Pontifícia Universidade Católica
do Rio de Janeiro

Fernando Benedito Veras Magalhães

Distributed CEP for Context-Aware Adaptive Acquirement and Processing of Information

PUC-Rio - Certificação Digital Nº 1820993/CA

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Markus Endler

Rio de Janeiro
April 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Fernando Benedito Veras Magalhães

Distributed CEP for Context-Aware Adaptive Acquirement and Processing of Information

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Markus Endler
Advisor
Departamento de Informática – PUC-Rio

Prof. Noemi de La Rocque Rodriguez
Departamento de Informática – PUC-Rio

Prof. Francisco José da Silva e Silva
Departamento de Informática – UFMA

Rio de Janeiro, April 14^{th}, 2021

PUC-Rio - Certificação Digital Nº 1820993/CA

All rights reserved.

Fernando Benedito Veras Magalhães

Bachelor's degree in Computer Science at Federal University of Maranhão (UFMA) in 2018.

PUC-Rio - Certificação Digital Nº 1820993/CA

Bibliographic data

Magalhães, Fernando Benedito Veras

Distributed CEP for Context-Aware Adaptive Acquirement and Processing of Information / Fernando Benedito Veras Magalhães; advisor: Markus Endler. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2021.

v., 61 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Internet das Coisas. 2. Ciência de Contexto. 3. Processamento de Eventos Complexos. 4. Processamento de Fluxos Distribuído. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

To God for everything. To my Advisor, Markus Endler, for guidance in this journey and for his patience. To my family and friends for their support. To Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for partially financing this research. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

PUC-Rio - Certificação Digital Nº 1820993/CA

Abstract

Magalhães, Fernando Benedito Veras; Endler, Markus (Advisor). **Distributed CEP for Context-Aware Adaptive Acquirement and Processing of Information**. Rio de Janeiro, 2021. 61p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

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). Initially, a single computer/cluster would concentrate all the CEP execution. However, a centralized execution of CEP is not suitable for coping with the high volume, velocity, and volatility of IoT sensors' data streams. Instead, applications using CEP should deploy a distributed CEP Event Processing Network, preferably having CEP agents both in the cloud and at edge devices. Also, deciding the arrangement used to split the processing among these tiers and their devices can be just as important. That said, being aware of each of the devices' current context, for instance, their location and available sensors, can help to collect and (partially) process the data on devices close to the data's production site. This work presents a context-aware distributed CEP platform called Global CEP Manager (GCM). GCM is a service of the ContextNet middleware that supports the context-based deployment, and dynamic rearrangement of CEP queries to CEP engines executing in the cloud, stationary edge devices, and M-Hubs, which are ContextNet's mobile edge devices. GCM uses the ContextMatcher, which is also part of this work. ContextMatcher is a module for ContextNet applications that enables the delivery of messages for nodes that match a specified set of contextual requirements.

PUC-Rio - Certificação Digital Nº 1820993/CA

Keywords

IoT; Context-awareness; Complex Event Processing; Distributed Stream Processing.

Resumo

Magalhães, Fernando Benedito Veras; Endler, Markus. **CEP Distribuído para Aquisição e Processamento de Informação Adaptativos Cientes de Contexto**. Rio de Janeiro, 2021. 61p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A disseminação atual da IoT aumenta a implantação de soluções de processamento de fluxo de dados para monitorar e controlar elementos do mundo real. Uma dessas soluções é o Processamento de Eventos Complexos (CEP). Inicialmente, um único computador ou cluster concentraria toda a execução do CEP. No entanto, a execução centralizada do CEP não é ideal para lidar com o alto volume, velocidade e volatilidade dos fluxos de dados dos sensores IoT. Em vez disso, as aplicações CEP devem criar e decentralizar o processamento de eventos CEP, de preferência tendo agentes CEP na nuvem e em dispositivos na borda. Além disso, tão importante quanto a descentralização, é decidir como o processamento será dividido entre esses dispositivos. Dito isso, estar ciente do contexto atual de cada dispositivo, por exemplo, sua localização e sensores disponíveis, pode ajudar a coletar e (parcialmente) processar os dados em dispositivos próximos ao local onde os dados foram produzidos. Este trabalho apresenta uma plataforma de CEP distribuído com ciência de contexto chamada Global CEP Manager (GCM). GCM é um serviço do middleware ContextNet que oferece suporte à implantação e ao rearranjo dinâmico de consultas CEP baseados em contexto para motores CEP em execução na nuvem, em dispositivos na borda estacionários e M-Hubs, que são dispositivos na borda móveis do ContextNet. O GCM usa o ContextMatcher, que também faz parte deste trabalho. ContextMatcher é um módulo para aplicações ContextNet que permite a entrega de mensagens para nós cujo contexto esteja de compatível com um determinado conjunto de características contextuais.

PUC-Rio - Certificação Digital Nº 1820993/CA

Palavras-chave

Internet das Coisas; Ciência de Contexto; Processamento de Eventos Complexos; Processamento de Fluxos Distribuído.

Table of contents

PUC-Rio - Certificação Digital Nº 1820993/CA

PUC-Rio - Certificação Digital Nº 1820993/CA

List of figures

PUC-Rio - Certificação Digital Nº 1820993/CA

List of tables

PUC-Rio - Certificação Digital Nº 1820993/CA

List of abreviations

AC – Air Conditioner
API – Application Programming Interface
CEP – Complex Event Processing
CQ – Continuous Query
DB – Database
DDS – Data Distribution Service
EPA – Event Processing Agent
EPL – Event Processing Language
EPN – Event Processing Network
GCM – Global CEP Manager
GPIO – General-purpose input/output
IoMT – Internet of Mobile Things
IoT – Internet of Things
MQTT – Message Queuing Telemetry Transport
OS – Operating System
PA – Processing Agent
REST – Representational State Transfer
WPAN – Wireless Personal Area Network
SoC – System on a Chip

PUC-Rio - Certificação Digital Nº 1820993/CA

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 high volume data flows as fast as possible. These systems constitute what (1) classify as Information Flow Processing (IFP), or else, stream processing systems.

The authors of (2) 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. It is possible to define the first characteristic as focusing on immediate or low latency responses instead of the traditional method of storing all the data for future analysis. In contrast, the second means that instead of processing the entire, eventually stale, set of received events, the stream processing model will focus on analyzing and detecting specific patterns of events in time-bounded sliding/batch windows, disregarding the old data (i.e., the data outside of the current window).

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. They offer a series of operators to process the data and detect when a specific situation occurs. Usually, it is possible to categorize these operators as filters, aggregation, or patterns. Filters select only data entries that have an attribute within a specific interval. Aggregation combines multiple streams. Finally, patterns define a sequence of entries with particular characteristics.

In the initial approaches to stream processing, many stream processing systems would concentrate all data stream processing at a centralized site, such as a central server, a cluster, or a cloud. Such configuration has some advantages; we may cite: simpler maintenance, easier enforcing of standards and security, and less data redundancy. However, many modern stream processing solutions started to adopt a distributed approach, even pushing some of the processing to the edge (3), closer to where the raw data (e.g., sensor data) is acquired.

PUC-Rio - Certificação Digital Nº 1820993/CA

This distributed approach has several benefits. For instance, it can reduce processing bottlenecks since it is not concentrated at the cluster/cloud, but pushes some of the processing to the edge (3). That approach means that a fair amount of data processing can be done locally, at the edge, reducing the latency to generate results and possibly the latency to react to undesired/unexpected situations. For example, continuous queries that operate on data only from sensors directly connected to an edge node and also drive some local action, such as raising an alert can be fully processed locally. Moreover, the distributed approach can also reduce the network bandwidth usage since instead of forwarding all the raw data, edge nodes can send only pre-processed (e.g filtered, aggregated or summarized) data. And since almost all stream processing systems essentially analyze data that "flows from the periphery to the center" (1), it is only natural to think of stream processing systems that divide the global data processing logic between the nodes along the path of the data flow (4).

Many flavors and approaches for stream processing have been proposed. For instance: Active Databases (5), Data Stream Management System (DSMS) (6), and ultimately Stream Reasoning (7) and Complex Event Processing (CEP). CEP (8) 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 label (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 that may include multiple events from different event streams and consider characteristics like the sequence in which the events occurred. 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. Lastly, the defining advantage of CEP is the possibility of producing complex events, which are events derived from other, more basic, events and assembled/created by the CEP engine.

The CEP engine is the component that processes every event based on the provided continuous queries. When one of those queries is activated, for instance, upon the occurrence of those complex patterns, a complex event may be generated. For example, the previous pattern could generate a FireWarning event type holding the average temperature readings of the original TemperatureOutOfBounds events. Those complex events will then

PUC-Rio - Certificação Digital Nº 1820993/CA

constitute new event streams, meaning that other continuous queries can further process them.

Distributed CEP implementations are often not only composed of cloud nodes but also edge nodes. Those edge nodes can be distributed in the monitored environment and may even be mobile. In those cases, the nodes can differ in many aspects. In addition to location, they may differ, for instance, in the availability or proximity of sensors, their processing power, and they may be connected to a power grid or may run on a battery with a particular amount of remaining charge. That means that each node will capture a particular set of information from the environment and may have a certain amount of resources that allow them to act as a remote CEP engine. Even two nodes equipped with thermometers will provide data relevant to different applications if they are far from each other. That means that being aware of the context of each node can help the distributed CEP system provide richer and more precise information.

1.1 Problem statement

PUC-Rio - Certificação Digital Nº 1820993/CA

The process of designing and deploying distributed CEP solutions presents some challenges. For instance, it is frequent to modify 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 in the desired situations. This characteristic justifies using an infrastructure where the user can state and deploy new queries or new versions of a query at runtime. This infrastructure could also check if a query will function adequately (internally and in conjunction with the other queries) before the query's deployment. This check would enable the user to fix mistakes that perhaps would only be detected at the trial phase.

Furthermore, to ensure the event type oriented coupling of the queries dispersed in the system, it is essential to ensure that the processing nodes that will consume the events a query outputs already know the event type of these events. For instance, imagine that query A produces events that a subsequent query B consumes where B and A are executed in different nodes. The node that implements B should previously know the characteristics of the events produced by A. Just as well, updates on queries and events should be checked to make sure they stay compatible with previous queries that consume the events.

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, a distributed CEP system may also be adaptable, automatically allocating and reallocating queries to CEP-enabled devices. For instance, the user may specify a set of contextual requirements that a device needs to answer to execute a query correctly. The system would be responsible for finding one (or all the devices) that fill the requirements and then automatically allocate and deploy the query. Those contextual requirements may include characteristics like available sensors, battery level, processing power, and location of the CEP-enable devices. The automatic reallocation 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 the devices may change their contextual attributes, for instance by disconnecting from sensors from one location and discovery new sensors when they move to a new location and by the battery dropping below a certain threshold.

PUC-Rio - Certificação Digital Nº 1820993/CA

1.2 Objectives

1.2.1 General Objective

The general objective of this master thesis is to discuss and provide a platform for distributed CEP applications that divide the stream processing between heterogeneous nodes in different IoMT tiers. That includes the software that enables the execution of CEP in those nodes and the distribution of Continuous Queries and Events between those nodes.

1.2.2 Specific Objectives

- Support the submission of new versions of queries and automatically distribute the new versions to every node executing the old version.
- Considering a set of continuous queries where some queries process the output of previous queries, assist in the interoperability of continuous queries. First, generating the event definition for the event types that will be output by each rule, then disseminating these event types transparently to the necessary nodes. Also, facilitate the connection of queries running in different tiers by selectively distributing generated events and event types.

– Assist in preserving the internal and mutual consistency of continuous queries before their deployment by validating the query when it is declared and when it is assigned to a node or a set of nodes.
– Investigate the use of the nodes' context as a parameter to decide how to allocate continuous queries. Given CEP queries with contextual requirements (e.g., location, available sensors, processing power), enable the automatic allocation and reallocation of this type of query to the CEP-enabled nodes based on these requirements. Also, reallocating them in case of disconnections.
– Evaluate the solution presented in this work by demonstrating a use case scenario and performance tests.

1.3 Items out of Scope

Given that our general objective is to provide a platform for distributed CEP applications, there are some aspects that, although relevant, we don't tackle in this master thesis:

– Distribute the service of query and event deployment. Our solution includes a central node that manages the query deployment. We don't offer support for replication or redundancy of this node. Our solution offers a level of reliability to disconnections since continuous queries can be reallocated from disconnected nodes to available nodes. However, our solution is not reliable to the disconnection of this central node.
– Replay of events lost in incomplete event windows due to a query reallocation. After a query is reallocated, the new node will process only the new events using the reallocated query. Old events that the query would process process are lost. Our solution does not include the replay of old events.
– Load balancing of queries. Our solution does not offer the meanings to automatically allocate or reallocate queries to nodes with the objective of balancing the processing load between nodes. The nodes chosen to handle the queries that execute on a single node are picked randomly among the available compatible nodes.
– Granting security of the data. Our solution does not use methods to grant the security of the data in any data acquisition or processing stage. To implement security we would recommend using a firewall and/or controlling the access to the network.

PUC-Rio - Certificação Digital Nº 1820993/CA

1.4 Allocation and Assignment Operations

During the development of this work, we found two operations in the query distribution that can cause ambiguity since it is possible to denominate them using the same words. The first one is when the user/system administrator defines how to distribute the queries. That definition can be to send the query to a specific node or to send to the nodes that fill a set of contextual requirements. The second one is when the system picks the nodes that will receive the queries based on these requirements. Although these are very different operations, terms such as assign, allocate, appoint and designate could refer to both of them. To avoid this ambiguity, we define here the terms used in this document:

- **Assign:** When the system administrator define how to distribute the query.
- **Unassign:** When the system administrator undoes the assignment of query, meaning that the query should not be deployed based on the undone assignment.
- **Allocate:** When the system picks the devices that will receive the query based on the assignment.
- **Deallocate:** When the system undoes the allocation of a query to a node because of an unassignment, because the node disconnected or because the node does not fill the set of contextual requirements anymore.
- **Reallocate:** The process of deallocating the query from one node and allocating to another one.

PUC-Rio - Certificação Digital Nº 1820993/CA

1.5 Outline

The remaining of this master thesis is organized as follows: Chapter 2 presents the enabling concepts and technologies used in this work. Chapter 3 presents related works. Chapter 4 presents concepts of the approach adopted to achieve this work's objectives. Chapter 5 presents how this concepts were implemented. Chapter 6 presents how this works validate continuous queries and the assignment of continuous queries. Chapter 7 presents the evaluation of this work. Finally, Chapter 8 concludes this master thesis.

2
Background

This chapter presents the main theoretical and technology background that enables our work. First, it further explains CEP and introduces Event Processing Networks. Then it introduces the ContextNet middleware.

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. It specializes in detecting the occurrence of patterns within these flows. The coining of the term Complex Event Processing is attributed to David Luckham(9) (8).

An event can represent the change of state of an analyzed entity or the regular measurement of one or more properties of that entity (e.g., a departure, an arrival, or a location update). An event can aggregate multiple attributes. A computer status event, for example, can contain the level of processor and main memory usage, temperature, and other attributes that are relevant to the application that will consume that event. Every event has an event type; **Event types** are to events what DB tables are to rows, they define a name/label (e.g., `DepartureEvent`) and the event attributes (e.g., `station_id`:int, `bus_plate`:string, `timestamp`:long). That said, what differentiates CEP from the simple processing of event flows is the generation of complex events that later constitute new event flows to be processed as well. Therefore, complex event processing refers not to the complex processing of events but to the processing of complex events where the higher the complexity of the event, the higher the abstraction level of the initially acquired data it represents.

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 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

PUC-Rio - Certificação Digital Nº 1820993/CA

constitute a complex event since it represents higher-level information (12).

Finally, as a visual aid to understanding CEP, Figure 2.1 contains a diagram that depicts the stages of data stream analysis using complex event processing. CEP can process data originated from multiple **data sources**. The first stage of processing this data is to encapsulate it in **simple events**, which are also named atomic events or raw events. Next, there is a software module commonly named **CEP engine** that process the **event streams**. This CEP engine knows the previously stated **continuous queries**. Continuous or standing queries are the mechanism offered by CEP technologies to state how the events should be analyzed. Every new event provided as input to the CEP Engine is tested based on the previously stated continuous queries. The CEP engine can also derive **complex events** by combining information from one or more simple events, such as a `FireEvent` generated from `TemperatureRaise` events followed by a `SmokeDetected` event. After generated, complex events are fed back to the CEP engine to also be processed. At last, the result of event processing can be automatic **actuation** or the production of **information**.

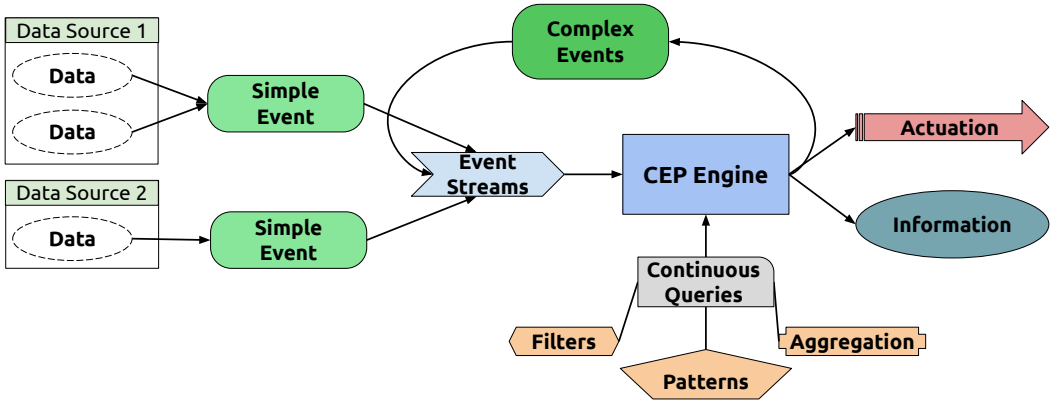

Figure 2.1: Data-stream analysis using CEP

The complex events produced by a continuous query can be consumed by subsequent queries constituting a multi-level processing pipeline. A directed graph called Event Processing Network (EPN) (10) is commonly used to represent this pipeline. 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 constructing a distributed CEP system since EPAs can be deployed on different devices. That means the designing of a distributed EPN with continuous queries interconnected, however executing in different devices. Figure 2.2, for instance, can provide an overview of an EPN in an IoT scenario with devices spread in different tiers. In this scenario, edge devices can execute filtering

PUC-Rio - Certificação Digital Nº 1820993/CA

rules that require low processing power. In contrast, cloud devices execute rules that aggregate the output of those filtering rules and rules that detect the occurrence of complex patterns.

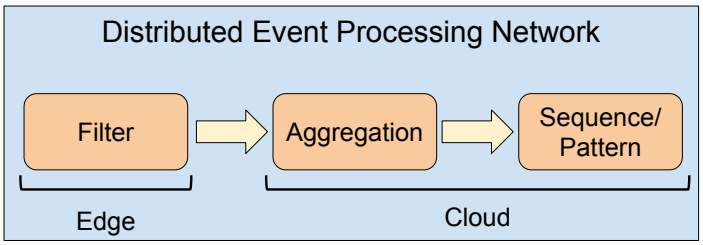

Figure 2.2: A distributed Event Processing Network

2.2 ContextNet and M-Hub

The ContextNet (11) 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 can automatically discover and connect to smart things with Bluetooth WPAN. 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 our solution.

PUC-Rio - Certificação Digital Nº 1820993/CA

[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/`

3
Related Work

Distributed CEP is already well established among information processing technologies. For instance, (13) dates back to 2003 and presents a framework for Distributed CEP implementation. However in this section we would like to focus on works that are more representative of the current development state of distributed CEP systems.

3.1
Query distribution strategy

One characteristic that usually diverges in different distributed CEP works is how they allocate the continuous queries among the CEP nodes. One of the possible approaches is the one adopted by (16). It proposes a implementation of 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 with the objective of reducing the message overhead. It assumes tree types of nodes: A single sink node that assemble the information from the MANET; Stationary sensor nodes; And mobile CEP-enabled nodes. Queries that acquire sensor data are allocated to the mobile node closest to the source sensor node. Queries that aggregate data from different event streams (flows of events of the same type) are allocated to the node where the event streams intersect in the MANET.

Another approach on allocating the CEP queries is the one adopted by (18) which focus on load balancing. Before allocating the queries, the authors' system needs to receive 3 parameters: All the event types and their expected input rate (the frequency in which the events are produced); The whole EPN with the event types consumed by each query; And the number of CEP-enabled nodes. They named allocation cost the metric used by the system to decide the allocation plan. The allocation cost for each node N_i is calculated multiplying two values: $InputRate_i$ and $UnitRate_i$. Each query allocated to N_i consumes a set of event types and uses a number of data operators. The $InputRate_i$ is the sum of the input rates of all those event types. The input rate of each event type is only summed once, even if it is consumed by more than one query allocated to N_i. The $UnitRate_i$ is the sum of all operators in all queries allocated to

PUC-Rio - Certificação Digital Nº 1820993/CA

N_i. The $UnitRate_i$ counts the operator usage, so if the same operator is used by multiple queries it will be counted multiple times. Then the system uses a greedy algorithm to decide the allocation of queries, trying to reduce the sum of the allocation cost of all nodes.

Going even further, some works like (14), allow the execution of queries in multiple nodes in parallel. First it divides each event stream (flow of events of the same type) in multiple event streams based on an identifier attribute. For example $RoomTemperature$ events can be divided in to multiple streams according to the $room\_id$ attribute. Each division is called a partition of the original event stream. Then each query is split into operators that should be executed in a sequence. The authors use the term pipeline for each operator sequence that represent a query. In that work, each operator of a pipeline could be allocated to a node. Also, some operators can be executed in parallel in multiple nodes, as long as each event stream partition is directed to a single node. For instance, a join on $RoomTemperature$ and $RoomHumidity$ can be executed in parallel as long as events with the same $room\_id$ are processed on the same node.

One characteristic that usually diverges in different distributed CEP works is how they allocate the continuous queries among the CEP nodes. One of the possible approaches is the one adopted by (16). It proposes an implementation of 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 to reduce the message overhead. It assumes three types of nodes: A single sink node that assembles the information from the MANET; Stationary sensor nodes; And mobile CEP-enabled nodes. The authors' system allocates queries that acquire sensor data to the mobile node closest to the source sensor node. In the same way, it allocates queries that aggregate different event streams to the node where the event streams intersect in the MANET.

Another approach on allocating the CEP queries is the one adopted by (18), which focuses on load balancing. Before allocating the queries, the authors' system needs to receive 3 parameters:

1. All the event types and their expected input rate (the frequency of production of events);
2. The whole EPN with the event types consumed by each query;
3. The number of CEP-enabled nodes.

They named allocation cost the metric used by the system to decide the allocation plan. The allocation cost for each node N_i is the multiplication of two

PUC-Rio - Certificação Digital Nº 1820993/CA

values: $InputRate_i$ and $UnitRate_i$. Each query allocated to N_i consumes a set of event types and uses a series of data operators. The $InputRate_i$ is the sum of the input rates of all those event types. The input rate of each event type is only summed once, even if more than one query allocated to N_i consumes the event type. The $UnitRate_i$ is the sum of the number of data operators in all queries allocated to N_i. The $UnitRate_i$ counts the operator usage as indicative of the query's complexity, so if multiple queries use the same operator, it will be counted multiple times. Then the system uses a greedy algorithm to decide the allocation of queries, trying to reduce the sum of the allocation cost of all nodes.

Going even further, some works like (14) allow the execution of queries in multiple nodes in parallel. First, it divides each event stream into multiple event streams based on an identifier attribute. For example, it is possible to divide `RoomTemperature` events into multiple streams according to the `room_id` attribute. Each division is called a partition of the original event stream. Then each query is split into operators that should be executed in a sequence. The authors use the term pipeline for the representation of a query as a sequence of operators. Then, the authors' solution allocates each operator of a pipeline to a node. Also, some operators can be executed in parallel in multiple nodes, as long as each event stream partition is directed to a single node. For instance, a join on `RoomTemperature` and `RoomHumidity` can be executed in parallel as long as events with the same `room_id` are processed on the same node.

PUC-Rio - Certificação Digital Nº 1820993/CA

3.2 Multi-tiered distributed CEP

By combining the execution of CEP in mobile and stationary edge nodes and cloud nodes, it is possible to take advantage of the proximity of the edge nodes to the sensor and actuators that assess and interact with the target entity/environment while also exploiting the cloud's processing power. In this sense, (17) provides a Remote Patient Monitoring (RPM) solution that uses smartphones as edge IoT gateways. The authors propose a multi-tiered architecture that can distribute the CEP workload between the edge and the hospital server. The sensors send health sensors data streams to the smartphone wirelessly. The patient or health agent chooses one of the queries packed with the smartphone application to process the data streams. The complex events produced by the smartphones are then sent to the hospital server to be further processed. There are some limitations in this solution, as the process of adding new CEP queries would require a recompilation of the

smartphone application. Also, the allocation of the CEP queries is performed manually instead of automatically.

3.3 Context-aware message distribution

Context-awareness is a well-explored field in middleware and communication software. One of the uses of context-awareness is delivering messages to nodes that fill a set of contextual requirements, especially location. In this scope, (19) is a recent work discussing geo-context aware message delivery in IoT environments. The idea is to forward messages only to nodes within a given geo-context. This approach can reduce unnecessary data transmissions or even enable data producers to restrict the information to a given area. The authors argue, for instance, that weather alert systems can take advantage of location awareness to notify only the nodes in the area affected by the weather conditions. The solution proposed in this master thesis indicates that the same principle of context-aware delivery is applicable in distributed CEP to deploy CEP queries to nodes depending on their contextual status.

3.4 This Work

Our work proposes a distributed CEP platform 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 runs on Android devices and can acquire data from sensors, local CEP processing, and sending resulting events to ContexNet gateways. Likewise, our solution includes the automatic allocation and reallocation of continuous queries based on the node's contextual status, also performing reallocations in case of disconnections. Compared to other methods of query distribution, our contextual approach does not inherently increase the scalability or load balancing. However, the node's context may be a crucial aspect of the data acquisition. For instance, our solution enables allocating queries that process temperature only to nodes with a thermometer available. Or, even more specifically, among these nodes, only the ones located around a point of interest. Lastly, our query deployment sequence includes a validation stage before dispatching the query to a CEP-enabled node. This validation can prevent the dispatching of poorly written queries that would generate errors on the node when trying to instantiate the query on the CEP engine. Our validation can also prevent the dispatching of new queries that would be incompatible with the current queries.

PUC-Rio - Certificação Digital Nº 1820993/CA

4
Proposed Approach

This chapter presents the concepts designed to achieve the objectives of this master thesis. It presents the model for a platform to build distributed CEP applications. The model provides the deployment of continuous queries on heterogeneous devices in an Internet of Mobile Things (IoMT) environment. The idea is to support the design and implementation of connected Event Processing Networks (EPNs) that distribute the data processing among nodes in different tiers (e.g., Edge and Cloud) and with different processing capabilities. Our model focuses on the following aspects:

- **Easy system management:** Our model comprises an simple interface for stating and assigning continuous queries. This interface 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 generally are 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 (i.e., the event's label 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.
- **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.

PUC-Rio - Certificação Digital Nº 1820993/CA

– **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 the event types that are produced by a previous query or originated in a data source (e.g., a sensor event). It also checks if new versions of a query will not impair other queries that consumed the output of the previous version because they can not consume the output of the new version.

– **Context-based query deployment:** Besides assign queries to specific nodes, the user/system administrator can assign queries to all or one of the nodes that fill a set of contextual requirements. The platform takes care of allocating the queries to the necessary nodes by acquiring and analyzing the contextual attributes of each node.

– **Adaptability:** Our solution aims to offer adaptable EPNs as it can automatically allocate and deallocate the continuous queries based on the contextual status of the nodes disperse in the environment.

PUC-Rio - Certificação Digital Nº 1820993/CA

4.1 General Architecture

Figure 4.1 displays a general view of the model established in this work. First, there is a **context-based message distribution layer** that runs on top of ContextNet, a middleware with reliable delivery of messages. Additionally, our model is composed of two elements, a **processing component**, and a **core component**. The processing component does the processing of data using CEP 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 and execute continuous queries that may produce events to be consumed internally or in other processing components. The core component handles the query distribution and assists in the correct coupling of the EPN parts. In the figure, the Core Node implements the core component. A **system administrator** interacts with the core component to issue and assign continuous queries. Section 4.3 present our idea of a Context-based Message Distribution Layer, Section 4.4 details the core component, and Section 4.5 details the processing component.

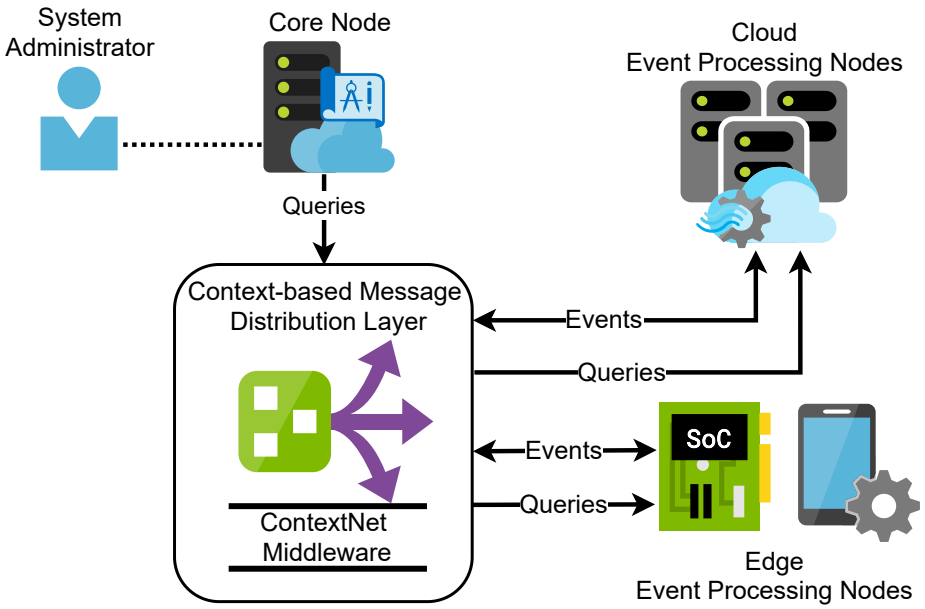

Figure 4.1: The general architecture of the proposed model.

4.2 Query Deployment Sequence

Figure 4.2 depicts the four steps of the query deployment in our model.

The system receives new or updated continuous queries in the query input step. This step also comprises the assignment of these queries. We consider the assignment of queries to a particular processing component or a target context. A target context may include one or more of the following context attributes: IoMT tier, processing power, location, minimum/maximum battery level, and sensor availability. By the IoMT tier, we mean cloud, stationary edge, and mobile edge. Context-assigned queries may be multicast or unicast. The system allocates multicast queries to every processing component that fills the necessary context attributes. However, if the query is unicast, the system allocates the query to only one of the processing components that fill the necessary context attributes.

After the query input, the query is validated and distributed to the processing components. Suppose the query was assigned to a context. In that case, the distribution step also comprises the query allocation (i.e., selecting the processing components that will receive the queries based on the contextual attributes).

The last step is the actual query execution. However, the processing components may disconnect or change their contextual attributes (e.g., move out of or into a context's location). In this case, it may be necessary to return to the distribution step, so the necessary continuous queries are allocated or deallocated.

PUC-Rio - Certificação Digital Nº 1820993/CA

It is essential to mention that our solution does not grant that queries assigned as unicast or multicast to a context will at all times be executed in at least one processing component since no processing component may fill the necessary contextual attributes. However, it ensures that if there is a processing component that fills the contextual attributes, the system will eventually deploy the query on that processing component (i.e., after the time necessary to process the node's connection and updated contextual attributes).

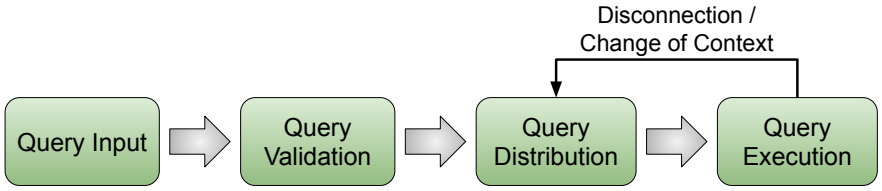

Figure 4.2: Query deployment sequence.

4.3 Context-based Message Distribution Layer

Our model includes a layer for the selective delivery of any message based on the nodes' contextual attributes to enable context-based query deployment. Figure 4.3 depicts this layer. It allows the definition of target contexts. **Target contexts** delineate a set of required contextual attributes, and any node that fills these requirements will match the context. The **nodes** send context update messages with their contextual attributes to a **server module**. The server module is then responsible for detecting when a node matches a context and storing the matches on a **matching table**. In short, this layer receives context-addressed messages as **input** and forwards them to the nodes that match the context according to an automatically updated matching table.

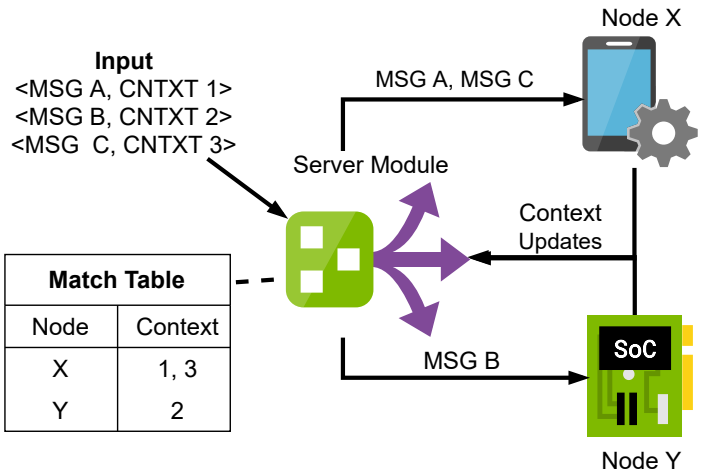

Figure 4.3: A general view of the Context-based Message Distribution Layer.

PUC-Rio - Certificação Digital Nº 1820993/CA

4.4 The core component

The core component is responsible for offering a system management interface while also handling the distribution of queries and events. Figure 4.4 contains the representation of the core component and its modules.

Our model 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 assigning or unassigning them to processing components or contexts.

The **validation module** receives user requests (queries and assignments) from the input interface and verifies them before they are registered or executed. Chapter 6 discusses the validation process in more detail. The assigned queries are encapsulated in context-addressed messages or messages directly addressed to a specific processing component. These messages then go to the connection module. The **connection module** is the core component's link with the context-based message distribution layer. It communicates with the processing components. It also reports connections and disconnections of processing components to the system status monitor. The **system status monitor** is responsible for ensuring the consistency between the queries allocated to a processing node and the queries executing on the processing node. For instance, when a processing component connects/reconnects, the system status monitor makes sure that it has the latest version of every query allocated to it.

PUC-Rio - Certificação Digital Nº 1820993/CA

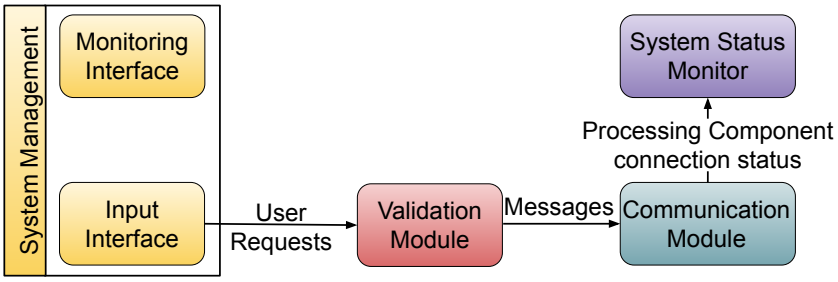

Figure 4.4: The Core Component.

4.5 The Processing Component

We propose a general processing component model that can have specific implementations for heterogeneous devices. This model is pictured in Figure 4.5.

Like in the core component, the **connection module** is a link with the context-based message distribution layer and handles all external communication. When it receives a message, the message is forwarded to the **message manager**. The manager 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. In our model, the messages containing a query also contain some metadata indicating the destination of the events that the query outputs; Namely, if they should be fed back to the local CEP engine and/or distributed to other (remote) processing components.

If the message indicates that the processing component should distribute the output events, the message manager will subscribe to the instantiated query. The generated events are then encapsulated in messages, and the message manager requests the communication module to transmit them. Suppose the message containing a query indicates that the processing component should handle the output locally. In that case, 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. Applications developed using our model can implement the event handler interface to obtain the output of data processing at the local processing component. Some examples would be using this data for actuation, storing the data on a database, or displaying the data on a dashboard.

The data input occurs in the **local event source** of processing components. The local event sources generate events and feed them 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 believe it is better to always pre-process the events locally before distributing them. This pre-processing can reduce network band consumption and even

PUC-Rio - Certificação Digital Nº 1820993/CA

save energy(15).

The last module of the processing component is the **context monitor**. It monitors the context attributes, issuing context updates when necessary. The context updates are sent to the message manager that sends them as a message directed to the server module of the context-based message distribution layer.

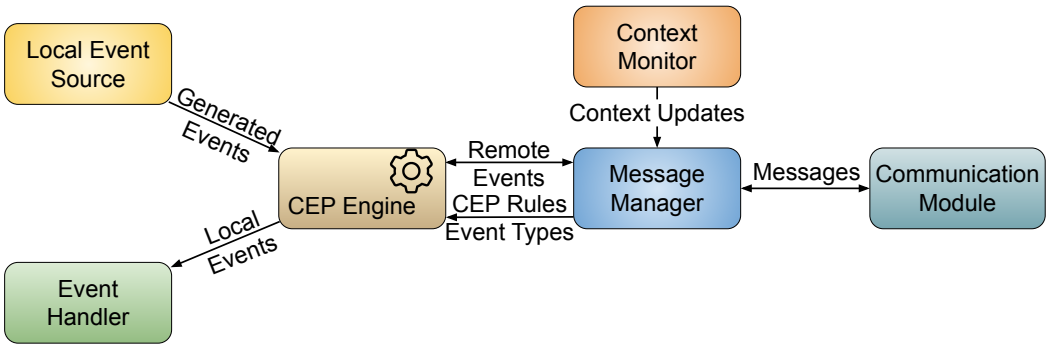

Figure 4.5: The processing component.

PUC-Rio - Certificação Digital Nº 1820993/CA

5
Implementation

This chapter presents details of how we implemented the approach proposed in the previous chapter. First, it discusses the ContextMatcher, a context-aware distribution module, and then the Global CEP Manager, our distributed CEP platform. Our implementation uses Java 1.8 and a PostgreSQL 12 database with the PostGIS extension and custom stored procedures.

5.1
ContextMatcher

5.1.1
Target Contexts

A target context defines a set of preconditions that client nodes need to satisfy to receive context-addressed messages. Target contexts are composed of one or more contextual attributes that can be:

- **Tier:** An IoMT tier that can be Cloud, Static Edge, or Mobile Edge;
- **Processing Power:** An integer that represents the node's minimum processing power. The user/application defines the processing power value of each device. For instance, the user may define a processing power of 3 for a particular smartphone and 6 for a particular desktop computer;
- **Battery Level:** A minimum or a maximum percentage of remaining battery;
- **Location:** An area where the nodes need be located to match the target context. It is expressed as geographic coordinates and a radius (in meters);
- **Sensor List:** A set of sensors that needs to be available at the node (e.g., temperature, humidity, accelerometer)

After the registration of a target context at the ContextMatcher, it is possible to address messages to that target context. In this sense, a target context is

PUC-Rio - Certificação Digital Nº 1820993/CA

similar to a topic in pub/sub communication[1]. However, while in the pub-sub mechanism, client nodes need to explicitly subscribe to topics; using the ContextMatcher, client nodes are automatically matched to a target context whenever they fill the target context preconditions (tier, location, etc).

5.1.2 ContextMatcher Client Nodes

Client nodes are context-aware nodes that report their contextual attributes to the ContextMatcher module and, therefore, can receive context-addressed messages. Upon connecting, the client nodes send a connection report containing their id and their static context attributes: the nodes' IoMT tier and processing power. The other contextual attributes are reported whenever there is some change: When the client node moves, it produces location updates; When the node connects to, or disconnects from, a sensor, it sends an update of its sensor list. When the node's battery drops below some level, or is being charged, it produces battery level updates.

The primary example of a client node with context reporting capacity is the M-Hub. The original M-Hub middleware (3) already provided updates of the battery level and of the location. We added new features to the original M-Hub project so it would also report the other contextual attributes.

5.1.3 ContextMacher Messages

The messages sent using the ContextMatcher have some attributes or meta-data that define how they will be distributed.

5.1.3.1 Multicast and Unicast

Messages addressed to a target context can be multicast or unicast. In the first case, all the nodes that match the target context will receive the multicast message. In the second case, ContextMatcher only sends unicast messages to a single node that matches the target context. Hence, the ContextMatcher randomly chooses one of the nodes that match a target context to receive all unicast messages, referred to as the **unicast node** for that target context. If the chosen node disconnects or unmatches the target context, a new unicast node is randomly selected. When no nodes match the target context, the first node that matches will be the unicast node.

[1]Publish-subscribe is a communication approach that decouples message producers and consumers. Consumers subscribe to a topic to receive the messages published on the topic while producers publish messages on the topic

PUC-Rio - Certificação Digital Nº 1820993/CA

In the use case proposed by this work of distributing queries, unicast can be valuable to avoid data duplicates. Imagine there is a continuous query Q_1 that produces `TemperatureRaise` events when the temperature surpasses a certain threshold. We assign that query to a target context that requests a temperature sensor and to be located in a particular area. There is also a continuous query Q_2 that counts the `TemperatureRaise` events each day produces a report. We assign this second rule to a specific cloud node. If a new node with a temperature sensor enters the area of Q_1, there will be 2 `TemperatureRaise` events for each real temperature rise, causing Q_2 to produce reports with wrong information.

Another usage of unicast messages, outside of query allocation, can be the following. Imagine a home with two senior parents that sometimes receive visits from their adult children. Both parents have smartphones that can send commands to the house garage door. Like in many other garage doors, the same command is used to open or close the door, depending on the door state. Their children's smartphones cannot control the garage door. However, they can send context-addressed messages (using a location target context) to the parents' smartphones requesting to open/close the door. If both parents are home, the command will be sent twice, possibly causing the door to open and quickly close back.

PUC-Rio - Certificação Digital Nº 1820993/CA

5.1.3.2 Retained Messages (*OnMatch* / *OnUnmatch*)

We refer to the regular messages that ContextMatcher just sends right away to matching nodes as instant messages. Besides instant messages, the ContextMatcher also supports retained messages. Retained messages are saved in the ContextMatcher DB for posterior re-sending and can be of two kinds: *onMatch* or *onUnMatch*. When assigned, *onMatch* messages are both stored and sent right away as if they were instant messages. Because they are stored, eventually, when a client node matches a target context, it will receive all the target context's multicast *onMatch* messages. Likewise, when the node unmatches the target context, it will receive all the target context's multicast *onUnMatch* messages. However, unlike *onMatch* messages, *onUnMatch* messages are not sent right away to any node. They are only sent when a unmatch occurs. Retained messages can also be unicast. Nevertheless, since unicast messages are addressed only to the unicast node, the system only sends unicast *onMatch* messages when it selects a new unicast node. Also, the system sends only sends unicast *onUnMatch* messages when the unicast node unmatches the target context. It is also possible to define retained messages directly addressed

to a client node. The system sends those messages when the node connects.

Again in the subject of context-based query distribution, if we wish to deploy queries to client nodes that match a target context and remove the queries when the nodes unmatch the context, we could use an *onMatch* message with the query and an *onUnMatch* that requests the removal of the query.

As an example of usability of retained messages outside of the query distribution scope, imagine electric cars as client nodes. Then we could create a target context for nodes with battery charges under 20% located near a charging station. An *onMatch* message could request the car to display on the dashboard a note suggesting a stop in the charging station. An *onUnMatch* message could request the removal of the note.

5.1.3.3 Message Priority

Context updates and node re-connections can trigger the sending of multiple retained messages to the same node. In those cases, it may be essential to deliver some messages first. For instance, before implementing a query, a node first needs to know the event type of the events that the query takes as input. That is similar to how on a database, it is only possible to run `SELECT` queries on tables that are already created. Therefore we wanted to enable delivering messages containing event types before delivering messages containing continuous queries. Ordered delivery of messages could also have other uses outside of this work's query delivery scope. For example, in a hypothetical use case where client nodes can receive over-the-air software updates, messages caring critical software updates could be delivered first.

To tackle this kind of situation, ContextMatcher offers message priority for retained messages. When registering a retained message, the priority is a simple 4 bytes integer. When the system has to deliver multiple messages to the same node, it sends first the messages with higher priority.

5.1.4 ContextMatcher Server Module

Besides the methods that the client nodes need to implement to report their contextual status and receive context-addressed messages, the ContextMatcher Server Module concentrates all operations of the ContextMatcher. This module can be incorporated into a ContextNet Core application, handling the communication with client nodes. It is also possible to deploy the Server Module as a standalone service accessible through a REST API.

PUC-Rio - Certificação Digital Nº 1820993/CA

Figure 5.1 presents the main components of the ContextMatcher Server Module. The API offers methods to declare new target contexts and send or register context-addressed or directly addressed messages. The database stores the target contexts, the retained messages, and the information about the client nodes (i.e., id, contextual attributes, and connection status). It also stores the current matches of nodes and target contexts. Besides storing data in the DB, the ContextMatcher Server Module also installs a series of stored procedures in the database. Stored procedures are new functions that can be programmed in the database using a procedural language (e.g., PL/pgSQL[2]). These stored procedures transfer some of the data operations performed by the ContextMatcher from the Java software to the database. For instance, automatic triggers on the database perform the matches of client nodes and target contexts. We explain why we decided to use stored procedures in Subsection 5.1.5.

The client message handler receives the messages from the client nodes. The application using the ContextMatcher Server Module can listen to any messages received in the client message handler using the Java API (that option is not available in the REST API). Besides that, the client message handler will always handle the messages with connection reports or context updates, using the information extracted from those messages as parameters for calls to the database's stored procedures. These calls can trigger several operations. Client connection reports from nodes not registered in the database will cause the addition of the node to the database. Both the connection reports and context updates may trigger context matches and unmatches. As a result of matches and unmatches, the database will return a list with retained messages to be sent to the node. They can be messages directly addressed to the nodes, *onMatch* messages to the target contexts that the node matched, or *onUnMatch* messages to the target contexts that the node unmatched. If the client node matches a target context that had no other matches, the message list will also include the target context's unicast *onMatch* messages. Suppose the node is a target context's unicast node, and the node unmatches the target context. Besides including the target context's unicast *onUnMatch* messages in the list, the database will select one of the other nodes (if there is any) that match the target context as the new target context's unicast node. In that case, the database will return an additional list of messages addressed to the new unicast node and containing the target context's unicast *onMatch* messages. Besides the application using the ContextMatcher API, client nodes

[2]PL/pgSQL is the procedural language used to create stored procedures in PostgreSQL databases. `https://www.postgresql.org/docs/12/plpgsql-overview.html`

PUC-Rio - Certificação Digital Nº 1820993/CA

can also send context-addressed messages. These are delivered to the necessary nodes by the message forwarder after getting the list of nodes that match the target context from the database. The message forwarder is a simple module that receives messages with a list of recipients and delivers the messages using the ContextNet middleware.

The ContextMacher also takes advantage that the ContextNet middleware enables listening to client node disconnections. When a client node disconnects, the node's contextual attributes and matches are deleted from the database. Also, if the node was the unicast node for any target context, the ContextMatcher will try to select a new unicast node.

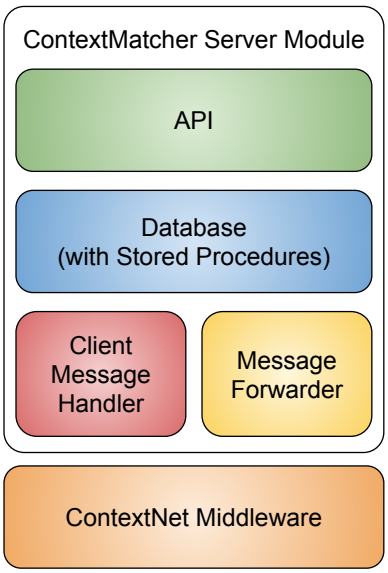

Figure 5.1: The ContextMatcher Server Module.

PUC-Rio - Certificação Digital Nº 1820993/CA

5.1.5 Justificaton for Stored Procedures

Given that the database stores all the information of target contexts and client nodes, we decided to delegate a great portion of the contextual data handling to the DB as stored procedures instead of concentrating it all in the Java software. To perform all the operations in Java, it would be necessary to load a lot of the stored information in the Java software to process and then update the data stored in the database. This loading could potentially represent a significant overhead. For instance, with every new client node context update, it would be necessary to load the data of all target contexts to check for matches and unmatches. Then the database would be updated based on the new matches and unmatches. Matches and unmatches would also mean that other database accesses would be required to

get the necessary *onMatch* and on *UnMatch* retained messages. By using stored procedures, when a context update is received, the Java portion only performs one procedure call instead of multiple database requests. This procedure call performs triggers all the necessary operations, ultimately returning the same retained messages.

5.2 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 as the GCM Core, a service that runs in a ContexNet core node. We developed distinct implementations of the processing component for different devices; we call them Processing Agents (PA). We developed a cloud PA for regular computers or servers and a stationary edge PA for the Raspberry Pi. We also adapted the M-Hub and the MEPA service to make it compatible with GCM as a mobile PA. Every PA is a ContextMatcher client node as well.

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.
- An optional **target context**. The GCM dispatches the events outputted by the query as instant context-addressed messages to the target context.
- A boolean attribute named `consumeLocally` indicating if the events outputted by the query should be fed back to the CEP engine of the local PA.
- 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.
- A boolean attribute indicating if the query contains a **high priority statement**. High priority statements are queries that do not actually perform event processing. They contain statements that create structures necessary for other queries, for instance, statements that create variables, views, etc. GCM deliveries queries containing high priority statements before regular queries by using the message priority offered by the ContextMatcher. Event types do not use high priority statements; they

PUC-Rio - Certificação Digital Nº 1820993/CA

have their particular message structure with an even higher priority in the delivery.

5.2.1 GCM Core

The Global CEP Manager Core acts as the core component we defined in subsection 4.4. The Figure 5.2 displays the architecture of the GCM 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 continuous queries or allocate them to PAs and methods to check their status. If the API receives a new query, it forwards the query to the validator. The **validator** implements the validation module. It checks if the queries are syntactically correct in Esper EPL, if they process existing event streams, and assists in maintaining the mutual consistency of continuous queries. The validation process is better detailed in Chapter 6. If the validation fails, the validator produces a syntax error. If the validation is successful, the validator generates three messages and register them on the ContextMatcher:

- A message with the event type of the events that will be output by the query. First, the validator generates the event type using the items that the query selects as attributes and *EvtOf<query label>* as the event type label. Then the event type is registered as a retained message on the ContextMatcher with priority 3. If the query's meta-information includes a target context, this message is designated as a multicast *onMatch* message addressed to the target context
- A message with the query itself. It will contain both the query and the query's meta-information. The message's priority depends if the query is a high priority statements. If it is a regular query, the priority will be 1; if it is a high priority statement, the priority will be 2. After registration, the message is not designated to any target context nor processing agent since the system administrator needs to assign the query.
- A message that requests the removal of the query. These messages will always have priority 1. Also, similar to the messages with queries, the validator does not designate the message to any target context nor processing agent until the query's assignment.

So, the registered messages obey the following order from highest to lowest priority: event types, high priority statements, regular queries and query removal.

PUC-Rio - Certificação Digital Nº 1820993/CA

If the GCM Core API receives a query assignment, the validator also validates the assignment. Suppose the query is not assigned to a target context but directly assigned to a PA. In that case, it needs to be a PA previously registered. Since PAs are ContextMatcher client nodes, their registration occurs automatically when they first connect. Nevertheless, the GCM API also offers methods that allow the system administrator to register PAs manually. The validator also prevents the assignment of a query both as unicast and multicast to the same context. After the assignment validation, if the query was assigned to a target context, the query's message will be designated as an *onMatch* message addressed to the target context. The validator also registers the message requesting the query's removal as *onUnMatch* for the target context. That way, the ContextMatcher will automatically send the query to any PA that matches the target context. Also, if the PA unmatches the target context, the ContextMatcher will automatically send to the PA the message requesting the query's removal. Whenever a query is directly assigned, the validator registers it as a direct retained message, and the ContextMatcher will send the query as soon as the node connects. The **query set comparator** uses the ContextMatcher's API to listen to the client messages. If the message contains a connection report, the comparator issues a direct instant message to the connected PA requesting the set of queries present in the PA. For each query, the PA will send the label and timestamp (version). Then the comparator contrasts the set sent by the PA with the queries that should be present in the PA according to the ContextMatcher's DB. These are the queries directly assigned or the queries assigned to the PA's matched target contexts, including unicast queries if the PA is the target context's unicast node. It will then issue new instant messages to the PA to ensure that the PA has all and only the queries it should have, and they are all on the latest version. Subsequently, the query set comparator will send a last message to notify the PA that its queries are consistent with the queries assigned to it.

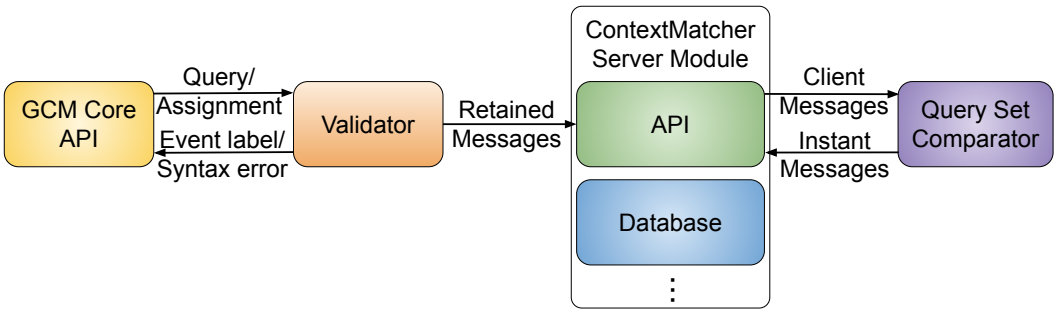

Figure 5.2: The Global CEP Manager Core.

PUC-Rio - Certificação Digital Nº 1820993/CA

5.2.2 Processing Agent (PA)

The Figure 5.3 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** perform the local event source functions and the event handler, respectively. They are Java interfaces the applications using our platform can implement. The **S2PA** service acts as an event input interface in the M-Hub, collecting sensor data and encapsulating in raw events. It is also possible to implement event input interfaces for data from sensors connected to a Raspberry. A possible event input interface implementation in Cloud Processing Agents could be a web crawler that gathers information from web sites and feed as event data. We included an **engine manager** that handles adding and removing continuous queries from the **EsperCEP engine** and feeding it events and collecting the queries' output. The **context reporter** acts as the context monitor, using the device APIs to get contextual information. The M-Hub for instance already offers a location API and a battery monitor API, we added another API to the M-Hub that keeps a list of the connected sensors and pushes every change to this list to the context reporter. We also added a field in the M-Hub's GUI where the user can state processing power for that particular device.

PUC-Rio - Certificação Digital Nº 1820993/CA

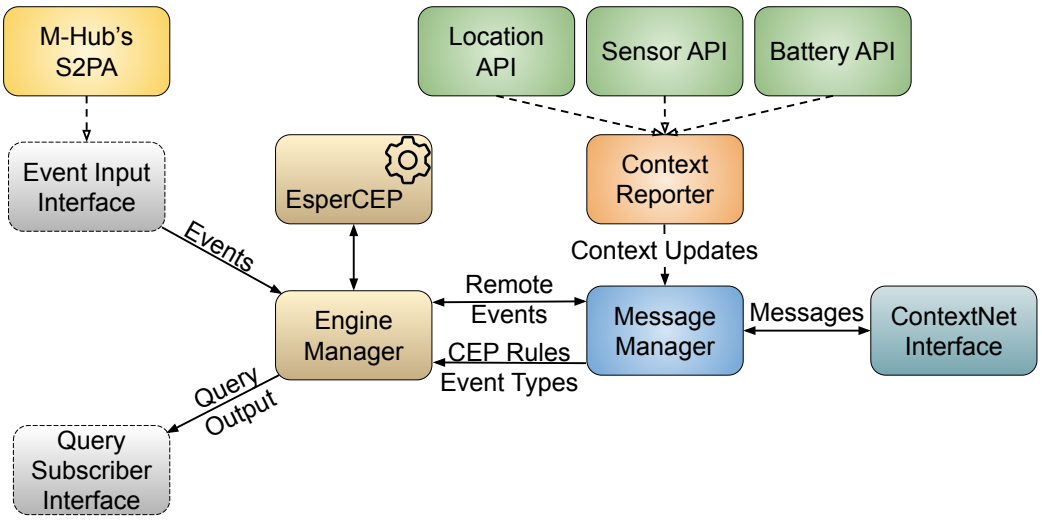

Figure 5.3: The Global CEP Manager Processing Agent.

5.2.3 Offline PA Behavior

When a Processing Agent (PA) disconnects, the GCM Core cannot send any requests to remove queries to that particular PA. It means that before the

PA reconnects and the core can make sure that the PA is running all (and only) the queries assigned to it, the PA may be running queries it should not. The Global CEP Manager API allows the system administrator to establish that when a PA disconnect, it should disable some of its functionality until the GCM Core restores the consistency between the queries assigned to the PA and the queries running on it. It is possible to disable only the emission of external events (events that would be forwarded to other PAs) or completely disabling the CEP engine until the consistency is restored. Upon disconnection, the PA will disable the selected functions until it receives the query set comparator's message that notifies that the consistency has been re-established.

PUC-Rio - Certificação Digital Nº 1820993/CA

6
Validation of Continuous Queries

The GCM includes a validation module to assist in preserving the internal and external consistency of Continuous Queries (CQ) in the EPN before they are deployed to the Processing Agents (PA). This chapter presents in discursive text which consistencies the validation module assures and when validation module actuates to assure them. Also, a formal definition of those consistencies in the form of invariants that are forced on CQs as well as how they are assigned will be presented.

6.1
Basic Concepts and Notations

PUC-Rio - Certificação Digital Nº 1820993/CA

The basic elements and concepts managed by the GCM are the following.

Event Type (E) has:
label/name - *Label*
set of typed Attributes - *Attrib*

Continuous Query (CQ) has:
label/name - *Label*
the query version - *Version*
set of Precedent Event Types - *Precedent*
a Consequent Event Type - *Consequent*

GCM **Processing Agent (PA)** (i.e. a CEP capable node) has:
set of deployed Continuous Queries (CQs) - *DeployedQueries, (or CQ ⊗ PA)*
is (momentarily) in a set of Contexts - *CurrContexts*

Context (Cxt) has:
a set of context attributes as listed in the subsection 5.1.1
a distribution mode (Unicast or Multicast)

GCMCore has:
set of registered CQs - *RegQueries*
set of registered Es - *RegEventTypes*

set of registered Processing Agents - *RegAgents*
set of registered contexts - *RegContexts*
table of CQ assignments to Processing Agents - *AssignedQueries(PA)*
table of CQ assignments to Contexts -*AssignedQueries(C)*

6.2 The invariants about system consistency

The invariant enforcement is related to each query individually (e.g., if it is synthetically correct), and the inter-query consistency, meaning that queries that are dependent on each other must keep playing well together. The corresponding validations occur in the five situations below:

6.2.1 When a new query is submitted

When a new CQ is submitted, the validator performs a series of tests. First it checks if the query respects the syntax of the Esper EPL version 4.8. It also checks if the query consults only *valid* event types where *valid* event types are those previously registered in the GCM Core. The events may be produced by queries or by event sources. Lastly, the validator checks if the query does not try to process an event attribute that is not present in the event type.

Invariants

- $valid(CQ_y) \implies CQ_y \quad syntactically \quad correct \quad EsperEPL \quad v4.8$
- $registered(E_1) \iff E_1 \in GCMCore.RegEventTypes$
- $valid(E_1, A) \iff registered(E_1) \wedge A \in E_1.Attributes$
- $E_x \in CQ_y.Precedent \vee E_x \in CQ_y.Consequent \implies registered(E_x)$
- $E_x.A \wedge E_x \in CQ_y.Precedent \implies valid(E_1, A)$

6.2.2 When a new version of a query is submitted

- Consider that events produced by query CQ_x are consumed by another query CQ_y and a new version of the previous query, say CQ'_x is introduced.
- Then this new version must generate events of a Event Type with the same label, and (at least) all the attributes of the events of the previous version, CQ_x .

PUC-Rio - Certificação Digital Nº 1820993/CA

Invariant

- $E_1 \in CQ_y.Precedent \land E_1 = CQ_x.Consequent \land E_2 = CQ'_x.Consequent \Longrightarrow$
  $name(E_1) = name(E_2) \land label(E_1) = label(E2) \land (\forall A_i \in E_1.Attrib \Longrightarrow A_i \in E_2.Attrib)$

6.2.3 When a query is directly assigned to a Processing Agent

When a continuous query CQ is directly assigned to a GCM Processing Agent, the query and the GCM Processing Agent must have been previously registered in the GCM Core.

Invariants

- $CQ_y \otimes PA_i \Longrightarrow PA_i \in GCMCore.RegAgents \land CQ_y \in GCMCore.RegQueries$
- $\land \forall E_i \in CQ_y.Precedent \lor CQ_y.Consequent \Longrightarrow .registered(E_i)$

6.2.4 When a query is assigned to a context

When a query is assigned to a context, the query and the context must have been registered on the GCM Core. And the query must not have been already registered to the same context in another distribution mode (unicast or multicast).

Invariant

- $CQ_y \otimes Cxt_i \Longrightarrow Cxt_i \in GCMCore.RegContexts \land CQ_y \in GCMCore.RegQueries$
  $\land \neg(CQ_y \; in \; Cxt_i.DistrMode = Unicast \land CQ_y \; in \; Cxt_i.DistrMode = Multicast)$

6.2.5 When a GCM Processing Agent reconnects to the GCM Core

When a GCM Processing Agent disconnects and reconnects, say PA_i, the set of CQ allocated to it may have changed. That is: there may be new queries, there may be new versions of deployed queries or old queries may have been removed from the set of allocated queries.

Hence, when the GCM Processing Agent reconnects to the system, the invariants have to be re-applied (and verified) for this agent and will eventually

PUC-Rio - Certificação Digital Nº 1820993/CA

hold again. Since it may take some time until a reconnected PA will be correctly reintegrated and updated with the CQs, we can say only that eventually the invariant will hold, if the PA stays connected for a sufficient period of time. Operator diamond, $\diamond$, is the "eventually qualifier" used in the formulations below.

Invariants

- PA_i *is connected* $\Longrightarrow$ $\diamond$ $(GCMCore.AssignedQueries(PA_i) = PA_i.DeployedQueries)$
- PA_i *is connected* $\wedge$ $Cxt \in PA_i.CurrContexts \Longrightarrow \diamond (GCMCore.AssignedQueries(Cxt) = PA_i.DeployedQueries)$

PUC-Rio - Certificação Digital Nº 1820993/CA

7
Evaluation

This chapter presents the tests and experiments we executed to evaluate our work. The objective was to demonstrate the applicability and test the performance of the Global CEP Manager (GCM) and the ContextMatcher. Section 7.1, presents a possible use case for the GCM. Section 7.2 presents the real setup we used to implement the use case. Section 7.3 presents the GCM tests and Section 7.4 the ContextMatcher tests.

7.1
Case Study

In this section we demonstrate a smart office use case to evidence a possible application of GCM. Latter, in the performance tests, the same use case will be adopted with some variations. The objective in this application scenario is to automatically turn on and off the air conditioner (AC) in the office rooms based on the current temperature and the presence or not of people in the room.

Figure 7.1 displays the sensors, actuators and devices available. Each room is equipped with a Bluetooth beacon and multiple temperature sensors. The beacon broadcasts the room number every 2 seconds and the temperature sensors dispersed in the room can connect to M-Hubs and provide temperature readings. The M-Hub application can be installed in the office workers smartphones, providing multiple Mobile Edge Processing Agents. The scenario also includes the GCM Core and a Cloud Processing Agent running in the cloud. Lastly, a Raspberry Pi that can actuate as a Stationary Edge Processing Agent is installed in each room, the Raspberries are equipped with a presence sensor and an infrared transmitter both connected to the GPIO ports.

The M-Hubs will automatically connect to the sensors and produce `SensorData` events with the temperature readings. To reduce the effect of the natural noise in sensor readings, the M-Hubs calculate the average temperature of each 3 `SensorData` events producing `EvtOfAvgTemp` events. The M-Hubs will also produce `Beacon` events with the room number when they receive the beacon broadcast. The current room is stored in the variable `var_currRoom`, which is updated after each new `Beacon` event is received, and cleaned (set to

PUC-Rio - Certificação Digital Nº 1820993/CA

-1) if no `Beacon` event is received in 5 seconds. Finally, if the `var_currRoom` has a valid value, the M-Hubs will produce `EvtOfRoomTemp` events with the temperature and the current room after each `EvtOfAvgTemp` event. These `EvtOfRoomTemp` events will be sent to the Cloud Processing Agent. The Cloud Processing Agent gathers all the `EvtOfRoomTemp` events for each room from all M-Hubs. If more than 60% of the events have temperatures above 25°C, it produces `EvtOfRoomTempOver` events which are sent to the Raspberries. The Raspberries store if the AC is on or off in the `var_isACOn`. `Presence` events with the room number will be produced in the Raspberry every 2 seconds if the presence sensor detects presence. If the AC is off and a Raspberry receives an `EvtOfRoomTempOver` within 5 seconds after a `Presence` event, the Raspberry will produce an `EvtOfTurnACOn` event. If the AC is on and there is a 10 minutes window with no Presence events, the Raspberry will produce an `EvtOfTurnACOff` event. The `EvtOfTurnACOn` and `EvtOfTurnACOff` events will trigger an actuation with the IR transmitter and the update of `var_isACOn` value. Listing 7.1 contains the EPL code of continuous queries for the M-Hubs. They were assigned as multicast since they should run on every M-Hub. Listing 7.2 contains the EPL code of continuous queries for the Cloud Processing Agent. They were assigned as unicast since they should aggregate the `EvtOfRoomTemp` events produced by all M-Hubs in a single place. Listing 7.3 contains the EPL code of continuous queries for the Raspberries, they were assigned as multicast since we assume there is only one Raspberry installed in each room.

PUC-Rio - Certificação Digital Nº 1820993/CA

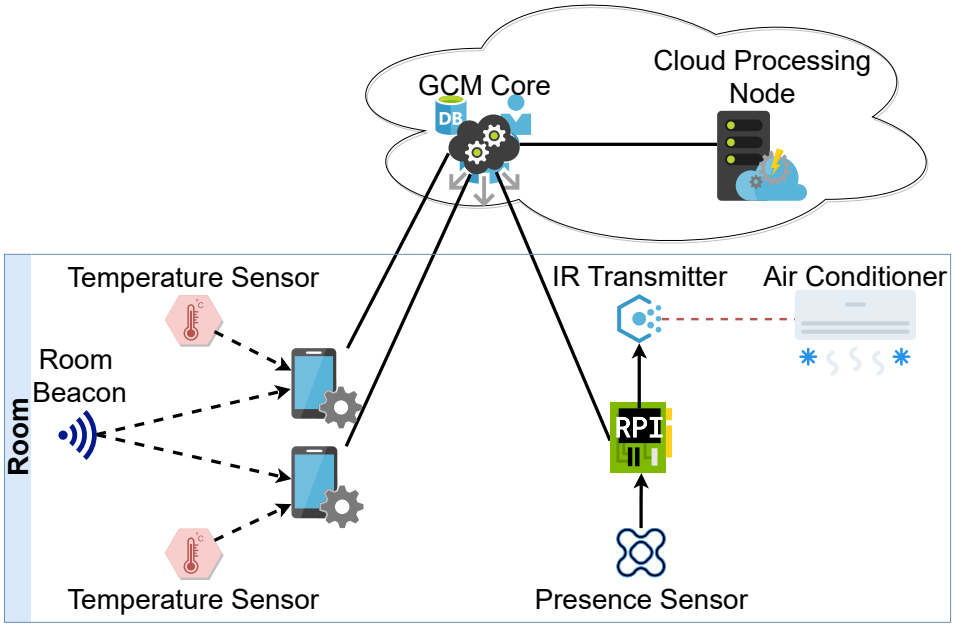

Figure 7.1: Use Case

This use case presents a real-world scenario where our solution could be applied and is useful. By automating turning the AC on and off, it is possible to save energy, and the occupants don't need to manually perform these operations anymore.

Listing 7.1: M-Hub queries.

```
-- Label: createCurrRoom | Create the variable that stores the
    current room.
create variable int var_currRoom = -1

-- Label: setCurrRoom | Sets the current room upon a new Beacon
    event.
on Beacon as b set var_currRoom = b.room

-- Label: unsetCurrRoom | Cleans the current room upon 5 seconds
    without a Beacon event
on pattern[every (timer:interval(5 sec) and not b=Beacon)] set
    var_currRoom = -1

-- Label: AvgTemp | Calculate the average of each 3 SensorData
    events with temperature readings.
select avg(sensorValue[0]) as temp from
    SensorData(sensorName='Temperature').win:length_batch(3)

-- Label: RoomTemp | Suplement the average temperature with the room
    id when available
select var_currRoom as room, a.temp as temp, current_timestamp() as
    timestamp from EvtOfAvgTemp as a where var_currRoom >= 0
```

Listing 7.2: Cloud Processing Agent queries.

```
-- Label: SegmentedByRoom | Enable the segmentation of EvtOfRoomTemp
    events by the room id
create context SegmentedByRoom partition by room from EvtOfRoomTemp

-- Label: RoomTempOver | Each 10 seconds, gathers all the
    EvtOfRoomTemp received for each room. Triggers if more than 60%
    of the events have temperature values are over the threshold
context SegmentedByRoom
select context.key1 as room, count(*, temp>25) as ct_out, count(*)
    as ct_all, current_timestamp() as timestamp
from EvtOfRoomTemp.win:time_batch(10 sec) having count(*,
    temp>25)>=(0.6*count(*))
```

PUC-Rio - Certificação Digital Nº 1820993/CA

Listing 7.3: Raspbery queries.

```
-- Label: createIsAcOn | Create the variable that stores if the AC
    is on.
create variable boolean var_isACOn = false

-- Label: TurnACOff | Turns the AC off if no presence is detected in
    10 minutes
select p.room as room, current_timestamp() as timestamp
from pattern [every (timer:interval(10 minutes) and not p=Presence)]
    where var_isACOn = true

-- Label: TurnACOn | Turns the AC on if presence is detected and
    RoomTempOver is activated within 5 seconds
select p.room as room, current_timestamp() as timestamp
from pattern [every (timer:interval(10 minutes) and not p=Presence)]
    where var_isACOn = true

-- Label: setIsACOnTrue | Sets the var_isACOn as true when a
    EvtOfTurnACOn is received
on EvtOfTurnACOn set var_isACOn=true

-- Label: setIsACOnFalse | Sets the var_isACOn as fasle when a
    EvtOfTurnACOff is received
on EvtOfTurnACOff set var_isACOn=false
```

7.2 Real Test Setup

This subsection will detail the setup we used to test the presented use case. First, we simulated some devices instead of using real hardware. Instead of using real sensors, the smartphones play a synthetic dataset in a loop to produce the `SensorData` events. They also play simulated `Beacon` events, always indicating the same room. We used a virtual presence sensor in the Raspberry, producing Presence events every 2 seconds, and the Turn On and Turn Off commands are printed on the screen instead of sent to an IR transmitter.

Other than this simulated equipment, we used the real devices presented in Table 2. We used both smartphones simultaneously, running the same code and EPL queries. The Dell laptop is responsible for the GCM Core, the ContextNet gateway, and the Cloud Processing Agent, each executing inside

PUC-Rio - Certificação Digital Nº 1820993/CA

its own Docker[1] container using the Windows Subsystem for Linux[2] 2 (WSL 2) backend. We also execute the Stationary Edge Processing Agent on a Docker container; however, this container is hosted in the Raspberry.

Device	Processor	RAM	Operating System
Samsung Galaxy S6	Exynos 7420 4x 1.5GHz + 4x 2.1GHz	3 GB	Android 7.0
Sony Xperia Z2	Snapdragon 801 4x 2.4GHz	3 GB	Android 6.0.1
Dell G7 7588	Intel Core i7-8750H 6x 2.2-4.1GHz	16 GB	Ubuntu 20.04 on Docker over WSL2 Windows 10
Raspberry Pi 4 model B	Broadcom BCM2711 4x 1.5GHz	4 GB	Alphine 3.13 on Docker over Raspbian 10

Table 7.1: List of devices.

7.3 Global CEP Manager Performance Tests

We used the setup presented in the previous sections to execute performance tests of the GCM. We executed multiple tests measuring the CPU usage by each container or the M-Hub modifying the speed of the generation of `SensorData events`. To execute this modification, we used different delays between the playing of entries in the temperature dataset. For each delay, we repeated the test 50 times, running for 10 minutes each time.

Our objective was to measure and compare how each component behaves as we demand the production and processing of more and more temperature events per second. This test can provide indications of the system's scalability and limits. The values of average CPU usage in each test are displayed in Figure 7.2. To acquire the CPU usage of the M-Hubs, we used the Android Profiler[3] available on Android Studio[4]. Therefore the values are presented as a percentage of the full mobile CPU processing capacity. In the DELL laptop and Raspberry, we adopted Docker's CPU usage values, using the `docker stats` command. Therefore, the CPU usage values for the containers are a percentage of a single CPU core capacity. That means that the container running the

[1]Docker is a virtualization platform to run applications inside containers. `https://www.docker.com/`

[2]WSL is a compatibility layer for running Linux executables on Windows. `https://docs.microsoft.com/en-us/windows/wsl/`

[3]Android Profiler is a tool to monitor resource usage of Android applications `https://developer.android.com/studio/profile/android-profiler?hl=en`

[4]Android Studio is an IDE for developing Android applications. `https://developer.android.com/studio?hl=en`

PUC-Rio - Certificação Digital Nº 1820993/CA

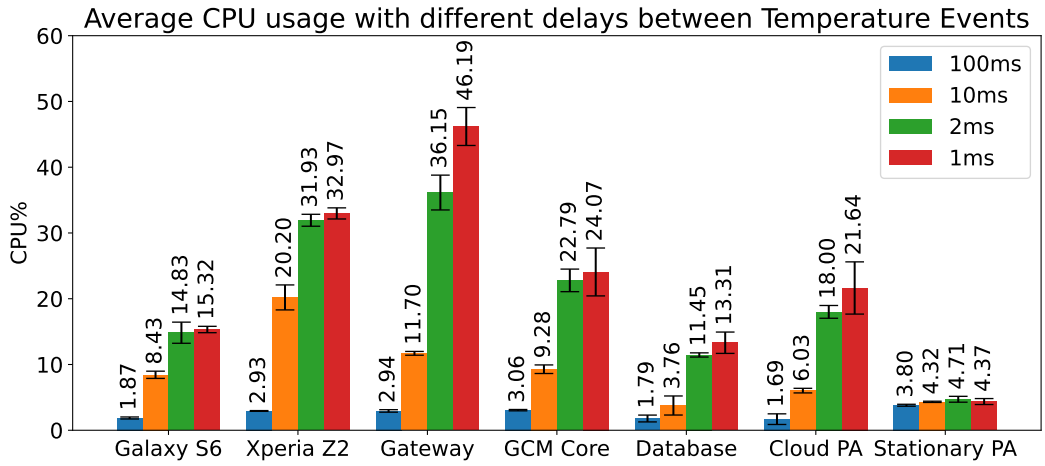

Figure 7.2: Global CEP Manager CPU Usage

Device\Delay	1ms	2ms	10ms	100ms
Galaxy S6	160.85	152.78	33.32	3.33
Xperia Z2	171.06	166.63	33.33	3.33
Expected	333.33	166.66	33.33	3.33

Table 7.2: `EvtOfAvgTemp` produced per second in each test.

gateway used on average 46.19% of a single core of the Dell laptop's CPU in the tests with 1ms delay. In the same test, the M-Hub used on average 15.32% of the total (all cores) processing power offered by the Galaxy S6.

Using the same test parameters, we also measured how many `EvtOfAvgTemp` each android device was able to produce in each case. Table 7.2 contains our measurement of how many events were produced and how many were expected given the delay.

This set of tests evaluate if the system can handle a very fast event output. The results indicate that the system performs well in the tests with delays between 100ms and 2ms, hitting a bottleneck at 1ms. We can notice this bottleneck in Table 7.2 as the values are very close to the expected in every column. Except for the 1ms column that drops significantly below the expected value. We believe that is the limit of how fast the devices can replay the events since this operation is not parallelizable and the CPU usages did not hit close to 100%. Nevertheless, our tests indicate that our system can handle well up to 500 `SensorEvents` per second from each smartphone in this use case, totaling 1000 `SensorEvents` per second.

PUC-Rio - Certificação Digital Nº 1820993/CA

7.4 ContextMatcher Performance Tests

To evaluate the performance of ContextMatcher we tested how long it takes to perform two kind of operations: the match of nodes and target contexts, and the reallocation of queries or messages from one node to another.

7.4.1 Matching Tests

To evaluate the performance of the ContextMatcher module in the process of matching a client node with a target context considering different context requirements, we created 4 target contexts, one for each supported contextual requirement:

1. A target context that requires the Client Node to have humidity, temperature, smoke, radiation and a thermistor sensors available.

2. A target context that requests that the Client Node to be located within 30 meters of the -22.980012557839675, -43.23316465427693 coordinates.

3. A target context that requests that the Client Node is in the Stationary Edge tier;

4. A target context that requests that the Client Node's battery has above 60% charge.

Our objective was to measure how long it would take for the ContextMathcher to match a client node with each target context. A context match occurs when a node sends a message with a context update as we previously explained. The ContextMatcher server module process that message, checking for new matches. To measure how long this process takes, ideally, we would measure the time interval from the moment the node sends the message to the moment the ContexMatcher performs the match. However, since the node and the ContextMatcher sever module would run in different devices that do not share a clock, we decided to use only the clock in the device running the ContextMatcher server module to measure time. To execute this test, we created a method to request the client Node to issue a mock context update that would trigger the match with the target contexts. That way, the computer running the ContextMatcher server module could issue these requests. That said, in our tests, we measured the time interval starting when the request was sent and ending when the ContextMatcher performed the match. That interval also encompasses the time necessary for the client node to receive the request

PUC-Rio - Certificação Digital Nº 1820993/CA

and send the context update. We executed the tests both running the client node and the ContextMatch server module in the same computer and using a different computer connected to the same wi-fi network. That way, we could have an idea of how much the link between the two devices would impact our results. We used the Dell laptop to run the ContextMatch server module and the local client and a Raspberry Pi to run the external client. Table 7.1 lists both devices. In these tests, we did not use Docker containers. We ran the software directly on the host OS, since we did not wanted to measure the CPU usage of each component. We populated the database with 100 dummy Client Nodes, and for each combination of target context and local or external client, we executed the match test 50 times and calculated the averages. Figures 7.3 and 7.4 display the results for the local and external client, respectively.

The results indicate that the ContextMatcher can perform matches of client nodes and target contexts with a delay in the order of milliseconds, even for the node connected via wi-fi. The matches with the target context that requires a set of sensors take significantly longer. That is because the target context's sensors and the client node's sensors are lists of strings, and it takes longer to compare them than to compare the battery, for instance, which is represented as an integer. Also, even though checking if a coordinate is within an area sounds a lot more complex than comparing integers, the location matches take just a little longer than the tier and battery tests. That is because of how optimized the spatial databases (e.g., PostgreSQL with PostGIS) are in calculating spatial comparisons.

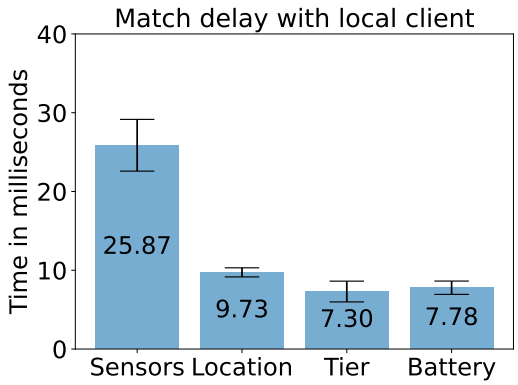

Figure 7.3: Tests with a locally hosted Client Node.

7.4.2 Reallocaction Tests

Our second test of the ContextMatcher measured how long the system would take to reallocate unicast queries. In this test, we used the setup from

PUC-Rio - Certificação Digital Nº 1820993/CA

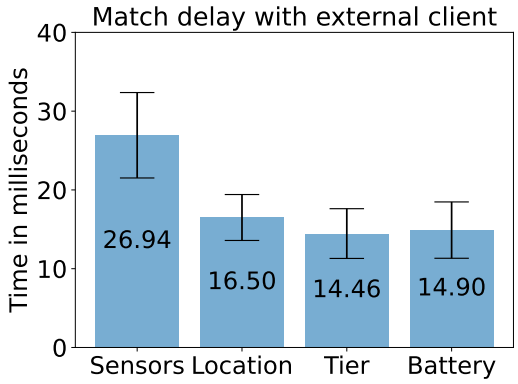

Figure 7.4: Tests with the Client Node running in the Raspberry.

the case study (Sections 7.1 and 7.2). ContextMatcher server module is a component of the GCM Core as depicted in Section 5.2, Figure 5.2. In this test, we assigned the M-Hub queries (Listings 7.1) as unicast instead of multicast. We connected the Samsung smartphone first so the ContextMatcher would allocate the queries for it. Then, after connecting the Sony smartphone and waiting a few seconds, we disconnected the Samsung smartphone so the system would reallocate the queries for the Sony smartphone. We measured the time interval from the moment the disconnection was detected on the GCM Core until the GCM Core received query status reports from the Sony smartphone stating that the queries are active. We repeated this test 50 times. Table 7.3 presents the average delay from detecting the disconnection until the first query status report and until the last query status report.

The query reallocation involves a series of operations, including sending messages, updating the DB allocation table, and the time the Esper CEP engine takes to instantiate new queries. The test results display that those operations take around 100 ms in the environment tested. Depending on how long the ContextNet middleware takes to report the node disconnection, reallocating queries this fast means we can avoid most event losings while the system is reallocating queries.

Metric Delay	Until First Query	Until Last Query
Average (ms)	91	118
Standard Deviation	12.87	14.01

Table 7.3: Delay until the query is relocated.

PUC-Rio - Certificação Digital Nº 1820993/CA

8
Conclusions

This chapter concludes this master thesis, presenting our final remarks, and directions for future works. Also, this chapter lists the contributions of this work and the publication that originated from it.

With the increasing use of the Internet of Things (IoT), more and more stream processing applications and services are deployed for monitoring and controlling real-world activities, people and assets. In these systems, situation-aware sensors and IoT devices embedded in the environment generate distributed, asynchronous and steady streams of heterogeneous data that have to be processed by stream processing technologies, in our case CEP. But a centralized execution of CEP is not ideal for coping with the high volume, velocity, and volatility of data streams from IoT sensors. Instead, the CEP Event Processing Network (EPN) should be distributed, preferably having CEP agents both in the cloud/cluster and in edge devices. The contextual status (e.g., location, processing power, battery level, and available sensors) can be fundamental in planning how to distribute this EPN. That is because these factors can determine which devices are suited to acquire and process the data.

In this regard, this master thesis presents a platform for distributed CEP applications that allows the context-aware deployment of continuous queries on heterogeneous devices in an Internet of Mobile Things (IoMT) environment. It also provides validation of those continuous queries testing if they will function adequately (internally and in conjunction with the other queries) before their deployment. We evaluated this platform using a real-word inspired use case scenario and found out that it achieves the objectives of this work.

8.1
Publications

Our work resulted in the following publication:
MAGALHÃES, F.; SILVA, F. ; ENDLER, M.. **Support for Adaptive and Distributed Deployment of CEP Continuous Queries for the IoMT**. In: ANAIS DO XXXVIII SIMPÓSIO BRASILEIRO DE REDES

PUC-Rio - Certificação Digital Nº 1820993/CA

DECOMPUTADORES E SISTEMAS DISTRIBUÍDOS, p. 99–112, Rio de Janeiro, RJ, Brasil, 2020. SBC.

8.2 Contributions

The main contributions of this work are are as follows:

- We developed the ContextMatcher, a context-based message distribution layer.
- We developed the Global CEP Manager, a platform for the creation of distributed CEP applications that disperse the event processing in cloud and (mobile or stationary) edge devices.
- We propose the deployment of continuous queries based on contextual requirements allowing the dynamic allocation and deallocation of queries to devices based on the device's current context. This represents a novel approach to query distribution strategies. To the best of our knowledge, there is no other work that considers the devices' context when deciding where to allocate each query. Most previous works focus on load balancing. Our implementation and sample use case indicate that our approach is applicable in real-world scenarios.
- We designed a set of validation procedures that precede the query deployment, testing aspects of the queries' internal correctness and inter-query consistency. This validation can help the detection of problems before the execution of the query in a processing node.

8.3 Future Works

This master thesis presented beneficial results. However, there is always space for future work. We believe some of the most productive further developments to this work would be the following:

- **Support for other middleware:** We developed both the ContextMatcher and the GCM based on using the ContextNet middleware as our communication bus. However, we could adapt both for other middleware, for instance, MQTT.
- **Time as a contextual attribute:** Time is an instrumental and far-reaching aspect of context. The target contexts (Subsection 5.1.1) of ContextMatcher could include parameters of relative time (e.g., before, after, during, or within a time frame around an event or happening) and absolute time (e.g., dates, time of day, days of the week).

PUC-Rio - Certificação Digital Nº 1820993/CA

- **Load balancing:** A natural next step in the development of GCM would be to include the load balancing of queries. Many papers on the subject of distributed CEP focus on load balancing. In this work, we opted to focus on other issues. However, we could either design and include a novel load balancing model or incorporate a load balancing model proposed in another work.
- **Further tests:** We could perform other tests to evaluate the efficiency of our work in scenarios with thousands of Processing Agents and constant connections, disconnections, and context changes.

PUC-Rio - Certificação Digital Nº 1820993/CA

Bibliography

[1] CUGOLA, G.; MARGARA, A.. **Processing flows of information: From data stream to complex event processing**. ACM Comput. Surv., 44(3):15:1–15:62, June 2012.

[2] CARNEY, D.; ÇETINTEMEL, U.; CHERNIACK, M.; CONVEY, C.; LEE, S.; SEIDMAN, G.; STONEBRAKER, M.; TATBUL, N. ; ZDONIK, S.. **Monitoring streams: A new class of data management applications**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB '02, p. 215–226, Hong Kong, China, 2002. VLDB Endowment.

[3] RIOS, L. E. T.. **An energy-aware iot gateway, with continuous processing of sensor data**. Master's thesis, PUC-Rio, Rio de Janeiro, Brasil, 2016.

[4] BALAZINSKA, M.; BALAKRISHNAN, H.; MADDEN, S. ; STONEBRAKER, M.. **Fault-tolerance in the borealis distributed stream processing system**. In: PROCEEDINGS OF THE 2005 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD '05, p. 13–24, New York, NY, USA, 2005. ACM.

[5] GOLDIN, D.; SRINIVASA, S. ; SRIKANTI, V.. **Active databases as information systems**. In: PROCEEDINGS. INTERNATIONAL DATABASE ENGINEERING AND APPLICATIONS SYMPOSIUM, 2004. IDEAS '04., p. 123–130, Coimbra, Portugal, 2004.

[6] GEISLER, S.. **Data Stream Management Systems**. In: Kolaitis, P. G.; Lenzerini, M. ; Schweikardt, N., editors, DATA EXCHANGE, INTEGRATION, AND STREAMS, volumen 5 de **Dagstuhl Follow-Ups**, p. 275–304. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[7] DELL'AGLIO, D.; DELLA VALLE, E.; VAN HARMELEN, F. ; BERNSTEIN, A.. **Stream reasoning: A survey and outlook**. Data Science, 1:59–83, 2017.

[8] LUCKHAM, D. C.. **The power of events**, volumen 204. Addison-Wesley Reading, 2002.

PUC-Rio - Certificação Digital Nº 1820993/CA

[9] LUCKHAM, D. C.; FRASCA, B.. **Complex event processing in distributed systems**. Technical report, Stanford University, 1998.

[10] ETZION, O.; NIBLETT, P. ; LUCKHAM, D. C.. **Event processing in action**. Manning Greenwich, 2011.

[11] ENDLER, M.; E SILVA, F. S.. **Past, present and future of the contextnet iomt middleware**. Open Journal of Internet Of Things (OJIOT), 4(1):7–23, 2018.

[12] LUCKHAM, D. C.. **Event Processing for Business**. John Wiley Sons, Ltd, 2012.

[13] PIETZUCH, P. R.; SHAND, B. ; BACON, J.. **A framework for event composition in distributed systems**. In: PROCEEDINGS OF THE ACM/IFIP/USENIX 2003 INTERNATIONAL CONFERENCE ON MIDDLEWARE, Middleware '03, p. 62–82, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[14] JAYASEKARA, S.; KANNANGARA, S.; DAHANAYAKAGE, T.; RANAWAKA, I.; PERERA, S. ; NANAYAKKARA, V.. **Wihidum: Distributed complex event processing**. Journal of Parallel and Distributed Computing, 79-80:42 – 51, 2015. Special Issue on Scalable Systems for Big Data Management and Analytics.

[15] RIOS, L. T.; ENDLER, M. ; COLCHER, S.. **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, May 2016.

[16] STARKS, F.; PLAGEMANN, T. P.. **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), p. 83–90, Oct 2015.

[17] DHILLON, A.; MAJUMDAR, S.; ST-HILAIRE, M. ; EL-HARAKI, A.. **Mcep: A mobile device based complex event processing system for remote healthcare**. In: 2018 IEEE INTERNATIONAL CONFERENCE ON INTERNET OF THINGS (ITHINGS) AND IEEE GREEN COMPUTING AND COMMUNICATIONS (GREENCOM) AND IEEE CYBER, PHYSICAL AND SOCIAL COMPUTING (CPSCOM) AND IEEE SMART DATA (SMARTDATA), p. 203–210, Halifax, Canada, 2018.

PUC-Rio - Certificação Digital Nº 1820993/CA

[18] SHIN, Y.; YOON, S.; TRIRAT, P. ; LEE, J.. **Cep-wizard: Automatic deployment of distributed complex event processing**. In: 2019 IEEE 35TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), p. 2004–2007, Macao, China, 2019.

[19] HASENBURG, J.; BERMBACH, D.. **Towards geo-context aware iot data distribution**. In: Yangui, S.; Bouguettaya, A.; Xue, X.; Faci, N.; Gaaloul, W.; Yu, Q.; Zhou, Z.; Hernandez, N. ; Nakagawa, E. Y., editors, SERVICE-ORIENTED COMPUTING – ICSOC 2019 WORKSHOPS, p. 111–121, Cham, 2020. Springer International Publishing.

PUC-Rio - Certificação Digital Nº 1820993/CA