Support for Adaptive and Distributed Deployment of CEP Continuous Queries for the IoMT

Fernando B. Veras Magalhães[1], **Francisco J. da Silva e Silva**[2], **Markus Endler**[1]

[1]Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
Caixa Postal 38097 – Rio de Janeiro – RJ – Brasil

[2]Departamento de Informática – Universidade Federal do Maranhão (UFMA)
São Luís – MA – Brasil

{fmagalhaes,endler}@inf.puc-rio.br, fssilva@lsdi.ufma.br

Abstract. *The current dissemination of IoT increases the deployment of stream processing solutions for monitoring and controlling elements of the real-world. One of those solutions is Complex Event Processing (CEP), and to handle the high volume, velocity and volatility of data streams from IoT sensors the CEP pipeline should be distributed, preferably having CEP operators both in the cloud/cluster and in edge devices. In this paper, we present a model for a distributed CEP platform and an implementation of this model called Global CEP Manager (GCM). GCM is a service of the ContextNet middleware that supports the deployment and dynamic rearrangement of CEP queries to CEP engines executing in the cloud and in M-Hubs, that are ContextNet's mobile edge devices.*

1. Introduction

The increasing demand to get relevant information about real-world processes, entities and interactions faster, along with the ever-growing generation of data, drives the demand for computer systems capable of processing of high volume data flows as fast as possible. These systems constitute what [Cugola and Margara 2012] classify as Information Flows Processing (IFP), or else, stream processing systems.

The authors of [Carney et al. 2002] highlight two essential characteristics of stream processing: the analysis of data immediately on its receipt, and the ability to take into account the temporal attribute of the data/events. The first characteristic is describable as putting a focus on immediate, low latency responses. In contrast, the second means that instead of processing the whole, historic set of received events, the stream processing model will focus on analyzing and detecting specific patterns of events in time-bounded sliding/batch windows, ignoring to observe variations among the current, windowed data and the old data. As a manner to declare how the data streams should be processed, stream processing technologies commonly offer continuous queries. Continuous or standing queries are business logic rules that define how data should be processed and outputted.

Many stream processing systems concentrate all data stream processing at a centralized site, such as a central server, a cluster or a cloud. Such configuration has the advantage of simpler maintenance, but suffers from the problem of overloading the network access to the central site with the sending of a large volume of very basic data/events. And

since almost all stream processing systems essentially analyze data that "flows from the periphery to the center" [Cugola and Margara 2012] it is only natural to think of stream processing systems which divide the data processing between the nodes along the path of the data flow [Balazinska et al. 2005].

This distributed approach has several benefits. For instance, it can reduce processing bottlenecks since it is not concentrated at the cluster/cloud. It also supports local processing queries that do not depend on external information, such as queries that operate on data only from sensors directly connected to an edge node, and which may also drive to some local action, such as raising an alarm. Moreover, the distributed approach will probably be able to reduce the network bandwidth usage as it is common to cause some filtering of the raw data stream at each processing step.

Many flavors and approaches for stream processing have been proposed. For instance: Active Databases, Data Stream Management System (DSMS), and ultimately Stream Reasoning and Complex Event Processing (CEP). CEP [Luckham 2002] has some properties that favor it among the other approaches. One of them is the use of events to encapsulate data. Events are meaningful pieces of data as they represent a fact or a status update (e.g., a departure, an arrival, or a location update). Also, each event belongs to an event type, which is characterized by a name (e.g., DepartureEvent) and a set of attributes (e.g., timestamp, train, station).

Furthermore, CEP has a distinct power to detect the occurrence of patterns across event streams. It means that CEP allows the declaration of (continuous) queries that identify a complex pattern of event occurrences, such as three or more events of type TemperatureOutOfBounds, followed by one or more SmokeDetected event, all within a time window of one minute. Another advantage of CEP is the possibility to produce complex events upon the detection of of those complex patterns, e.g., a FireWarning event type holding the average temperature readings of the original temperature events. Those complex events will then constitute new event streams, meaning that other complex queries can further process them.

The process of designing and deploying distributed CEP solutions presents some challenges. For instance, it is frequent to make modifications to a CEP query both before and after its deployment. Commonly, the developer may need to adjust the event window or the event pattern to ensure that the query will trigger on the desired situations. Therefore, a distributed CEP system should support the upload of new versions of a query and grant that the new version will be propagated to the necessary processing nodes.

Also, to ensure the event type-oriented coupling of the queries dispersed in the system, it is essential to ensure that event type definitions are already known in the processing nodes that will consume those events. For instance, if query A produces events that will be consumed by a subsequent query B (that may be in by a different node), the characteristics of the events produced by A should be previously known by the node that implements B.

Moreover, an essential step in distributed stream processing is the actual partition of the processing load. In distributed CEP, that means allocating or relocating queries to the nodes. The system may require the developer to choose himself/herself explicitly a node to run each query. However, the system may also provide the option to assign

or reassign queries to the nodes automatically. This automatic reassignment can be especially useful on the Internet Of Mobile Things (IoMT), where smart objects and edge gateways can be mobile. That is for two main reasons: First, because mobile nodes are more susceptible to disconnections, therefore the reassignment of queries from a disconnected node to an available node is frequent; Second, because it enables the assignment of queries depending on the node's location.

The main contributions of this paper are: (i) a model for a distributed CEP platform and (ii) an implementation of this model called Global CEP Manager (GCM), and (iii) some initial performance tests of GCM. Our model is composed of a core component that handles query distribution and a processing component that does the actual processing of data. While the core component runs on a central node, the processing component can be instantiated on cloud/cluster nodes, as well as on edge nodes, stationary or mobile smartphones. The general goal of GCM is to facilitate the creation of such distributed applications while also checking if the queries will function properly on a processing pipeline before their distribution.

The remainder of the paper is goes as follows: In Section 2 we present the basis used to produce this work, that is, CEP and event processing networks ,and the ContextNet middleware. We discuss some related work in Section 3. Then, in Section 4 we present our model for a distributed CEP platform, followed by 5 were we present the Global CEP Manager, an implementation of this model. As evaluation, in Section 6 we demonstrate an use case and the results of our initial performance tests. Finally, Section 7 summarizes this paper and presents future work.

2. Fundamental Basis

2.1. CEP and Event Processing Networks

Complex Event Processing (CEP) is a software technology for the dynamic analysis of large amounts of data or event flows in near real-time. An event can represent the change of state of an analyzed entity or the regular measurement of one or more properties of that entity. CEP analyzes events based on CEP rules or queries, which are similar to queries to a database. However, while in database systems queries are made to information already stored, CEP uses continuous queries, that is, every new event received is tested against queries that had already been instantiated before the arrival of the event. Each continuous query uses one or more stream operators, such as filter, aggregation, and pattern recognition. The output of a continuous query may constitute a complex event since it represents higher-level information [Luckham 2011]. Those complex events can then be consumed by subsequent queries constituting a multi-level processing pipeline. This pipeline can be represented as a directed graph called Event Processing Network (EPN) [Etzion et al. 2011]. In this graph, each vertex is called an Event Processing Agent (EPA), which is the processing stage of a query. Also, each directed edge (x, y) on the EPN represents that the EPA y consumes the output of the EPA x. Finally, representing a CEP processing pipeline as an EPN can facilitate the construction of a distributed CEP system since EPAs can be deployed on different devices.

2.2. ContextNet and M-Hub

The ContextNet [Endler and e Silva 2018] is a scalable middleware focused on supporting IoT and IoMT environments. It offers two main protocols of communication SDDL and

MRUDP. SDDL is a protocol based on the Data Distribution System (DDS) pattern for communication between cloud-based services. MRUPD is a protocol for establishing a connection between gateways, that also run SDDL, and client devices. The M-Hub is an Android application that acts as a mobile edge gateway for the ContextNet. Two main services of the M-Hub were relevant to our work. The first one was the S2PA; it has the ability to discover and connect to smart things with Bluetooth WPAN automatically. S2PA enables the M-Hub to collect sensor data automatically that will feed the input of our distributed CEP solution. The second one is the MEPA service, which is a CEP service that uses Asper[1], an Android port of the Esper[2] engine. We adapted the MEPA service to ensure it implements the processing node we displayed on our model and is compatible with GCM.

3. Related Work

Distributed CEP is already well established among information processing technologies, for instance, [Pietzuch et al. 2003] dates back to 2003 and presents a framework for Distributed CEP implementation. However, this section will focus on works that are more representative of the current development state on distributed CEP systems. Another relevant work on the subject is Wihidum [Jayasekara et al. 2015], it distributes the processing load using pipelines and distributed operators. Pipelines are ordered series of simple operations (e.g.: filtering and mapping) where each operation can be executed on one node and them send the output to the node that is responsible for the next operation on the pipeline, it requires the specification of each simple operation and their other of execution. Distributed operations supported are join and pattern matching. For instance, a distributed join of RoomTemperatureStream and RoomHumidityStream on the roomId attribute would partition the events of those streams among the processing nodes in a way that events with the same roomId would be directed to the same node.

A relatively new approach on distributed CEP is the use of mobile devices as processing nodes, this approach can be especially effective in the ContextNet environment since it uses the M-Hub to establish a link with sensors and actuators. Thus the processing would be closer to the physical objects that receive information of and interact with the external world. Also as indicated by Talavera [Rios et al. 2016], local processing can reduce the smartphone's data and battery usage. In this subject we found [Starks and Plagemann 2015] a work that focuses on distributed CEP in Mobile Adhoc networks (MANETs) it provides a mechanism for an automatic statement of CEP rules on the mobile processing nodes closest to the source. However, that work consists of a prototype developed on a network simulator, it does not provide a solution usable in the real world.

Our work proposes a solution that combines stationary edge nodes, mobile edge nodes and cloud or clustered processing nodes. This approach can work very well in the ContexNet environment since the M-Hub run on Android devices and has the capabilities of acquiring data from sensors, local CEP processing and sending resulting events to ContexNet gateways.

[1]Asper, Android port of Esper. https://github.com/mobile-event-processing/Asper
[2]Esper CEP is an open-source CEP engine. http://www.espertech.com/esper/

4. A Model for a Distributed CEP Platform

The first contribution of this paper is a conceptual model for a platform to build distributed CEP applications. It supports the deployment of continuous queries on heterogeneous devices in an Internet of Mobile Things (IoMT) environment. The idea is to create a connected Event Processing Network (EPN) that distributes the data processing among nodes in different layers (e.g., Edge and Cloud) and with different processing capabilities. Our model focuses on the following aspects:

- **A Simple System Management Interface**: Our model comprises an interface for stating and allocating queries. It also allows checking aspects of the system status, for instance, the active nodes and queries.
- **Support for heterogeneous devices dispersed in the cloud and edge of the IoMT**: Our solution supports the deployment of queries to mobile or stationary edge nodes that are generally closer to the data sources, but lack processing power. It also supports the deployment of queries to cloud nodes, which can generally perform complex operations that consume the output of edge nodes.
- **Facilitate query connection**: Each query will produce events as its output. Those events should then be distributed to the nodes that may process them. However, before receiving the events, the nodes should already be able to recognize their event type (e.g., the event name and its set of attributes). Our solution ensures that the nodes that consume the output of a query will receive both the type of the outputted events and the events themselves.
- **Validation of rules before they are deployed**: Our model considers a validation step previous to the actual query deployment. The validation checks the syntax of queries and if they consume valid event types. We consider valid event types that are produced by a previous query or originated in a data source (e.g., a sensor event).
- **Easy modification of executing queries**: During the development of an EPN, it is common to make adjustments to an already running query. In our solution it is possible to issue a new version of a query. We ensure the distribution of the new version to every connected processing component that was running the old version.
- **Adaptability**: Our solution aims to offer adaptable EPNs as it provides automatic query reallocation. First, it considers query relocation in case of node disconnection. It also considers associating the query with a context (e.g., a location). The system should then allocate the query to a node that meets the context and relocate if necessary (e.g., if the device moves out of the location).

The Figure 1 displays a general view of the model established in this work. We propose the use of middleware with reliable delivery of messages as a communication bus, and our model is composed of two elements, a **processing component** and a **core component**. The processing component does the processing of data and can have multiple implementations for different, heterogeneous devices. In the figure, both Cloud Processing Nodes and Edge Processing Nodes run implementations of the processing component. They receive queries and may produce events to be consumed on other processing components. The core component handles the continuous query distribution and ensures the correct coupling of the EPN parts. In the figure, the Core Node implements the core

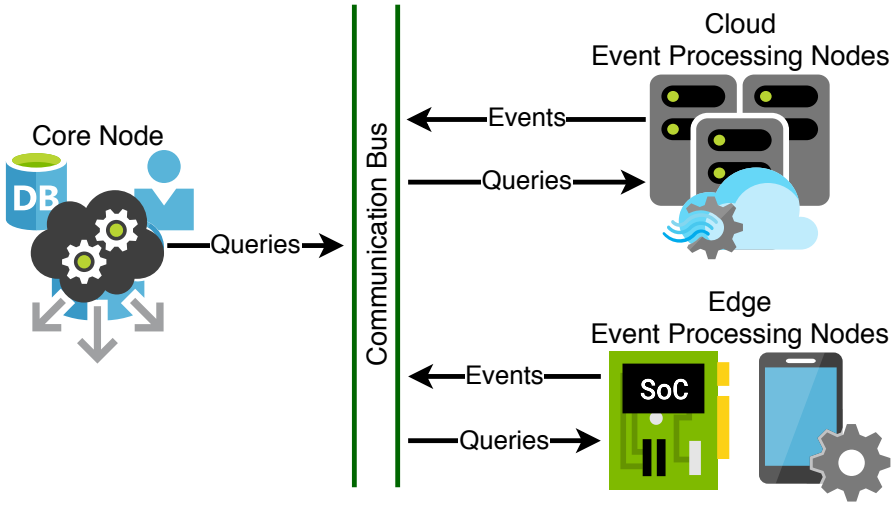

Figure 1. The general architecture of the proposed model.

component. A **system manager** interacts with the core component to issue queries and allocate them to processing components. We will further detail each of those components on the next two subsections.

4.1. The core component

The core component is responsible for offering a system management interface while also handling the distribution of queries and events. The Figure 2 contains the representation of the core component and its modules.

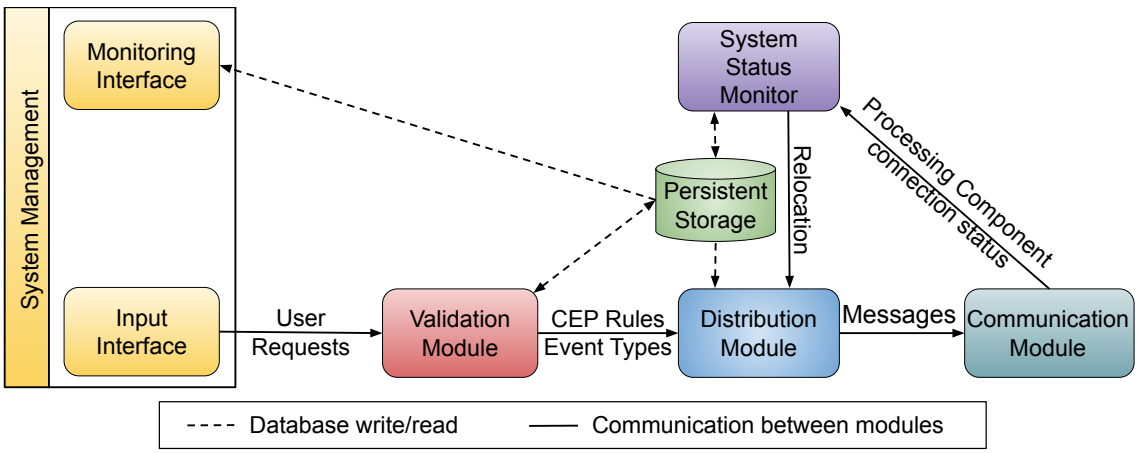

Figure 2. The Core Component.

Our solution offers two interfaces for system management: a **monitoring interface** and an **input interface**. The monitoring interface allows consulting the status of queries and processing components. For instance, it is possible to check which processing components are running a query and which queries are running in a processing component. The input interface allows creating and updating queries and allocating or deallocating it from processing components.

The **validation module** receives user requests (queries and allocations) from the input interface and verifies them before they are shipped to the processing components. It

will check the syntax of queries and ensure they consult existing event types, that is, event types that are produced by previous queries or by data sources. Also, prior to the query allocation, the validation module will check if the designated processing component is valid and known. After validation, the persistent storage is updated and the **distribution module** handles the query shipping. It encapsulates the queries in messages and requests the communication module to send them to each designated processing component.

The **connection module** handles all communication with the processing components. It also reports connections, disconnections and context updates of processing components to the **system status monitor**. When a processing component connects/reconnects, the system status monitor saves/updates its information on the persistent storage and makes sure that it has the latest version of every query allocated to it. When a processing component disconnects, the system status monitor relocates its queries to another component if possible. If a processing component was running a query associated to a context and it the component reports a change in its context, the system status monitor will check it and may relocate the query.

4.2. The Processing Component

We propose a general processing component model that can have specific implementations for heterogeneous devices. This model is pictured in the Figure 3:

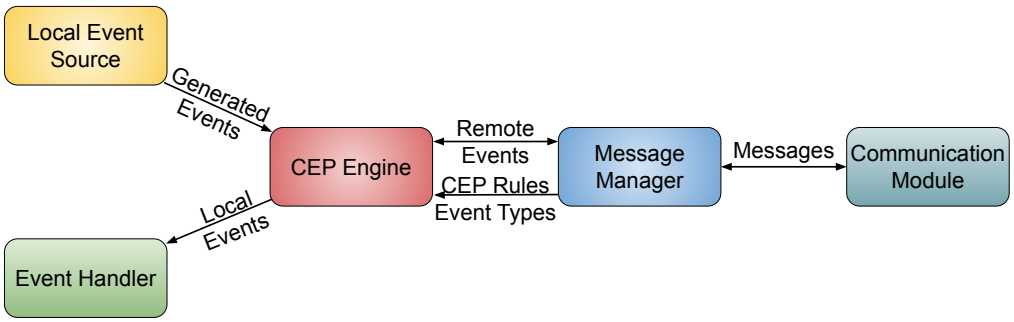

Figure 3. The processing component.

Similar to the core component, the **connection module** handles all external communication. When it receives a message, it is forwarded to the **message handler**. The handler then unwraps the message and extracts its contents. The messages may contain queries, the definition of event types, or external events. The **CEP engine** instantiates the new queries and registers the event types. If the message contains an event, it is processed based on the previously instantiated queries. If the processing component receives a new version of a query that it is already executing, it removes the old version and instantiates the new version of the query. The messages containing queries also contain a tag indicating if the outputted events should be handled locally, distributed to other processing components, or both.

If the message indicates that the outputted events should be distributed, the message handler will subscribe to the instantiated query. The generated events are then encapsulated in messages and the message handler requests the communication module to transmit them.

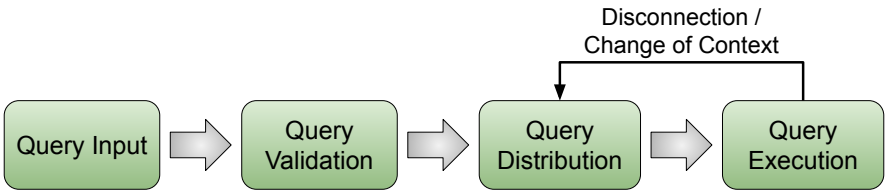

Figure 4. Query deployment sequence.

If the query tag indicates that the output should be handled locally, the outputted events can be further processed by other continuous queries on the same processing component. Also, the **event handler** subscribes to those queries. The event handlers on the processing components function as the sinks of the EPN. They are the interface that applications developed based on our model can use to obtain the output of the data processing.

The data input occurs in the **local event source** of processing components. The local event sources generate events that are fed to the CEP Engine. For instance, a local event source may gather sensor readings or get information from online data sources. Our model does not contain a direct connection between the local event source and the message manager. We stimulate pre-processing the events locally before distributing them. This pre-processing can reduce network band consumption and even save energy[Rios et al. 2016].

4.3. Query Deployment Sequence

In our model, the query deployment can be divided into four steps, which are depicted in the Figure 4.

The system receives continuous queries to be deployed in the query input step. This step also comprises the definition of where the query should be allocated. We consider the specification of queries to a particular processing component, to an IoMT layer, or a context. By the IoMT layer, we mean cloud, stationary edge, and mobile edge. A context, however, may include many aspects, for instance, location, minimum battery level, or sensor availability.

After the query input, the query is validated and distributed to a specific processing component. If the query was allocated to a layer or a context, the distribution step also comprises choosing a processing component to execute the query. That means checking if one of the connected processing components fills the allocation specification. If there is none, the system will wait until one is available to deploy the query.

The last step is the actual query execution. However, the device that was choosen to execute a query may disconnect or change its context; for example, move out of the query's location. In those cases, the query is deallocated from the old processing component and goes back to the query distribution step.

5. The Global CEP Manager

To implement the proposed model, we developed the Global CEP Manager (GCM). GCM is a distributed CEP platform for ContextNet nodes. We implemented the core component has the GCM Core, a service that runs in a ContexNet SDDL node. We developed

distinct implementations of the processing component for different devices; we call then Processing Agents (PA). We developed a cloud PA for regular computers or servers and an edge PA for the Raspberry Pi. Furthermore, we also adapted the M-Hub and the MEPA service to make it compatible with GCM as a mobile PA.

We adopted the Esper CEP engine as it is open source and uses Event Processing Language (EPL) similar to SQL to describe continuous queries. In our implementation, each continuous query has the following meta-information:

- A user provided label that identifies the query.
- A user-provided tag that identifies which IoT layer should receive the query output. We support five tags: Cloud, S_Edge, M_Edge Local, and Global. Cloud: the cloud PAs will receive the query output. S_Edge or M_Edge: the stationary or mobile edge PAs respectivelly will receive the query output. Local: the output is not distributed and is handled locally on the processing agent. Global includes all PAs.
- A timestamp of the query creation or modification. This timestamp is automatically generated on the GCM Core and represents the query version since we support query updates.

The Global CEP Manager Core functions as the core component we defined in our model. The Figure 5 displays the architecture of the GCM Core.

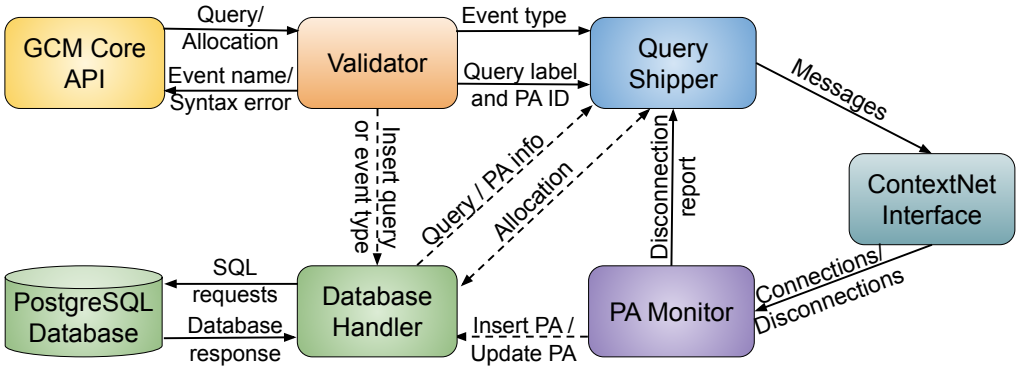

Figure 5. The Global CEP Manager Core.

The **GCM Core API** implements both the monitoring interface and the input interface described in our model. It provides Java methods to create and update queries or allocate them to PAs, as well as methods to check their status. The **validator** implements the validation module. It checks if the queries are syntactically correct in Esper EPL and if they process existing event streams. If the validation fails, the validator produces a syntax error. If the validation is successful, the validation module generates the event type for the events that will be generated by the query. Then it returns the name of the event type and requests the **query shipper** to distribute the event type to the necessary PAs. Those are the PAs related to the query output tag we described at the begin of this session. Then the validator saves the query on the database.

Our present implementation supports assigning queries to a specific processing agent, to all processing agents of a type (cloud PA's, stationary Edge PAs or M-Hubs),

or one automatically selected processing agent of a type. Currently, the GCM does not support assigning queries to a context. If the query is designated to a specific PA, the validator checks if the designated PA is already registered on the database. Then the query label and the assignment info are passed on to the query shipper. The query shipper implements the distribution module and is responsible for shipping the query to the necessary processing nodes. If the query was designated to a specific PA, the query shipper just encapsulates the query in a message and requests the ContextNet interface to send it. However, other designation options will require the query shipper to perform additional steps. First, the query shipper will get information about the active processing agents of the designated type from the database. If the query was designated to all nodes of the type, the query shipper will request the ContextNet interface to send the message to each one of them. If the query was designated to be executed in one automatically chosen PA of a given type, the query shipper picks one randomly. Then the query shipper also saves information about the query allocation on the database. That includes the PA currently running the query and if the query is reallocatable.

The **ContextNet interface** is an implementation of the communication module for the ContexNet middleware. The functions of the system status monitor were divided between the **PA Monitor** and the **Query Shipper**. The ContextNet Interface reports PA connections and disconnections to the PA Monitor. The PA monitor then inserts or updates the PA information on the database. The PA Monitor also reports PA disconnections to the Query Shipper. If the disconnected PA was running a query that was allocated to one automatically chosen PA of its type, the query is reallocated. The query shipper picks another PA of the same type currently active and ships the query to that PA.

We used a PostgreSQL database as persistent storage. Our implementation includes a database handler that offers methods for data access. The database handler performs SQL requests to the database.

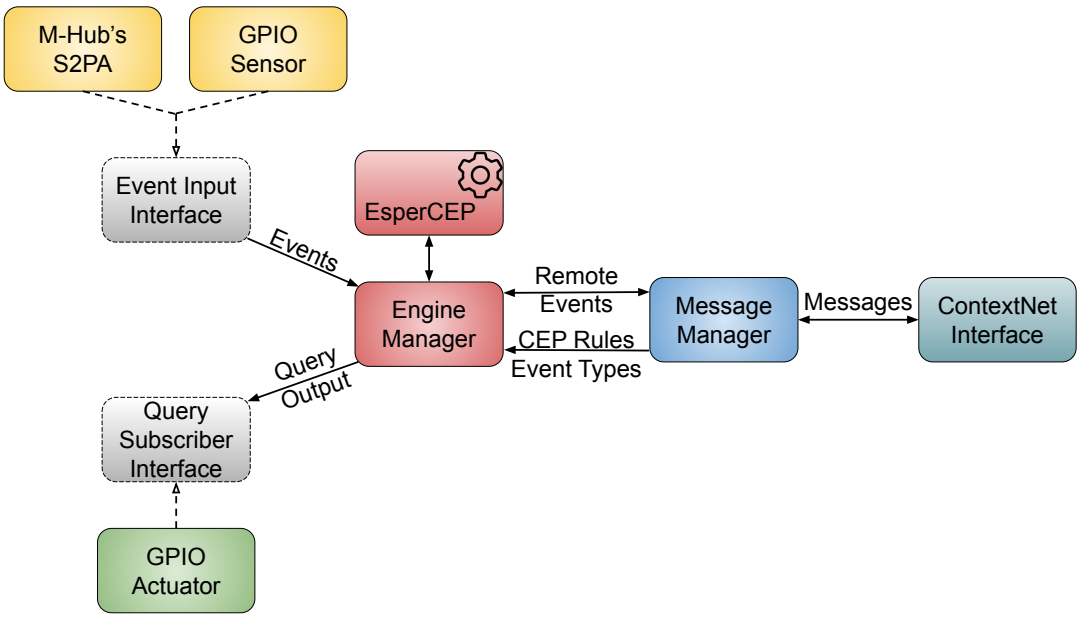

Figure 6. The Global CEP Manager Processing Agent.

The Figure 6 contains the general architecture of the PAs. This architecture is very

similar to the processing component proposed in our model. The **event input interface** and **query subscriber interface** are implementations of the local event source and the event handler, respectively. They are Java interfaces that can be implemented by the user applications developed using our platform. In the edge PA that runs in Raspberry Pi we included implementations of those interfaces that can use the sensors and actuators connected to the General Purpose Input/Output (GPIO). We included an **engine manager** that handles adding and removing continuous queries from the EsperCEP engine as well as feeding it events and collecting the queries' output.

Our implementation includes some additional operations to ensure consistency. When a PA connects (or reconnects) to the GCM Core, the ContextNet Interface issues a notification to other components in the PA using EventBus[3]. Upon receiving this notification, the engine manager produces a list with the label and version of each query running in the processing agent. This list will be sent through the system infrastructure and ultimately reach the query shipper on the GCM core. The query shipper checks if the list is consistent with the information on the database. If the database contains a query that should be allocated to the PA that was not listed or there is a new version of the query, the query is shipped. If the list contains a query that is not currently allocated to the PA according to the database, the query shipper sends a message requesting the removal of the query.

6. Case Study

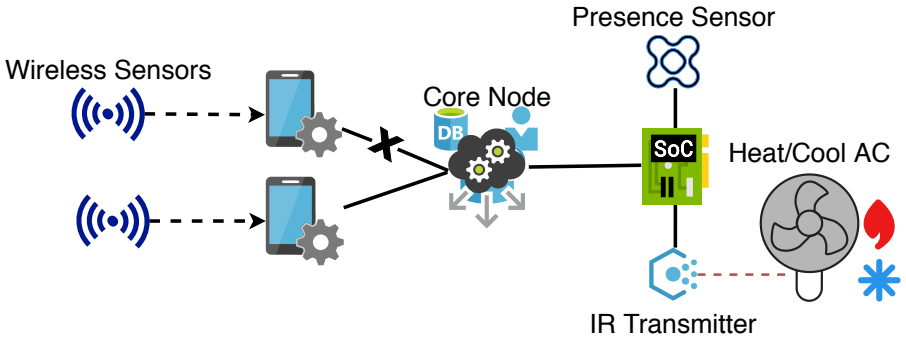

Figure 7. Use Case

We chose a smart home/office use case to demonstrate the adaptability or our solution as well as to measure its communication overhead. The scenario, depicted in Figure 7, contains two real temperature sensors that are connected to different M-Hubs through Bluetooth Low Energy. It also contains a Raspberry Pi emulating a presence sensor and an IR transmitter that controls a heat/cool AC system. Both the M-Hubs, the Raspberry, and the Core node are all connected to the same wi-fi network.

The M-Hubs automatically produce **SensorData** events collected from the connected sensor. We created a query that collects the average of the temperature sensor's readings and produces **AverageTemperature** events every 5 seconds and are sent to the Core Node (running on a server). That query is displayed on the Listing 1 We used the

[3]EventBus is a library that offers a pub/sub protocol for loose coupling application classes that may run in different threads. http://greenrobot.org/eventbus/

GCM to assign this first query to an automatically selected M-Hub and to distribute those events to stationary edge nodes (e.g., the Raspberry Pi). We then configured the Raspberry Pi to produce **Presence** events when the presence sensor is activated. We also used the GCM to issue some queries to the Raspberry Pi displayed on Listing 2. The first query detects when there are no Presence events for 20 minutes. This query produces **TurnACOff** events. Another query aggregates AverageTemperature events and Presence events. This second query activates when a presence is detected, and the temperature is outside of a comfortable interval (e.g.: between 22°C and 26°C). This query produces **TurnACOn** events. We used a variable to store if the AC is on or off, to avoid sending multiple TurnACOff or TurnACOn events. A query subscriber may consume either TurnACOff and TurnACOn events and use the IR transmitter to turn AC off or turn it on at 24°C.

Listing 1. M-Hub query.

```
insert into AverageTemperature
select avg(sensorValue[1]) as temperature from
SensorData(sensorName='Temperature').win:time_batch(5 sec)
```

Listing 2. Raspbery queries.

```
-- There is no Presence event during an interval of 30 minutes,
   and the AC is on
insert into TurnACOff select true from pattern
[ every (timer:interval(30 minutes) AND NOT Presence) ]
where var_isACOn=true

-- A Presence event is followed by an out of bounds
   AverageTemperature event, and the AC is off
insert into TurnACOn select true from pattern
[ every Presence ->
AverageTemperature(temperature not in [22:26])
where timer:within (20s) ]
where var_isACOn=false

 -- These queries change the value of the variable automatically
on TurnACOff set var_isACOn=false
on TurnACOn set var_isACOn=true
```

First, we measured the message overhead of our solution. We developed a static implementation of the scenario composed of applications that implement the same queries hard-coded on the same devices. Since the static implementation would not support query relocation, we used only one M-Hub and one sensor. We compared the number of messages over time produced by our solution and the static implementation. Figure 8 displays the results of this first test. As we can see, the message overhead only occurs in the beginning, that is due to the initial steps performed when a PA connects. These steps are the handshake between the PA and the Core Node as well as the actual query shipping process. If we consider that the nodes are already connected and measure only the impact of

Test	1	2	3	4	5	6	7	8	9	10
Delay (s)	3.285	2.381	3.893	3.610	2.947	2.4260	3.594	3.899	3.659	2.562
Average	**3.226**									

Table 1. Delay until the query is relocated.

shipping the queries, the overhead greatly reduces. This overhead is a natural consequence of the dynamism and flexibility offered by our solution. On the static implementation, the overhead does not occur; however, being hard-codded, it does not offer some key benefits of our solution. Those benefits are: a central platform to distribute queries among devices spread in multiple locations, easy modification of executing queries, and query relocation. On the static implementation, it would be necessary to re-code, recompile, and reinstall an application to include new queries or new versions of a query. Furthermore, in most use cases, even if the system manager often issues new queries or new versions of queries, the event distribution tends to surpass the network usage of the query distribution largely.

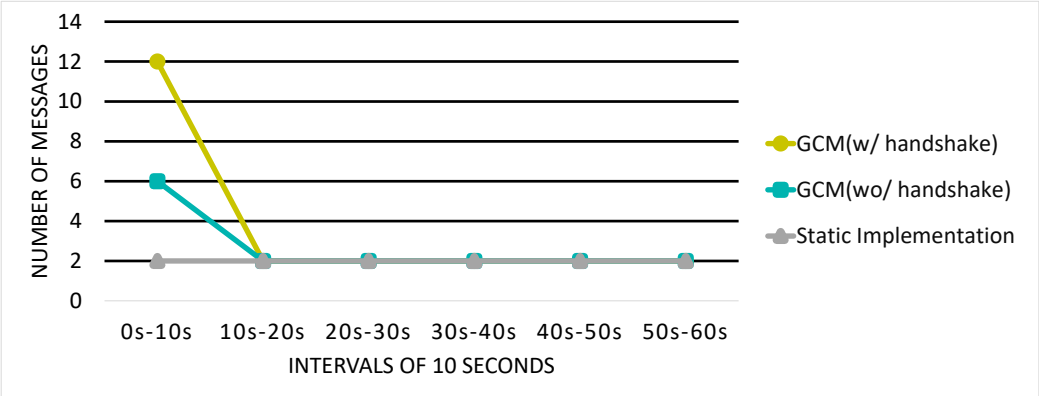

Figure 8. Message overhead.

To test the adaptability of our solution, we used the full scenario, with two M-Hubs and two wireless sensors. After the queries are deployed, and the full system is running, we simulated a disconnection. We turned off the wi-fi on the M-Hub running the query. Then we waited until the query was redistributed to the other M-Hub. We repeated this test ten times and measured how long does it take for the query to be automatically redistributed to the other M-Hub. We display our measurements in Table 1, and on average, it takes 3.23 seconds. In the presented use case, this delay does not represent a significant loss since the AverageTemperature events are outputted every 5 seconds. On more time-critical applications, it would be necessary to detect the disconnection faster or include redundancy.

7. Summary and Future Work

In this paper, we presented a model for a distributed CEP platform that facilitates the engineering and deployment of distributed CEP solutions. We also presented GCM, an implementation of this model that supports running queries on heterogeneous devices. Our solution is flexible and adaptative as it supports issuing new queries and new versions of queries that are automatically distributed and can automatically allocate and relocate queries if necessary. We adopted a use case scenario to demonstrate the advantages of our

solution and also used this scenario to measure the message overhead generated by our solution. This overhead concentrates on the start and does not impact the network over time.

As future work, we are currently developing the support to assign queries to contexts, as discussed on or model, to the GCM. Then it will be possible to assign a query to a specific location, for instance, and the system will automatically allocate or relocate the query to devices on this location. We also intend to test the GCM on other scenarios further.

References

Balazinska, M., Balakrishnan, H., Madden, S., and Stonebraker, M. (2005). Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 13–24, New York, NY, USA. ACM.

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment.

Cugola, G. and Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62.

Endler, M. and e Silva, F. S. (2018). Past, present and future of the contextnet iomt middleware. *Open Journal of Internet Of Things (OJIOT)*, 4(1):7–23.

Etzion, O., Niblett, P., and Luckham, D. C. (2011). *Event processing in action*. Manning Greenwich.

Jayasekara, S., Kannangara, S., Dahanayakage, T., Ranawaka, I., Perera, S., and Nanayakkara, V. (2015). Wihidum: Distributed complex event processing. *Journal of Parallel and Distributed Computing*, 79-80:42 – 51. Special Issue on Scalable Systems for Big Data Management and Analytics.

Luckham, D. (2002). *The power of events*, volume 204. Addison-Wesley Reading.

Luckham, D. C. (2011). *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons.

Pietzuch, P. R., Shand, B., and Bacon, J. (2003). A framework for event composition in distributed systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 62–82, New York, NY, USA. Springer-Verlag New York, Inc.

Rios, L. T., Endler, M., and Colcher, S. (2016). An energy-aware iot gateway, with continuous processing of sensor data. In *SBRC2016*, XXXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC2016, Salvador, Brasil.

Starks, F. and Plagemann, T. P. (2015). Operator placement for efficient distributed complex event processing in manets. In *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 83–90.