DCEP-Sim: An Open Simulation Framework for Distributed CEP

Fabrice Starks
University of Oslo
Gaustadallen 23B
Oslo, Norway
fabriceb@ifi.uio.no

Thomas Peter Plagemann
University of Oslo
Gaustadallen 23B
Oslo, Norway
plageman@ifi.uio.no

Stein Kristiansen
University of Oslo
Gaustadallen 23B
Oslo, Norway
steikr@ifi.uio.no

ABSTRACT

Distributed Complex Event Processing (CEP) is gaining increasing interest for two reasons: (1) to scale system performance to handle higher workloads in real-time, and (2) to perform in-network processing, e.g., in mobile networks to reduce the amount of data that has to be transferred through the network. System scalability and the complexity of mobile systems are some of the major challenges when evaluating the performance of new Distributed CEP solutions. We propose an open framework for distributed CEP (DCEP-Sim) built on a well-established network simulator, i.e, ns-3. The design of DCEP-Sim is based on the engineering principles of separation of concerns and the separation of mechanisms and policies. By leveraging the ns-3 feature of object aggregation it is very easy to add new policies, e.g., placement or selection policies, and evaluate them without changing anything else in the DCEP-Sim. The fact that ns-3 includes many accurate network models implies that Distributed CEP simulation with DCEP-Sim will also be much more accurate than ad-hoc handcrafted simulations. We demonstrate in a use case how easy it is to configure performance evaluation experiments and we perform experiments to confirm that the integration of the Distributed CEP in ns-3 is good foundation for large-scale experiments. The evaluation results demonstrate that DCEP-Sim substantially reduces the effort and costs of Distributed CEP evaluation.

CCS CONCEPTS

•Networks → Network simulations; •Computing methodologies → Simulation support systems; Simulation tools;

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference format:
Fabrice Starks, Thomas Peter Plagemann, and Stein Kristiansen. 2017. DCEP-Sim: An Open Simulation Framework for Distributed CEP. In *Proceedings of ACM International Conference on Distributed Event-Based Systems, Barcelona, Spain, June 19 fi 23, 2017 (DEBS fi 17),* 11 pages.
DOI: 10.1145/3093742.3093919

DEBS fi 17, Barcelona, Spain

DOI: 10.1145/3093742.3093919

1 INTRODUCTION

The need to analyse high velocity and high volume data is continuously increasing. The Internet of Things is one good example for this trend. A popular prediction estimates for Year 2020 about 50 Billion connected devices [1]. In this context, real-time analysis of data is important for two reasons: (1) the velocity and volume of the data might make it impossible to store it on disk before analysis, and (2) many application domains, like smart cities or automated traffic control, aim to maintain continuous situational awareness to be able to react as fast as possible to certain events. The concepts of Complex Event Processing (CEP) are well suited to addressing these processing needs and there is a clear trend that Distributed CEP will be of increasing importance. We identify two core reasons to distribute CEP: (1) to scale system performance and handle higher workloads in real-time, and (2) to perform in-network processing to reduce the amount of data that has to be transferred through the network. The latter is especially important in mobile networks and networks with bandwidth limitations. Consequently, Distributed CEP is one important active research area and evaluation of Distributed CEP solutions is one central part of this research.

However, performing proper performance evaluation of Distributed CEP is a major challenge in itself due to the potentially large scale and especially due to the complexity of distributed systems, which is particularly high if mobile networks are involved. In [11], we analysed the performance evaluation approaches of 13 key publications on Mobile Distributed CEP systems. Only two report results from real world experiments and two from mathematical analysis, while all include at least one simulation or emulation study. The fact that simulation is the most popular approach is not surprising since a proper evaluation of Distributed CEP approaches might require networks with several hundred nodes, making real-world experiments unfeasible. This problem is further exacerbated for wireless networks where the shared medium and node mobility implies a high degree of network dynamicity, making it exceedingly difficult to conduct controlled and repeatable experiments. The majority of these simulation studies is performed with simulators that are created for the specific experiment and only four experiments are performed with non-propriety simulators, i.e., J-Sim [10], OMNeT++ [12], and PeerSim [4].

Creating a new simulator for a particular experiment means either a huge amount of work, or the simulation models are (over-) simplified to reduce the amount of work, which in turn might lead to inaccurate results. Proprietary simulators make it also harder or impossible for peer researchers to repeat experiments and to compare results. One important conclusion from the study in [11]

and further literature studies is that there are currently no evaluation techniques and tools for Distributed CEP research that enable low effort and cost experiments that are repeatable and enable comparison of alternative solutions. CEP-Sim [8] has similar goals, but focuses on cloud environments. In general, CEP-Sim extends CloudSim [6], which models the network through a latency matrix denoting the end-to-end latencies between the involved CloudSim entities. Thus, only static network topologies with fixed end-to-end latencies can be modeled and bandwidth is not modeled. Therefore, CEPSim is not sufficient for Distributed CEP experiments to gain representative and accurate results, e.g., when evaluating placement policies in mobile networks. We provide a more detailed comparison between our framework and CEP-Sim in the conclusion.

It is the aim of this work to introduce DCEP-Sim: a tool for representative and accurate evaluation of Distributed CEP systems. The goals of DCEP-Sim are to achieve a high degree of accuracy, enable low effort and cost experiments, and to make it easy to compare alternative DCEP solutions. We propose to rely on the results achieved in the network research community. This community has put substantial amounts of work into the development of scalable and accurate network simulators, like OMNeT++ and ns-3 [2]. Furthermore, these simulators are designed to be easy to use and to enable efficient experimentation with low effort. Probably the most popular network simulator, ns-3, has been continuously improved from the first generation to the third generation. Many models for network components at all layers exists for ns-3 and are ready to use, e.g., for wired and wireless media, medium access protocols, network and transport protocols, etc. Since the boundary between a simulated network and a real world prototype on top of the simulator implies a substantial performance and scalability penalty [9], we decided to implement the Distributed CEP functionality as simulation models within ns-3 instead of using emulation. Since it is our goal to provide a tool to researchers that simplifies comparison of alternative Distributed CEP solutions we design an open and extensible Distributed CEP skeleton for ns-3. This skeleton is based on the well-accepted CEP fundamentals [7]. Through separation of concerns we identify the core Distributed CEP components and within the components we separate mechanisms and policies. For example, the placement component contains a mechanism for event notification forwarding and hooks to add new or re-use existing implementations of placement policies. To realize these hooks, we use the ns-3 feature of object aggregation, which makes it very easy to add new policies, e.g., placement or selection policies. Consequently, it is also easy to evaluate new policies and compare them with existing policies, because there is no need to change anything else in the Distributed CEP implementation in DCEP-Sim. Thanks to a wealth of validated and accurate ns-3 network models, it is possible to perform simulations of Distributed CEP systems in various network environments and produce reliable and useful results.

We demonstrate in this paper how easy it is to configure performance evaluation experiments in DCEP-Sim and how easy it is to deploy alternative policies. The performance evaluation experiments show that DCEP-Sim adds only a very small fraction of computational overhead such that DCEP-Sim inherits the performance and scalability properties of ns-3, in addition to inheriting the accuracy of ns-3 network models.

The remainder of this paper is structured as follows. In Section 2, we present the design of DCEP-Sim after a brief discussion of its requirements. In Section 4, we describe how DCEP-Sim was implemented. In Section 5, we present and discuss the results from DCEP-Sim evaluation experiments. In Section 6, we discuss the motivation and requirements for DCEP-Sim, and how they were addressed in the design and implementation. We close our discussion with a brief summary of the results and future work.

2 DCEP-SIM DESIGN

In this section, we address DCEP-Sim requirements and the design principles used to achieve them.
At the heart of a Distributed CEP system is the CEP engine which is responsible to perform the actual processing of events. Additionally, to enable Distributed CEP, a Placement component is necessary to build the event broker overlay network and perform event routing. Therefore, following the discussion about the design principles, we present the design of the two core components of DCEP-Sim: the CEP engine and Placement components. Afterwards, we briefly discuss the design of the remaining components, and close this section with a presentation of how all the components in DCEP-Sim work together.

2.1 Requirements

We aim to develop a Distributed CEP simulator which is generic, scalable, accurate, and easy to use. The simulator needs to be useful for a broad range of research studies and must not be fixed for a particular application domain or scenario. It should therefore be easy to extend DCEP-Sim by adding new models or extending extisting ones. As an example, it should be possible to implement and integrate arbitrary CEP operators without changing its core architecture. DCEP-Sim should enable large-scale experiments with Distributed CEP solutions. To enable simulation with a high number of CEP instances, it is preferable that DCEP-Sim adds minimal simulation time overhead to the simulation time incurred by underlying ns-3 network models. Moreover, the simulator should enable the production of both realistic and accurate results which can be easily reproduced.

2.2 Design principles

To ensure the generality of the simulator, we use as the foundation for our work the abstract CEP architecture proposed in [7] [1]. The components of the proposed architecture are based on elementary and common features found in CEP engines. By using this architecture, we argue that the simulator will be usable in a broad range of research studies and not be fixed for a particular application domain or scenario. While the proposed components are found in various CEP systems, we do not expect them to behave in the same way for all research studies. Therefore, our design follows the ***separation of mechanism and policy*** principle [13]. The policy refers to ***what*** the component does, and the mechanism specifies ***how*** it is done. The mechanism is the underlying technique used

[1] CEP is used here instead of Information Flow Processing (IFP) which is used in [7]

to implement a specific policy, as an example, a placement mechanism specifies the technique used to place operators on processing nodes, while a placement policy is used to decide where the operators should be placed. By separating the two aspects, we allow the user to focus on implementing custom policies without having to think about how they are applied. The assumption is that placement components can be differentiated by their placement assignment policies not the techniques used to send operators to their processing host. The separation of mechanism and policy design principle contributes to the flexibility of the system as it makes it possible for the user to easily adapt the components based on their specific needs. Furthermore, it is possible to change the mechanisms, e.g., to improve efficiency without affecting the existing policies. We have therefore designed DCEP-Sim as a platform with Distributed CEP mechanisms which can be used to deploy and evaluate CEP and Placement policies. The separation of mechanisms and policies ensures the desired ease of use of the DCEP-Sim.

2.3 CEP engine

A CEP system takes streams of events from different sources and processes them based on stored rules. When new events are produced, they are forwarded to either the sink or an event broker. As such, four CEP engine sub-components are identified in [7]: a Receiver, a Detector, a Producer, and a Forwarder component. In this section, we show how these sub-components are designed and how they work together. The tasks performed by the Receiver component are not modeled in DCEP-Sim, therefore, it is ignored in the following discussion.

CEP rules describe how to process incoming events and produce new ones. Conceptually, a CEP rule is composed of two parts: a ***Condition*** part and an ***Action*** part [7]. Considering the two parts of a CEP rule, we can deduce a two stage event processing iteration. In the first stage, the CEP engine asserts the condition part of a CEP rule with incoming streams of events as input. When a sequence of events matches the condition, the second stage is triggered to apply the predefined action. This view of CEP suggests two core components: the ***Detector*** component responsible for event pattern detection, and the ***Production*** component responsible for producing an event notification for the detected pattern or a command towards an actuator. We design two classes of objects which are used by the Detector to perform its task. One is an operator class which implements one or more unary, logic or sequence CEP operators. The Detector relies on instances of this class to do the actual processing. Consequently, we enable the implementation of a large variety of operators which can be used to process events in different manners. The expressiveness of CEP rules lies in their ability to capture ordering and timing relationships between sequences of events. Therefore, a Detector component needs to maintain a history of events it has seen, to capture ordering and timing relationships between them. Therefore, we have designed an additional class (***Buffer Manager***), responsible for maintaining the history of the events received by the CEP engine. The Buffer Manager is complex in that it needs to implement the actual timing constraints of CEP logic operators. It implements selection, consumption, and load shedding policies, and mechanisms to store and retrieve events from their respective buffers. The selection policy determines which events are used to match the condition part of a CEP rule, and the consumption policy specifies whether matched events can be reused in the following CEP iterations. The load shedding policy is used to deal with bursty incoming events. In this paper, we focus on the selection and consumption policies, and leave load shedding for future work. To ensure the generality and flexibility requirements for DCEP-Sim, we design and implement the mechanism of the Buffer Manager, and define an interface which can be used to implement custom buffer management policies.

When an expected pattern is matched in the event stream, the corresponding event sequence is forwarded to a ***Producer*** component. The Producer component is responsible for creating new events based on the Action and event Type encoded in the CEP query.

The ***Forwarder*** component is responsible for forwarding events produced by the Producer component towards their destination. As such, once an event notification is created by the Producer component, the Forwarder components uses the Placement component to determine where the event should be forwarded.

2.4 Placement

The Placement component is responsible for assigning operators to event brokers and performing event routing and forwarding between them. The Placement component maintains an event routing table which the Forwarder uses to forward events to other event brokers or the Sink. The Placement component uses a placement policy to determine where operators should be placed during the initial placement and subsequent placement adaptation. To ensure the generality and flexibility of DCEP-Sim, we separate the implementation of the Placement component into two classes: a placement mechanism class, and a placement policy class. The two classes are combined at run time to perform placement, placement adaptation, event routing, and event forwarding. The placement mechanism implements the actual forwarding and placement of operators inside the network, in addition to event forwarding. The placement component policy class implements placement assignment and event routing algorithms. We design and implement placement mechanisms as part of DCEP-Sim, and define an interface which should facilitate easy implementation and deployment of placement policies.

2.5 Additional components

We design additional components responsible for generating events, CEP queries, and network communication; the ***Data source***, the ***Sink***, and the ***Communication*** components. To enable internal communication of DCEP-Sim components we design a ***Dispatcher*** component which facilitates communication between the Placement, CEP engine, Communication component, Sink and Data source object.

The Data source component is responsible for generating events at a predefined rate to simulate, for example, a sensor. The Sink component generates CEP queries which are then sent to DCEP-Sim. The Data source and Sink component must be pre-configured with the type of events and Query the generate. The Communication component is used by DCEP-Sim for network communication

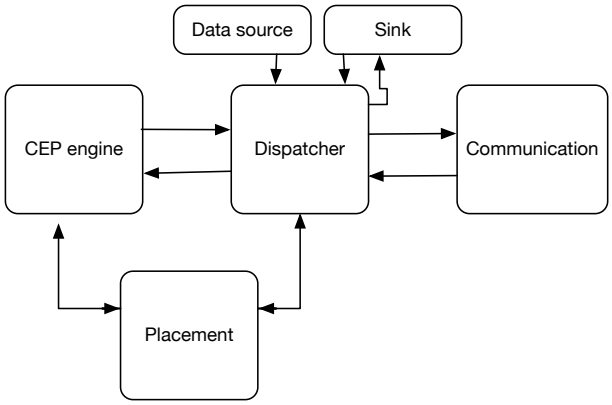

Figure 1: An overview of DCEP-Sim components

in Distributed CEP scenarios. The communication component uses the UDP transport protocol for network communication. The Dispatcher component interacts with the Communication component to send or receive events, operators or meta data.

2.6 Component interaction

When the Sink generates a CEP Query, it is sent to the Dispatcher which forwards it to the Placement component. Based on the current CEP deployment, the Placement component uses a placement assignment policy object to determine where the query should be sent. The query can either be placed locally or sent to a remote node. If the query is placed locally, it is forwarded to the local CEP engine.

The CEP engine creates and stores Operator instance(s) which are used by the Detector component to match incoming events. Before the operator instance is stored, it is initialized by configuring the selection and consumption policies and a Buffer Manager instance.

When a data source generates an event, it is sent to the Dispatcher. The Dispatcher sends the event to the Placement component, which uses an event routing table to determine whether to forward the event to the local CEP engine or to a remote one. In a centralized scenario, the event must be forwarded to a remote sink for processing. In this case, the Placement component sends the event to the Dispatcher with a destination address for the sink node.

The Dispatcher sends the event to the Communication component along with the destination address. The Communication component uses UDP to send the event to its destination.

When an event is received by the communication component, it is sent to the Dispatcher. The Dispatcher sends the event to the Placement component which needs to determine where the event should be forwarded. If it is a final event, it is forwarded directly to the sink component. Otherwise, the event is sent to the local CEP engine.

Inside the CEP engine, the detector loads all operators awaiting the current event and passes it to each one of them, one after the other. Each operator has its own Buffer Manager which it uses to apply its selection and consumption policies.
If there is a match, all the events that were selected are forwarded to the Producer.
The Producer uses the action part of the query to determine the type of the event to produce. Once a new event is created, it is sent to the Forwarder component which uses the Placement component to determine where the new event should be sent. An event notification with a destination address is created and forwarded to the Dispatcher.
If the destination address corresponds to the local node, the event is forwarded to the sink application, otherwise, it is sent to the Communication component.

3 NS-3

In the following subsections, we present an overview of ns-3 concepts that are used as building blocks for the implementation of DCEP-Sim. Afterwards, we describe how the ns-3 simulation scripts and modules are developed. The aim is to lay the foundations for the description of the implementation of DCEP-Sim.

3.1 Overview

ns-3 is a discrete event network simulator with a large number of network protocol models. ns-3 models are designed to be highly realistic and are therefore popular within the network research community.

The ns-3 simulation framework can be used in two ways: performing simulations with existing models, or extending existing models. It is possible to add new models to existing ns-3 modules, or create a new module with models. We have developed the DCEP-Sim simulator as an ns-3 module.

In essence, ns-3 constitutes a set of libraries which can be statically or dynamically linked to a C++ main program (referred to as a script) to create and run a simulation. The ns-3 simulation script typically creates ns-3 nodes, installs the Internet stack and applications on them, and builds a network topology. It is also responsible for starting and stopping the simulation [3]. ns-3 models represent internet protocols and networks, allowing the network research community to build simulations of Internet systems on top of them. Furthermore, ns-3 provides additional tools to support researcher with tasks such as simulation data gathering and analysis.

3.2 Concepts

The core abstractions used in ns-3 are Node, Application, Netdevice, and Channel. The Node is a computing device abstraction which can be connected to a network. An ns-3 Application is a user program abstraction which runs on a ns-3 Node and generates some activities to simulate the real world. The Channel abstraction represents the media over which data flows in a network (e.g., IEEE 802.11). The Netdevice abstraction represents both the device driver and the simulated hardware. More complex models are built on top of these core abstractions and are organized into modules. For example, the CSMA module contains models for CSMA Channel and Net Device which are necessary to simulate a CSMA network topology.

Each of the ns-3 abstractions is represented in C++ by classes which model their respective behavior. A class representing an ns-3 model can use one of the three ns-3 base classes: the ns-3 *Object*, the *ObjectBase*, and the *RefCountBase*. The ns-3 Object base class offers three main special properties:

- the ns-3 type and attribute system,
- the object aggregation system, and
- a smart-pointer reference counting system.

The attribute system enables easy parameterization of ns-3 models. This is achieved through the ns-3 attribute namespace which makes it possible to access internal properties of a model without having to use C++ pointers. This makes ns-3 simulations very flexible as any model property in a simulation can be accessed, modified, or monitored at run time without using C++ pointers.

The aggregation system makes it possible to easily extend ns-3 models without the limitations of object-oriented inheritance. The aggregation system solves the ***Fragile base class problem*** of object-oriented programming systems. The fragile base class problem occurs when seemingly unharmful modifications to the base class break its sub-classes. Therefore, instead of using inheritance to extend ns-3 models with new features, new classes implementing the features are defined and their instances aggregated to the base model at runtime. The aggregation system is used to extend any of the models from the simulator.

ns-3 models are grouped into modules which model specific Internet sub-systems, the protocols or networks. An ns-3 module contains a set of C++ programs examples on how to use the module, a set of models which constitute the model and helper classes to easily create and configure instances of the module.

4 IMPLEMENTATION

DCEP-Sim is developed as an ns-3 module. The components described in the design section are implemented as sub-classes of the ns-3 Object class. The aim is to be able to use the attribute and aggregation systems offered by ns-3 and ensure maximum flexibility and extensibility of the simulator. In addition to the models described in the design section, we have developed a CEP engine wrapper class for the Detector, Producer and Forwarder components. Additionally, we have developed an ns-3 application (DCEP application) works both as a dispatcher in the simulator and as a configuration interface for the simulation script. All the simulator objects are aggregated to the ns-3 application, and the application is responsible for configuring and initializing them. From an ns-3 simulation script point of view, the DCEP application 'is' the DCEP-Sim simulator. Consequently, the models of the simulator are configured through the DCEP application from the simulation script. Finally, the simulation is run and stopped through the DCEP application.

To apply the separation of policy mechanism principle discussed in the design section, the Placement and Buffer Manager were implemented as abstract classes. To ensure the extensibility of the simulator, any additional features can be added to the models through the ns-3 aggregation system. We have also developed a simulation script for demonstration and evaluation purposes.

In the following sections, we present the implementation of DCEP-Sim data structures, models and the simulation script used for evaluation.

4.1 Data structures

The main data structure classes developed are: the ***Event***, the ***EventPattern***, the ***Query*** and the ***Operator***. The Event class defines states for the type, location, and timestamp of an event. The EventPattern defines a logical function which represents the condition to match. The Query defines the type of events it consumes and the event type to produce when an event pattern is matched. The Operator class is designed as an abstract class with two pure virtual functions: ***configure*** and ***evaluate***.The configure function must perform operator initialization tasks such as: creating a concrete Buffer manager and aggregating it to the operator instance, initializing the buffers, etc. A concrete sub class of the operator abstract class should either define member functions which implement different logic constructs such as: conjunction, disjunction, repetition, or negation. Alternatively, objects which implement these logic functions can be aggregated to the operator instance during operator initialization. The evaluate function must implement the actual event matching with the help of the Buffer manager aggregated to it. The evaluate function returns a sequence of events which matched the condition defined by a stored query.

To enforce the separation of policy and mechanism, we implement the Buffer manager model as an abstract class. The Buffer manager abstract class defines functions which implement common buffer mechanisms, and virtual functions which need to be implemented by Buffer manager sub-class creators to implement selection, consumption, and load shedding policies.

4.2 Models

Algorithm 1 instantiate_query

```
1: function INSTANTIATE_QUERY(query)
2:     cop ← CompositionOperator  ▷ the CompositionOperator object is aggregated to the query
3:     query ← cop
4:     cop CONFIGURE
```

The CEPEngine class acts as a wrapper for the Detector, Producer, and Forwarder. The CEPEngine class defines functions to process an event and process queries. The ***process_event*** function uses two helper function to initialize the query (see Algorithm 1) and store the query in a local query pool. The function ***instantiate_query*** is responsible for creating an instance of an Operator class, configuring it, and aggregating it to the query being intialized. The ***process_event*** function simply forwards the incoming event to the Detector.

The Detector class defines a method to process events (see Algorithm 2). The purpose of the method is to retrieve all queries stored locally and use the instance of an Operator subclass aggregated to each one of them to process the incoming event. For each Operator instance retrieved, the detector calls the evaluate function which returns a sequence of events if there was a match. When the evaluate function returns a sequence of events, the process_event

Algorithm 2 process_event

```
1: function PROCESS_EVENT(Event)
Require: queries                ▷ all queries expecting this event
2:     for all queries do
3:         cop   ▷ CompositionOperator aggregated on the query
4:         cop EVALUATE(Event)
```

function forwards them (along with the corresponding query) to the Producer.

The Producer class uses the information from the Detector to produce a new event object and forwards it to the Forwarder.

The Forwarder class defines a function which forwards the event to the Placement class which has the knowledge concerning the destination of the event, i.e., the sink or the next event broker.

The Placement component is implemented as an abstract class with pure virtual functions representing placement policies for: initial placement and placement adaptation. In addition, the abstract class defines functions and states (event routing table) which implement forwarding mechanisms. To use the simulator, a concrete placement class must be derived from the abstract class and the pure virtual functions implemented. Alternatively, one could use one of our placement classes which are based on our own placement policies. The ns-3 aggregation system can be used to extend the placement class with additional features. This enables a clean and easy way to implement new and complex placement classes without breaking the simulator or the class itself.

The ns-3 application (DCEP application) serves as a 'Facade' to the simulator from the simulation script side. Using the ns-3 attribute system, the simulator is configured from a simulation script through the attributes defined in the DCEP application. The DCEP application uses the attribute values to configure the other models in the simulation. In particular, the DCEP application attributes are used to set the role of the current node where the CEP system is installed (sink, data source, or broker), the deployment model, and any other simulation parameters that might be needed. For example, a DCEP application attribute was created to set the number of events generated by the data sources. In the ***StartApplication*** function, all the components of the simulator are created, initialized, and aggregated by the DCEP application. This makes it possible for them to access it when trying to communicate with other components. The DCEP application also serves as a dispatcher inside the simulator. It defines functions which act as intermediaries between the Sink, data source, Communication, and Placement components.

4.3 Simulation script

The simulator is started, configured, and stopped through the DCEP application in a simulation script. The DCEP application is installed on each ns-3 node in the network topology that is created by the simulation script, configured and started. We have created a DCEP application helper class which is used to collectively create and install DCEP applications on ns-3 nodes: see Algorithm 3.

Afterwards, it is possible to configure DCEP applications individually as shown in Algorithm 4.

Algorithm 3 Bulk DCEP application installation

```
1: DCEPAppHelper DCEPAppHelper
2: ApplicationContainer DCEPApps = DCEPAp-
   pHelper.Install(nodes)
```

Algorithm 4 DCEP application configuration

```
1: DCEPApps.Get(0)-¿SetAttribute("attribute name", Attribute-
   Value(value))
```

Running and stopping the DCEP applications on the network nodes is as simple as shown in Algorithm 5. In Algorithm 5, we are scheduling all applications to run in 1 second, and stop after 12 seconds.

Algorithm 5 Running and stopping the DCEP application

```
1: DCEPApps.Start (Seconds (1.0));
2: DCEPApps.Stop (Seconds (12.0));
```

From this description, it appears how easy it is to configure and run a Distributed CEP simulation. For more details on how to write ns-3 simulation scripts see [5].

5 EVALUATION

The aim of DCEP-Sim is to enable researchers to perform scalable and accurate experiments with Distributed CEP solutions. To achieve this, DCEP-Sim takes advantage of the highly accurate ns-3 network and topology models, while adding minimal overhead to the computational cost of the ns-3 models. Furthermore, it is the objective of DCEP-Sim to facilitate experiments with Distributed CEP solutions. DCEP-Sim achieves this through an easy and comprehensive interface for DCEP-Sim configuration and simulation.

These goals capture qualitative and quantitative aspects. We analyze the qualitative aspects, i.e., DCEP-Sim user friendliness by showing how the experiments were configured and run. The quantitative aspects, i.e., scalability and accuracy are analyzed through the experiments.

DCEP-Sim is not a custom Distributed CEP solution, instead, it implements core CEP mechanisms which are used to implement policies for Distributed CEP solutions. Consequently, the aim is not to evaluate a custom Distributed CEP solution, but to measure the overhead DCEP-Sim adds to the processing cost from ns-3 network and topology models. Ideally, the overhead introduced by DCEP-Sim should be minimal in order to support large-scale Distributed CEP solutions with many brokers, a high workload and complex policies. Minimal overhead from DCEP-Sim implies that we inherit the scalability properties of ns-3.

The overhead introduced by DCEP-Sim cannot directly be measured as computational complexity in the form of additional CPU utilization, because it is the nature of discrete event simulators to finish experiments as fast as possible by claiming as much CPU time as possible. However, higher computational complexity in

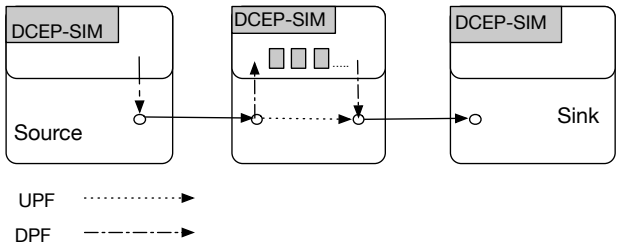

Figure 2: The network topology and simulation setup

ns-3 models leads to a longer simulation time. For example, an experiment with one hour virtual time and low computational complexity might last *x*ms, while one hour virtual time and high computational complexity might last *2*x*ms.
Therefore, we use simulation time as the metrics to study the overhead of DCEP-Sim.

To quantify the overhead of DCEP-Sim; we use as baseline a wired network with chain topology in which UDP packets are forwarded hop-by-hop from one end to the other without any further intermediate processing see Figure 2. We refer to packet forwarding with no intermediate processing as UDP Packet Forwarding (UPF). To measure the overhead of DCEP-Sim we enable on all nodes in the chain CEP and refer to this as DCEP-Sim Packet Forwarding (DPF) when CEP is enabled.

Simulation initialization is performed only once at the beginning of the simulation and is the main contributing factor to the simulation time in large scale simulations. The simulation initialization time is induced by ns-3 network models, not DCEP-Sim. Therefore, we do not include the simulation time related to simulation initialization in the measurements. Instead, we focus on the simulation time between the transmission of the first packet from the source, and the reception of the last packet at the sink.

For event processing we use an OR operator for two reasons: (1) the OR operator is simple and introduces minimal overhead such that we can better isolate the overhead of the basic DCEP-Sim framework, and (2) the OR operator ensures that in the UPF and the DPF experiments the same amount of packets traverses all links in the network.

In the experiments, we study the effect of three input parameters: the number of intermediate nodes (event brokers), the number of transport packets (event notifications), and the number of operators deployed in each event broker.

In Section 5.1, we describe the evaluation setup and demonstrate how easy it is to configure DCEP-Sim and run Distributed CEP experiments. In Section 5.2, 5.3, 5.4, and 5.5, we present and discuss results from the experiments for each of the input parameters studied.

5.1 Evaluation setup

Our simulations were run on a personal computer running ubuntu operating system (16.10), with the following system specification:

- Memory 7.7 GB
- Processor Intel Core i7 CPU 870 @ 2.93GHZ*8
- Graphics Gallium 0.4 on NV98
- OS type 64-bit
- Disk 69.8 GB

To run our ns-3 simulations, we have created a ns-3 simulation script which does the following:

(1) creating ns-3 nodes,
(2) creating a topology,
(3) installing the internet stack,
(4) installing and configuring the DCEP applications on all nodes, and
(5) running and stopping the DCEP applications and the simulation.

The first, second, and fourth steps are explained in Section 4.3, where we also demonstrate how easy it is to perform them. The DCEP applications are used to configure DCEP-Sim models. They expose parameters to study, for example, the number of operators, number of events to generate. The applications are also used to configure the role of different network nodes in the simulation: whether a node is a sink, data source or broker. This is done from the ns-3 simulation script as follows:

```
.....
DCEPApps.Get(index)->SetAttribute("attribute name",
      AttributeValue(value));
```

The attribute name corresponds to the name of the parameter to configure with the given value.

There are three types of nodes in the topology: a source, intermediate nodes or event brokers, and a sink. The DCEP applications are configured based on the role of the node they are installed on. The DCEP application installed on the source is configured with the number of events to send, the event broker is configured with the number of operators to deploy, and the sink is configured with the number of events to expect. This is done as follows:

```
....
/* data source configuration*/
DCEPApps.Get(i)->SetAttribute("IsGenerator", BooleanValue(
      true));
DCEPApps.Get(i)->SetAttribute("number of events",
      UintegerValue (numberOfEvents));
.....
/* event broker configuration */
DCEPApps.Get(i)->SetAttribute("Dummy Processor",
      BooleanValue(true));
 DCEPApps.Get(i)->SetAttribute("number of operators",
       UintegerValue (numberOfOperators));
 .....
 /* sink configuration */
 DCEPApps.Get(i)->SetAttribute("IsSink", BooleanValue(true));

 DCEPApps.Get(i)->SetAttribute("number of events",
       UintegerValue (numberOfEvents));
```

The experiments are grouped based on the DCEP-Sim parameters: the number of nodes, the number of events and the number of operators. In each experiment group, we compare the simulation time between UPF and DPF scenarios. Furthermore, we run 10 simulations for each scenario in each experiment group. In all

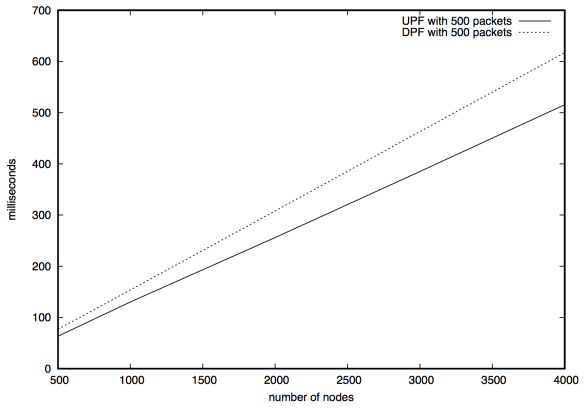

Figure 3: Results showing the overhead introduced by DCEP-Sim when the number of DCEP-Sim instances increases.

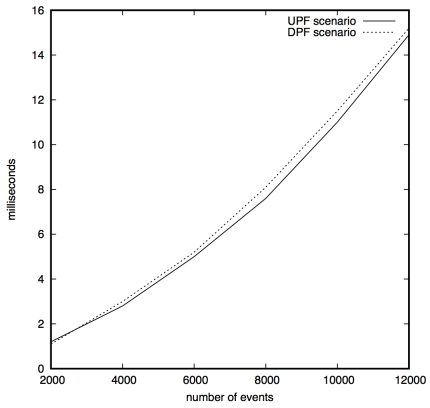

Figure 4: Results showing the overhead introduced by DCEP-Sim when the number of events increases.

simulations, events are generated at a constant rate of one event per virtual time second.

To automate the experiments we use a shell scripts which runs the simulation with command line arguments corresponding to the simulation parameters:

```
....
./waf --run "DCEP-chain --NumberOfNodes=1000 --
    NumberOfOperators=20 --NumberEvents=500 --cep=
    true"
```

After the configuration of the DCEP applications, they are collectively configured to start and stop according to the duration of the virtual simulation time, as follows:

```
....
DCEPApps.Start (Seconds (1.0));
DCEPApps.Stop (Seconds (200));
```

As shown in the code listings, it is easy and straightforward to configure and run experiments with Distributed CEP experiments.

5.2 Varying the number of event brokers

In this experiment, we study the impact that the number of DCEP-Sim instances has on the simulation time overhead. We run simulations with 500, 1000, 2000, 3000, and 4000 brokers/intermediate nodes in addition to the source and sink nodes. At the event brokers, one operator is deployed, and 500 events are sent from the source node towards the sink. The results in Figure 3 are based on the average simulation time from 10 simulations in UPF and DPF. Both UPF and DPF simulation time exhibit a linear increase when the number of nodes increases. DCEP-Sim percentage overhead is low and steady at around 20%. The relative standard deviation for the results varies from 0.4% to 0.6%.
The results show that DCEP-Sim scales well when the number of nodes increases, maintaining a steady overhead.

5.3 Varying the number of events

In this experiment we study the impact of the number of events on the simulation time overhead from DCEP-Sim. The network chain consists of one event broker in DPF, and one intermediate node in UPF. The event broker deploys one operator which is used to process every incoming event. The intermediate node in UPF directly forwards every incoming event towards the sink. Only one operator is deployed at the event broker in order to emphasize the impact of increasing the number of events. For the same reason, we use one event broker for DPF and one intermediate node for UPF. We study the simulation time for UPF and DPF with 2000, 4000, 6000, 8000, 10000, 12000 events.
The results in Figure 4 show a minimal DCEP-Sim percentage overhead which varies between 7% and 2%. The relative standard deviation varies from 2.6% to 8.4%. The significantly low and stable DCEP-Sim overhead shows that an increase in the number of events to process has a minimal impact on the total simulation time.

5.4 Varying the number of operators

In this experiment, we vary the number of operators deployed at the event broker in order to study how the increase in operator workload affects DCEP-Sim overhead. We run the experiment with one source, an event broker for DPF and an intermediate node for UPF, and a sink. The source generates a total of 10000 events. We use one event broker in order to remove the overhead introduced by the number of nodes and emphasize the overhead introduced by the number of operators.
Results from Figure 5 show that DCEP-Sim percentage overhead increases by ca 15% every time the number of operators increase by 10. The increase in overhead is relatively low considering the number of operators added on the intermediate node and the small simulation time scale. Another positive insight is the fact that the simulation time per operator decreases when the number of operators increases.

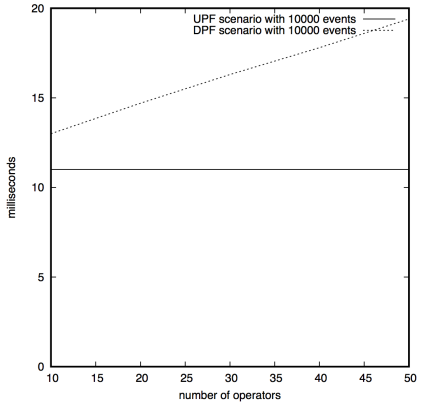

Figure 5: Results showing the overhead introduced by DCEP-Sim when the number of operators per event broker increases.

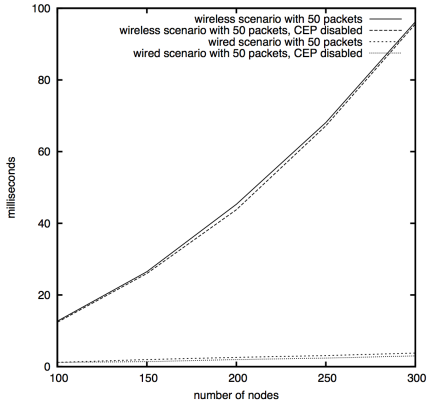

Figure 6: simulation time from wireless vs wired network scenario

5.5 Wireless vs wired network scenarios

In this experiment, we create a wireless chain network topology with no networking infrastructure, i.e., the nodes run in promiscuous mode. The nodes are connected in a adhoc manner in such away that only consecutive nodes are within each others' range. This enables us to simulate a wireless chained topology.
The remaining simulation setup is similar to previous experiments.

ns-3 wireless topology models are significantly more complex than models for wired networks, and introduce a high processing cost during simulations. The accuracy of these models comes with a penalty in high simulation time. Our aim is to add a minimal overhead in such network scenarios in order enable large scale experiment with Distributed CEP solutions implementing complex CEP policies. Figure 6 shows the high simulation time cost from the wireless scenario compared to the wired scenario. DCEP-Sim overhead is only 0.6% to 3% in the wireless scenario. In the wired network scenario, DCEP-Sim overhead varies from 43% to 26%. Because the simulation time from DCEP-Sim is the same in wireless and wired scenario, its overhead is dwarfed by high simulation time from the wireless network scenario.

6 CONCLUSION

Evaluation of distributed, potentially mobile systems, like Distributed CEP is a challenging task. To be able to perform a proper analysis of the system under test sufficiently representative, accurate, and understandable data is needed. For experimental approaches it is important that the experiments are repeatable and that results from systems with similar purposes are comparable. Obviously, results from real world experiments are representative, but only for the context in which the experiments are performed. The Internet is a dynamic system with continuous changes in available bandwidth, queue length in routers, background load of computing nodes etc. This dynamicity, which is especially high in mobile networks, implies that results might not be reproducible and comparable in real world experiments. Furthermore, this approach is quite expensive and time consuming if the systems under test are large. Therefore, simulation and emulation play an valuable role in the evaluation of Distributed CEP. However, it is important that the network simulators and the models they use are accurate, because the network can have a strong impact on the performance of Distributed CEP solutions. For example, to perform operator placement, the metric of network usage, which is the sum of the delay bandwidth product of all links in an operator network, is used in many recent placement proposals. Obviously, network simulators that have been developed over many years in the networking community are better suited to model accurately the network than simulators that are crafted for one particular research study.

The decision whether to use emulation (i.e., deploy a real world Distributed CEP prototype on top of a network simulator) or simulation is a trade-off between convenience to use the same Distributed CEP prototype in emulation studies and real-world studies versus generality, openness, and scalability of the simulation approach that is proposed in this work. The boundary between the network simulator and a real world prototype implies performance penalties and limits the scalability of emulation experiments. This is avoided in DCEP-Sim by implementing it as simulation models in the network simulator ns-3. The basic components of DCEP-Sim are based on the CEP fundamentals introduced by Cugola et al [7] such that Distributed CEP implementations in DCEP-Sim are representative for a broad range of CEP prototypes. By applying the well established engineering principle to separate policy and mechanisms we achieve an architecture and implementation of a Distributed CEP skeleton that makes it very easy to deploy in DCEP-Sim the experimenters' choice of selection, production, load shedding, and placement policies. ns-3 facilitates this by providing besides inheritance in class hierarchies the concept of object aggregation to specialize policies.

The motivation to design and implement DCEP-Sim was given by the need for an appropriate tool to test and evaluate Distributed

CEP systems in our ongoing research work. The only simulator with similar goals of DCEP-Sim we could identify is CEP-Sim.
DCEP-Sim and CEP-Sim share the goal to enable simulation of Distributed CEP systems for research purposes. CEP-Sim models user queries as Directed Acyclic Graphs composed of data producer, operators, and data consumers. Data generators are used in CEP-Sim to feed events into the data producers, which forward the events towards the data consumers through a set of operators. The CEP-Sim query and processing model is similar to the event processing overlay in DCEP-Sim, which is composed of data sources, event brokers and event consumers (or sinks). We model user queries as sets of atomic and composite operators which are placed on data sources and processing nodes respectively. Events are generated by the atomic operators and forwarded towards the event consumers through the composite operators. In DCEP-Sim, it is possible to define an operator which is used to process incoming events. It is, therefore, possible to implement CEP-Sim operators (both stateless and windowed) using DCEP-Sim operator abstraction and use them to process incoming events using their original event processing algorithms. However, CEP-Sim and DCEP-Sim were designed to be used by researchers studying different system environments, i.e., cloud environment vs. static and mobile networks. Additionally, the two simulators are built on two different simulation platforms (CloudSim and Ns-3) which focus on different computing and networking aspects. Ns-3 focuses on networking simulation, while CloudSim focuses on cloud computing. The choice of these simulation platforms has a direct impact on the design of the two simulators. CEP-Sim targets cloud computing environments and is therefore designed for efficient processing and resource consumption in a cloud environment. The basic unit of simulation is a set of operators placed on one Virtual Machine with predefined algorithms for operator scheduling. A scheduling algorithm therefore controls operator processing instead of incoming events. This particular event processing approach in CEP-Sim is due to the fact that CloudSim offers a batch processing model, which is somewhat unsuited for streaming systems like CEP. In CEP-Sim, a batch or the basic processing unit is an 'event set'. As such, in CEP-Sim, operators exchange event-sets instead of individual events. In DCEP-Sim, operator processing is event driven and operators exchange individual events in streams between them.

The goal of the DCEP-Sim evaluation in this paper is not to evaluate particular policies, but instead evaluate the usefulness of DCEP-Sim for experimental evaluation of Distributed CEP solutions. Main aspects of usefulness relate to ease of use, scalability and accuracy. Ease of use in turn refers to (1) the effort to configure and run experiments and to collect measurement data for the evaluation; and (2) to the complexity and effort to add respectively change policies, for example to compare alternative placement policies. As shown in the evaluation section, the DCEP application is used to easily configure DCEP-Sim instances in bulk or individually. Furthermore, it is possible to define DCEP-Sim parameters through the command line arguments passed to the simulation script. This makes it possible to automate the simulation with varying DCEP-Sim parameters.

To inherit the scalabilityof ns-3, it is important that DCEP-Sim does not introduce substantial overhead in terms of simulation time. In complex network scenarios such as MANETs, it is critical for DCEP-Sim to introduce minimal overhead due to high simulation time cost from ns-3 models. Results from the evaluation show that DCEP-Sim scales well when as the number of nodes increases, maintaining a steady overhead of 20%.

In wireless networks, DCEP-Sim simulation time varies between 0.6% to 3%. Since the simulation time from DCEP-Sim is the same in wireless and wired scenarios, its overhead is dwarfed by the high simulation time from the wireless scenario. The high simulation time in the wireless scenario is due to significantly more complex network models.
Increasing the number of events has minimal impact on DCEP-Sim percentage overhead which varies between 2% and 7%.
As the number of operators increases by 10, DCEP-Sim overhead grows by 15%; a minimal increase considering the time scale (15% corresponds to 1.6 milliseconds). Furthermore, the simulation time overhead per operator decreases from 11% in a scenario with 10 operators, to 3.5% in a scenario with 50 operators.
Overall, the results from our evaluation show that DCEP-Sim scales well when the number of nodes, events or operators increases.

Currently, we develop in DCEP-Sim particular selection and consumption policies to use them for our research in placement policies for Mobile Distributed CEP. Ultimately, the goal is to establish a DCEP-Sim open source project.

The most crucial performance parameters in Distributed CEP are all related to network traffic and its consequences. Therefore, it is important to use good network models as they can be found in ns-3. While CEP processing is modeled in DCEP-Sim, the fact that ns-3 is a discrete event simulator leads to the fact that software execution time, like operator processing, cannot be measured because it is zero. Introducing models for software execution in discrete event simulation is a big future challenge to address.

ACKNOWLEDGEMENT

We want to thank the reviewers for their extensive and insightful comments.

REFERENCES

[1] 2017. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. (2017). http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf retrieved 2017-01-20.
[2] 2017. NS-3. (2017). https://www.nsnam.org/ retrieved 2017-01-20.
[3] 2017. ns-3 network simulator. (2017). https://www.nsnam.org/docs/manual/ns-3-manual.pdf retrieved 2017-01-20.
[4] 2017. PeerSim. (January 2017). http://peersim.sourceforge.net/ retrieved 2017-01-20.
[5] 2017. Writing scripts. (2017). https://www.nsnam.org/support/faq/writing-scripts/ retreived 2017-02-10.
[6] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, A. F. De Rose, and Rajkumar Buyya. 2011. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Softw. Pract. Exper.* 41, 1 (Jan. 2011), 23–50. https://doi.org/10.1002/spe.995
[7] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages. https://doi.org/10.1145/2187671.2187677
[8] Wilson A. Higashino, Miriam A.M. Capretz, and Luiz F. Bittencourt. 2016. CEPSim: Modelling and simulation of Complex Event Processing systems in cloud environments. *Future Generation Computer Systems* 65 (2016), 122 – 139. https://doi.org/10.1016/j.future.2015.10.023 Special Issue on Big Data in the Cloud.
[9] Stein Kristiansen and Thomas Plagemann. 2011. Accuracy and Scalability of Ns-2's Distributed Emulation Extension. *Simulation* 87, 1-2 (Jan. 2011), 45–65. https://doi.org/10.1177/0037549710370215

[10] Ahmed Sobeih, Wei peng Chen, Jennifer C. Hou, Lu chuan Kung, Ning Li, Hyuk Lim, Hung ying Tyan, and Honghai Zhang. 2005. J-Sim: A simulation and emulation environment for wireless sensor networks. *IEEE Wireless Communications magazine* 13 (2005), 2006.

[11] Fabrice Starks, Vera Goebel, Stein Kristiansen, and Thomas Plagemann. February 2017. Mobile Distributed Complex Event Processing - Ubi Sumus? Quo Vadimus? (February 2017). accepted for publication in Mobile Big Data- A Roadmap from Models to Technologies, Springer 2017, available at https://www.duo.uio.no/bitstream/handle/10852/55328/MobileDCEP.pdf.

[12] András Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools '08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, Article 60, 10 pages. http://dl.acm.org/citation.cfm?id=1416222.1416290

[13] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. 1974. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM* 17, 6 (June 1974), 337–345. https://doi.org/10.1145/355616.364017