INetCEP: In-Network Complex Event Processing for Information-Centric Networking

(Extended Version)

Manisha Luthra*, Boris Koldehoff†, Jonas Höchst†, Patrick Lampe†, Ali Haider Rizvi*, Ralf Kundel*, Bernd Freisleben†

*Technical University of Darmstadt, Germany
†University of Marburg, Germany

{firstname.lastname}@kom.tu-darmstadt.de
{lastname}@informatik.uni-marburg.de

Abstract—Emerging network architectures like Information-Centric Networking (ICN) offer simplicity in the data plane by addressing named data. Such flexibility opens up the possibility to move data processing inside network elements for high-performance computation, known as in-network processing. However, existing ICN architectures are limited in terms of (i) in-network processing and (ii) data plane programming abstractions. Such architectures can benefit from Complex Event Processing (CEP) as an in-network processing paradigm to efficiently process data inside the data plane. Yet, it is extremely challenging to integrate CEP because the current communication model of ICN is limited to consumer-initiated interaction that comes with significant overhead in number of requests to process continuous data streams. In contrast, a change to producer-initiated interaction, as favored by CEP, imposes severe limitations for request-reply interactions.

In this paper, we propose an in-network CEP architecture, INetCEP that supports unified interaction patterns (consumer- and producer-initiated). In addition, we provide a CEP query language and facilitate CEP operations while increasing the range of applications that can be supported by ICN. We provide an open source implementation and evaluation of INetCEP over an ICN architecture, Named Function Networking, and two applications: energy forecasting in smart homes and a disaster scenario.

Index Terms—Complex Event Processing, Information-Centric Networking, In-Network Processing

I. INTRODUCTION

Emerging network architectures like Information-Centric Networking (ICN) simplify the data plane of the current Internet by changing its addressing scheme from named hosts to named data. ICN has evolved as a key paradigm towards a content-centric Internet, as currently adopted by academia and industry, e.g., by Internet2, Cisco, and Intel [6] for real-world deployment [37]. The data plane abstractions of ICN are particularly useful since users can define what data they need instead of identifying where to get it from. Additionally, exploiting data plane programmability on in-network elements of ICN can offer high throughput by processing packets at line rate, while delivering them at low latency, typically known as in-network processing. However, existing ICN architectures like Named Data Networking (NDN) and Named Function Networking (NFN) are restricted in terms of data plane programmability due to lack of (i) in-network processing and (ii) data plane programming abstractions.

This makes Complex Event Processing (CEP) a paradigm of choice for an ICN architecture. CEP is a powerful in-network processing paradigm that takes a query as an input to describe the correlations over a set of incoming data streams in order to deliver data notifications in response to the query. For instance, in a disaster scenario, a heat map query can describe correlations over a set of data streams, e.g., location updates from victims to deliver a heat map distribution of survivors to better coordinate the activities of rescue workers.

However, employing CEP on top of an ICN architecture is extremely challenging. An important challenge is that the communication model of current ICN architectures has strong limitations in supporting the processing of periodic data streams. For instance, NDN uses a consumer-initiated interaction pattern where a consumer pulls data by sending an interest (request) to the network. The NDN network forwards this request to one or more producers that satisfy the request and then forward the data (reply) back to the consumer. For continuous data streams, consumer-initiated interaction poses significant overhead in terms of number of request messages and in the delay until fresh data becomes available.
available. On the other hand, changing to a pure producer-
initiated interaction as favored by CEP is problematic for many
Internet-of-Things (IoT) applications that build on the request-
reply interactions. For example, applications like Amazon
Alexa need personalized request-reply interaction.

In fact, ICN ideally should offer efficient support for both
interaction patterns as part of a unified communication model
as illustrated in Fig. 1. Initial work in the context of content
routing has shown the potential of a unified communication
model [19]. However, performing CEP operations inside the
network while efficiently realizing a unified communication
model in ICN is a challenge that we aim to address in this
paper. Thus, we present a novel INetCEP architecture [8]
with the following contributions:

(i) a unified communication layer to provide the functional-
ility of CEP at the network level,
(ii) a general CEP query language that specifies patterns for
meaningful event detection over the ICN substrate, in the
form of query interests,
(iii) a query processing algorithm to resolve query interests,
and
(iv) an open source implementation and evaluation of the
proposed approach on a state-of-the-art ICN architecture,
ICN, with two IoT case studies.

The paper is organized as follows. In Section II we present
preliminaries required to understand our approach. In Section
III we describe the problem space and our system model.
In Section IV, we present the INetCEP architecture, and in
Section V, we provide an evaluation. In Section VI, we
provide a comparison with related work. Finally, in Section
VII, we discuss possible extensions to our architecture before
concluding in Section IX.

II. BACKGROUND

In this section, briefly explain the building blocks of our
work: Content-Centric Networking (CCN) that evolved
into Named Data Networking (NDN) and Named Function
Networking (NFN), and Complex Event Processing (CEP).

Content-Centric Networking. Jacobson et al. [26]
proposed CCN[1] where communication is consumer-initiated,
consisting of two packets: Interest and Data. A data
object (payload of a Data packet) satisfies an interest if the
name in the Interest packet is a prefix of the name in the
Data packet. Thus, when a packet arrives on a face (identifed
by face_id) of a CCN node, the longest prefix
match is performed on the name and the data is returned based
on a lookup.

CCN data plane: Each CCN node maintains three major
data structures: Forwarding Information Base (FIB), Content
Store (CS) (also known as in-network cache), and Pending
Interest Table (PIT). Once an Interest arrives on a face,
the node first checks its Content Store for a matching Data
packet by name. Upon a match, the Data packet is sent via
the same face it arrived from. Otherwise, the node continues
its search in the PIT that stores all the Interest packets (along
with its incoming and outgoing face) that are not satisfied. If an
entry exists in the PIT, the face is updated and the Interest
is discarded, because an Interest packet has already been
sent upstream. Otherwise, the node looks for a matching FIB
entry and forwards the Interest to the potential source(s)
of the data.

NDN and NFN: NDN [45] emerged as a prominent
architecture that builds on the principles of CCN’s named
data. NFN is another emerging architecture that focuses on
designing functions in addition to named data
by extending the principles of NDN. NFN blends
CPP data computations with network forwarding, by performing
computational tasks across the CCN network. It represents
named functions on the data as Church’s λ-calculus
expressions that are the basis of functional programming. We
to encapsulate CEP operators (cf. next section) as NFN
named functions and hence resolve them in the network.
Yet, the proof-of-concept design of NFN focuses mainly on
resolving functions on top of the CCN substrate. In contrast,
we focus on continuous and discrete computations (push and
pull), expressive representation of the computation tasks and
their efficient distribution (cf. Section V).

Complex Event Processing. CEP can process multiple
online and unbounded data streams using compute units called
operators to deliver meaningful events to the consumers.
The consumers specify interest in the form of a query comprising
of multiple operators. Some of the commonly used operators
are defined in the following.

(i) Filter (σ) checks a condition on the attribute of an
event tuple and forwards the event if the condition is
satisfied.
(ii) Aggregate applies an aggregation function such as
max, min, count, sum, avg, etc., on one or more event
tuples. Hence, the data stream must be bounded to apply
these operations. For this purpose, window can be used.
(iii) Window limits the unbounded data stream to a window
based on time or tuple size, such that operators like
Aggregate can be applied on the selected set of tuples.
(iv) Join (×) combines two data streams to one output
stream based on a filter condition applied on a window
of limited tuples.
(v) Sequence (→) detects causal or temporal relationship
among two events applied on a window of selected event
tuples from a data stream, e.g., if event a caused event b
and event a happened before event b.

These operators can be stateless or stateful. Filter and
Aggregate are stateless operators, while the other operators
are stateful and maintain the state of input tuples before
emitting the complex event and therefore depend on multiple input tuples to be accumulated before actual emission.

III. MOTIVATING USE CASES

Use case I: Disaster Scenario. A natural disaster scenario is a prominent use case of [ICN] architecture research. It is drafted as one of the baseline scenarios in an active Internet Research Task Force (IRTF) draft of [ICN] working groups [36]. A typical disaster management application is to generate a heat map showing live distributions of survivors in a disaster area [29]. An important property of such an application is that the information must be updated continuously to delegate rescue workers to the hot spots and to monitor their operations, and hence is producer-initiated. However, there are limitations of developing such an application using current [ICN] architectures such as [NDN] alone, due to the absence of support for producer-initiated communication.

Use case II: Internet of Things. We consider an IoT application as the second use case for our approach since: (i) it is one of the baseline scenarios for [ICN] architectures [36] and (ii) IoT traffic is among the most used type of traffic in the current Internet [9]. An intrinsic property of such applications is that IoT devices produce continuous data streams, e.g., sensor data that needs to be analysed, filtered, and derived to retrieve meaningful information for the end consumers, and hence it is oriented towards producer-initiated interaction. One such application in the context of smart homes is short term load forecasting of energy consumed by smart plugs, which is useful, e.g., for energy providers. The DEBS grand challenge 2014 focuses on this application.

Although we target the above use cases in the following sections, our solution is not limited to these use cases, but the presented scenarios are representative to cover the design space of our solution.

IV. PROBLEM SPACE

In this section, we first discuss the limitations of using straightforward solutions to motivate the need of our architecture (cf. Section IV-A) and then we explain the system model of INETCEP (cf. Section IV-B).

A. Design Challenges

We discuss the limitations of using straightforward solutions, e.g., standard consumer-initiated, producer-initiated...
Fig. 4: INETCEP communication model supports pull-based communication and fetches latest Data packet. However, this will require additional synchronization mechanisms.

Alternatively, another possibility is to support just producer-initiated transmission while using CCN primitives (cf. Fig. 3). Although this is a viable option for some applications [20], it results in a three-way message exchange of what amounts to a one-way message. The producer sends an asynchronous Interest packet that is not intended to fetch a Data packet from the network but to announce the data name and the callback from the consumer. The consumer then shows interest in the data name, which is fulfilled by a Data packet from the producer.

**Limitation 3:** Besides the overhead generated by a 3-way message, this design has other major issues. The interests leave an in-network state in the PIT of CCN nodes such that data can be fetched by the same path, CCN typically performs a unicast of the Interest packet, so the Data packet can follow the same path to the consumer. Such an application has to support two kinds of Interest packets: a packet that is not supposed to fetch data, and a packet that is supposed to fetch data.

**Limitation 4:** Long-lived Interest packets can be used in place of a query, but this also has multiple side-effects. Similar to multiple interests, long-lived Interest packets will also result in large in-network state (cf. Fig. 2a). In addition, the long-lived Interest packets will have to deal with stale data as explained earlier in Limitation 2 (cf. Fig. 2b). To solve the aforementioned issues, we propose to have both consumer-initiated and producer-initiated interaction patterns coexisting under a unified CCN communication layer. A CCN architecture is unable to achieve this using existing packets and data structures as we saw above. Hence, we propose additional packets as a part of the communication model and handle them while processing CEP queries in the network as defined in Section V-A.

**Limitation 5:** CCN/NDN assumes a hierarchical naming scheme to address named data, e.g., /nodeA/temperature, in order to fetch data objects e.g., 35°C, from the producers. A simple way to specify CEP operations over data would be to represent this using the standard naming scheme, e.g., a min operator as /nodeA/min/temperature. However, there are problems with this approach: (i) the name cannot be used to correlate data from multiple producers, (ii) this would mean the processing is performed always at the consumer, which is inefficient and (iii) this is not extensible and not expressive, since adding more operators would mean appending them in the naming scheme, which reduces readability.

Hence, we need more than just Interest packets that encapsulate name prefixes as stated above to represent CEP queries over a CCN network. We propose an expressive query language that can correlate data from multiple producers and an efficient query parser to execute queries in the network (cf. Section V-B).

**Limitation 6:** The query specified by the consumers must be processed within the CCN network. The CCN resolution engine can resolve only Interest packets to retrieve Data packets based on the matching name prefix, but it cannot express query. A naive way to deal with this is to process the query at the consumer. However, this would overload the network with all the unnecessary data that could have been filtered on the way to the consumer and overload the consumer with all the processing. This might result in a single node of failure, when the data becomes very big. Thus, the processing needs to be performed in the CCN network, e.g., at the broker while being transmitted to the consumer. We provide this in two ways: (i) centralized query processing, where the entire query is processed at a single broker and (ii) distributed query processing, where the query operators are assigned to in-network nodes for processing (cf. Section V-C).

### B. INETCEP System Model

Every CCN node can act either as a producer, a consumer or a broker. Here, a broker is an in-network element, i.e., an INETCEP aware CCN router, while a producer or consumer is an end device, e.g., a sensor or a mobile device. On the one hand, consumers can request a specific data item using an Interest packet, where broker(s) forward(s) the request received by consumers to support anonymous request-reply communication, as illustrated in Fig. 4. The producer replies with a data object contained in a Data packet. On the other hand, broker(s) process(es) the unbounded and ordered data streams generated by producers to provide event-based
communication, which happens as illustrated in Fig. 5 and is explained below.

A producer multicasts the data stream (Data Stream packet) towards the broker network, which disseminates the stream all over the network (push). The Data Stream packet is forwarded to further brokers in the network if there are consumers downstream for query interest (qi). An efficient event dissemination can be achieved by using routing algorithms, e.g., defined in this work [13], by looking at the similarity score of the qi. A consumer issues a query by sending an Add Query Interest packet comprising qi (top Fig. 5). Each qi encapsulates a CEP query q that is processed by interconnected brokers in B forming a broker network. The qi is stored in the Pending Interest Table (PIT) of the receiving broker until a Remove Query Interest packet is received that triggers the removal of qi from the PIT (bottom Fig. 5). Unlike a conventional CEP system, the event-based communication in the INETCEP happens in the underlay CCN network.

The query q induces a directed acyclic operator graph G, where a vertex is an operator ω ∈ Ω and an edge represents the data flow of the data stream D. Each operator ω dictates a processing logic $f_\omega$. We explain the constituents: the communication model, the query model, and the operator graph model below.

**Communication Model.** We provide five types of packets to support both kinds of interaction patterns. The Interest (request) packet is equivalent to CCN's Interest packet that is used by the consumer to specify interest in any named data or named function. The Data (reply) packet is a CCN data packet that satisfies an Interest. It also encapsulates the complex event (ce) as described later. The Data Stream packet represents a data stream of the form $<ts,a_1,\ldots,a_m>$. Here, ts is the time at which a tuple is generated and $a_i$ are the attributes of the tuple. The Add Query Interest packet represents the event of interest in the form of a CEP query q. The Remove Query Interest packet represents the CEP query that must be removed for the respective consumer (so that it no longer receives complex events). The CCN forwarding engine (data plane) is enhanced to handle these packets, as below.

**Query Model.** The INETCEP query language is based on two main design goals: it should deal with both pull (data from relations) and push (time series data streams) kind of traffic, and support standard CEP operators (as identified in Section II) over the CCN data plane.

Thus, a query (q) must be able to capture time series data streams as well as relations of the form $<ts,a_1,\ldots,a_m>$ and define an operator ω with processing logic $f_\omega$ in a way that it is extensible.

**Operator Graph Model.** The operator graph G is a directed acyclic graph of plan nodes. The vertex of the graph is a plan node that encapsulates a single operator (ω), while the links between plan nodes represents the data flow from the bottom of the graph to the top. The operator graph can be processed centrally or collaboratively in a distributed manner by mapping it to the underlay CCN network. In distributed CEP, typically, an operator placement mechanism defines a mapping of an operator graph G onto a set of brokers, to collaboratively process the query. The placement needs to be coordinated with the forwarding decisions for efficient processing over the CCN data plane.

V. INETCEP ARCHITECTURE

We identify the following three broad requirements for the INETCEP architecture from our discussion in the previous section.

**R1** A unified communication layer supporting both producer and consumer initiated communication (cf. Section V-A).

**R2** An expressive CEP query language for specifying the event of interest (cf. Section V-B).

**R3** Resolution of CEP queries by efficient and scalable inter-network query processing (cf. Section V-C).

We address each of these requirements below.

**A. Unified Communication Layer**

In this section, we explain the extension of the CCN data plane to enable CEP. In our approach, each CCN node n ∈ N maintains a Content Store or cache (CS), a Pending Interest Table (PIT), a Forwarding Information Base (FIB) and a CEP engine. In the following, we explain the function of these main building blocks and the data plane handling. Subsequently, we detail on the newly introduced packets Add Query Interest, Remove Query Interest and Data Stream and simultaneously the solution to the limitations identified in Section IV-A.

1) Node Components: The CS stores all the data objects associated with the CCN Interest, additionally, the timestamped data objects associated with the query interests (qi). The data object returns a value associated with the qi, e.g., if the qi is the sum of victims in a disaster location, then the data object contains value 20. Hence, if multiple consumers are interested in the same qi, the query is not reprocessed but the data is fetched directly from the CS. To deal with Limitation 2 of stale cache entries for qi (cf. Section IV-A), we store logical timestamps along with cache entries while storing qi...
TABLE I: Description of differences in traditional ICN vs INetCEP architecture ("-" means no support).

| Characteristic            | Old Architecture | Description | INetCEP | Our Architecture | Description |
|---------------------------|------------------|-------------|---------|------------------|-------------|
| Packet Types (cf. § V-A2) | Interest         | Consumer request | Interest | Consumer request |
|                           | Data             | Producer reply | Data    | Producer reply   |
|                           | -                | -            | Stream  | Data Stream      |
|                           | -                | -            | Add Query Interest | CEP query subscription |
|                           | -                | -            | Remove Query Interest | CEP query unsubscription |
| Data Structures (cf. § V-A1) | PIT             | Stores pending interests of consumer | PIT | Stores pending interests and query interests |
|                           | CS               | Stores data packets | CS | Stores data packets and buffers the data stream for stateful operators |
|                           | FIB              | Stores forwarding information towards producers | FIB | Stores forwarding information towards producers and consumers interested in CEP query |
| Data Processing (cf. § V-B § V-C) | -                | -            | CEP engine | Parse, process and derive complex events |

The PIT stores the pending qi so that the complex event (ce) could follow the path, i.e., the in-network state created in PIT to the consumer. The face information is also stored in the PIT entry to keep track of qi’s consumers. In contrast to consumer-initiated interaction, the ce must be notified to the interested consumers as and when detected in real-time. Thus, as new data is received, the query interests in the PIT are re-evaluated as explained later in Algorithm 1. In the former, we are referring to only continuous Data Stream packet since we also support fetching data from relations (this is handled conventionally using a CCN Interest packet as explained in Section II).

The reasons why we distinguish between Add Query Interest (qi) and CCN Interest packets are: (i) qi is invoked on receipt of Add Query Interest as well as the Data Stream packet, (ii) removal of the PIT entry is not based on a Data packet retrieval but on the reception of the Remove Query Interest packet and (iii) qi retrieves Data packets asynchronously. In summary, we deal with Limitation 3 and Limitation 4 by asynchronously handling the qi instead by 3-way message exchange and efficiently managing PIT entries, respectively. We deal with Limitation 1 by storing qi in PIT and asynchronously delivering event notifications to the consumers.

The FIB table gets populated as the producer multicasts to the broker network leaving a trail to the data source. In this way the data processing is performed efficiently along the path from producer and consumer. Finally, the CEP engine holds the processing logic f_ω for each operator ω and is responsible for parsing, processing, and returning the result to the next node towards consumer (cf. Sections V-B and V-C).

2) Data Plane Handling: In Algorithm 1 (lines 1-12) and Fig. 6, we define the handling of Add Query Interest and Data Stream packets at the broker end in a CCN network. The processing of qi stored in PIT is triggered based on the receipt of these two packets as follows: (i) when an Add Query Interest packet is received at a broker (line 1) and (ii) due to the continuous arrival of new Data Stream packets (line 13). This is in contrast to the PIT entry of CCN in the CS. Hence, always up to-date data objects are stored, while old entries are discarded (cf. Fig. 4).

Algorithm 1: Add Query Interest and Data Stream packet handling.

```plaintext
function ADDQUERYINTEREST(qi)
if qi is found in CS
  data ← CS.FetchContent(qi);
  return data;
  (Discard AddQueryInterest)
else if qi found in PIT; lookup(qi) then
  PROCESSQINPIT(qi, AddQueryInterest)
else if qi found in FIB; lookup(qi) then
  CREATEOPERATORGRAPH(qi) (Refer Algorithm 2);
  Forward AddQueryInterest;
else
  (Discard AddQueryInterest)

function PROCESSDASSTREAM(DataStream)
for each qi ∈ PIT do
  if DataStream satisfies qi
    PROCESSQINPIT(qi, DataStream);
  else
    Forward DataStream;

function PROCESSQINPIT(qi, packet)
if packet == DataStream and packet.ts > qi.ts
  CREATEOPERATORGRAPH(qi) (Refer Algorithm 2);
  Forward packet;
else
  facelist ← PIT.getFaces(qi);
  if qi.face is not found in facelist then
    P.I. AddFace(qi);
  Discard packet;
```
Interest, which is checked only on the receipt of a new Interest packet.

When an Add Query Interest arrives, the broker checks if the (up to-date) data object corresponding to the qi is already processed at one or more brokers and a matching Data packet (with latest timestamp) is found in the CS or cache. The resolution to the Data packet is explained later in Section V-B. If the cache entry is not found, the broker continues its search in the PIT table (lines 617).

If qi is found in PIT and the face corresponding to the query interest does not exist (lines 24-26), a new face_id (from which the interest is received) is added. Conversely, if the face entry is found in PIT, this means the qi is being processed and hence the packet is discarded (lines 25-27). However, if no entry in PIT exists, this means that the consumer’s interest reaches first time at the broker network. Therefore, a new entry for qi is created and the qi is processed by first generating an operator graph (cf. Section V-B) and then processing it (cf. Section V-C) (lines 8-12).

A Data Stream packet is also handled similarly to the Add Query Interest packet (lines 13-18), except for the fact that the query processing is triggered if Data Stream satisfies a query interest in PIT and it is a new packet (lines 20-22). This means that the qi performs an operation on the data object contained in the Data Stream packet. In this case, query processing is triggered because it may contribute to the generation of a new ce. In addition, if the broker does not have a matching qi in PIT, this means it is not allocated to operator graph processing and hence it is forwarded to the next broker (line 18). The Data Stream is forwarded if there are consumers downstream by looking at the FIB entry.

When a Remove Query Interest packet is received at a broker, the node looks up its PIT table for an entry of the qi. If found, it removes the PIT entry for qi and the Remove Query Interest packet is forwarded to the next node. It is done in a similar way as the PIT entry corresponding to a CCN Interest packet is removed when a matching Data packet is found.

To summarize, in Table 1 we show the differences of the INETCEP architecture in comparison to the standard CCN architecture in terms of the packet types, the data plane, and the processing engine. We show that with minimum changes in the data plane, we support both consumer- and producer-initiated traffic.

B. A General CEP Query Language

In this section, we present a general CEP language to resolve Limitation 5. By doing this, we provide a means to resolve CEP queries expressed as qi (query interests) on the data plane of CCN. The grammar definition of the query language can be found in Appendix A. We aim for three main design goals for the query language and the parser: (i) distinguishing between pull and push based traffic, (ii) translating a query to an equivalent name prefix of the CCN architecture, and (iii) supporting conventional relational algebraic operators and being extensible such that additional operators can be integrated with minimum changes. This is to ensure easy integration of existing and new operators. We provide the definition of INETCEP query language in Section V-B1 and the parser in Section V-B2.

1) Query Language: Each operator in a query behaves differently based on the input source type, i.e., consumer- and producer-initiated interaction, which is done based on the reception of a Data packet or a Data Stream packet, respectively. The Data packet is processed and returned as a data object, as conventionally done in the CCN architecture. For instance, a Join operator placed on broker C can join two data objects, <lat1,long1> with name prefix /node/A and <lat2,long2> with /node/B to produce <lat1,long1,lat2,long2> with /node/C.

In contrast, a Data Stream is processed and transformed either into an output stream (another Data Stream packet), or can be transformed to derive a Data packet, containing, e.g., a boolean variable depending on the CEP query. For instance, a join of two continuous data streams expressing location attributes of producer A with location attributes of producer B leads to the generation of a new data stream, as illustrated by broker C in Fig. 7.

We express the standard CEP operators as explained in Section II using the INETCEP language below.

```
WINDOW(GPS_S1, 4s)
Query 1: Selects a sliding window of tuples for 4s from gps source 1.
FILTER(WINDOW(GPS_S1, 4s),'latitude'<50)
Query 2: Selects tuples with latitude value less than 50 pts from window Query 1.
JOIN(
  FILTER(WINDOW(GPS_S1, 4s),'latitude'<50),
  FILTER(WINDOW(GPS_S2, 4s),'latitude'<50),
  GPS_S1,'ts' = GPS_S2,'ts'
)
Query 3: Performs join of the two resulting tuples from Query 2 with gps source 1 and 2.
```

The stateful operators e.g., Window and JOIN must store the accumulated tuples in some form of a readily available storage. For this, we make use of in-network cache, the CS.

3We have implemented all standard CEP operators defined in Section II including SEQUENCE operator but only present the representative operators relevant for our use cases.
The broker that first receives the
query starts by instantiating top down the
binary tree starting from the root operator \( \omega_{\text{cur}} \) (line 3), where \( \text{cur} = \text{root} \) in the first step. The traversal is performed in a
depth-first pre-order manner (visit parent first, then left (L) and then right (R) children) (lines 11-13).

The workflow of the query parser algorithm for an example
query of the form of Query 3 is illustrated in Fig. 8 and
explained in the following. We start by extracting the operator
name (\( \omega \)) by separating the parameters into a list. We create
a logical operator graph for each query by instantiating the
operators and their data flow. An operator is created only
after the semantic checks on the operator are verified, e.g.,
if the \( \omega \) is valid, and/or it has valid parameters. Once all the
semantic checks are verified, we continue processing the query recursively as in Algorithm 2 by generating the corresponding
NFN query and creating logical plan nodes for operators that
are assigned to the broker network for processing the operator
graph.

### C. Operator Graph Processing

This module processes the query either centrally or in a
distributed way by mapping the operator graph plan nodes to the
brokers (B) or INETCEP aware CCN routers after operator
graph construction (cf. Section V-B). This is because central
processing might not be sufficient for all the use cases, e.g.,
when the amount of resources required to process the queries
increases with the number of operators and/or queries. It is,
therefore, necessary to distribute operators on multiple brokers.
In this way, the network is also not unreasonably loaded by
the queries, while the network forwarding is not disturbed.

In Fig. 9, we show the timeline of distributed query processing.
In case of central processing, only parsing and deployment
is required. 1 The broker that first receives the \( q_i \) from the
consumer parses the CEP query and forms an operator
graph (as described in Section V-B). This broker becomes
the placement coordinator and coordinates the further actions
taken for operator graph processing. 2 The coordinator builds
the path where the operator graph is processed based on a
criteria, e.g., minimum latency and selects other broker nodes
for operator placement. This is along the path from producer
towards the consumer. It is important because the Data
packets are forwarded as well as processed along this path (in-network processing). 3 The coordinator recursively traverses
the operator graph, while assigning the CEP operators or

---

**Algorithm 2: Recursively generating the operator graph**

**Variables**
- \( q \) = the input CEP query
- \( \omega_{\text{cur}} \) = current operator

```plaintext
1 function CREATEOPERATORGRAPH(\( \omega_{\text{cur}} \))
  2 \( \text{curList} \) = top down list of 3 \( \omega \) of tuple \( \tau \)
  3 PARSEQUERY(\( \text{curList} \));
  4 function PARSEQUERY(\( \text{curList} \))
  5 \( \omega_{\text{cur}} \) ← GETOPERATOR(\( \text{curList} \));
  6 \( \text{nfnExp} \) ← CONSTRUCTNFPQUERY(\( \omega_{\text{cur}} \));
  7 \( \text{node} \) ← new Node(\( \text{nfnExp} \));
  8 if size(\( \text{curList} \)) == 1 then
    9 return \( \text{node} \);
  10 else if size(\( \text{curList} \)) > 1 then
    11 \( \text{PARSEQUERY(\( \text{curList.left} \))};
    12 \( \text{PARSEQUERY(\( \text{curList.right} \))};
    13 return \( \text{node} \);
```
In principle, this module can be extended to laboriously processing the deployed query and delivering the network discovery service, (ii) placing the plan node with one or more QoS metrics (based on the knowledge from vide operator placement decisions. This module is responsible for operator placement as dictated in the placement literature [33]. By contrast, we provide unified communication layer for co-existing consumer- and producer-initiated interactions, while doing CEP operations in the network. As a consequence, we embedded CEP operators as named functions while leveraging NFN’s abstract machine to resolve them. NFN works together with CCN-lite [7], which is a lightweight implementation of CCNx and NDN protocol. We have developed unified interfaces of our design on top of NFN (v0.1.0) and CCN-lite (v0.3.0) for the Linux platform [8].

We selected the NFN architecture [42] to implement our solution, due to its built-in support of resolving named functions as so-called λ expressions on top of the ICN substrate. However, a major difference to our architecture is that the communication plane in NFN is only consumer-initiated. In contrast, we provide unified communication layer for co-existing consumer- and producer-initiated interactions, while doing CEP operations in the network. As a consequence, we embedded CEP operators as named functions while leveraging NFN’s abstract machine to resolve them. NFN works together with CCN-lite [7], which is a lightweight implementation of CCNx and NDN protocol. We have developed unified interfaces of our design on top of NFN (v0.1.0) and CCN-lite (v0.3.0) for the Linux platform [8].

We evaluate the INETCEP architecture by answering two questions:

**EQ1** Is the INETCEP system extensible and expressive?

**EQ2** How is the performance of INETCEP system?

To this end, we explain the evaluation setup in Section VI-A, EQ1 in Section VI-B and EQ2 in Section VI-C.

### A. Evaluation Environment

We selected the NFN architecture [42] to implement our solution, due to its built-in support of resolving named functions as so-called λ expressions on top of the ICN substrate. However, a major difference to our architecture is that the communication plane in NFN is only consumer-initiated. In contrast, we provide unified communication layer for co-existing consumer- and producer-initiated interactions, while doing CEP operations in the network. As a consequence, we embedded CEP operators as named functions while leveraging NFN’s abstract machine to resolve them. NFN works together with CCN-lite [7], which is a lightweight implementation of CCNx and NDN protocol. We have developed unified interfaces of our design on top of NFN (v0.1.0) and CCN-lite (v0.3.0) for the Linux platform [8].

We have enhanced the NDN protocol implementation in the CCN-lite and the NFN architecture by: (i) including the additional packet types and their handling, as described in Section V-A; (ii) implementing the extensible general CEP query language, parser, and CEP operators as NFN services, as described in Section V-B; and (iii) implementing a network discovery service with modifications in both CCN and NFN, and operator placement as an NFN service (cf. Section V-C).

We evaluated our implementation using the CCN-lite emulator on two topologies: centralized (cf. Fig. 10a) and distributed (cf. Fig. 10b). Each node in our topology is an Ubuntu 16.04 virtual machine (VM) with 8 GiB of memory. Here, each VM (node) is a CCN-NFN relay, which hosts a NFN compute server encapsulating the CEP operator logic. For running the experiments, we first created a CCN network topology as illustrated in Fig. 10. Second, we deployed the INETCEP architecture that works on the NFN compute server, the CCN-NFN relay, as well as on the links. Here, as intended, the nodes communicate using the NDN protocol instead of IP.

In the centralized topology, we have two producers, a single broker that processes the query and one consumer. In the distributed topology, we have one consumer, two producers and six brokers, as shown in the figure. The data structures CS and PIT are utilized as explained in the previous sections (cf. Section V-A). We use Queries 1-3 (cf. Section V-B) for our evaluation with the DEBS grand challenge 2014 smart home dataset and the disaster field dataset. The dataset is explained in Section VI-B.
B. Evaluation Question I: Extensibility

To show the extensibility and expressiveness of our approach, we extended the INETCEP architecture for the two representative IoT use cases that we introduced in Section III with a heat map query and a load prediction query. We extended the INETCEP query language and CEP operators to include the heat map and prediction operators by making a few additions to our implementation in our extensible query language and parser. We used real world datasets to evaluate the queries: the 2014 DEBS grand challenge and a disaster field dataset.

**Dataset 1.** For the heat map query, we use a dataset of a field test mimicking a post-disaster situation. The field test mimics two fictive events, a lightning strike and a hazardous substance release from a chemical plant, which resulted in a stressful situation. The collected dataset consists of sensor data, e.g., location coordinates. It was collected from smartphones provided to the participants. Each sensor data stream has a schema specifying the name of the attributes, e.g., the GPS data stream has the following schema:

\[ <ts, id, value, property, plug id, household id, house id > \]

**Query.** We use the latitude and longitude attributes of this schema to generate the heat map distribution of the survivors from the disaster field test. A typical heat map application joins the GPS data stream from a given set of survivors, derives the area by finding minimum and maximum latitude and longitude values, and visualizes the heat map distribution of the location of the survivors in this area. For simplicity, we consider a data stream from two survivors, as shown in the operator graph in Fig. 11. Here, \( p_1 \) and \( p_2 \) are the producers of smart plugs, \( \omega_\text{M} \) is the join operator, and \( \omega_\text{H} \) is the heat map generation operator (39). This is easily possible using the INETCEP language and parser implementation that follows an Abstract Factory design pattern. First, we included the algorithm for heat map generation, which is ET extended the IN approach, we extended the INETCEP architecture for two

\[
\begin{align*}
\text{Query 4: } & \text{Display the heat map distribution of GPS source 1 and 2 in the given area with a given cell size.} \\
\text{Dataset 2. } & \text{The second dataset comes from the 2014 DEBS grand challenge scenario focused on solving a short-term load forecasting problem in a smart grid. The data for the challenge is based on real-world profiles collected from smart home installations. The dataset captured load measurements from unique smart plugs with the following schema:}
\end{align*}
\]

\[ <ts, id, value, property, plug id, household id, house id > \]

**Query.** We apply an existing solution to perform prediction by extending the INETCEP architecture for two smart plugs. The corresponding operator graph for such a prediction is illustrated in Fig. 11 and listed below. Here, \( p_3 \) and \( p_4 \) are the producers of smart plugs, \( \omega_\text{M} \) is a join operator and \( \omega_\text{pr} \) is a prediction operator based on the algorithm. In the first query, we notify the consumer about the predictions for five minutes into the future, while in the second query we notify only if the predictions of load are above a threshold. Similarly to the heat map application, we implemented a prediction algorithm by adding 50 LOC and the language implementation with 20 LOC.

**Dataset 2.** The second dataset comes from the 2014 DEBS grand challenge scenario focused on solving a short-term load forecasting query for smart homes.

**Dataset 2.** For the load forecasting query, we use a dataset of home installations. The dataset captured load measurements from unique smart plugs with the following schema:

\[ <ts, id, value, property, plug id, household id, house id > \]

**Query.** We apply an existing solution to perform prediction by extending the INETCEP architecture for two smart plugs. The corresponding operator graph for such a prediction is illustrated in Fig. 11 and listed below. Here, \( p_3 \) and \( p_4 \) are the producers of smart plugs, \( \omega_\text{M} \) is a join operator and \( \omega_\text{pr} \) is a prediction operator based on the algorithm. In the first query, we notify the consumer about the predictions for five minutes into the future, while in the second query we notify only if the predictions of load are above a threshold. Similarly to the heat map application, we implemented a prediction algorithm by adding 50 LOC and the language implementation with 20 LOC.

**Query 5:** Performing a prediction every 5 minutes for 5 minutes into the future on the load observed by plug source 1 and 2.

**Filter-Join:**

\[
\begin{align*}
\text{Query 6: } & \text{Filter the load prediction values that are greater than 20 from Listing 5} \\
\text{Dataset 2. } & \text{The second dataset comes from the 2014 DEBS grand challenge scenario focused on solving a short-term load forecasting query for smart homes.}
\end{align*}
\]

**Query 6: Filter the load prediction values that are greater than 20 from Listing 5.**

**C. Evaluation Question II: Performance**

We evaluated the performance of the INETCEP architecture on standard CEP queries including the Queries 1-3. Additionally, we evaluated the heat map and prediction queries (Queries 4-6). Our aim is to understand the performance of: (i) the query parser on increasing the number of nested operators in operator graph, (ii) the operator graph module for different kind of queries, and (iii) the operator placement module on increasing number of queries.
Communication delay.

Fig. 12: Performance of the query parser depending on increasing the number of nested operators.

Fig. 13: End-to-end and communication delay observed in Queries 1 to 6.

1) Query Parsing: In the query parser design (cf. Section V-B2), we performed a complexity analysis of the Algorithm 2. In this section, we verify the analysis experimentally by increasing the number of operators in the operator graph while using the centralized topology shown in Fig. 10.

In Fig. 12, we show the performance of query parser algorithm in terms of runtime (in ms) for Query 2 or Filter operators and Query 3 or Join operators. We show in a line plot with a confidence interval of 95% for 20 runs of the emulation that the two queries scales reasonably and can be processed in a few milliseconds.

2) Centralized Query Processing: We measured the end-to-end delay in processing the six queries defined above using the operator graph module (cf. § V-C). In Fig. 13, we show the results as box plots with a confidence interval of 95% for 10 runs of the emulation. For Query 1 the total delay perceived is less than a few milliseconds. It increased for the new queries Queries 4-6, where we introduced prediction and heat map operators, primarily due to increased consumption of data and the computational complexity of the algorithms for prediction [34] and heat map [29], respectively. We further show the distribution of the mean end-to-end delay in Table II to understand the primary reason for the delay. We listed the time spent in each of the modules of the INETCEP architecture, namely, operator graph creation, placement of operators, and communication of events. The values shown in the table are the mean of the values observed for 10 executions. For basic CEP queries [12], the major portion of time (98%) is spent in communication. This can be explained, as also confirmed in other works [20], by the limitations of the CCNx implementation of the NDN protocol. Hence, we observe the communication delays for all the queries in Fig. 13b, where it takes up to 500 ms for delivering results of complex queries like prediction and heatmap.

3) Operator Placement: To understand the behavior of the operator placement module, we utilize the distributed topology of seven VMs, as illustrated in Fig. 10b, to place operators based on the information collected by the network discovery service. To take full advantage of distributed CEP, we increased the query load of Query 3 starting from node 1 to 5. For example, the first 10 queries were initialized at the first broker node followed by the next 10 queries at the second node, and so on. Hence, the number of consumers also increased in the network with the query load. In Fig. 14, we see that the total delay in retrieving the complex event increased incrementally with the query load, which is reasonable given the static network size (7 nodes) and the resources. However, the queries were distributed evenly using the operator placement algorithm at distinct nodes.

To summarize, we evaluated the performance of INETCEP on two topologies: centralized and distributed, using two IoT applications and six different queries. Our evaluation shows that IoT applications can be integrated seamlessly using the INETCEP architecture. CEP queries can be formulated and can be extended for more use cases, and simple CEP queries can be processed in milliseconds.

VII. RELATED WORK

We now review the state-of-the-art ICN architectures in terms of their support for consumer- and producer-initiated interaction patterns and existing in-network processing (INP) architectures in Section VII-A and CEP and networking architectures in Section VII-B.

A. ICN Architectures

Interaction Patterns in ICNs: In Figure 1, we highlight the main ICN architectures NDN [43], NFN [42], DONA [23], PURSUIT [2], PSIRP [1], in terms of their support of a unified communication layer as presented in our work. However, only those appearing in the green box, namely CONVERGENCE [35], GreenICN [3], and Carzaniga et al. [19].
provide support for both kind of invocation mechanisms. The CONVERGENCE system combines the publish/subscribe interaction paradigm on top of an information-centric network layer. In contrast, we provide a unified interface such that pull and push based interaction patterns could co-exist in a single network layer while performing in-network computations. GreenICN is an ICN architecture for post-disaster scenarios by combining NDN (pull-based) with COPSS [20] (push-based). However, GreenICN introduces additional data structures, e.g., a subscription table (ST), while we provide this combination using the existing data structures of ICN. Furthermore, it is not clear if GreenICN can function as a whole in a single ICN architecture [41]. Carzaniga et al. propose a unified network interface similar to our work, however, the authors only propose a preliminary design of their approach [19] without implementing it in an ICN architecture, and subsequently focus on routing decisions [18] rather than on distributed processing.

**B. CEP and Networking Architectures**

**ICP and IoT Architectures in ICNs:** Authors in work [43, 48] propose an approach to distribute computational tasks in the network by extending the NFN architecture similar to our work. However, the authors do not deal with the lack of abstractions required for processing continuous data stream. In contrast, we propose a unified communication layer to support CEP over ICN and an extensible query grammar and parser that opens a wide range of operators. Krol et al. [31] propose NFaaS based on unikernels, which is a container based virtualization approach to encapsulate named functions placed on NDN nodes. However, they do not provide support for stateful functions, while IoT functions can be stateful, e.g., involving time windows, which is supported by our architecture. Ahmed et al. [10] propose a smart home approach using NDN and support both push and pull interaction patterns similar to our work. However, in their architecture they only support retrieving raw data, e.g., humidity sensor readings, but not meaningful events as we do. Shang et al. [39] propose a publish/subscribe based approach for modern building management systems (BMS) in NDN. However, the authors build on standard consumer-initiated interaction, as described in Limitation 1 (cf. Section IV-A). Publish-subscribe deployment for NDN in the IoT scenarios has been discussed in previous works [24, 24]. These works confirm the need of integrating producer-initiated interaction in NDN; however, do not provide a unified layer for both interaction patterns as we do.

**CEP and Networking Architectures**

**CEP Architectures:** Several event processing architectures exist, ranging from, e.g., the open source Apache Flink [16] to Twitter’s Heron [32] and Google’s Millwheel [11]. One possibility is to interface one of them with an ICN architecture. Initial work implemented Hadoop on NDN [22] for datacenter applications. However, this requires changing the network model to push in contrast to our work, which would limit the support for a wide range of applications, as discussed above.

**Networking Architectures:** Another emerging network architecture is Software-Defined Networking (SDN) [30], which is gradually being deployed, e.g., in Google’s data centers. It allows network managers to program the control plane to support efficient traffic monitoring and engineering. The SDN architecture is complementary to our work, since SDN empowers the control plane, while ICN upgrades the data plane of the current Internet architecture.

**Data Plane and Query Languages:** The literature discusses many CEP query languages [18]. The novelty of the proposed query language is to allow for a mapping of operations to ICN’s data plane. Alternative designs build on P4 [14] in the context of SDN. Initial work on programming ICN with P4 [40] faced several difficulties due to lack of key language features and the strong coupling of the language to SDN’s data plane model.

**VIII. Discussion**

In this section, we discuss important future challenges that could be interesting to provide more sophisticated networking, reliability and optimization mechanisms in the INETCEP architecture.

**Sophisticated Flow and Congestion Control:** In CCN, the PIT table ensures the flow balance since one Data packet is sent for each Interest packet. The Data Stream packets of INETCEP could disturb the rule of flow balance since the producer could overflow the buffer on the broker side. For this, INETCEP implements a simple flow control mechanism where we restrict the receiver (consumer/broker) to specify maximum outstanding messages at a time. However, since the forwarding logic of Data Stream packets is similar to IP multicast, existing sophisticated multicast congestion control solutions like TCP-Friendly Multicast Congestion Control [43] and similar can provide sophisticated flow and congestion control.

**Reliability:** The brokers or consumers could miss packets when the available bandwidth and resources at their end is lower than the sending rate. This makes the presence of a module of reliability relevant, which can be catered by extending our work with existing reliable CEP solutions [27] or by looking into equivalent IP solutions such as Scalable Reliable Multicast [21].

**Query Optimization:** In INETCEP we provide a placement module that maps the operator graph to in-network elements of CCN. Another complementary direction could be to generate an optimal operator graph, e.g., based on operator selectivity or even partition operator graph by performing query optimization [15].

**Optimizing QoS:** In INETCEP we provide a programming abstraction for the developers to write CEP queries over ICN data plane substrate. In addition, the placement module can be extended to look into further decentralized solutions and even other QoS metrics [17] like throughput, availability, etc.

**IX. Conclusion**

In this paper, we proposed the INETCEP architecture that implements a unified communication layer for co-existing consumer-initiated and producer-initiated interaction patterns.
We studied important design challenges to come up with our design of a unified communication layer. In the unified layer, both consumer- and producer-initiated interaction patterns can co-exist in a single ICN architecture. In this way, a wide range of IoT applications are supported. With the proposed query language, we can express interest in aggregated data that is resolved and processed in a distributed manner in the network. In our evaluation, we demonstrated in the context of two case studies that our approach is highly extensible. The performance evaluation showed that queries are efficiently parsed and deployed, which yields - thanks to the in-network deployment - a low end-to-end delay, e.g., simple queries induce only few milliseconds of overall delay.

Interesting research directions for future work are: (i) enhancing the performance of query processing by using parallelization, (ii) porting CCNx implementation on real hardware to accomplish low communication delays, and (iii) developing CEP compliant caching strategies.

ACKNOWLEDGEMENTS

This work has been co-funded by the German Research Foundation (DFG) as part of the project C2, A3 and C5 within the Collaborative Research Center (CRC) SFB 1053 – MAKI.

REFERENCES

[1] FP7 PSIRP project. http://psirp.org/, 2008. Accessed on 24.02.2019.
[2] FP7 PURSUIT project. http://www.fp7-pursuit.eu/PursuitWeb/, 2009. Accessed on 13.02.2019.
[3] FP7 Green ICN project. http://www.greenicn.org/, 2013. Accessed on 24.02.2019.
[4] 2014 debs grand challenge. https://debs.org/debs-grand-challenges/2014/, 2014. Accessed on 30.08.2019.
[5] Amazon Alexa. https://developer.amazon.com/docs/alexa-voice-service/api-overview.html, 2016. Accessed on 30.8.2019.
[6] A Better Way to Organize the Internet: Content-Centric Networking. https://spectrum.ieee.org/telecom/internet/a-better-way-to-organize-the-internet-contentcentric-networking, 2017. Accessed on 30.8.2019.
[7] Cnc-lite github. https://github.com/cn-tofbase/ccn-lite, 2019. Accessed on 30.08.2019.
[8] INetCEP github. https://github.com/luthramanisha/INetCEP, 2019. Accessed on 30.08.2019.
[9] Idc worldwide internet of things installed base by connectivity forecast, 2017-2021. https://www.idc.com/getdoc.jsp?containerId=US41619617, March 2017. Accessed on 5.03.2019.
[10] S. H. Ahmed and D. Kim. Named data networking-based smart home. ICT Express, 2(3):130 – 134, 2016. Special Issue on ICT Convergence in the Internet of Things (IoT).
[11] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, M. Król and I. Psaras. Nfaas: Named function as a service. In Proceedings of the 43rd IEEE Conference on Local Computer Networks (LCN), 2018.
[12] M. Gibbens, C. Gniady, L. Ye, and B. Zhang. Hadoop on named data networking: Experience and results. Proceedings of the ACM Measurement and Analysis of Computing Systems, 1(2):1:2–1:21, June 2017.
[13] C. Gundogan, P. Kietzmann, T. C. Schmidt, and M. Waehlisch. HoPP: Robust and Resilient Publish-Subscribe for an Information-Centric Internet of Things. In Proceedings of the 43rd IEEE Conference on Local Computer Networks (LCN), 2018.
[14] C. Gundogan, T. Schmidt, H. Hamburg, and M. Waehlisch. Publish-Subscribe Deployment Option for NDN in the Constrained Internet of Things, July 2017. ICN Research Group, Internet-Draft, draft-gundogan-icnpub-pub-iot-01.
[15] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[16] V. Jacobson, D. K. Smetters, N. H. Briggs, and R. L. Braynard. Networking named content. In Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, CoNEXT ’09, pages 1–12, New York, NY, USA, 2009. ACM.
[17] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Volz. Rollback-recovery without checkpoints in distributed event processing systems. Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS ’13, page 27, 2013.
[18] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. SIGCOMM Computer Communication Review, 37(4):181–192, Aug. 2007.
[19] C. Köpp, H.-J. von Mettenheim, and M. H. Breitner. Decision analytics with heatmap visualization for multi-step ensemble data. Business & Information Systems Engineering, 6(3):131–140, June 2014.
[20] D. Kreutz, P. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. Proceedings of the IEEE, 103(1):14–76, Jan. 2015.
[21] M. Król and I. Psaras. Nfaas: Named function as a service. In Proceedings of the 4th ACM Conference on Information-Centric Networking, pages 134–144, New York, NY, USA, 2017. ACM.
[22] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron. Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD ’15, pages 239–250, 2015.
[23] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. IEEE Internet Computing, 12(6):50–60, 2008.
[24] A. Martin, R. Marinho, A. Brio, and C. Fetzer. Predicting energy consumption with streammine3g. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, pages 270–275, ACM, 2014.
[25] N. B. Melazzi. CONVERGENCE: Extending the Media Concept to Include Representations of Real World Objects. The Internet of Things, pages 129–140, 2010.
[26] K. Petikoukis, B. Ohlman, D. Conruo, G. Boggia, G. Tyson, E. Davies, A. Molinaro, and S. Eum. Computer-based networking: Baseline
scenarios, March 2014. Internet Research Task Force (IRTF), Internet Draft, draft-irtf-icnrg-scenarios-02.

[37] A. Rahman, D. Trossen, D. Kutscher, and R. Ravindran. Deployment Considerations for Information-Centric Networking (ICN), 2019. Internet Research Task Force (IRTF), Internet Draft, draft-irtf-icnrg-deployment-guidelines-04.

[38] C. Scherb, C. Marxer, U. Schnurrenberger, and C. Tschudin. In-network live stream processing with named functions. In 2017 IFIP Networking Conference (IFIP Networking) and Workshops, pages 1–6, June 2017.

[39] W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, J. Thompson, J. Burke, B. Zhang, and L. Zhang. Named data networking of things. In Proceedings of the 1st IEEE International Conference on Internet-of-Things Design and Implementation, pages 117–128. IEEE, 2016.

[40] S. Signorello, R. State, J. François, and O. Festor. Nda.p4: Programming information-centric data-planes. In IEEE NetSoft Conference and Workshops (NetSoft), pages 384–389, June 2016.

[41] A. Tagami, T. Yagyu, K. Sugiyama, M. Arunaihurai, K. Nakamura, T. Hasegawa, T. Asami, and K. K. Ramakrishnan. Name-based push/pull message dissemination for disaster message board. In 2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), pages 1–6, June 2016.

[42] C. Tschudin and M. Sifalakis. Named Functions and Cached Computations. In IEEE Consumer Communications and Networking Conference (CCNC), pages 851–854, 2014.

[43] Q. Wang, B. Lee, N. Murray, and Y. Qiao. CS-Man: Computation service management for IoT in-network processing. In 2016 27th Irish Signals and Systems Conference. ISSC 2016, pages 0–5, 2016.

[44] J. Widmer and M. Handley. Tcp-friendly multicast congestion control (tfmcc): Protocol specification., 2006. Internet Engineering Task Force (IETF) RFC 4654.

[45] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, T. Hasegawa, T. Asami, and K. K. Ramakrishnan. Name-based push/pull message dissemination for disaster message board. In 2016 IEEE Internet Research Task Force (IRTF), Internet Draft, draft-irtf-icnrg-scenarios-02.

APPENDIX

In this section, we define our query grammar in Appendix A the implementation details of our extensible design in Appendix B and the algorithms used for extending the system with load forecasting and a heat map application in Appendix C

A. Query Grammar

Definition A.1. A grammar consists of four components:

(i) A set of terminals or tokens. Terminals are the symbols that occur in a language.

(ii) A set of non-terminals or syntactic variables. Each of them represent a set of strings. We define them the way we want to use them.

(iii) A set of production rules that define which non-terminals can be replaced by which terminals, or non-terminals or a combination of both. Here, the terminal is the head of the left side of the production, and the replacement is the body or right side of the production. For example, head → body

(iv) One of the non-terminals is designated as start symbol for each production.

Following the above definition and Chomsky-Hierarchy, we selected a type 2 grammar or a context-free grammar for CEP over ICN language since a query may consist of multiple subqueries (or operators), which can be expressed (out of many possible ways) using parenthesis “()”.

| ω | ::= | σ | u | n | r | n | g | agg | seq |
|---|---|---|---|---|---|---|---|---|---|
| μ | ::= | JOIN( format , ω , ω , boolExp ) |
| σ | ::= | FILTER( format , ω ) |
| win | ::= | WINDOW( latinNumber , number ) |
| seq | ::= | SEQUENCE( format , ω → ω ) |
| agg | ::= | AGGFUN( format , latinNumber , win ) |
| number | ::= | REG(0-9)+ |
| latinNumber | ::= | REG([a-zA-Z0-9]+) |
| boolExp | ::= | latinNumber comparison latinNumber — boolExp concat boolExp |
| comparison | ::= | < | > | = |
| concat | ::= & — “—” |
| time | ::= | nn : nn : nn : nn |
| n | ::= | REG(0-9){1} |
| format | ::= | Data Stream — Data |

TABLE III: Initial context-free grammar for INETCEP language.

The context-free grammar allows us to combine the production rules. The head of each production consist only of one non-terminal and the bodies are not limited by only one terminal and/or one non-terminal. This is because the language needs to embed operators in parenthesis and with a regular grammar we cannot have an arbitrary number of parentheses. We need a way to memorize each parenthesis and this ability is given by context-free grammars.

We define an initial INETCEP language grammar and represent it using BNF (Backus-Naur form) in Table III considering the aforementioned design decisions. We use regular expressions represented as REG(...), where (...) can be literals \([a−z]\) and \([A−Z]\) in lower and upper case, respectively, and numbers \([0–9]\). The plus (+) sign in REG(\([a−z]\)+) means that at least one lowercase letter has to appear, while a 1 in REG(\([a−z]\){1}) means that exactly one lowercase letter has to appear. The relational operators given by comparison define a binary relation between two entities, e.g., two column names of a schema, a column name to a number or a column index to a number.

B. Extensibility

To make our query language extensible, we follow a well-known Abstract Factory design pattern from object-oriented programming for our operator definition as illustrated in Fig. 15 Algorithm 2 is the starting point of our operator graph creation. This is implemented in the OperatorTree class. Each operator inherits the abstract class OperatorA which defines the interpret (parseQuery in Algorithm) function as seen in the figure. The checkParameters verifies the correctness of the parameters.

If a new operator is to be included, it will override the existing methods of the abstract class and the parameters correctness has to be defined. This allows minimal changes in the implementation for each new application developed using our system.

C. Applications

1) Short term Load Forecasting: For the DEBS Grand Challenge in 2014, Martin et al. derived requirements for predicting future energy consumption of a plug. We
formulate our requirements with respect to our INETCEP system:

(i) To meet the goal of making an estimation on future energy consumption, it is necessary to use historical data as a reference.

(ii) In order to run on a machine with limited resources, the prediction algorithm needs to be lightweight in computation power and storage.

The formula for predicting future load is given by the publishers of the DEBS Grand Challenge and is as follows: 

$$\text{predicted}_\text{load}(s_{i+2}) = (\text{avgLoad}(s_i) + \text{median}([\text{avgLoad}(s_j)]))$$

Here, \(i\) is the current timestamp, \(s_i\) the currently recorded values at time \(i\) and \(s_j\) the past values at a corresponding time \(j = i+2\). The load two steps in the future is therefore made up of the current average electricity consumption and the average electricity consumption from the past.

In Fig. 16 we represent the flow of the prediction algorithm as per the requirements defined above and Query 5. For each time window of 1 minute, we first determine if it is the time for next prediction, which is provided as an input in the query. If the time has not come yet, a value for prediction (average load) is calculated and stored so that it can be used for the equation defined above. Inversely, if it is the time to make a prediction, a prediction tuple of the following form is emitted.

\[<ts, \text{plug}_\text{id}; \text{house}_\text{id}; \text{house}_\text{id}; \text{predicted}_\text{load}>\]

Here, \(ts\) is the timestamp of the prediction, \(\text{plug}_\text{id}\) identifies a socket in a household, \(\text{house}_\text{id}\) identifies a household within a house and \(\text{house}_\text{id}\) identifies a house and

predicted_load is the prediction as specified in the equation above.

2) Heat Map: Algorithm 3 describes the heat map creation and visualization for the location updates from survivors of the disaster field test used in this work based on [29].

Algorithm 3: Algorithm for the heat map operator

Data: \(\text{loc}\): Window of location \(<lat, long>\) tuple of the survivor

Data: \(\text{Lat}_\text{min}\): The minimum latitude value

Data: \(\text{Lat}_\text{max}\): The maximum latitude value

Data: \(\text{Long}_\text{min}\): The minimum longitude value

Data: \(\text{Long}_\text{max}\): The maximum longitude value

Data: \(HC\): The number of horizontal cells needed to map the values

Data: \(VC\): The number of vertical cells needed to map the values

Data: \(\text{cell}_\text{size}\): The granularity

Data: \(\text{Grid}\): A two dimensional array

1. \(HC = \lfloor \frac{\text{Long}_\text{max} - \text{Long}_\text{min}}{\text{cell}_\text{size}} \rfloor;\)
2. \(VC = \lfloor \frac{\text{Lat}_\text{max} - \text{Lat}_\text{min}}{\text{cell}_\text{size}} \rfloor;\)
3. for each line in \(SD\) do
4. \(\text{absLatVal} = \text{loc}[\text{lat}] - \text{Lat}_\text{min};\)
5. \(\text{absLongVal} = \text{loc}[\text{long}] - \text{Long}_\text{min};\)
6. \(\text{Grid}[\lfloor \text{absLatVal} / \text{cell}_\text{size} \rfloor][\lfloor \text{absLongVal} / \text{cell}_\text{size} \rfloor] += 1;\)
7. return \(\text{Grid}\)

In line 1, we calculate the number of horizontal cells required for the desired heat map. For this we divide the difference between the maximum and minimum longitude by the desired cell_size, which indicates how large and finely meshed the resulting heat map should be and then round this value down to the next smaller number (given by floor function). In line 2, similarly we compute the vertical cells. For each of these location tuples in the current window, the first absolute latitude and longitude values are computed in line 4 and 5, respectively. By dividing these values by the cell_size, we obtain the corresponding position in the heat map in line 6.