Tutorial: DCEP-Sim: An Open Simulation Framework for Distributed CEP

Introduction for Users and Prospective Developers

Fabrice Starks
University of Oslo
Norway
fabriceb@ifi.uio.no

Stein Kristiansen
University of Oslo
Norway
steikr@ifi.uio.no

Thomas Plagemann
University of Oslo
Norway
plageman@ifi.uio.no

ABSTRACT

Evaluation of Distributed Complex Event Processing (CEP) systems is a rather challenging task. To simplify this task, we developed the open simulation framework for Distributed CEP, called DCEP-Sim. The goal of this tutorial is to facilitate the process of using DCEP-Sim. Since DCEP-Sim is designed and implemented in the popular network simulator ns-3 we introduce the most important concepts of ns-3. Simulations in ns-3 are configured and executed though a main program called an ns-3 script. We use a simple example script to explain how simulations with DCEP-Sim are set up and executed. To give an idea how DCEP-Sim can be adjusted to particular needs, we explain how DCEP-Sim can be adapted (e.g., through changing the workload and the network topology) and how new Distributed CEP solutions can be added by explaining how to add a new operator to DCEP-Sim.

CCS CONCEPTS

• **Computing methodologies → Simulation tools**;

KEYWORDS

Distributed Complex Event Processing, Simulation

ACM Reference Format:
Fabrice Starks, Stein Kristiansen, and Thomas Plagemann. 2018. Tutorial: DCEP-Sim: An Open Simulation Framework for Distributed CEP: Introduction for Users and Prospective Developers. In *DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems, June 25–29, 2018, Hamilton, New Zealand.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3210284.3219501

1 INTRODUCTION

DCEP-Sim is an open simulation framework to simplify the evaluation of Distributed Complex Event Processing (CEP) systems, including the comparison of different Distributed CEP solutions [5], [4]. DCEP-Sim is implemented as an application in the popular network simulator ns-3 to achieve representative and accurate simulation results for all kinds of networks, including mobile networks [2], [3]. It fully leverages all features of ns-3 that (1) allow easy setup and configuration of simulations, (2) the large collection of existing networking related simulation models, and (3) the approach to easily adapt existing simulation models and add new simulation models. Therefore, it is useful for users of DCEP-Sim and mandatory for prospective developers to understand the concepts of ns-3. Consequently, this DCEP-Sim tutorial contains a brief introduction of the most important ns-3 concepts.

DCEP-Sim itself is designed and implemented to be open for arbitrary extensions, like adding new operators, new placement policies, and new placement adaptation policies and mechanisms. The existing open source solution of DCEP-Sim can be therefore seen as an extensible Distributed CEP skeleton for ns-3. The classical software engineering principles of separation of concerns and the separation of policies and mechanisms are applied to the CEP fundamentals presented in [1] and result in an architecture with four main components (see Figure 1).

The four main components are implemented as sub-classes of the ns-3 `Object` class and the CEP engine is a wrapper class for the detector, producer and forwarder. Due to space restrictions we refer the reader to [5] for a detailed explanation of the design and implementation of the DCEP-Sim components. The DCEP-Sim code base contains several example setups for simulations. In order to perform particular experiments, one might want to change (1) the workload which is defined by Source components that produce atomic events and a Sink component that provides the query, (2) the network topology in terms of number of nodes, connectivity, or mobility pattern, (3) change existing or add new placement policies, and (4) add new operators for the CEP engine. In order to explain how this can be done, we briefly introduce in the next section the most important ns-3 concepts. Afterwards, we follow a hands-on approach to explain how simulation experiments with DCEP-Sim are configured and executed with a so-called *script*. Furthermore, we explain how to change the workload for experiments and how to extend DCEP-Sim with new operators to illustrate the extensibility of DCEP-Sim.

2 BRIEF INTRODUCTION TO NS-3

ns-3 is one of the most widely used open-source discrete event network simulators. As a general-purpose network simulator, its model base spans a wide range of models of nodes, communication software and traffic, network devices, and network media. The models are relatively detailed compared to other simulators, facilitating realistic simulations, emulations, and rapid prototyping

ACM ISBN 978-1-4503-5782-1/18/06.
https://doi.org/10.1145/3210284.3219501

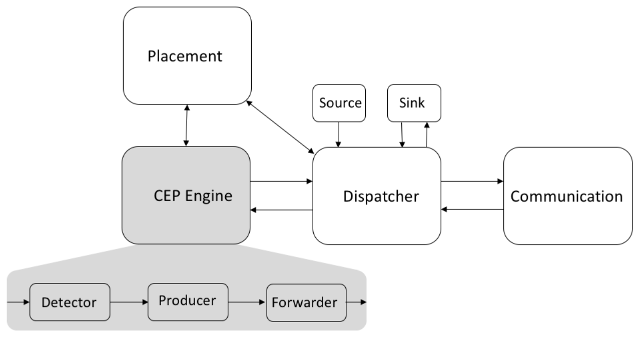

Figure 1: DCEP-Sim Architecture

of real-world implementations. At the same time the models impose sufficiently low computational overhead to enable practical simulations of large-scale networks.

ns-3 users can be roughly classified as *experimenters* that conduct network experiments and *model developers* that extend the ns-3 model base. Thus, ns-3 supports usability and extensibility and provides facilities that are tailored specifically to each user group.

2.1 Facilities for Experimenters

Experiments with discrete event simulators are typically specified via a simulation script. This script constitutes the primary interface for experimenters. In ns-3, this script is a `main()` function written in C++[1] and compiled and linked with the required models upon simulation execution. The script describes how to construct the models and events that make up the simulated network scenario. This includes the instantiation and parametrization of models for nodes, links, communication protocols and node movements, the specification of traffic workload, and the scheduling of pre-defined events at given points in simulation time. Examples of the latter are events that initiate and terminate the transmission of traffic, and the event that terminates the simulation.

ns-3 provides three key facilities to enable scripting style implementation of the above-mentioned `main()` function, such that the simulation script can be implemented with only rudimentary C++ knowledge: (1) *helper* and *containers*, (2) the *attribute system*, and (3) the *data collection framework*. Helpers leverage the fact that most network configurations are similar across experiments, e.g., mobile networks often consist of mobile nodes that use identical Internet stacks and network interface cards.

ns-3 helpers expose to the experimenter easy-to-use interfaces that encapsulate common tasks required to configure and install models on network nodes. Helpers are used in conjunction with containers, to allow the same configuration and model instantiation to be repeated for many nodes at once. For instance, with a single call to `InternetStackHelper::install( NodeContainer c)`, models for the complete Internet stack, including pre-configured instances of the ARP, ICMP, IP, UDP, and TCP protocols, are installed on all nodes in the container c. DCEP-Sim provides for example helpers to install DCEP applications on all nodes that are part of the DCEP overlay with the simple command `DcepAppHelper::Install (NodeContainer c)`.

[1]Alternatively, ns-3 allows to write the script in Python. This is however not as common, and is not covered in this tutorial.

Simulation models typically have attributes that affect their behavior. The *attribute system* provides an intuitive interface to expose model attributes. All objects inheriting from the `ObjectBase` class can be assigned such attributes. For instance, all `Dcep` objects have attributes to determine which placement policy and adaptation policy to use for the placement of operators and adaptation of the operator graph. Attributes can be set directly by calling the `SetAttribute(string name, const AttributeValue &value)` member function inherited from the ObjectBase class. Alternatively, they can be set by specifying default values at the beginning of the simulation script after which all objects of the specified type are initialized with the given attribute values.

ns-3 provides detailed tracing of simulated traffic, e.g., in the form of pcap-traces containing all data sent or received at the network devices. Furthermore, ns-3 provides a separate data collection framework to collect data from other sources. This framework is based on the more fundamental *trace sources* and *trace sinks*. In addition to attributes, classes that inherit from the `ObjectBase` class can be assigned *trace sources* that function as dispatch points for data that is useful for, e.g., simulation output. User-defined functions, called *trace sinks*, can be attached to these at simulation initialization time, and will be called upon the availability of data at the trace source. DCEP-Sim makes extensive use of trace sources. As an example, the `DataSource` class has a trace source named *Event* that provides information about every new event produced by a data source in the operator graph. The data collection framework builds on this tracing framework to provide higher-level services, like storing the data in a particular format (e.g., as SQLLite-formated output), automatically plotting the data using GnuPlot, and performing online statistical analysis during the simulation to, e.g., determine when enough data is available to yield sufficiently narrow confidence intervals in the results.

All attributes and trace sources are addressable via an intuitive, shared attribute namespace, that also supports expressive wildcard characters to address multiple model instances at once. This relieves the user from the need to iterate complicated data structures to gain access to the objects of interest.

2.2 Facilities for Model Developers

Developers benefit additionally from the three ns-3 features: *smart pointers*, *object aggregation*, and *run time typing information*. Extending the model base is required whenever simulation studies involve evaluation of new network components, and the simulator community thrives from the continuous sharing of such models via integration with the mainline model base. This process however puts stringent requirements on the fundamental design principles of the model base, requiring facilities that ensure sustainable extensibility. From more than 20 years of experience with ns-2, one particularly significant obstacle for extensibility has been identified: the problem of the *fragile base class*. As the number of classes inheriting from a given base class increases, there is an increasing probability of breaking any of these when modifying the base class. ns-3 addresses this problem via an engineering principle called *object aggregation.* Instead of inheriting from a base class, new functionality is added by aggregating objects to each other. All objects in an aggregation can access each other via their type, i.e., this depends on the ns-3 facility *run-time type information.* DCEP-Sim makes extensive use of object aggregation, e.g., to build particular DCEP instances according to user-provided attribute values. For instance, object aggregation is used by the DCEP-Sim helpers to aggregate particular placement and adaptation policies according to the values defined by the experimenter via the attribute system. This way, model developers can easily extend the set of available policies by adding new classes to the pool of policies.

One particularly cumbersome aspect of C++ programming is the lack of garbage collection. The *smart pointers* of ns-3 help to alleviate this problem. By consistently using smart pointers for a given object type, the smart-pointer sub-system automatically keeps track of the number of references to that object and deletes the object when the reference count reaches zero. Since only objects of classes that inherit from `RefCountBase` are managed by smart pointers, this does not entirely eliminate the possibility of memory leaks, but significantly helps to reduce the problem.

All above-mentioned facilities are provided by three core classes at the base of the ns-3 class hierarchy: `Object`, `ObjectBase` and `RefCountBase`. Each of these endow their sub-classes with a particular sub-set of facilities: (1) `ObjectBase` objects can be given ns-3 attributes and trace sources and can be aggregated with other `ObjectBase objects`, (2) `RefCountBase` objects can be managed by smart-pointers, and (3) the `Object` class provides all of the above features and is therefore often the best choice for developers to extend.

3 A SIMPLE DCEP-SIM SCRIPT

In this section, we take a step-by-step look at the example script called dcep-example.cc. At first, the logging levels for the Placement, Dcep, Detector and Communication components are set with the `LogComponentEnable()` function (see Listing 1).

Listing 1: Setting logging level

```
LogComponentEnable ("Placement", LOG_LEVEL_INFO);
LogComponentEnable ("Dcep", LOG_LEVEL_INFO);
LogComponentEnable ("Detector", LOG_LEVEL_INFO);
LogComponentEnable ("Communication",
 LOG_LEVEL_INFO);
```

The next step is to create a container for `numNodes` number of nodes (see Listing 2). Afterwards, a wireless network in which the nodes in the network container n move according to the `Constant PositionMobilityModel` is installed.

Listing 2: Create and install network topology

```
uint32_t numNodes = gridWidth*gridWidth;
NodeContainer n;
n.Create (numNodes);

NetDeviceContainer devices = SetupWirelessNetwork(
 n);
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::
 GridPositionAllocator", "MinX", DoubleValue (0.0)
 , "MinY", DoubleValue (0.0), "DeltaX",
 DoubleValue (distance), "DeltaY", DoubleValue (
 distance), "GridWidth", UintegerValue (gridWidth)
 , "LayoutType", StringValue ("RowFirst"));

mobility.SetMobilityModel("ns3::
 ConstantPositionMobilityModel");
mobility.Install (n);
```

The OLSR routing protocol is installed in Listing 3 with the `OlsrHelper`. The network layer protocol IP, and transport protocols TCP and UDP are set up with the `InternetStack-Helper` and installed on the nodes that are part of the network container `n` and IPv4 addresses are assigned to these devices.

Listing 3: Setting up network and transport layer

```
OlsrHelper olsr;
InternetStackHelper internet;
internet.SetRoutingHelper (olsr);
internet.Install (n);
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer iface = ipv4.Assign (
 devices);
```

The main complexity of setting up the Distributed CEP overlay is encapsulated in the `DcepAppHelper`, which installs on all nodes in the network container a Distributed CEP instance and groups them in an application container. Node 0 is configured to serve as sink. All Distributed CEP instances are provided with the sink address and the placement policy to be used. Furthermore, two Distributed CEP instances are configured to serve as sources (see Listing 4).

Listing 4: Setting up the Distributed CEP overlay

```
sinkAddress = Address (iface.GetAddress (0));
DcepAppHelper dcepApphelper;
ApplicationContainer dcepApps = dcepApphelper.
 Install (n);
uint32_t eventCode = 1;

for(uint32_t i = 0; i <= numNodes; i++) {
     dcepApps.Get(i)->SetAttribute("SinkAddress",
      AddressValue (sinkAddress));
     dcepApps.Get(i)->SetAttribute("placement
      policy", StringValue(placementPolicy));

     if(i == 0) { /* sink node*/
           dcepApps.Get(1)->SetAttribute("IsSink",
             BooleanValue(true));
     }
     else if ((i == (numNodes-1)) || (i == (
      numNodes-2))){
          dcepApps.Get(0)->SetAttribute("
           IsGenerator", BooleanValue(true));
          dcepApps.Get(0)->SetAttribute("event
           code", UintegerValue (eventCode++));
```

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

Finally, dcepApps is scheduled to run from Second 1 to 30 and the simulator to stop at Second 35. Afterwards, the simulator is executed and simulates 35 seconds before terminating (see Listing 5).

Listing 5: Running the simulation

```
dcepApps.Start (Seconds (1.0));
dcepApps.Stop (Seconds (30.0));
Simulator::Stop(Seconds(35.0));
Simulator::Run ();
Simulator::Destroy ();
```

4 ADAPTING AND EXTENDING DCEP-SIM

To give the reader a first idea of how to tailor DCEP-Sim to perform particular experiments we briefly explain how the workload can be changed and how new operators can be added.

Distributed CEP instances can be configured in the script as data sources. The current data source model supports uniform traffic and the event rate and the total number of events can be configured in the script. To introduce more complex traffic patterns, e.g., based on statistical distributions, or to generate trace driven event streams the data source model needs to be extended respectively new models created. The function of the DataSource object that generates atomic events is called GenerateCEPEvents() and can be found in the dcep.cc file. In this way it is possible to generate arbitrary event streams with well-defined or random patterns and to replay event stream traces from a file.

Operators are used in the detector class (which is member of the CEP engine wrapper class). The operator implementation is separated from the implementation of the selection and production policies that should be used. When a query is placed, the CEP engine instantiates it by creating an instance of the operator(s) given in the query. The DCEP-Sim operator abstract class defines two virtual functions which need to be implemented by derived classes to add a new operator: the Configure() function to set up the operator and the Evaluate() function to do the actual processing of incoming CEP events. The Configure() function is called to set up the operator. Typically, this function would configure the selection and consumption policies which the operator will apply, in addition to setting up the buffers for events as shown in Listing 6 for the AND operator.

Listing 6: Configuration of AND operator

```
Void AndOperator::Configure(Ptr<Query> q){
        this->queryId = q->id;
        this->event1 = q->inevent1;
        this->event2 = q->inevent2;

        Ptr<BufferManager> bufman =  CreateObject<
          BufferManager >();
        bufman->consumption_policy =
          SELECTED_CONSUMPTION; //default
        bufman->selection_policy =
          SINGLE_SELECTION; //default
        bufman->configure(this);

        this->bufman = bufman;
}
```

When an event is received by the CEP engine, it is passed to every operator expecting it by calling the Evaluate function. The Evaluate function returns true or false based on whether some events have matched the expected pattern. The function also returns a list of all events that matched the pattern through its output parameter. When the Evaluate function returns true, the detector component forwards the list of events that matched, along with the corresponding query, to the producer component.

5 CONCLUSIONS

Currently, DCEP-Sim serves as a tool for our ongoing research on adaptive placement in mobile networks. We provide the current code base as open source and hope that DCEP-Sim might be useful to other researchers and developers and are looking forward to future contributions in form of extensions of DCEP-Sim from the research community. Integrating the Distributed CEP simulation in ns-3 allows to benefit from all ns-3 advantages, including a large number of existing models for network simulation, easy extensibility, and powerful logging ang tracing facilities. To fully leverage these features, it is recommended to become familiar with ns-3. As a tool created for research in our lab it is naturally not yet fully complete and perfect. Some prospective users might find it unsatisfactory that a discrete event simulator like ns-3 is not simulating the execution time of software, e.g., the execution of the CEP instances. Creating models of the execution of CEP instances is subject for future research.

ACKNOWLEDGMENTS

This work has been supported by the Cesar project (The Research Council of Norway, project number 250239).

REFERENCES

[1] 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
[2] NS-3. 2017. https://www.nsnam.org/ (2017), last vistited 2018-5-11.
[3] ns-3 network simulator. 2018. Release ns-3-dev. https://www.nsnam.org/docs/manual/ns-3-manual.pdf (retrieved 2018-5-11 2018).
[4] F. Starks. 2018. DCEP-Sim repository. http://github.com/fabricesbDCEP-Sim.git, (retrieved 2018-5-14 2018).
[5] Fabrice Starks, Thomas Peter Plagemann, and Stein Kristiansen. 2017. DCEP-Sim: An Open Simulation Framework for Distributed CEP. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 180–190. https://doi.org/10.1145/3093742.3093919