i) support multiple queries with distinct QoS demands and (ii) cope with environmental changes. For example, in a traffic control application, the traffic density and the mobility pattern of vehicles is known to vary a lot in between rush hours (high density and slow movements) and normal hours (low density and fast movements) [11]. Consequently, a good OP mechanism will favor stability of the placement itself to cope with an environment of fast movements of cars. On the other hand, the placement should be highly delay- and load-efficient to keep up with the high event rates produced under environmental conditions with many vehicles. It is important to note that delay



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

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

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

Figure 1: Traffic control scenario showing the need of different OP mechanisms for dynamic user environments.

efficiency and stability (measured as control message overhead) are here conflicting demands that are hard – if not impossible – to meet by a single OP mechanism.

To support a wide spectrum of QoS demands, we propose the concept of *OP mechanism transitions*. An OP mechanism transition allows dynamically exchanging OP mechanisms in a *seamless* way depending on the environmental conditions at run time. This way, DCEP systems can benefit from a large number of existing OP mechanisms [12, 21, 36]. Yet, a crucial research challenge is to address the cost in terms of state migration caused when a placement mechanism is exchanged. In this paper, we propose Tcep, a transition-capable CEP system, supporting cost-efficient operator placement mechanism transitions. We show in the context of an IoT scenario and existing operator placement mechanisms that transitions help to adapt DCEP systems to a wider spectrum of QoS demands and that OP mechanism transitions can be achieved fast and at low cost. In more detail, the paper provides following contributions:

(1) We devise two novel transition strategies to realize a cost-efficient OP mechanism transition in a *live (online)* and a *seamless* manner.
(2) We present a *lightweight learning* algorithm that adaptively exchanges OP mechanisms at run time based on their performance during the execution.
(3) We provide an implementation and evaluation of Tcep to show that *cost-efficient* transitions between OP mechanisms can be beneficial to fulfill QoS demands of a large number of consumers in a dynamic user environment.

The remainder of the paper is structured as follows. We further detail on the problem, particularly for the traffic control scenario and motivate the need of *mechanism transitions* by a preliminary evaluation in Section 2. We introduce the Tcep system model in Section 3. We present the design of Tcep in Section 5. Section 6 evaluates the Tcep system. Sections 7 and 8 present the related work and conclude our paper, respectively.

2 CASE STUDY: IOT TRAFFIC CONTROL APPLICATION

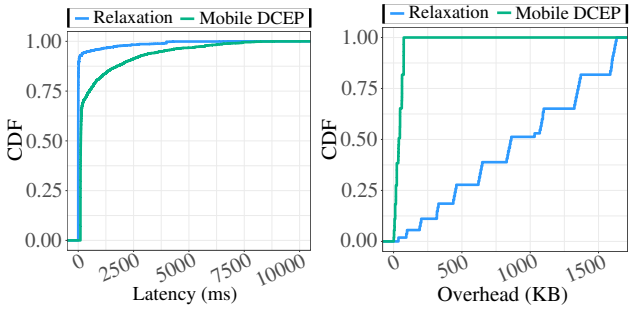
In this section, we introduce an IoT scenario, in the context of a traffic control application. In this scenario, state-of-the-art placement mechanisms [27, 37] fail to fulfill QoS demands that change because of dynamic environmental conditions. This demonstrates the need for a DCEP system that supports OP mechanism transitions.

We consider a continuous query[1] to detect that a road section on a highway is congested because of a traffic congestion (cf. Figure 1). Any consumer can pose the query (cf. Figure 1b), e.g., a vehicle, with respect to a specific road section on the highway, say SectionV1. The query correlates specific conditions – the observed traffic density and speed of vehicles – observed on SectionV1 and its succeeding road section, say SectionV2. The query detects a sequence (Line 9) of such conditions for SectionV1 (Lines 5–7) and SectionV2 (Lines 10–12). The events of both streams (vehiclesAtSectionV1 and vehiclesAtSectionV2) are generated by heterogeneous sensor sources available in the IoT infrastructure (e.g., speed sensors, radar sensors, traffic monitoring cameras and others). The complex event: congestion of road SectionV1 is detected whenever the sequence of conditions on SectionV1 and SectionV2 in a temporal timespan of one minute (Line 14) indicates (i) dense traffic and slow vehicles for SectionV1 and (ii) sparse traffic and fast vehicles for SectionV2 (cf. Figure 1a).

The execution of the query and its operators is performed on the available resources in the IoT infrastructure, such as mobile devices, that can directly interact via device-to-device communication [13]. The mapping of the operators to resources, i.e., the operator placement, must account for the QoS_DEMAND specified within the query. As part of the query specification[1], these demands (e.g., low latency) can be specified according to the user's requirements.

A premise underlying our work is that different demands cannot be accommodated by the same placement mechanism. Therefore, we analyze the ability to fulfill specific QoS demands for the query in Figure 1b for two popular state-of-the-art OP placement mechanisms: *Relaxation* [27] and *Mobile DCEP* [37]. The key idea of the *Relaxation* mechanism is to build on a model referred to as latency

[1] in the AdaptiveCEP query language in Scala [39]



(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 [27] and Mobile DCEP [37] OP mechanisms.

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. On the other hand, *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 that are capable of forming a device-to-device network closer to the data sources. In this way, the authors are able to achieve a near optimal embedding of the operator graph at low control message overhead.

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

Figure 2, shows 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. The cumulative distribution function (CDF) over the number of deployed queries confirms that *Relaxation* achieves consistently very low end-to-end latency for most of the queries (less than 100 ms for 80 % of the query workload). This is consistent with the findings of Pietzuch et al. [27], but the message overhead (to build the latency space) is increasing quickly with the number of deployed queries (up to 1500 KB on average). 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 impose a long end-to-end latency (~7.5 s on average).

The above evaluation shows that different QoS demands require building upon different OP mechanisms. Most importantly, depending on the changing environmental conditions, different mechanisms are required to fulfill the specified QoS demands. In a less dynamic environment with respect to node mobility, e.g., within a rush hour, we measured a significantly lower control overhead for *Relaxation*, i.e., Relaxation can be used to achieve low latency. However, when changing from "rush" hours (with lower dynamics) to "normal" operation (with higher dynamics), a transition from *Relaxation* to *Mobile DCEP* is important. Controlling the overhead improves stability of the OP under the increased dynamics.

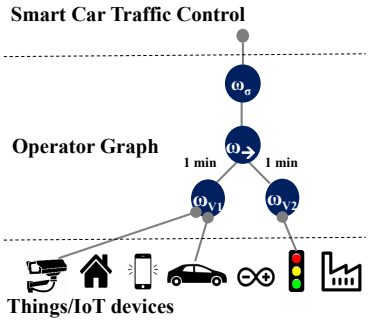


Figure 3: Operator graph for the query in Figure 1b.

3 SYSTEM MODEL

In this section, we present the system model by introducing the Tcep operator graph, the IoT resource model, and the transition mechanism and QoS demand model of Tcep.

3.1 Tcep Model

Tcep consists of a set of event producers (P), which generate continuous data streams (D), a set of event consumers (C), which express an interest in the inference of event patterns from the incoming data streams, and a set of event brokers (B), which host a set of operators (Ω) to process and forward events. Event consumers specify interest in the detection of an event pattern by means of a continuous CEP query. The query induces a directed acyclic *operator graph* $G = (\Omega \cup P \cup C, D)$, where each vertex corresponds to an operator $\omega \in \Omega$ and each edge corresponds to the processing flow of 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. For example, Figure 3 represents an operator graph for detection of a traffic congestion at road sections corresponding to the query in Figure 1b. Operators ω_{V1} and ω_{V2} corresponds 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 function processes ordered input event streams from the operator's input buffer B_I and produces output events that are stored in the operator's output buffer B_O. An operator can either work based on fixed computational parameters (immutable) or it can generate dynamically changing computational state (mutable), depending on the internal logic of the operator. For example, a mutable operator can dynamically change the selection of events determined by an operator-specific *selection policy* and *consumption policy*, e.g., of window and sequence operators.

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. We consider a hierarchical network infrastructure consisting of three layers: (mobile) Things referring to IoT devices interconnected over wireless communication, a fixed network layer comprising distributed resources in data centers (cloud), and a third layer of resources at the edge, which offer a low-latency link to the Things in physical proximity (cf. Figure 4).

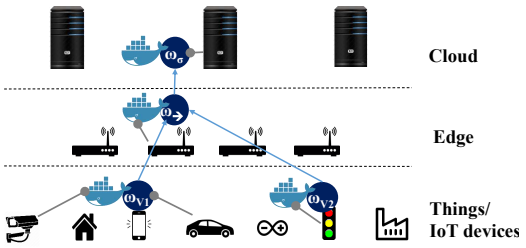


Figure 4: Example operator graph deployment on the IoT network resources.

It is important to note that cloud and edge resources are assumed to communicate via a fixed IP infrastructure, while IoT devices and edge resources can form different wireless network topologies including device-to-device communication between IoT devices.

In the IoT scenario, Things represent producers and consumers, while operators can be placed on any of the three layers. The end-to-end latency for this resource model is influenced by the physical proximity of resources, but also the computational power of resources. In general, we assume higher resource availability and processing power at the cloud. In contrast, IoT devices have resource-constraints because they are battery-powered. Edge nodes are computationally more powerful than mobile nodes. They, however, are constrained in their availability.

In order to access computational resources and place operators on the IoT infrastructure, each operator ω is encapsulated in a lightweight and portable Docker container that provides the runtime execution environment (*EE*) of Tcep.

3.3 Tcep Mechanism and Transition Model

Tcep follows a modular design as a composition of multiple OP mechanisms $M_1, M_2, \ldots, M_N$. An OP mechanism determines a mapping of an operator graph G onto a number of brokers $H = \{b_1, b_2, \ldots, b_k\}$ in the network infrastructure. The mapped network of brokers is well known as an operator network. We define the mapping of the operator network as follows:

$$\alpha : \Omega \times H \rightarrow \{0, 1\}, s.t.$$

$$\alpha_{i,j} = \begin{cases} 1, & \text{if } \omega_i \text{ is placed on } b_j \\ 0, & \text{if } \omega_i \text{ is not placed on } b_j \end{cases}$$

In this work, we study the concept of a transition, denoted as $T : M_A \rightarrow M_B$. A transition T performs an exchange from a mechanism M_A to M_B, e.g., OP mechanisms, at run time in a seamless or non-disruptive manner. To enable the seamless execution of a transition, multiple instances of an operator may actively process events at distinct brokers each executing different OP mechanisms.

3.4 QoS Demand Model

An important 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, e.g., end-to-end latency, bandwidth, control message overhead etc. Tcep allows specification of one or more QoS demands (*QoS*) and changing them at run time. The dynamics in the environmental conditions $(e_1, e_2, \ldots, e_k)$ such as varying workload and mobility influence the fulfillment of such QoS demands.

In this work, we consider two important performance metrics influencing the decision of operator placement in a dynamic environment [28]. First, we define *end-to-end latency* as the time for processing and transmitting a complex event along the path $\omega_1, \omega_2, \ldots, \omega_k$ between any event producer p to any consumer c. It is important to note that end-to-end latency can be time-varying due to the dynamic nature of the network. Second, we define *control message overhead* by the average number of packets that must be sent per broker b_i, until a complex event can be successfully delivered to the event consumer c.

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 (cf. Figure 4). Dependent on the environmental conditions $e(t)$ at time t, the QoS demands of consumers, say $QoS_{|e(t)}$ are changing. Furthermore, the ability and cost of an OP in terms of resource requirements to fulfill the QoS demands are changing over time.

The Tcep system aims to ensure that in spite of changing environmental conditions, the QoS demands of queries are fulfilled and the resource demands are satisfied. Therefore, we determine for changing environmental conditions $e(t)$ and corresponding $QoS_{|e(t)}$ 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 significant cost because of state migration in terms of time as well as 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, the state migration has to take place in a cost-efficient manner.

We define the objective function of the *transition problem* considering two key cost factors, the costs imposed in terms of (i) transition time ($C_{Time}(T_i)$) and (ii) 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 all operators $\omega_j \in \Omega$ to the target brokers ($Time_{mig.(m_j)}$) dependent on α. Thus, we define the cost in terms of transition time as:

$$C_{Time}(T_i) = Time_{select} + Time_{\alpha} + \sum_{j=1}^{n} Time_{mig.(m_j)}$$

Similarly, the transition overhead is the number of messages exchanged in order to perform a transition including the (i) selection of a placement mechanism, (ii) the transition coordination, and (iii) the placement and migration of the operators.

The transition problem in this paper, therefore, is to minimize a weighted sum of transition time and transition overhead in order to meet the QoS demands under the execution of transitions as stated below:

$$\min \left[w_t * C_{Time}(T_i) + w_o * C_{Overhead}(T_i) \right]$$

$$s.t.\ \alpha(t) \text{ satisfies } QoS_{|e(t)} \text{ under the execution of } T_1, \ldots, T_n$$

Here, w_t and w_o denote weights for transition time and overhead, respectively.

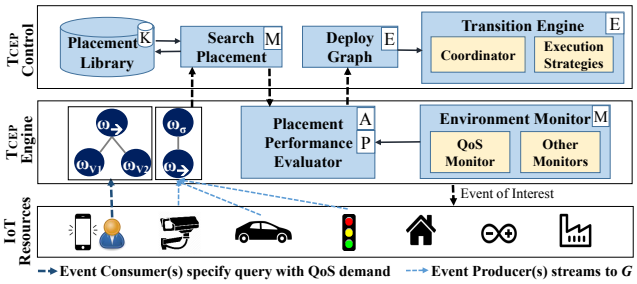


Figure 5: The TCEP system design.

5 THE TCEP SYSTEM DESIGN

The key components of the Tcep system are represented in Figure 5. Tcep system comprises three layers. *IoT resources* includes event consumers, at the bottom layer, can pose queries with specific QoS demands. The application components, e.g., the things, will receive in turn the events of interest. The *control layer* utilizes and manages a library of state of the art OP mechanisms. It decides when to perform a transition and which placement mechanism to select within a transition. It is also responsible to coordinate the transition, i.e., to perform operator migrations building on different transition execution strategies. The *Tcep engine* provides an execution environment to execute operators on the infrastructure of the IoT resources layer. Moreover, the execution environment of the Tcep engine provides mechanisms for monitoring the performance of the placement, as well as monitoring the environmental conditions.

Tcep follows the well-known *MAPE-K* [18] model for adaptation. The four processes of the loop, *Monitoring* (M), *Analyzing* (A), *Planning* (P) and *Executing* (E) are realized in a decentralized manner (cf. Figure 5) in the control layer and within the Tcep engine. In the following sections, we will focus on three research challenges, namely:

(1) How to realize transitions in a seamless manner?
(2) How to decide, when to perform a transition?
(3) How to adaptively select an OP mechanism as a consequence of a transition?

In Section 5.1, we investigate (1) seamless and concurrent transfer of multiple operators during a transition, while taking into account minimal-state for a cost-efficient transition. We address (2) and (3) in Section 5.2, where we first provide a monitoring mechanism for environmental conditions and the respective QoS demands, and second, we present a *lightweight learning* algorithm for an adaptive selection of OP mechanism such that the QoS demands are satisfied.

5.1 Transition Engine

The Tcep transition engine coordinates how a transition is performed over the life cycle of a transition [15], i.e., from its invocation to its completion. The life cycle of a transition is defined by the two transition strategies. 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

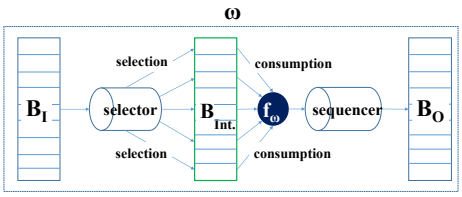


Figure 6: Intermediate buffer represented in the operator state model [40].

and orchestrates the transition life cycle. Tcep currently supports two transition strategies (detailed below). The difference in the life cycle of the proposed transition strategies 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 our primary goal. To do this, we specifically look into costs associated with performing a transition in terms of time and overhead. The transition execution strategy dictates the logic of how *cost-efficient* operator migrations are performed while fulfilling the QoS demands. Taking into account these requirements, we present two transition execution strategies, where we 1) coordinate the transition, 2) perform operator migrations while ensuring the correctness and completeness of the delivered *complex* events to the consumers, and 3) perform the *live* and *seamless* transition.

Moving Fine-Grained State (MFGS) Sequential Transition. In this strategy, 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 successors were successfully migrated. Here, the dependency of operators follows a bottom-up fashion, where leaf operators are successors of their predecessors 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 target OP mechanism (Line 5). It is important to note that the target OP mechanism is predetermined by the search placement 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 details in the next paragraph (Line 8). Afterwards, this state is sent to the target broker, to start the execution of the operator with the minimum migrated state.

In addition, the target broker subscribes to its producers or predecessors to receive event streams, from the time the intermediate state was captured (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,

Algorithm 1: Moving Fine-Grained State Sequential Transition.

```
Variables : OList ← bottom-up list of set of operators (Ω)
            ω ← current operator to be migrated
            producers ← list of producers to ω
            targetMechanism ← target OP mechanism
            targetBroker ← target broker host of ω
            φ_Int ← intermediate state of ω
1  function INIT-MFGS-SEQUENTIALTRANSITION()
2      OList ← BOTTOMUPASLIST(Ω);
3      MFGS-SEQUENTIALSTRATEGY(OList.HEAD, targetMechanism)
4  function MFGS-SEQUENTIALTRANSITION(ω, targetMechanism)
5      targetBroker ← targetMechanism.FINDTARGETBROKER();
6      if targetBroker ≠ ω.SOURCEBROKER then
7          ω.COPYEXECUTIONENVIRONMENT(targetBroker);
8          φ_Int. ← ω.COMPUTEINTERMEDIATESTATE();
9          targetBroker.STARTEXECUTIONWITHDATA(producers,
             φ_Int.);
10         if ω.NEXT().RECEIVEDACK(timeout, retries) then
11             STOPEXECUTION(ω.SOURCEBROKER);
12             MFGS-SEQUENTIALTRANSITION(ω.NEXT(),
                 targetMechanism);
13         else
14             MFGS-SEQUENTIALTRANSITION(ω,
                 targetMechanism);
```

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, wherein we perform m operator migrations sequentially 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).

Cost-efficient Operator Migrations. The TCEP transition engine computes fine-grained computational state of an operator (cf. Section 3.1) for *cost-efficient* operator migrations. We build upon the operator state model proposed by Wermund et al. [13, 40]. In the operator state model (cf. Figure 6), 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 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 are then forwarded to the ω's successor. The events which are acknowledged by all successors are then removed from the output buffer B_O.

Conventionally, a CEP system transfers the internal state ϕ_ω that comprises the state of 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 sliding window operator), or the corresponding event is either consumed (inserted into the output buffer B_O) or discarded by the correlation function.

For the cost and completion time of a migration, it is important when the target broker subscribes to its incoming event streams. Consider the intermediate state $\varphi_\omega(t_i)$ 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 $t_i - \delta$, the target broker subscribes to the input events from the producers or the predecessor operator. δ is a small value to ensure that the target broker receives input events before the processing starts. Yet, all input events until the source broker is executing, are discarded (Line 10).

Properties. The transition time taken by this transition strategy is within the bound $O(|\Omega| + |\varphi_\Omega|)$. Here, φ_Ω denotes the intermediate state of the set of operators Ω within the operator graph. This time includes the time required by the coordinator to transfer the state of an operator over the network (Lines 9 to 12). In this strategy, we reduce the time required to perform an operator graph transition by transferring a minimum amount of state. Yet, the operator processing at the target broker does not take place unless the source broker is in execution. This means that while the selected state is being transferred (i.e., it is on the wire), there are some events – sent to the target broker – that remain unprocessed. No output events are produced unless the intermediate state is transferred. Achieving a seamless transition, e.g., with no output disruption, however, remains an open problem. 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 strategy.

Seamless Minimal State (SMS) Concurrent Transition. In contrast to the above strategy, this strategy allows for more than one operator migrations at the same time (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 begins 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 [42]. The selection of time t_i is such that for each operator ω, SMS waits until the operator ω is purged from its old state (Line 8). Essentially, until B_{Int} and f_ω *are stateless*. For example, for a window-aggregate operator, the target broker waits until the last event of the window is processed, $w + \delta$, here w is the window size and δ 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 *minimal-state* time of an operator ($t_{imin}(\omega)$).

The target broker starts its execution with the minimal state (the last *SN*) at the *transition start time*, while the predecessor operators at the higher level are still under execution by the former OP mechanism. Thus, in this strategy, the transition coordinator allows an execution of two OP mechanisms concurrently. This is to deal with the output disruption discussed as follows.

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
            φ_sequencer ← state of sequencer
            waitTime ← time taken until current operator
            is purged from its state
1  function SMS-ConcurrentTransition(OGlevel, targetMechanism)
2      for all ω ∈ OGlevel do in parallel
3          targetBroker ← targetMechanism.findTargetBroker();
4          if targetBroker ≠ ω.sourceBroker then
5              ω.copyExecutionEnvironment(targetBroker);
6              NTPClockSynchronization,(targetBroker,
                 ω.sourceBroker);
7              minimalStateTime ←
                 ω.determineMinimalStateTime();
8              waitTime ← waitUntil(minimalStateTime));
9              φ_sequencer ← ω.lastSN;
10             targetBroker.StartExecutionWithData(producers,
                 φ_sequencer);
11             targetBroker.determineReferencePoint
                 (minimalStateTime);
12             if ω.parent().receivedACK(timeout, retries) then
13                 StopExecution(ω.sourceBroker);
14                 SMS-ConcurrentTransition(OGlevel.next(),
                     targetMechanism);
15             else
16                 SMS-ConcurrentTransition(OGlevel,
                     targetMechanism);
```

Seamless and Concurrent Operator Migrations. In order to explain the concurrent operator migrations, we refer to the operator graph from our example scenario in Figure 7. *Src* box refers to the placement of an operator at the source broker and *Trg* box refers to the placement at the target broker. 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. steps 2 to 3 in Figure 7) enables seamless execution in this strategy. Yet, migrations do not interfere with each other, while the operator network gradually transforms the placement (cf. step 4 in Figure 7). The transition coordination is accomplished atomically in the Tcep transition engine.

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). All the input events received after $t_{imin}(\omega)$ are redirected to the target broker. The source broker processes the remaining input events. The replicated operator at the target broker starts processing the input events concurrently after the migration of the operators at lower levels in the graph. The source brokers are gradually replaced by their targets as illustrated in Figure 7.

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) [24] at both ends (Line 6). To avoid duplicates in the output events, due to the concurrent processing, we use the reference point method [8] (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.

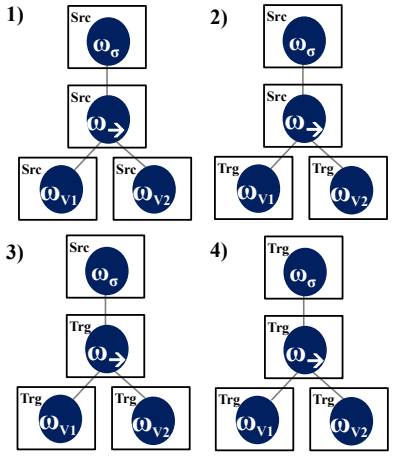


Figure 7: Sequence of operator migrations in the operator graph for SMS strategies.

Properties. In this strategy, 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 strategy is within $O(log(|\Omega|) + C)$, here $C = |\varphi_\Omega|$ that is constant (state of the sequencer) for a given set of operators Ω.

5.2 Placement Performance Evaluator

This component measures the performance of the OP mechanisms continuously and analyze their behavior. A *lightweight learning* algorithm is employed, to statistically determine which mechanism best meets the QoS demands building on a selection strategy of genetic algorithms [41]. *Lightweight* refers to the fact that learning does not rely on any training set but only use statistics that are collected online during the execution. The learned model is used by the search placement component to adaptively select an appropriate OP mechanism with best performance. The environment monitor component keeps track of the performance behavior (QoS demands and environment conditions via QoS monitor and other monitors) and reports any changes to this component – e.g., if the QoS demand specified in the query is violated. During initialization (when no empirical statistics are available), 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 this section, first, we 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.

Heuristic Fitness Score for OP Mechanism. We measure the performance of the current OP mechanism in execution 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. We define a heuristic fitness function with the objective to maximize the number of times an OP mechanism fulfills the current QoS demands. This means that if an OP mechanism fulfills 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 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 has performed over the time, in comparison to the mechanisms that were in execution before (since the query was first submitted). The goal is to find a 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 fulfill multiple QoS demands, the score is determined for each QoS demand separately. For each OP mechanism M_i, we maintain a score function $Score_{(M_{i,qos_j})}(t_t)$ which is 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.

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

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 $t_{s,t}$.

In Equation 1, $\mu_{qos_j}(t_{s,t})$, $max_{qos_j}(t_{s,t})$, 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 give priority to the data that is recently collected. The initial of value of decay is set to 0.

The overall score of an OP mechanism is computed based on the scores of 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 $qos_j \in [qos_1, qos_k]$, where k is the total number of 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 from the *Linear Ranking Selection Strategy* [41], a selection method from Genetic Algorithms (GA). The ranking based method is suitable for our OP mechanism selection problem, since it allows us (1) to perform a relative analysis suitable for the heuristic fitness function that indicates which OP mechanism is better, and (2) by an appropriate selection pressure it favors exploration over exploitation avoiding selecting worse OP mechanisms. In this method, OP mechanisms are sorted according to their fitness values, and then ranks are assigned to them. Rank N is assigned to the best OP mechanism while rank 1 to the worst. The selection probability P_i is linearly assigned according to the rank as follows.

$$P_i = \frac{1}{N}\left(\eta^- + (\eta^+ - \eta^-)\frac{i-1}{N-1}\right); i \in [1, N] \quad (2)$$

In Equation 2, $\frac{\eta^-}{N}$ is the probability that the worst OP mechanism is selected and $\frac{\eta^+}{N}$ 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 [9]. The probability of the OP mechanism to be selected is proportional to its fitness function score.

$$\eta_i = \frac{Score_{(M_i)}}{\sum_{i=1}^{n} Score_{(M_i)}}$$

The worst probability and the best probability are calculated as the minima and maxima respectively of the probability distribution function η. The selection of the mechanism means 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 which were classified to perform better (exploitation of the learning algorithm), but sometimes we also select worse OP mechanisms to update their score (exploration).

6 EVALUATION

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

(1) Does the mechanism transition concept satisfy changing QoS demands for dynamic environmental conditions?
(2) Can a transition for OP mechanism be performed in a live and seamless manner?
(3) 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 two ways: (i) in Section 6.2, we evaluate the ability of Tcep to meet QoS demands with respect to latency and message overhead, (ii) in Section 6.3, we evaluate the stability of the system subject to transitions, and the cost imposed by the distinct transition strategies proposed in Section 5.

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.

Table 1: Configuration parameters for the evaluation.

Number of producers	2
Number of brokers	10
Number of consumers	4
Number of queries	50
WINDOW_SIZE	5, 10, 20, 30, 40, 50 secs
QOS_DEMANDS	latency, message overhead
Delay and Jitter	min. 20 ± 10 ms, max. 100 ±50 ms (normal distribution)
OP mechanisms	Relaxation [27], Mobile DCEP [36]
Transition execution strategies	MFGS-Sequential, MFGS-Concurrent, SMS-Sequential, SMS-Concurrent

6.1 Evaluation Environment and Setup

The implementation of Tcep builds on an adaptive complex event processing system proposed in [39]. In particular, Tcep builds on the programming model for specifying QoS demands at run time (e.g, the query in Figure 1b). We extended the runtime environment based on the Akka actor system [2] to build a distributed network of Docker containers for easy deployment in the edge-IoT scenario. The Docker container helps to encapsulate each CEP operator as a lone Docker instance with all its dependencies. Furthermore, we realized extensions in the form of a placement module that integrates state-of-the-art OP mechanisms [27, 37] and measure the performance of the resulting placement.

We build Tcep's Docker image upon the Alpine Linux distribution,[2] which is much smaller and lightweight. 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. 2.5.9 [2], the Esper CEP engine v. 5.5.0 [1] and Docker v. 17.12.0-ce [3]. We deployed 16 Docker containers on a VM with 32 GiB of memory and 6 processors on a high-end server. The Docker overlay network is organized in a hierarchical fashion. We emulate wide area networks using `netem`[3] by adding variable delay and jitter on the interfaces connecting the Docker containers. The network delay and jitter are applied as minimum at the lower end and as maximum at the cloud nodes – consistently with the edge-cloud scenario.

We use the query in Figure 8 – a subquery of the traffic detection query from Section 2. The query performs a join ($\omega_{\bowtie}$) over a window of the event streams from road sections V1 (ω_{V1}) and V2 (ω_{V2}) respectively (Lines 6 - 7). Last, the road sections with a heavy penetration of vehicles are selected by ω_{σ} (Line 9).

We generate the primary events as continuous data streams as an input to Tcep synthetically for each execution. For example, for the input streams `vehiclesAtSectionV1` and `vehiclesAtSectionV1` we produce randomly generated identifiers and count corresponding to the stated types (cf. Figure 8). We emulate each execution for

[2]https://github.com/gliderlabs/docker-alpine [Accessed on 03.05.2018]
[3]https://wiki.linuxfoundation.org/networking/netem [Accessed on 03.05.2018]

```
1  case class VehiclesAtSection(highwayId: Int,
2       sectionId: Int, count: Long, time: Long)
3  val vehiclesAtSectionV1: Stream[VehiclesAtSection] =
       ...
4  val vehiclesAtSectionV2: Stream[VehiclesAtSection] =
       ...
5  val vehicleInfo =
6    ((vehiclesAtSectionV1 window WINDOW_SIZE) join
7     (vehiclesAtSectionV2 window WINDOW_SIZE) on
          'highwayId
8     where { (enteredVehicles, exitVehicles) =>
9       enteredVehicles.count > exitVehicles.count
10    }
11    demand QOS_DEMANDS)
```

Figure 8: Evaluation query.

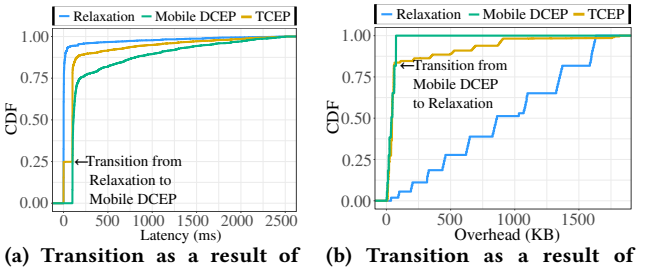
20 minutes, and initiate the measurements after 2 minutes warm-up. Each measurement is taken at a regular interval of 5 seconds. We incrementally increase the query workload for up to 50 queries. The metrics in the evaluation are influenced by multiple parameters such as the number of input events, the 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 1.

6.2 Performance of OP Mechanism Transitions

In order to understand whether the mechanism transition is able to fulfill changing QoS demands for dynamic environmental conditions, we evaluate the performance of Tcep. We consider the following metrics: (i) average end-to-end latency from query subscription until the complex-event was received at the consumer end, (ii) average control message overhead associated with establishing the broker network. Figure 9 shows a comparison of latency and control message overhead samples dependent on the current query load (as CDF) for Relaxation [27], Mobile DCEP [37] and Tcep.

Tcep enables execution of transition as a result of dynamics in the environmental condition and henceforth changes in the QoS demands as shown in Figure 9a and Figure 9b. Figure 9a shows that Tcep initially executes the Relaxation mechanism to comply with the latency demand of the event consumer. Later, as shown by an arrow in the figure, a change in QoS demand (latency to overhead) leads to the transition T : Relaxation $\rightarrow$ Mobile DCEP. The transition ensures that the specified or changed QoS demands are met. In Figure 9a, a change from end-to-end latency to control message overhead is detected. Tcep ensures the demands are met by means of a transition.

In Figure 9b, a change in the QoS demand, namely from control message overhead to end-to-end latency, is issued. Tcep identifies this change and selects a mechanism that is suitable to meet the QoS demand by means of Tcep's selection algorithm. The Relaxation placement mechanism is chosen, to ensure that the latency demands are met under very high workload. We specify the trigger of the transition T : Mobile DCEP $\rightarrow$ Relaxation by an arrow in the Figure 9b. Here, clearly, an increase in the control message overhead can be observed as a consequence of the run time transition to Relaxation.



(a) Transition as a result of change in QoS demand from latency to message overhead.

(b) Transition as a result of change in QoS demand from message overhead to latency.

Figure 9: TCEP vs. Relaxation and Mobile DCEP.

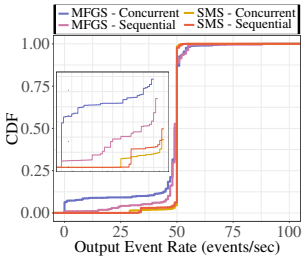


Figure 10: SMS transition strategies consistently deliver output events enabling seamless execution of a transition.

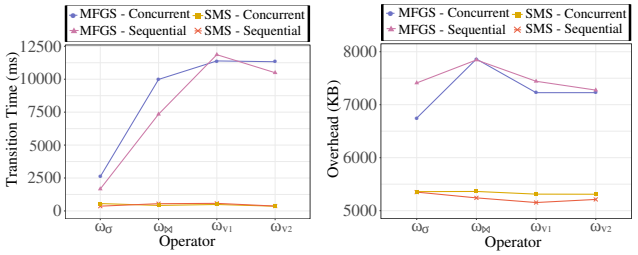
6.3 Performance of Transition Strategies

In this section, we aim to understand, how far the transitions are disruptive and what is 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 strategies in a fair manner, we extend Algorithm 1 to concurrently migrate the operators. Similarly, we extend Algorithm 2 to sequentially migrate the operators. The four approaches are enlisted in Table 1.

To verify the seamless execution of transitions, we measured the output event rate (x-axes) produced while TCEP's strategies were in execution (cf. Figure 10). In this evaluation, we increased the query load with the input event rate of 10 events/s incrementally. A small output disruption for MFGS-Sequential and Concurrent strategies was observed as seen in the magnified Figure 10 (5 and 8 % times respectively). However, SMS-Sequential and Concurrent strategies do not exhibit any disruption and continuously deliver output events with an output event rate of 50 events/s, 99 % times.

Cost of Transition Strategies for different operators. Now, we analyze the cost incurred by the transition strategies in detail. In Figure 8, we measure the performance of the transition strategies for each operator ω_{V1}, ω_{V2}, ω_{σ} and $\omega_{\bowtie}$ of the query. To recall, operators ω_{V1} and ω_{V2} are window-aggregate operators, $\omega_{\bowtie}$ is a join operator and ω_{σ} is a filter operator. The transition time comprises of operator migration time as well as the time an operator has to wait for migration until the predecessor starts its operation at the target broker (cf. Section 5.1). For example, $\omega_{\bowtie}$ waits for migration until ω_{V1} and ω_{V2} start their operation at the target brokers. Leaf operators have no wait time as they have no predecessors.

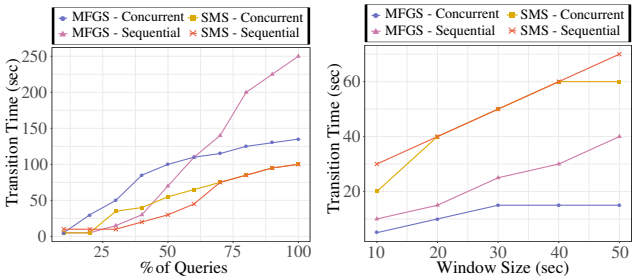
Figures 11a and 11b show the transition time and overhead per operator for each transition execution strategy. As expected,



(a) SMS transition strategies perform operator migrations in the order of a few seconds.

(b) SMS transition strategies require minimal state transfer during operator migrations.

Figure 11: Transition cost for different operators.



(a) Transition times of sequential transition strategies grow significantly faster than concurrent strategies.

(b) Increasing window sizes impact stronger transition times for SMS than MFGS under the network conditions (cf. Table 1).

Figure 12: Transition strategies under increasing workload (number of queries and window size).

the MFGS-Sequential and the MFGS-Concurrent strategies exhibit longer transition times, since the intermediate state that has to be transferred is larger (up to 8 MB for a single operator). Operators ω_{V1} and ω_{V2} do not have any wait time but have longer transition time as a result of ~ 7.5 MB state transfer. Operator $\omega_{\bowtie}$ has a lower transition time (~7.5 sec) for the Sequential strategy, while it is a little longer for the Concurrent strategy as a result of longer wait time in the latter. This small difference is perhaps due to the concurrent transfer of two operators ω_{V1} and ω_{V2}, $\omega_{\bowtie}$ has to wait longer for migration (until both are functioning at the target brokers). $\omega_{\bowtie}$ has the highest state size, as it stores all the input events that have been processed from the input stream ω_{V1} to join with future input events from the other input stream ω_{V2}.

In conclusion, our results show that the SMS-Sequential and Concurrent strategies perform better in both transition time and overhead, with the time within a range of 358 – 575 milliseconds in comparison to 35 seconds (if the transition is performed naively) for a single operator. Yet, the two strategies also have a longer wait time, as a result of deferment of operator migrations until the old operator is purged from its state, which is further analyzed in the next section.

Cost of Transition Strategies with Varying Parameters. We analyze the effect of increasing the number of queries (and henceforth increasing number of operators) and the window size on the transition strategies (cf. Table 1).

Figure 12a shows the effect of varying the number of queries (0 – 100 %) – hence the number of operators – on the transition

time. As expected, both the Sequential strategies– MFGS and SMS shows a linear increase in transition time with the increasing query load (due to sequential operator migrations). On the other hand, Concurrent strategies pay off for a larger workload – as expected – since operator migration involves less synchronization. MFGS-Sequential has the highest transition time among all the strategies, under a larger workload of queries. This is because, due to the increasing workload, the number of operator increase, the state that has to be transferred also increases, and henceforth does the transition time. Yet, SMS Concurrent and Sequential strategies both performed at par for this load of queries (Table 1). The increase in the window size results in longer transition time (Figure 12b), because more state has to be transferred. Interestingly, window size affects SMS strategies more than MFGS strategies due to the longer waiting time of SMS strategies – equivalent to the window size for each operator ω_{V1} and ω_{V2}. Thus, for larger window sizes, SMS strategies may have longer transition time, but then only the SMS strategies offer a seamless execution.

7 RELATED WORK

It is highly important to fulfill QoS demands in a DCEP system for a wide range of application domains [28]. By enabling transitions, Tcep allows to exchange operator placement mechanisms, and in this way, fulfill QoS demands under dynamic environmental conditions. In this section, we analyze and compare to related work in three key areas: operator placement, adaptive event processing systems, and existing methods for mechanism transitions.

7.1 Operator Placement

Operator placement (OP) is widely studied to fulfill QoS demands while incurring minimum cost in terms of performance [21, 36]. A wide range of OP mechanisms have been proposed considering different QoS demands such as to achieve low latency [5], to minimize bandwidth [27, 34, 35], to lower message overhead [37], as well as to preserve trust and privacy [40].

The fulfillment of QoS demands, however, is only feasible under limited changes of the environmental conditions. For instance, most existing work [5, 10, 34] builds on stationary networks. Approaches considering dynamic changes, e.g. in the cause of mobility, introduces i) redundancy by means of duplication [37] or checkpointing [20], ii) placement update at regular intervals [27], or, iii) operator migrations when changing the placement [19, 25, 26, 40].

Overall, it is important 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.

7.2 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 [39] proposed AdaptiveCEP, a programming model and CEP system that supports specifying QoS demands at run time. This work is complementary to Tcep since AdaptiveCEP is not focusing on the the adaptive selection and execution of transition strategies. However, in Tcep the query language is used to specify changes in the QoS demands in order to instantiate a transition.

Heinze et al. proposed an elastic data stream processing system (DSPS) [16], where the number of active hosts can be scaled up and down and operator migration is coordinated accordingly. Based on this work, the same authors proposed an adaptive replication scheme for DSPS [17] that performs adaptation at runtime between active replication and upstream backup schemes for fault tolerance. Another line of work by Matteis et al. [23] addressed the problem of elasticity by *Model Predictive Control* method that accounts for system behavior over a future time horizon to predict the best reconfiguration to be executed. Aniello et al. [6] proposed an adaptive online scheduling algorithm for Apache Storm using two placement mechanisms. Sutherland et al. [38] developed an adaptive scheduling selection framework for continuous queries in DSPS.

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.

7.3 Mechanism Transitions

The idea of mechanism transitions origins from the collaborative research center MAKI, in which researchers investigate mechanism transitions for the Future Internet [29]. Within MAKI, mechanism transitions are investigated in the context of a wide range of communication mechanisms [15, 30–33]. For example in publish-subscribe systems, mechanism transitions between filtering schemes [30] and event dissemination mechanisms [31] are studied. Another line of work by Froemmgen et al. [14, 15] proposed transition strategies to always execute the best suitable search overlay. Richerzhagen et. al [33] 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 [14, 30]. 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. In particular, 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 environmental conditions. Tcep allows an integration of state-of-the-art OP mechanisms and dynamically executes the best matching OP mechanism to meet QoS demands of applications. To this end, we have explored how to perform transitions and analyzed their cost and performance. Moreover, we proposed two transition execution strategies for efficient migrations of operator state during a transition. Our evaluation in the context of an IoT scenario and

based on the state-of-the-art OP mechanisms shows that, Tcep fulfills changing QoS demands by performing transitions in a seamless manner, i.e., without any output disruption. Furthermore, the cost analysis shows that by means of our transition execution strategies the transition execution time can be decreased to a few milliseconds for a single operator. In this work, we performed a transition only in a *reactive* manner. We are currently investigating *proactive* adaptation methods to further improve the overall performance of Tcep. In addition, we are considering proactive transitions in other mechanisms of DCEP such as operator migration strategies. We will also consider trade-offs between different learning strategies for the adaptive selection of an appropriate OP mechanism.

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, by DFG grant SA 2918/2-1 and by the European Research Council (grant No. 321217). The authors would like to thank Pratyush Agnihotri for his valuable contribution in the evaluation of the system.

REFERENCES

[1] EsperTech – Esper. http://www.espertech.com/esper/, 2006. Accessed 03.05.2018.
[2] Akka. http://akka.io/, 2009. Accessed 03.05.2018.
[3] Docker – Community edition. https://www.docker.com/community-edition, 2013. Accessed 03.05.2018.
[4] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017. https://www.gartner.com/newsroom/id/3598917, 2017. Accessed 03.05.2018.
[5] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 456–467. VLDB Endowment, 2004.
[6] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218, 2013.
[7] M. Baguena, G. Samaras, A. Pamboris, M. L. Sichitiu, P. Pietzuch, and P. Manzoni. Towards enabling hyper-responsive mobile apps through network edge assistance. In *13th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 399–404, 2016.
[8] J. V. d. Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 168–179. Morgan Kaufmann Publishers Inc., 1996.
[9] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.
[10] V. Cardellini, V. Grassi, F. Lo Presti, and 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*, pages 69–80, 2016.
[11] G. Cookson and B. Pishue. INRIX Global Traffic Scorecard. Technical report, 2018.
[12] G. Cugola and A. Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.
[13] R. Dwarakanath, B. Koldehofe, and 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, pages 13–18, 2016.
[14] A. Frömmgen, S. Haas, M. Stein, R. Rehner, A. Buchmann, and M. Mühlhäuser. Always the best: Executing transitions between search overlays. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 8:1–8:4, 2015.
[15] A. Frömmgen, B. Richerzhagen, R. Julius, D. Hausheer, R. Steinmetz, and A. Buchmann. Towards the Description and Execution of Transitions in Networked Systems. In *Intelligent Mechanisms for Network Configuration and Security*, pages 17–29. Springer International Publishing, 2015.
[16] T. Heinze, Z. Jerzak, G. Hackenbroich, and 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 '14, pages 13–22, 2014.
[17] T. Heinze, M. Zia, R. Krahn, Z. Jerzak, and 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 '15, pages 150–161, 2015.
[18] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
[19] G. G. Koch, B. Koldehofe, and K. Rothermel. Cordies: expressive event correlation in distributed systems. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS 2010)*, pages 26–37, 2010.
[20] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and 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 '13*, page 27, 2013.
[21] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.
[22] M. Luthra, B. Koldehofe, and R. Steinmetz. Adaptive Complex Event Processing over Fog-Cloud Infrastructure Supporting Transitions. In *GI/ITG KuVS-Fachgespraech Fog-Computing*, 2018.
[23] T. D. Matteis and G. Mencagli. Proactive elasticity and energy awareness in data stream processing. *Journal of Systems and Software*, 127:302 – 319, 2017.
[24] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.
[25] D. O'Keeffe, T. Salonidis, and P. Pietzuch. Network-aware stream query processing in mobile ad-hoc networks. In *MILCOM 2015 - 2015 IEEE Military Communications Conference*, pages 1335–1340, 2015.
[26] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran. MCEP: A Mobility-Aware Complex Event Processing System. *ACM Transactions on Internet Technology (TOIT)*, 14(1):1–24, 2014.
[27] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, 2006.
[28] M. A. Rahmani, L.-S. P. Preden, and A. Jantsch. *Fog Computing in the Internet of Things*. Springer International Publishing, 2018.
[29] B. Richerzhagen, B. Koldehode, and R. Steinmetz. Immense Dynamism. *German Research*, 37:24–27, 2015.
[30] B. Richerzhagen, N. Richerzhagen, J. Zobel, S. Schönherr, B. Koldehofe, and R. Steinmetz. Seamless transitions between filter schemes for location-based mobile applications. In *IEEE 41st Conference on Local Computer Networks (LCN)*, pages 348–356, 2016.
[31] B. Richerzhagen, M. Schiller, M. Lehn, D. Lapiner, and R. Steinmetz. Transition-enabled event dissemination for pervasive mobile multiplayer games. In *IEEE 16th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–3, 2015.
[32] B. Richerzhagen, S. Wilk, J. Rückert, D. Stohr, and W. Effelsberg. Transitions in live video streaming services. In *Proceedings of ACM VideoNEXT 2014*, 2014.
[33] N. Richerzhagen, P. Lieser, B. Richerzhagen, B. Koldehofe, I. Stavrakakis, and R. Steinmetz. Change as chance: Transition-enabled monitoring for dynamic networks and environments. In *14th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, 2018.
[34] S. Rizou, F. Durr, and K. Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–6, 2010.
[35] B. Schilling, B. Koldehofe, and 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)*, pages 355–364, 2011.
[36] F. Starks, V. Goebel, S. Kristiansen, and T. Plagemann. Mobile Distributed Complex Event Processing - Ubi Sumus ? Quo Vadimus ? *Mobile Big Data- A Roadmap from Models to Technologies, Springer 2017,*, (1):1–34, 2017.
[37] F. Starks and T. P. Plagemann. Operator placement for efficient distributed complex event processing in manets. In *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 83–90, 2015.
[38] T. M. Sutherland, B. Pielech, Y. Zhu, L. Ding, and E. A. Rundensteiner. An adaptive multi-objective scheduling selection framework for continuous query processing. In *9th International Database Engineering Application Symposium (IDEAS'05)*, pages 445–454, 2005.
[39] P. Weisenburger, M. Luthra, B. Koldehofe, and 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*, SEAMS '17, pages 140–151, 2017.
[40] R. Wermund. *Privacy-Aware and Reliable Complex Event Processing in the Internet of Things*. PhD thesis, Technical University of Darmstadt, 2018.
[41] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121. Morgan Kaufmann Publishers Inc., 1989.
[42] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 431–442, 2004.