An Edge Computing-based Complex Event Processing Technique for Sensor-based Systems

by

Amarjit Singh Dhillon

A thesis submitted to the
Faculty of Graduate and Postdoctoral Affairs
in
partial fulfillment of the requirements for the degree of
Masters of Applied Science in Electrical and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
Canada

Abstract

Complex Event Processing (CEP) on sensor-based systems often uses a mobile gateway agent to forward raw sensor data streams to a remote back-end server. Complex events that are triggered by multiple raw events are then detected at the back-end server. This approach relies on a persistent network connection between the back-end server and the mobile device.

This thesis proposes an edge computing-based mobile CEP technique in which CEP is performed on the mobile edge device using an embedded CEP engine and the detected complex events are sent to the back-end server for further processing. A proof-of-concept prototype for this system has been built using a Siddhi CEP engine and a WSO_2 server. A thorough performance analysis is performed for comparing the proposed system with the back-end server-based system. The proposed system can handle intermittent network disconnections and leads to reduced user cost and energy consumption for the mobile device.

Acknowledgments

"No problem can be solved from the same level of consciousness that created it"

—Albert Einstein

The work presented in this dissertation would not have been realized without the assistance of many individuals. I take this opportunity to express my sincere gratitude to everyone who helped me during this journey.

Foremost, I would like to thank my supervisors Dr. Shikharesh Majumdar and Dr. Marc St-Hilaire for their continuous guidance, motivation, and feedback throughout the process. Secondly, I would like to thank my parents for providing me with monetary support and moralistic encouragement throughout my academic achievements.

Many thanks to Ali El-Haraki for discussing various technical challenges concerned with our prototype. I am grateful to TELUS and Natural Sciences and Engineering Research Council of Canada (NSERC) for providing financial support for this research.

A special thank goes to my friend Jasmeet Singh for sharing all the ebbs and flows with me during the past two years. I am indebted to Dr. Parvinder Kaur Sandhu for her unremitting support and continuous enthusiasm. Many thanks to Ranjit Singh Saini, Manpreet Singh Dhillon, and Amrinder Singh for various astute conversations. Solving certain problems would not have been possible without assistance from the *Stack Overflow* community and some technical blog posts.

To my grandparents . . .

Table of Contents

Abstract **ii**

Acknowledgments **iii**

List of Tables **xi**

List of Figures **xv**

List of Algorithms **xvi**

List of Symbols **xvii**

List of Abbreviations **xviii**

1 Introduction **1**

1.1 Motivation for the Thesis 3

1.2 Proposed Technique 4

1.3 Contributions of the Thesis 5

1.4 Scope of the Thesis 6

1.5 List of Publications 7

1.6 Thesis Outline 7

2 Background Information **9**

2.1 Big Data Analytics 9

2.2 What is CEP? . . . 12
2.2.1 DBMS vs. DSMS and SQL vs. CQL . . . 14
2.2.2 SP Systems vs. CEP Engines . . . 16
2.3 Components of CEP Engines . . . 17
2.3.1 Event Sources . . . 18
2.3.2 Event Sinks . . . 19
2.3.3 Source Mappers and Sink Mappers . . . 19
2.3.4 Blocking Queues . . . 19
2.3.5 In-Memory Tables . . . 20
2.3.6 Callbacks . . . 21
2.3.7 CEP Operators . . . 21
2.3.8 Windows . . . 23
2.3.9 Query Execution Plan . . . 24
2.3.10 Event Time-stamping . . . 25
2.4 State-of-the-art CEP engines . . . 26
2.4.1 Apache Flink . . . 26
2.4.2 Apache Siddhi . . . 26
2.5 Pub-Sub Systems . . . 27
2.5.1 MQTT . . . 30
2.6 Cloud and Edge Computing . . . 31
2.7 IoT Server . . . 31
2.7.1 WSO_2 IoT Server . . . 32
2.8 WSO_2 Agent . . . 34
2.9 Android Essentials . . . 34
2.10 Real-Time Dashboards . . . 35

3 Related Work **37**

3.1 CEP and Stream Processing systems 37

3.1.1 Open-Source CEP engines 38

3.1.2 Commercial off-the-shelf (COTS) CEP engines 43

3.1.3 CEP, Cloud, and IoT 44

3.2 CEP for Smart Buildings and Smart Homes 47

3.3 CEP for Remote Patient Monitoring 48

3.4 Concluding Remarks 56

4 System Architecture and Prototype **57**

4.1 Server CEP System 57

4.1.1 Interaction Among Various Components 58

4.1.2 Device Enrollment/Registration Process 60

4.1.3 An Initial Login into SCEP Application 60

4.1.4 Components of the SCEP Application 61

4.1.5 IoT Hospital Server 62

4.1.6 Components of CEP-as-a-Service 63

4.1.7 Performance tuning for ActiveMQ 67

4.1.8 Device Management Dashboard 68

4.1.9 Setting up an Analytics Dashboard 69

4.2 Mobile CEP System 71

4.2.1 Key Features of Mobile CEP 73

4.2.2 Components of the Mobile CEP Application 74

4.2.3 Screenshot of Mobile CEP Application 76

4.2.4 Mobile CEP Algorithm 77

4.2.5 MCEP Sensor Handler Algorithm 78

4.2.6 MCEP MQTT Service Algorithm . . . 80
4.2.7 MCEP Average Network Consumption . . . 82
4.3 System Prototype Implementation . . . 82
4.3.1 Dependencies used in Mobile CEP . . . 85
4.4 Sensor Simulator . . . 85
4.4.1 Sensor Simulator Algorithm . . . 87
4.5 Timekeeper . . . 88
4.5.1 Timekeeper Algorithm . . . 89

5 Performance Analysis **90**
5.1 The Remote Patient Monitoring Use Case Modeling . . . 90
5.2 Various types of Event Time-stamping . . . 92
5.3 Workload and System Parameters . . . 93
5.4 Performance Metrics . . . 95
5.5 Experiments for Server CEP system . . . 101
5.5.1 Impact of T_{run} on RBL_{SCEP} . . . 101
5.5.2 Impact of T_{win} on L_{SCEP} . . . 102
5.5.3 Impact of λ_{RE} on L_{SCEP} . . . 102
5.5.4 Impact of λ_{RE} on Q_{SCEP} . . . 104
5.5.5 Impact of λ_{RE} on E_{SCEP} . . . 104
5.6 Experiments for Mobile CEP system . . . 105
5.6.1 Impact of T_{run} on RBL_{MCEP} . . . 105
5.6.2 Impact of T_{win} on L_{MCEP} . . . 106
5.6.3 Impact of $Count_B$ and Th_B on L_{MCEP} . . . 107
5.6.4 Impact of λ_{RE} on $\mu_{CE-MCEP}$ . . . 108
5.6.5 Impact of λ_{RE} on L_{MCEP} . . . 108

5.6.6 Impact of λ_{RE} on Q_{MCEP} . . . 110
5.6.7 Impact of λ_{RE} on E_{MCEP} . . . 110
5.7 Performance Comparison of SCEP and MCEP Systems . . . 111
5.7.1 Comparison of RBL_{SCEP} and RBL_{MCEP} when T_{run} is increased . . . 111
5.7.2 Comparison of L_{MCEP} and L_{SCEP} . . . 112
5.7.3 Comparison of Q_{MCEP} and Q_{SCEP} . . . 113
5.7.4 Comparison of E_{MCEP} and E_{SCEP} . . . 115
5.7.5 Comparison of $CU_{IHS-MCEP}$ and $CU_{IHS-SCEP}$ . . . 115
5.7.6 Comparison of CU_{SCEPA} and CU_{MCEPA} . . . 116
5.7.7 Comparison of MU_{SCEPA} and MU_{MCEPA} . . . 117
5.7.8 Comparison of UC_{SCEP} and UC_{MCEP} . . . 118

6 Conclusions and Future Work **120**
6.1 Synopsis of the Proposed Research . . . 120
6.2 Characteristics of SCEP and MCEP systems . . . 121
6.3 Performance Comparison of SCEP and MCEP systems . . . 123
6.4 Future Work . . . 124

Appendices **126**

A Dependencies & Algorithms **126**
A.1 Dependencies . . . 126
A.2 Algorithms . . . 127

B Sensor Simulator Algorithm **128**
B.1 $Sensor_X$ Generator Algorithm . . . 128

C Timekeeper Algorithms **131**

C.1 $Sensor_X$ Listener Algorithm 131

C.2 $DequeThreadSensor_X$ Algorithm 133

C.3 *ComplexEventListener* Algorithm 133

References **135**

List of Tables

Table 2.1 List of various CEP operators given in [29], [50], [51] and [52] 22
Table 3.1 Type of operators supported in various systems 45
Table 4.1 Java Memory Configurations 84
Table 4.2 A sample tuple generated by the sensor simulator 86
Table 4.3 Tuple format 86
Table 5.1 CEP Query 91
Table 5.2 Various times associated with an event 92
Table 5.3 Workload and system parameters 95
Table A.1 Dependencies for the MCEP application 126

List of Figures

Figure 2.1 Four dimensions of the big data [12] 11
Figure 2.2 Spark streaming architecture [21] 12
Figure 2.3 A data stream 13
Figure 2.4 Example of a query tree 14
Figure 2.5 (a): A DBMS [25] (b): A DSMS [25] 15
Figure 2.6 SP and CEP systems [22] 17
Figure 2.7 A storm topology [33] 17
Figure 2.8 Overview of the CEP architecture [35] 18
Figure 2.9 A blocking queue [45] 20
Figure 2.10 Data stream and relation conversion concept proposed in [2] 20
Figure 2.11 Classification of windows [53] 23
Figure 2.12 A query tree 24
Figure 2.13 Automata for ABC [54] 25
Figure 2.14 Automata for AB+C [54] 25
Figure 2.15 Siddhi CEP architecture [58] 27
Figure 2.16 Example of a Pub-Sub system [61] 28
Figure 2.17 (a) QoS_0 [63] (b) QoS_1 [63] (c) QoS_2 [63] 29
Figure 2.18 Key components of the WSO_2 IoT server [71] 32
Figure 2.19 Java 8 language feature support [77] 35
Figure 3.1 Evolution of CEP systems [5] 39
Figure 3.2 The architecture of CEPSim [111] 44

Figure 3.3 The architecture of a smart home system as proposed in [111] 47
Figure 3.4 System architecture for RPM system [130] 49
Figure 3.5 System architecture for RPM system [135] 51
Figure 3.6 A simplified view of a system architecture used in [141] 53
Figure 3.7 An outdoor experiment conducted in [141] 54
Figure 3.8 System architecture used in [151] and [152] 55
Figure 4.1 Server CEP system architecture 58
Figure 4.2 Sequence diagram showing a high-level interaction of the components in the SCEP system 59
Figure 4.3 (a) Registration page (b) Accepting policy agreement (c) Setting secure PIN 61
Figure 4.4 Various components of the SCEP application 62
Figure 4.5 Key components of IHS 62
Figure 4.6 Components of the CEPaaS module deployed on the IoT server . . . 63
Figure 4.7 A screen-shot of the web GUI for ActiveMQ 65
Figure 4.8 Thrift server 66
Figure 4.9 Administrator dashboard for a particular user 69
Figure 4.10 Dashboard setup 69
Figure 4.11 Screen-shot on the DataDog dashboard running at the IHS 70
Figure 4.12 Mobile CEP system Architecture 71
Figure 4.13 The sequence diagram for mobile CEP system 72
Figure 4.14 Components of mobile CEP application 74
Figure 4.15 Screenshot of mobile CEP application 76
Figure 4.16 Data sent and received by the mobile device 82
Figure 4.17 System prototype setup 84

Figure 5.1 Time-stamping in MCEP 93
Figure 5.2 Time-stamping in SCEP 93
Figure 5.3 CEP specific metrics 97
Figure 5.4 Impact of application usage time on the battery consumption in SCEP system 101
Figure 5.5 Impact of time window length on the average CEP latency in SCEP system 103
Figure 5.6 Impact of raw event arrival rate on the average CEP latency in SCEP system 103
Figure 5.7 Impact of λ_{RE} on the average complex event queuing delay in SCEP system 104
Figure 5.8 Impact of λ_{RE} on the average end-to-end delay in SCEP system . . . 105
Figure 5.9 Impact of experiment runtime on the battery power in MCEP system 106
Figure 5.10 Impact of time window on the average CEP latency in MCEP system 107
Figure 5.11 Impact of Th_B and $Count_B$ on average CEP latency in MCEP system 108
Figure 5.12 Impact of raw event arrival rate on $\mu_{CE-MCEP}$ 109
Figure 5.13 Impact of raw event arrival rate on average CEP latency in MCEP system 109
Figure 5.14 Impact of raw event arrival rate on average queuing latency in MCEP system 110
Figure 5.15 Impact of λ_{RE} on average end-to-end latency in MCEP system 111
Figure 5.16 Impact of runtime on battery usage in MCEP and SCEP systems . . . 112
Figure 5.17 Comparison of average CEP latency in SCEP and MCEP systems . . 113
Figure 5.18 Impact of average raw event arrival rate on Q_{MCEP} and Q_{SCEP} 114
Figure 5.19 Impact of average raw event arrival rate on E_{MCEP} and E_{SCEP} 115

Figure 5.20 Impact of average raw event arrival rate on the IHS CPU utilization 116
Figure 5.21 Impact of average raw event arrival rate on CU_{SCEPA} and CU_{MCEPA} . 117
Figure 5.22 Impact of average raw event arrival rate on MU_{SCEPA} and MU_{MCEPA} 118
Figure 5.23 Impact of average raw event arrival rate on UC_{SCEP} and UC_{MCEP} . . 119

List of Algorithms

Algorithm 4.1 Mobile CEP Algorithm 77
Algorithm 4.2 MCEP Sensor Handler Algorithm 79
Algorithm 4.3 MQTT Service Algorithm 81
Algorithm 4.4 Sensor Simulator Algorithm 87
Algorithm 4.5 Timekeeper Algorithm 88
Algorithm A.1 Algorithm for computing average CPU utilization in MCEP . . . 127
Algorithm A.2 Algorithm for computing average memory usage in MCEP 127
Algorithm B.1 $Sensor_X$ Generator Algorithm 129
Algorithm C.1 $Sensor_X$ Listener Algorithm 132
Algorithm C.2 $DequeThreadSensor_X$ Algorithm 133
Algorithm C.3 *ComplexEventListener* Algorithm 134

List of Symbols

Symbol	Description	Unit
L	Average CEP latency	Milliseconds
Q	Average queuing latency	Milliseconds
E	Average end-to-end latency	Milliseconds
λ	Average arrival rate	Events/second
μ_{CE}	Average complex event throughput	Events/second
γ	Selectivity	-
CU	Average CPU utilization	%
MU	Average memory usage	MB
BU	Average battery usage	%
RX	Average received bytes/sec	MBps
TX	Average transmitted bytes/sec	MBps
T_g	Event generation time	-
T_{gg}	Event global generation time	-
T_a	Event arrival time	-
T_i	Event ingestion time	-
T_d	Event detection time	-
T_n	Event notification time	-
T_{gn}	Event global notification time	-

List of Abbreviations

ADB	Android Debug Bridge
AES	Advanced Encryption Standard
AGeCEP	Attributed Graph Rewriting for Complex Event Processing
AI	Artificial Intelligence
AJAX	Asynchronous JavaScript and XML
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
APK	Android Package Kit
APNS	Apple Push Notification Service
AWS	Amazon Web Services
BAM	Business Activity Monitoring
BG	Background
BIH	Boston's Beth Israel Hospital
BLE	Bluetooth Low Energy
BP	Blood Pressure
BPM	Business Process Management
BYOD	Bring Your Own Device
CE	Complex Event
CEL	Cayuga Event Language

CEP	Complex Event Processing
CEP4HFP	Complex Event Processing for Heart Failure Prediction
CEPaaS	Complex Event Processing as-a-Service
CHF	Congestive Heart Failure
CMS	Container Management System
COPD	Chronic Obstructive Pulmonary Disease
COPE	Corporate Owned Personally Enabled
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
CQL	Continuous Query Language
CRHMS	CEP-based Remote Health Monitoring System
CSA	Connected Streaming Analytics
CSV	Comma Separated Values
DAG	Distributed Acyclic Graph
DAHP	Data Active Human Passive
DAS	Data Analytics Server
DBA	Data Bridging Agent
dBm	Decibel milliwatts
DBMS	Database Management System
DML	Data Manipulation Language
DSMS	Data Stream Management System
E2E	End-to-End

EC Elastic Compute

ECA Event Condition Action

ECG Electrocardiogram

EEG Electroencephalogram

EHR Electronic Health Record

ELK ElasticCompute Logstash Kibana

ELS Event Listening Service

EMR Amazon Elastic MapReduce

EPL Event Processing Language

ESB Enterprise Service Bus

FCM Firebase Cloud Messaging

FG Foreground

FIFO First-In-First-Out

FSM Finite State Machine

GB GigaByte

GCE Google Compute Engine

GCM Google Cloud Messaging

GCP Google Cloud Platform

GPRS General Packet Radio Services

GPS Global Positioning System

GUI Graphical User Interface

HA High Availability

HADP	Human Active Data Passive
HD	Hard Drive
HL7	Health Level-7
HR	Heart Rate
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HVAC	Heating Ventilation and Air Conditioning
IaaS	Infrastructure-as-a-Service
IBM	International Business Machines Corporation
IDC	Interactive Data Corporation
IE	Intermediate Event
IHS	IoT Hospital Server
IMEI	International Mobile Equipment Identity
IoT	Internet of Things
IP	Internet Protocol
JMS	Java Message Service
JMX	Java Management eXtensions
JNDI	Java Naming and Directory Interface
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KB	KiloByte
LoRaWAN	Long Range Wide Area Network

LTS Long Term Support

LWT Last Will and Testament

M2M Machine-to-Machine

MB MegaByte

MBeans Managed Beans

MCEP Mobile Complex Event Processing

MCEPA MCEP Application

MD Mobile Device

MEC Mobile Edge Computing

MIT Massachusetts Institute of Technology

MOM Message Oriented Middleware

MQTT Message Queuing Telemetry Transport

MVC Model View Controller

NFA Non-deterministic Finite Automata

NSERC Natural Sciences and Engineering Research Council of Canada

OAuth Open Authorization

OS Operating System

OSGi Open Service Gateway Initiative

PaaS Platform-as-a-Service

PHR Personal Health Record

POJO Plain Old Java Object

QoS Quality of Service

RAM Random Access Memory

RBAC Role Based Access Control

RBL Remaining Battery Life

RDD Resilient Distributed Dataset

RE Raw Event

REST REpresentational State Transfer

RFID Radio Frequency Identification

RMD Remote Monitoring Device

RPC Remote Procedure Call

RPM Remote Patient Monitoring

RPTM Remote Patient Tracking and Monitoring

RR Respiration Rate

RTT Round Trip Time

SaaS Software-as-a-Service

SASE Stream-based And Shared Event processing

SCEP Server Complex Event Processing

SCEPA SCEP Application

SD Secure Digital

SLA Software Level Agreement

SNR Signal to Noise Ratio

SOA Service Oriented Architecture

SP Stream Processing

SpO2 Arterial Oxygen Saturation

SQL Structured Query Language

SQuAl Stream Query Algebra

SS Sensor Simulator

SSD Solid State Drive

SSL Secure Socket Layer

STOMP Streaming Text Oriented Message Protocol

TB TeraByte

TCP Transmission Control Protocol

TELUS Telus Corporation

TIBCO The Information Bus COmpany

TLS Transport Level Security

UDP User Datagram Protocol

UID Unique ID

URI Uniform Resource Identifier

URL Uniform Resource Locator

USB Universal Serial Bus

VM Virtual Machine

VPN Virtual Private Network

WHO World Health Organization

WNS Windows Notification Service

WSO2 Web Services Oxygen

XML eXtensible Markup Language

XMPP Extensible Messaging and Presence Protocol

Chapter 1

Introduction

Stream processing on sensor-based systems deals with processing the sensor data streams in a near real-time fashion using various stream processing engines to perform streaming analytics [1]. Complex Event Processing (CEP) is a technique which ingests raw events from one or more sensor data streams in order to detect various complex events using Continuous Query Language (CQL). CQL is a declarative query language built on top of Structured Query Language (SQL) with some additional query constructs such as time windows and length windows which can be sliding or tumbling in nature depending upon the window expiration policy [2]. A window provides a temporal bound on an unbounded data stream that flows continuously. This enables the processing of the unbounded data steam over the window length. Generally, a Complex Event (CE) corresponds to the occurrence of multiple raw events each of which may correspond to the crossing of a specific threshold by a sensor data for example. The major difference between general stream processing systems (such as Apache Storm [3] and Apache Spark [4]) and CEP systems is that CEP systems are tuned for lower event processing latency and have a smaller setup time. Nowadays, CEP is widely used in various fields such as Business Activity Monitoring (BAM), Business Process Management (BPM), operations management and healthcare [5]. With the increase in the use of mobile devices and the availability of low-cost sensors, the popularity of Internet of Things (IoT) based appli-

cations and services are growing rapidly. A number of example use cases of CEP for sensor-based systems are discussed next.

1. Healthcare: The integration of health sensors with smartphones has enabled healthcare providers to offer Remote Patient Monitoring (RPM) as-a-Service. A survey by Berg Insight (a market research firm) in 2013 found that nearly 3 million patients use RPM [6]. Another survey in 2015 showed that 84% of the service providers use mobile devices for remote patient monitoring [7]. The RPM technique enables service providers to remotely monitor diseases such as sleep apnea, arrhythmia and Congestive Heart Failure (CHF). The streaming data from wearable health sensors is forwarded as raw data streams to a mobile device using a bluetooth or a WiFi connection. The mobile device forwards these raw sensor data streams to a centralized hospital server where various events are co-related using CQL to generate higher level complex events. The streaming data from various mobile-based embedded sensors such as a gyroscope, accelerometer and Global Positioning System (GPS) is also fused with data from raw health sensor streams for providing context enrichment.

2. Smart home: CEP systems can be used in smart homes to provide enhanced security, remote monitoring, and other home automation tasks. A smart home is a home in which various sensor-based appliances such as motion detectors, temperature sensors, a smart light system and a Heating Ventilation and Air Conditioning (HVAC) system are installed. These devices send the sensor data streams to a gateway device such as a smartphone which forwards the sensor data streams to a back-end server where complex events are detected and actions are performed [8].

3. Smart building: CEP systems are also used in smart buildings where various sen-

sors installed in appliances such as elevators, lighting systems, temperature control systems and energy consumption meters that send the sensor data streams to a centralized CEP system where complex events are detected [9].

1.1 Motivation for the Thesis

With the advent of the Internet of Things (IoT), CEP can be provided as-a-Service (CEPaaS) for providing solutions for smart homes, smart buildings, and RPM. In this research, we have considered RPM as a typical CEP use case. CEPaaS uses a centralized back-end server-based approach for detection of complex events using the data streams received from the sensors. This technique forwards all the sensor data through the edge device that collects the sensor data and sends it to a back-end server for the detection of complex events. Examples of edge devices include smartphones and wireless tablets. Such a system has several limitations as mentioned below:

1. Persistent connection: Current RPM methodologies collect the sensor data streams using a tablet or smartphone and forward them to a remote IoT server for complex event detection using Complex Event Processing as-a-Service (CEPaaS). This technique necessitates the mobile edge device to remain connected to the network at all times.

2. High bandwidth consumption and user cost: The main problem associated with using RPM-as-a-Service is that the user cost is increased a lot, as all of the raw sensor data streams have to be forwarded to the IoT server for processing. An interesting alternative method to reduce cost would be to perform the CEP at the edge device and to send only the CEP alerts to the IoT server.

3. Energy consumption: The power consumption of the mobile device is increased

as more power is consumed in transmitting the raw sensor data streams from the mobile device to the IoT server in comparison to processing the streams locally in the mobile device itself.

4. Data privacy: Security is a big concern in CEPaaS as sensor stream data needs to be encrypted and other state-of-the-art authentication and authorization measures need to be applied for the communication between the mobile device and the IoT server.

5. Transmission throttling: This technique can lead to a decrease in the transmission rate due to the additional checks and measures which need to be performed for authentication and authorization.

6. Out-of-order-delivery: Sending multiple sensor data streams to the centralized IoT server (in parallel to one another) can result in out-of-order delivery of various raw sensor data streams which may further result in false alarms.

7. Delivery guarantee: The sequence of the events is very crucial in event detection. To make sure that all the events have arrived on the IoT server exactly-once and in the exact same sequence they were generated at the sensor, various control signals have to be sent which lead to additional overhead.

1.2 Proposed Technique

This research focuses on devising an edge-based Mobile Complex Event Processing (MCEP) system which can effectively handle network unavailability. The system comprises sensor devices connected to a mobile device that, in turn, is connected to an IoT back-end server. The complex event processing is done by a CEP engine embedded within the mobile device which is used at the edge. The IoT server can be deployed in a standalone mode on

a single server, on a High Availability (HA) cluster or a cloud. The IoT server which has been used in prototype implementation provides support for batch, real-time, interactive and predictive analytics. In order to compare the performance of the proposed system with a CEPaaS system, a server CEP system, providing CEP service has also been devised. In comparison to the Server CEP system, the proposed mobile CEP solution leads to a decrease in user cost, a decrease in the energy consumption of the mobile device and a decrease in event processing latency. The end-to-end latency is also reduced in addition to mobile device memory usage as discussed later in Chapter 5.

1.3 Contributions of the Thesis

This research focuses on devising an edge-based mobile CEP system which can effectively handle network unavailability, reduce user cost and device power consumption. To the best of our knowledge, this is the first CEP system where the entire complex event processing is done on the mobile device deployed at the edge. The key contributions of this dissertation are as follow:

- A novel edge-computing-based CEP technique called mobile CEP which has the ability to detect complex events and generate local notifications even when the mobile device is unable to connect to the remote IoT server due to poor network connectivity. This technique lead to a number of advantages that are listed:
  - Connectivity: It can effectively handle temporary disconnection in the network connecting the edge device with the back-end server
  - User benefits: Reduction in the user cost, mobile device power consumption, and various event processing latencies of the proposed system in comparison to the state-of-the-art CEPaaS system.

  - Better data privacy: The proposed technique leads to more data privacy as the complete sensor data processing is done at the edge and the raw sensor data is not transmitted over the network.
  - Out-of-order delivery: The proposed solution circumvents the probability of out-of-order delivery occurrences as the sensors are connected to the edge device such as a smart phone using a local dedicated bluetooth/Wi-fi connection instead of a remote connection used in the case of CEPaaS system.

- Prototype: A proof-of-concept prototype using an Android smartphone has been developed to demonstrate the effectiveness of the proposed technique.
- Performance Insights: A rigorous performance analysis of the prototypes that leads to insights into system behavior and performance is performed.

1.4 Scope of the Thesis

At the time of writing of this dissertation, the CEP support for the Siddhi CEP system [10] used in this thesis is only available for Android and Raspberry-Pi devices and to the best of our knowledge, no other CEP engine other than Siddhi CEP is available for embedding within mobile devices. However, if in the future more CEP engines are ported to another mobile Operating System (OS) such as iOS, Blackberry, Ubuntu, and Microsoft, then the proposed system can be extended to support these devices with a modest effort. Our technique can also be extended to achieve a novel hybrid CEP architecture [11] which can perform stream processing at the mobile device using the embedded CEP engine and the batch processing at the IoT server using the Spark SQL for example [4]. Also, the proposed architecture can be used with the edge devices connected to the sensors with a wired connection. Please note that the accuracy of detection of the complex event

depends upon the accuracy of the sensor and the complex event detection software used . Controlling the accuracy of the detection is the beyond the scope of this thesis.

1.5 List of Publications

The list of publications resulting from this research is presented.

- A. S. Dhillon, S. Majumdar, M. S. Hilaire, and A. E. Haraki, "A Mobile Complex Event Processing System for Remote Patient Monitoring," in *Proceedings of the IEEE International Conference on Internet of Things (IEEE ICIOT)*, pp. 1–4, 2018.

- A. S. Dhillon, S. Majumdar, M. S. Hilaire, and A. E. Haraki, "MCEP: A Mobile device based Complex Event Processing System for Remote Healthcare," in *Proceedings of the 11th IEEE International Conference on Internet of Things (iThings)*, pp. 203–210, 2018. [IEEE Best Paper Award]

- A. S. Dhillon, (Advisors: S. Majumdar, M. S. Hilaire, and A. E. Haraki), "Complex Event Processing in Mobile Device for Remote Patient Monitoring," Poster, Data Day 5.0, Ottawa, ON, Canada, June 2018. [3^{rd} Place - Student Poster Competition]

1.6 Thesis Outline

The rest of the dissertation is organized as follows. Chapter 2 provides the background information related to various types of big data analytics, CEP systems, Pub/Sub systems, and IoT servers. Chapter 3 presents a literature survey. Chapter 4 describes the system prototype and discusses the architecture implementation details for server CEP and mobile CEP architectures. Chapter 5 focuses on the performance analysis and discusses various performance metrics, system/workload parameters, and experimental results comparing the various systems investigated. Finally, Chapter 6 concludes the thesis

and discusses possible directions for future work.

Chapter 2

Background Information

In this chapter, background information on big data analytics, complex event processing, and various CEP system components is provided in Section 2.1, Section 2.2 and Section 2.3 respectively. State-of-the-art CEP engines such as Apache Flink and Apache Siddhi are presented in Section 2.4. A brief discussion on Pub/Sub systems, cloud computing, and IoT server is presented in Section 2.5, Section 2.6, and Section 2.7 respectively. Further, an overview of the WSO_2 agent used in the prototype implementation is provided in Section 2.8. Finally, Section 2.9 discusses some essential Android concepts and Section 2.10 provides an overview of real-time dashboards.

2.1 Big Data Analytics

The systems processing streams of large volumes of data are often referred to as big data. Various challenges associated with big data include data storage, data visualization, data indexing, data retrieval, and data analysis. The primary goal in analyzing big data is to extract meaningful information from the data in order to get insights for making better decisions and devising strategies for system management. Four important characteristics of big data [12] are shown in Figure 2.1. A brief discussion of these characteristics is provided below.

1. Volume: The volume is related to a large amount of data to be analyzed coming from

various data sources such as different server or sensors as shown in Figure 2.1[1]. A survey by Interactive Data Corporation (IDC) has forecasted that the global data will reach 163 trillion GigaByte (GB) by 2025 [13].

2. Velocity: This parameter is related to data streams that are generated at a high rate from devices such as sensors, actuators, and Radio Frequency Identification (RFID) tags. As per the survey conducted by IBM in [12], every one minute, 72 hours of video footage is uploaded on the YouTube and 216,000 Instagram posts are shared (see Figure 2.1).
3. Variety: As shown in Figure 2.1, big data systems deal with data which is present in many different forms such as audio, video, and tweets, etc.
4. Veracity: This dimension deals with the uncertainty in the truthfulness of the available data (see Figure 2.1).

Data analytics concerns the extraction of knowledge from big data. The various types of data analytics used in big data are described next.

- Batch analytics: The batch analytics deals with the processing of large volumes of datasets which have been collected over a period of time and persisted in a database. The query latency used for performing batch analytics is directly proportional to the batch size. Hadoop [14] is a well-known batch analytics platform which is built upon the MapReduce framework developed by Google [15].
- Stream analytics: Stream analytics deals with the live streaming of data. The objective is to process streaming data as soon as it arrives on the stream analytics platform, without persisting it to a database. Stream analytics is widely used in networking, IoT, market analysis etc. Most of the industry standard stream analyt-

[1]Some of the icons used in various figures in this thesis are taken from https://www.flaticon.com/

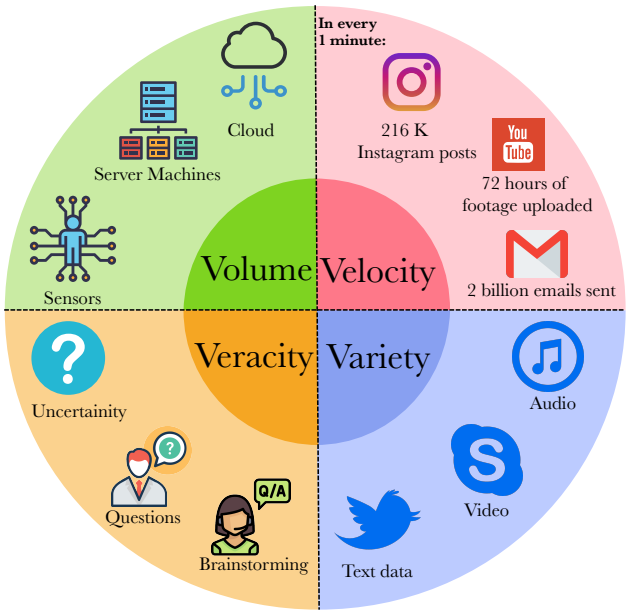


Figure 2.1: Four dimensions of the big data [12]

ics platforms have processing latency in the sub-second range and are able to handle high throughput rates such as 1GBps [16]. Some of the open-source steam analytics platforms are Apache Flink [17], Apache Storm [3], Apache Samza [18], The Information Bus COmpany (TIBCO) stream analytics [19], and the WSO_2 stream processor [20]. Some of the big data platforms support only batch or streaming analytics while only a few of them can handle both types.

- Micro-batch Analytics: In this technique, the incoming data streams are packaged to form small micro-batches upon which the analytics is performed. The Apache Storm supports micro-batching using the Trident Application Programming Interface (API) which is the extension of Storm developed by Twitter. Apache Spark is another platform which supports micro-batching [4]. As shown in Figure 2.2, Apache Spark discretizes the data streams at server 1 to form micro-batches and

then processes them on server 2. These micro-batches are stored as Resilient Distributed Dataset (RDD) (see Figure 2.2). This technique is also known as *discretized stream processing* in big data.

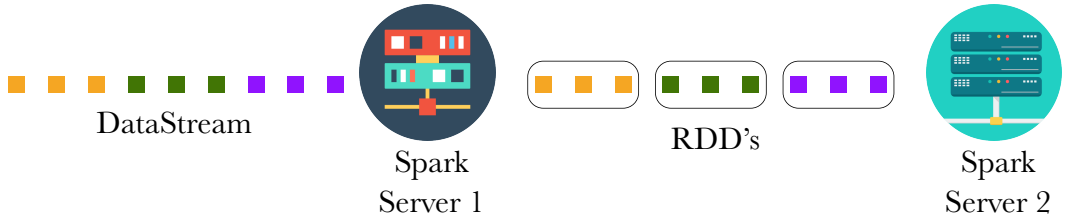


Figure 2.2: Spark streaming architecture [21]

- Predictive analytics: The predictive analytics uses advanced concepts of statistics, machine learning, data modeling and Artificial Intelligence (AI) to predict the future events based on the historical data. Various data mining techniques are used to find patterns in historical data so as to predict the events such as threats and opportunities that are expected to occur in the near future.

2.2 What is CEP?

Complex event processing deals with the real-time analytics component of the big data analytics. It is useful in monitoring and analyzing one or more sensor data streams in a real-time fashion to generate complex events representing alerts or opportunities. This is achieved by fusing data streams from multiple sensors inside the CEP engine to match patterns written using CQL [22].

- Data stream: As shown in Figure 2.3, a data stream consists of an unbounded number of events which arrives continuously. These data streams could be coming from sensors and actuators such as an RFID tag, a GPS device, and a gyroscope sensor or in the form of network packets, tweets, click-streams, and stock-ticks. All the

events in a data stream follow a particular schema. The schema represents the set of attributes included in an event. Some of the commonly used attributes are event generation time-stamp, sensor value/values, sensor name and event/tuple id.

Figure 2.3: A data stream

- Dataset: A dataset consists of a bounded number of events which are usually persisted in a database. The size of the dataset usually varies from a GB to a TeraByte (TB) range and hence analyzing it requires distributed processing.
- Event: An event can be modeled by using a tuple data-type or an object in most programming languages. Thus, the term event and tuple can be used interchangeably. A brief description of various types of events used in CEP systems is presented below.
  - Raw Event (RE): Events generated by a sensor are called as raw events.
  - Complex Event (CE): Event generated by the CEP engine as a result of a pattern match.
  - Intermediate Event (IE): Each raw event stream that is processed within the CEP engine produces an intermediate event stream. Multiple intermediate event streams are further processed within the CEP engine and can lead to a complex event. The intermediate events are also named as expired events in CEP terminology. Some advanced CEP engines such as Apache Siddhi and Apache Flink provide a mean to handle such events.

As the stream is a continuation of the occurrence of various events, thus the terms RE

stream, IE stream, and CE stream represent the streams consisting of the respective events. For example, Figure 2.4 shows a query tree consisting of two RE streams, two IE streams, and a CE stream. The CQL query is internally represented as a query tree consisting of various CEP operators in the CEP engine which are explained later in Section 2.3.7. In this query tree, the raw events are received by a selection operator (π) and a projection operator (σ) to generate two intermediate event data streams. Further, these intermediate event data streams are received by a join operator ($\bowtie$) to generate a complex event data stream.

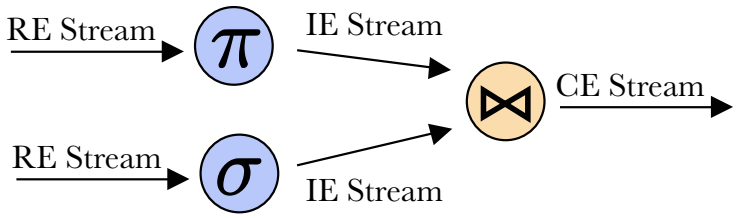


Figure 2.4: Example of a query tree

2.2.1 DBMS vs. DSMS and SQL vs. CQL

This section discusses the differences between a Database Management System (DBMS) and a Data Stream Management System (DSMS) system as well as the differences between SQL and CQL. This discussion is important as DSMS and CQL are two core components of a CEP system. These differences are enumerated next.

1. The traditional DBMS works on a persisted and finite dataset stored in a database whereas DSMS works on unbounded data streams arriving in a continuous fashion.
2. In DBMS, the dataset is static and SQL queries are dynamic as the user can perform rewrite query again and again over the same static dataset. However, in DSMS, the data streams are dynamic and CQL queries are static as they are installed once and

continue to run forever unless terminated by the user [23].

3. In DBMS, the dataset is usually large in size and SQL queries are usually small relative to the large dataset whereas, in DSMS, the data streams are small in size while CQL queries are relatively larger than the data streams.

4. As streams are arriving in a continuous fashion, a one-time processing model is followed in CEP systems. In other words, the data access in DSMS is a single pass (done in a sequential manner) while the data access in DBMS can have multiple passes in random or sequential fashion.

5. A DBMS follows the Human Active Data Passive (HADP) model as they produce results when queried by a user. On the other hand, DSMS systems follow the Data Active Human Passive (DAHP) model as data arrives continuously while the user is passive and results are continuously produced by installed CQL queries [24].

Figure 2.5(a) and Figure 2.5(b) show the DBMS and DSMS respectively. The superscript shown in red shows the order in which the execution is performed. For DBMS, the incoming data streams are persisted to a storage to form a dataset. Then, SQL queries are written to fetch the dataset into main memory and perform the required computation. However, in DSMS the CQL queries are installed first and they wait for the data streams.

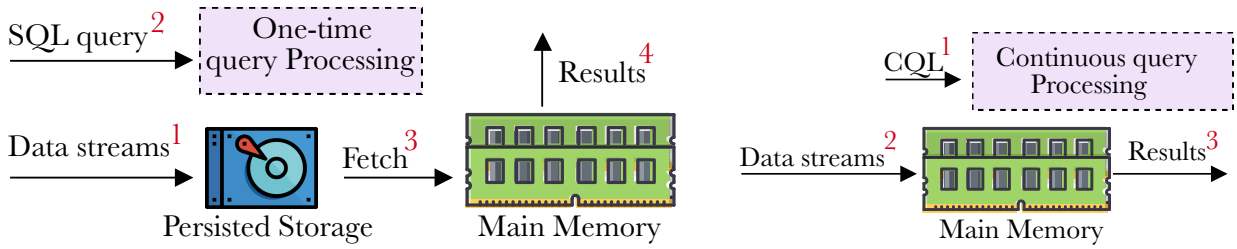


Figure 2.5: (a): A DBMS [25] (b): A DSMS [25]

As soon as the data streams arrive in the system, the processing is done in-memory to generate complex events.

Please note that CQL is the name of the query language developed at Standford University which was used for Aurora [23] and the Borealis system [26] they have developed. Later on, many such query languages have been devised such as Event Processing Language (EPL) by EsperTech [27], Siddhi query language by WSO_2 [10], TESLA by Cugola *et al.* [28], Oracle continuous query language and Stream-based And Shared Event processing (SASE) developed by the University of Massachusetts [29]. Some attempts to set the standard benchmark for CQL languages are discussed in [30] [31].

2.2.2 SP Systems vs. CEP Engines

The various real-time analytics platforms such as Apache Storm [3], Apache S4 [32], and Apache Spark [4] are categorized as Stream Processing (SP) systems. As shown in Figure 2.6, CEP systems and SP systems share some common functionalities as both perform real-time analytics. Some of the differences between these two types of systems are provided next.

1. The stream processing engines follow the imperative programming approach, as they require the user to write the logic for processing nodes, such as creating bolts in Apache Storm as shown in Figure 2.7. The spouts are used to ingest events from data streams and the processing nodes (bolts) are pipelined to generate a topology. The topology consists of various operators connected in a Distributed Acyclic Graph (DAG) fashion. On the other hand, the CEP engines follow the declarative programming approach, as they do not require the user to write logic, but instead a CEP query which uses inbuilt CEP operators as shown in Figure 2.4.

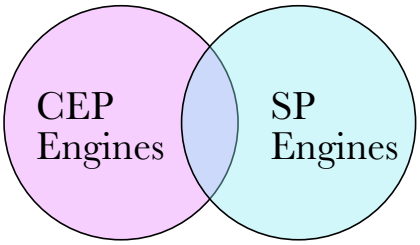


Figure 2.6: SP and CEP systems [22]

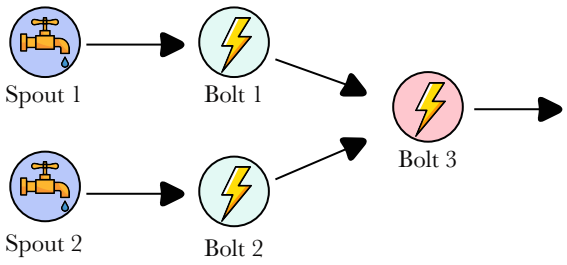


Figure 2.7: A storm topology [33]

2. The SP engines are designed to work in a distributed fashion whereas the CEP engines tend to work in a centralized manner, as the various CEP operators share state information.

3. As the CEP engines were initially created for stock-market analysis, they are tuned for a nanosecond to millisecond range latency [34]. However, most of the SP systems are built with a focus on reliable message processing and usually have close to a second level latency.

With the advent of IoT, CEP systems are getting more attention as compared to SP systems because of the imperative programming approach and lesser latency associated with CEP systems.

2.3 Components of CEP Engines

Figure 2.8 shows the various components of a CEP system. Here a solid line represents multiple data streams whereas a dashed line represents a single data stream. Initially, multiple raw sensor data streams are ingested by using either one of the event sources form the list of the sources shown in Figure 2.8. Further, these data streams are passed through a source mapper which converts the event format from custom types such as eXtensible Markup Language (XML) or JavaScript Object Notation (JSON) to the native format used by the CEP engine. As shown in the figure, the source mapper can also persist

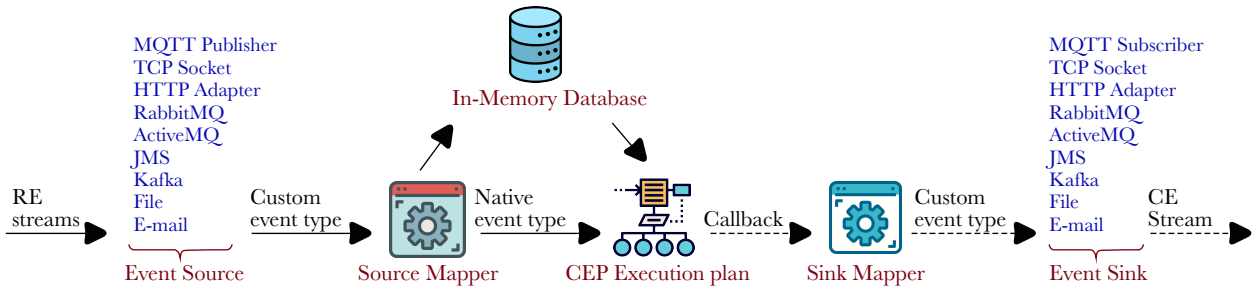


Figure 2.8: Overview of the CEP architecture [35]

these data streams in an in-memory database. Please note that in-memory databases are preferred because of the low latency requirements. If the event streams are temporarily persisted, then the execution plan consumes the data streams from the database. If the data streams are not persisted, then an execution plan consume data streams from the source mapper. The CEP execution plan consists of a CEP query tree which is used to find the complex events and sent to a sink mapper using various callback methods. The job of a sink mapper is to convert the event format to the type which is required by the event sink. Finally, an event sink forwards the CEP alerts to a specified destination such as a dashboard or an email address. A brief discussion of the various CEP components is provided in the following subsections.

2.3.1 Event Sources

An event source consumes the sensor data streams from the external sources. The various kinds of event sources used by the CEP systems include Message Queuing Telemetry Transport (MQTT) [36], Transmission Control Protocol (TCP) sockets, Kafka [37], WebSockets, Secure WebSockets, Email, Java Message Service (JMS) queues [38], files, RabbitMQ [39], etc. Please note that a CEP system may support all or a few of the aforementioned event sources. Also, various event receivers such as MQTT, JMS, and Kafka create an event queue for each data stream which provides temporal persistence and avoids

data loss in case the CEP engine is saturated.

2.3.2 Event Sinks

An event sink is responsible for publishing the CEP data streams to external endpoints such as e-mail, TCP sockets, Kafka, Hypertext Transfer Protocol (HTTP) adapter and real-time dashboard. The event publisher can further forward the streams to event receivers so as to form a pipelined architecture.

2.3.3 Source Mappers and Sink Mappers

The source mappers are responsible for format conversion for data streams from a custom format to a native format which is understood by the CEP engine. Similarly, the sink mappers convert the native event type generated by the CEP engine to a custom event type required by the event sink. The various format types that are used by events include XML, WSO_2-event, text, JSON, binary data, Plain Old Java Object (POJO), key values pairs or Comma Separated Values (CSV). The WSO_2 Siddhi CEP used in this dissertation does processing using the WSO_2-event format. The various commercial health sensors use Health Level-7 (HL7) as the default standard to transfer data streams [40]. The latest version of HL7 messages is 3.0 which is based upon XML encoding.

2.3.4 Blocking Queues

Queues are an important part of a CEP system, as events need to be temporarily persisted in a queue before sending them to the CEP engine to prevent event data loss. However, in a producer-consumer scenario, thread safety is required as both producer and consumer are accessing the shared queue at the same time. The blocking queues provide various thread safety mechanisms. Different types of blocking queues are available such as *LinkedBlockingQueue* [41], *ArrayBlockingQueue* [42], *PriorityBlockingQueue* [43], and *SynchronousQueue* [44] which are suitable for different needs. As shown in Figure 2.9, the

producer thread keeps on en-queuing the incoming elements from the data stream to the tail of a queue and the consumer thread keeps on de-queuing the data stream elements from the head of the queue in a parallel fashion. In blocking queues, if the producer thread tried to en-queue an event when the queue is full, then the producer thread is blocked and kept in a waiting state unless there is an empty space in the queue. Also, if the consumer thread tries to de-queue an event from an empty blocking queue, then that thread is blocked until the producer thread en-queues an event to the queue. Circular queues or ring buffers are also used by some of the CEP engines.

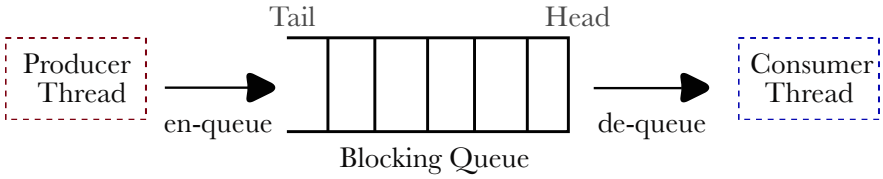


Figure 2.9: A blocking queue [45]

2.3.5 In-Memory Tables

Another option to persist the events is by using in-memory databases such as Sqlite3 and H2 [46]. As shown in Figure 2.10, the data stream events are converted to relations before persisting them to a database. Furthermore, the relation-to-relation modification can also be performed using Data Manipulation Language (DML) commands. The Flink stream processing platform provides a *TableAPI* [47] which is a language integrated query API in Java and Scala. Using this API, data streams are converted to relational datasets upon which SQL operations can be performed. This approach may lead to additional latency due to the data-type conversion and is not preferable when low latency is required.

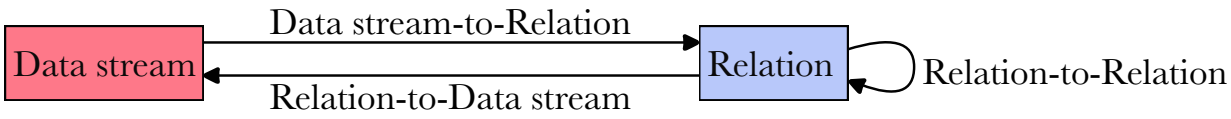


Figure 2.10: Data stream and relation conversion concept proposed in [2]

2.3.6 Callbacks

Callbacks are methods which are used to receive complex events from the CEP engine. Two types of callback methods are discussed next.

1. Stream callback: A CQL query installed in the CEP engine can generate many CE streams. The stream callback is used to subscribe to a particular CE stream using the stream identifier. Every time complex events are detected by the query, the stream callback will receive those events. The CEP event detection time-stamping is done at stream callback.

2. Query callback: It receives all the data streams which are generated by the CQL query. It includes multiple complex event streams and intermediate event streams.

2.3.7 CEP Operators

An operator is a reserved keyword in CQL such as *select*, *where*, and $\rightarrow$. These operators can be classified as arithmetic, logical, and comparison operators. Further, these operators can be divided into two categories as discussed below [48].

1. Blocking/stateful operators: These operators require the complete input data stream before producing a result as they perform the processing based on the previous state information of an operator. These operators save the state information of previous tuples to compute the result. For example, the *max* operator needs to save the state information of all the data stream events in the specified window to find the maximum value [24]. Aggregate, Join and Cartesian product are few more examples of blocking operators.

2. Non-blocking/stateless operators: They do not require to save the state information of the data stream tuples to compute the results. The results can be computed as

soon as the element is seen by the operator. For example, the conjunction, disjunction, map, and filter operators will produce the results as the data stream arrives [49].

Table 2.1 shows a brief description of various CEP operators.

Table 2.1: List of various CEP operators given in [29], [50], [51] and [52]

Name	Symbol	Description
Kleene star	*	Fires when zero or more events match the condition.
Kleene plus	+	Looks for one or more matches.
Optional	?	Looks for zero or one matches.
Conjunction	&	Used when all the conditions should be true.
Disjunction	||	Used if either of the condition is true.
Join	$\bowtie$	Used to join two or more streams based on a key.
Union	$\succ$	Used to merge two or more streams without any condition.
Negation	!	The event is triggered if the event does not happen.
Followed by	$\rightarrow$	Represents the order in which events happen.
Every	*every*	Looks for a match for every arrival of the event.
Next	*next(x)*	Looks for immediate next occurrence of x event.
Selection	π	Selects the predicates based on some selection condition.
Projection	σ	Selects all the stream elements.

The various aggregate operators such as *average*, *max*, *min*, *count distinct*, *max forever*, *min forever*, and *standard deviation* are also supported by most of the CEP engines. The selectivity of an operator is defined in Equation (2.1).

$$\gamma = \frac{\text{Number of events emitted by an operator}}{\text{Number of events received by an operator}} \tag{2.1}$$

2.3.8 Windows

As data streams are unbounded and continuous in nature as compared to finite datasets, a certain bound has to be specified in a query for performing computations. Windows help in selecting a bounded subset of data from a continuously arriving data stream for analysis. These windows can be sliding or batch (based on window expiration policy), each of which can have time-based or length/count-based selection policy as shown in Figure 2.11. An example of a time-based window is to find the average value of all Google stock-ticks within the last 1 minute. On the other hand, finding the average of the last 100 stock-ticks is done using the length-based window of 100. The two types of expiration policies: sliding or batch are discussed next.

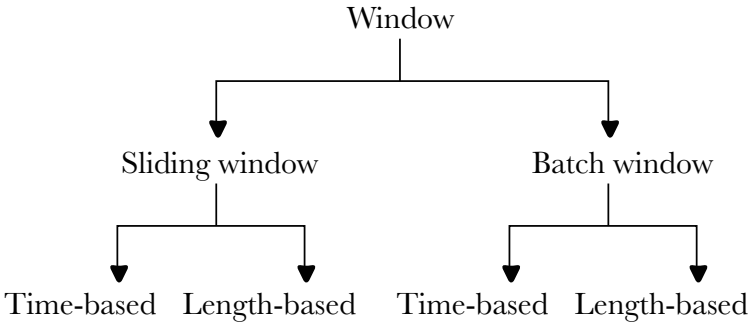


Figure 2.11: Classification of windows [53]

- Sliding windows: The sliding window has a window size and a window slide parameter. For example, a sliding window of 2 minutes with slide parameter of 1 minute will capture all the events that happened in last the 2 minutes and will update the window elements every 1 minute such that all the events in the previous window are flushed out after processing.
- Batch windows: The batch windows are non-overlapping and processing of the batch is done after the batch is complete. For example, a time-based batch window

of 1 minute will start processing once 1 minute worth of data has been collected. This is necessary as sometimes the whole data is required to perform the computation. All the aggregated operations on a data stream such as finding the max, min, count, average, std. deviation require the batch window.

2.3.9 Query Execution Plan

The query execution plan is the internal representation of the query in the CEP engine. CEP query can be broadly categorized as either a pattern query or a regular query, based on the query execution plan [50] [52].

1. Regular query: For these queries, the order of arrival of events is not important. The CEP engine executes this type of queries by using a tree-based computation. The query execution plan for an example CQL query is shown in Figure 2.12. In this query tree, two streams: Heart Rate (HR) and Respiration Rate (RR) are received by the leaf operators. If the value of HR is ≥ 30 and the value of RR is ≤ 20, then a complex event is generated.

2. Pattern query: For such a query, the order of the events is important and the order is specified using a *followed by* ($\rightarrow$) operator. A state machine-based execution model is used within the CEP for executing the pattern queries.

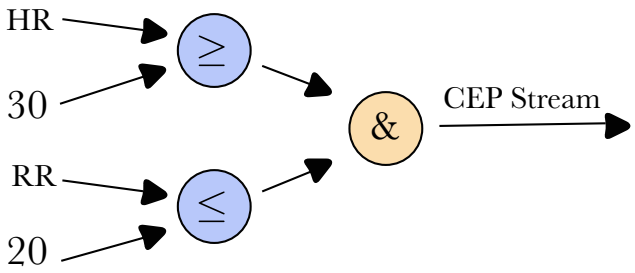


Figure 2.12: A query tree

As per [50], the various types of patterns contiguity are enumerated below.

- Strict contiguity: It means that an event occurs exactly after another, such that no other event comes in between these two events. For example, a pattern *A.next(B)* will be fired only if *B* arrives immediately after event *A* arrives.
- Relaxed contiguity: It means that the pattern will fire a complex event even if another event occurs in between the occurrence of the specified events. For example, a pattern $A \rightarrow B$ *within* 10 *seconds* will be fired if event *B* arrives after the arrival of event *A*.
- Non-deterministic relaxed contiguity: It provides more relaxation to the matching events by using operators such as $*$, $+$ or ?. For example, a pattern $A \rightarrow B \rightarrow C$ *within t seconds* will match if these events arrive in the specified order within *t* seconds as shown in Figure 2.13. Similarly, a pattern $A \rightarrow B* \rightarrow C$ *within t seconds* will result in a complex event if the first event *A* occurs, followed by one or more *B* events, followed by *C* event within *t* seconds as shown in Figure 2.14.

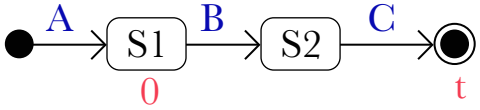


Figure 2.13: Automata for ABC [54]

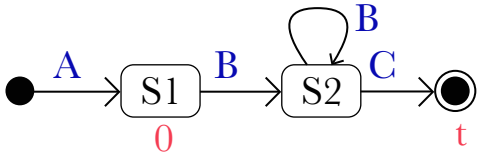


Figure 2.14: Automata for AB+C [54]

2.3.10 Event Time-stamping

The time-stamping is an important aspect of CEP engine. The various times associated with an event are:

1. Raw event generation time (T_g): It is the local time at which the raw event is generated in the sensor and usually time-stamped by the sensor itself.
2. Raw event ingestion time (T_i): The time at which the raw event is ingested by the CEP engine.
3. Complex event detection time (T_d): The time at which the complex event is detected in the CEP engine.

2.4 State-of-the-art CEP engines

A number of CEP engines are available in the market such as Apache Flink [17], Apache Siddhi [10], Drools Fusion [55], Esper [56], and Microsoft StreamInsight [57]. In this section, the two most widely used CEP engines are briefly discussed.

2.4.1 Apache Flink

Apache Flink provides both batch processing and stream processing capabilities by using the dataset and data stream API. The CEP library has been built on top of the data stream API using Java and Scala. At the time of writing this thesis, there is no CQL support available in Flink. However, the pattern API is written using lambda expressions using an imperative programming approach. Unfortunately, Apache Flink cannot be used in this research as it has not been ported to any mobile device yet.

2.4.2 Apache Siddhi

Siddhi is an open-source and lightweight CEP engine (less than 2 MB) which provides a Siddhi query language for writing CQL queries [10]. It is used by many fortune 500 companies including UBER to process 20 billion events/day (300,000 events per second) for fraud analytics [58]. It has been recently ported to Android and Raspberry-Pi devices. As shown in the Figure 2.15, the Siddhi CEP engine can receive sensor data streams from

various types of event sources and also publish the detected complex event stream using various types of event sinks such as RabbitMQ, Kafka, ActiveMQ and TCP sockets. Also, the sensor data streams can be in various different formats such as JSON, binary, XML or key-value pairs as shown in the Figure 2.15. The support for various types of SQL and NoSQL databases such as Cassandra [59], H2 [46], and MongoDB [60] is also available for the temporal persistence of sensor data streams.

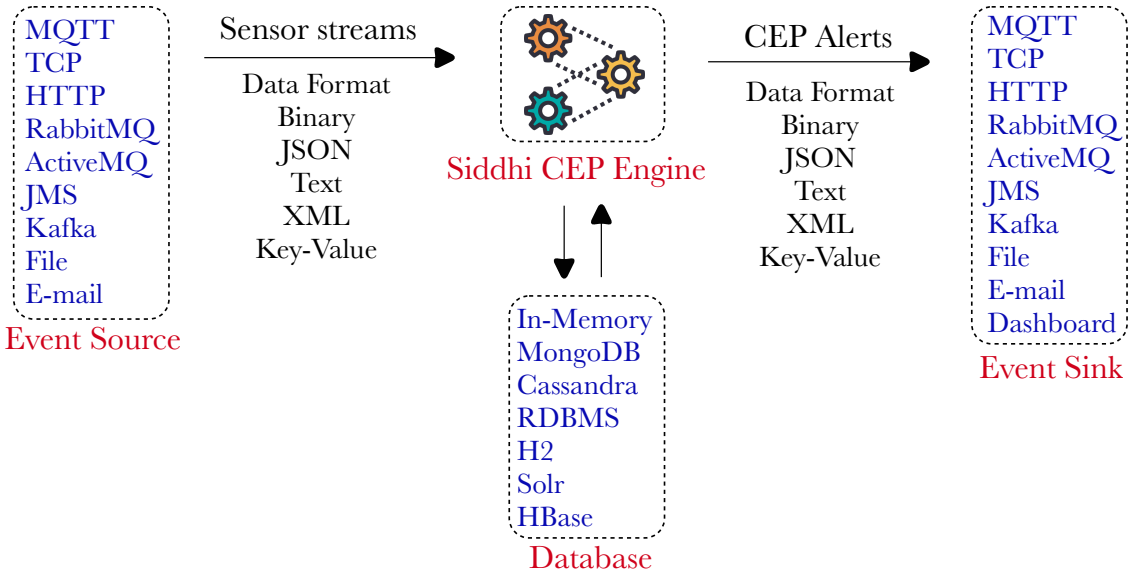


Figure 2.15: Siddhi CEP architecture [58]

2.5 Pub-Sub Systems

Publish/Subscribe messaging systems are often referred to as Pub-Sub systems. Pub-Sub systems are widely used in the context of CEP as event sources and event sinks. This system has been developed as an alternative to the client-server messaging model in which a client can communicate directly with the server. In Pub-Sub systems, the publishers and subscribers are agnostic of one another as messages as forwarded by a third-party broker. As shown in the Figure 2.16, the publishers and subscribers are both

connected to a broker and are subscribed to a given topic in the broker. The messages are sent by the publishers to a particular topic in the broker. The broker forwards these messages to all the subscribers which have subscribed to that topic.

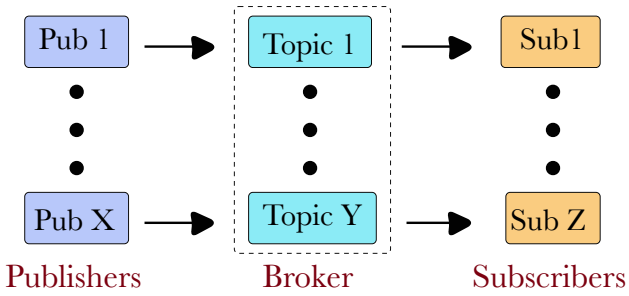


Figure 2.16: Example of a Pub-Sub system [61]

Some of the advantages of using the Pub-Sub systems are:

1. Scalability: The broker can be massively parallelized so as to increase scalability [62].
2. Space decoupling: The publishers and subscribers are space decoupled as they do not need to know the Internet Protocol (IP) address and port number of one another as needed in the case of the conventional client-server architecture.
3. Time decoupling: The publisher and subscribers do not need to be active at the same time as the messages can be persisted by the broker.

A detailed description of the various components of the Pub-Sub system is presented next.

1. Client: Both the publisher and a subscriber can be referred to as the client from a broker's perspective. The clients can be persistent or transient. The persistent clients maintain the session information with the broker in the registry files whereas the transient clients do not.

2. Topic: It is an endpoint to which the various publishers and subscribers can connect. It is encoded as a UTF-8 string and uses the forward-slash (/) as the delimiter.

3. Pub-Sub broker: Each topic has a queue associated with it. The broker receives the messages from various publishers and stores them in the queues associated with the respective topics. As the broker is the single point of failure, thus clustering the broker is often used.

4. QoS: As shown in Figure 2.17, various message delivery guarantees are provided by the broker using Quality of Service (QoS) parameters. The various types of QoS supported are given next.

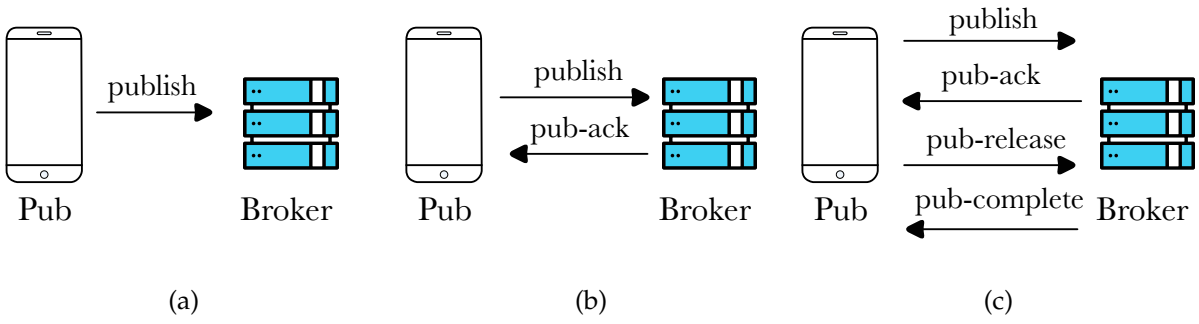


Figure 2.17: (a) QoS_0 [63] (b) QoS_1 [63] (c) QoS_2 [63]

(a) QoS_0: It provides at-most-once delivery of the messages as no acknowledgment is sent by the subscriber to the broker (see Figure 2.17a). So two things can happen, either the message is received by the subscriber or it is lost during transmission.

(b) QoS_1: It ensures at-least-once delivery of the message. As shown in Figure 2.17b, if the acknowledgment (*pub-ack*) is sent by the subscriber before the specified timeout, then the message is delivered once. Every message is appended with

a unique packet identifier which is also included in the *pub-ack* sent by the subscriber. If the *pub-ack* is not received within the specified timeout, the message is sent again by the broker which leads to more than once delivery of the message. Thus, the broker will store the copy of the original message until it has not received *pub-ack*. Thus with QoS_1, at least one data packet and a control packet are sent by the broker.

(c) QoS_2: It guarantees that every message is delivered exactly-once. QoS_2 is the slowest type of delivery among all other QoS types. Also, it may lead to more data transmission and memory usage. After the broker gets the message, it sends a *pub-ack* message to the publisher, as shown in Figure 2.17c. Upon receiving the *pub-ack*, the publisher sends another acknowledgment saying that it has received the acknowledgment through a *pub-release* control message. Upon receiving the *pub-release* message, the broker can delete the reference to the packet identifier and reply with the *pub-complete* control message.

Some of the widely used Pub-Sub systems are MQTT [36], Kafka [37], ActiveMQ [64], and RabbitMQ [39]. A brief discussion of the MQTT Pub-Sub system that is used in this research is provided next.

2.5.1 MQTT

MQTT is a lightweight and bandwidth efficient machine-to-machine protocol designed for low powered devices [36]. It was started as a joint project named *EclipsePaho* by IBM and Eurotech in 2011. Today, it is very popular among the mobile devices and it has API's available in multiple languages such as *C*, *C++*, *C#*, *Go*, *Java*, *Node.js*, and *Python*. MQTT runs on the top of TCP/IP stack. The MQTT broker is also known as a mosquito broker.

2.6 Cloud and Edge Computing

Both cloud computing and edge computing are relevant in the context of our research as CEP can be done at the edge or provided on the cloud as a CEP service. Both of them are explained next.

- Cloud computing provides the resources such as software, infrastructure, storage, analytics and many more as-a-service at low cost by using a pay-as-you-go model. Various companies such as Amazon, Microsoft, Google and International Business Machines Corporation (IBM) provide Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). However, providing CEP-as-a-service is still an ongoing research work in which the most relevant work has been done in [65].
- Edge computing is a technique in which the computation is done at the edge of the network and even on devices such as mobile devices, Arduino micro-controller [66] and Raspberry-Pi [67]. In this manner, the computational power of the edge devices is used which results in a number of benefits such as a reduction in the communication bandwidth between the edge device and the cloud data-center, providing near real-time data analytics as edge devices are in proximity to data sources.

2.7 IoT Server

The IoT servers are used for different purposes such as administration, monitoring, data gathering and analysis [68]. They are modular, based on open-source enterprise platforms that provide the capabilities that are needed for the server-side of the IoT architecture, to connect to the edge devices. The data that is transmitted through the server gateway is processed and stored securely and analyzed using big data analytics. Examples of the IoT

servers include Amazon Web Services (AWS) IoT Platform [69], IBM Watson IoT Platform [70], WSO_2 IoT server and Google Cloud Platform (GCP).

2.7.1 WSO_2 IoT Server

The WSO_2 IoT server can run on a workstation, a cluster or can be deployed on a cloud [68]. This server is an open-source Apache 2.0 licensed server. It provides support for batch analytics, real-time analytics, interactive analytics and predictive analytics. It can provide built-in support for Windows, iOS, Blackberry and Android devices along with support for Arduino and Raspberry-Pi devices. As shown in Figure 2.18, the IoT server uses a three-layered architecture consisting of a broker, core, and analytics.

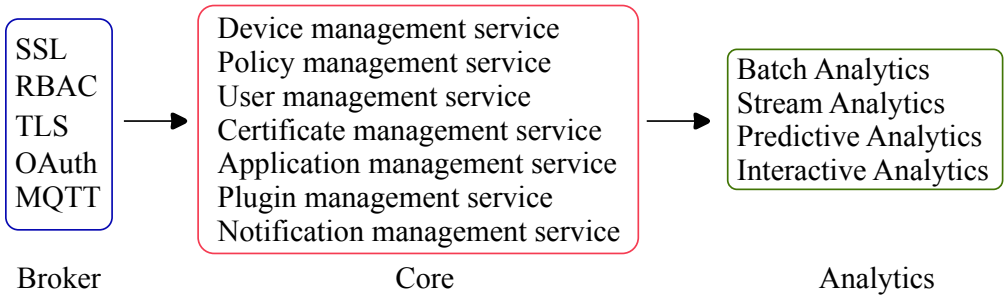


Figure 2.18: Key components of the WSO_2 IoT server [71]

A brief description of each component is presented next.

1. Broker: The broker is responsible for providing authentication, authorization and security features. Open Authorization (OAuth) and Secure Socket Layer (SSL) are used for patient authentication while Role Based Access Control (RBAC) is used for authorization. Mutual SSL and certificates are used to provide Transport Level Security (TLS). The MQTT broker runs inside the IoT broker which receives the events from various devices.

2. Core: The core provides various services as shown in Figure 2.18.

- Device management service: It manages various sensor devices and mobile devices with the server.
- Policy management service: Various policies such a white-listing or black-listing applications, policies related to access control for a tenant can be enforced from the IoT server for enrolled devices.
- User management service: As the server is multi-tenant, thus this service provides a different level of access for each user. It lets create or delete various users as well.
- Certificate management service: Each registered edge device has device certificates which are managed by using this service.
- Application management service: This service manages various applications which are running on the server.
- Plugin management service: A plug-in is an Open Service Gateway Initiative (OSGi) extension which a user can write to have custom functionality. The OSGi is a Java framework for developing the services. It facilitates the creation of the services/plugins which are compatible with the IoT server and various mobile devices.
- Notification management service: It uses the Apple Push Notification Service (APNS) [72], Firebase Cloud Messaging (FCM) [73], and Windows Notification Service (WNS) [74] for providing notifications.

3. Analytics: This layer provides the batch, stream, predictive and interactive analytics functionalities.

2.8 WSO_2 Agent

It is the application which helps in the communication between the various mobile edge devices such as Android and iOS-based devices with the WSO_2 IoT server. In other words, it provides the device management features like device enrollment, device authentication, and authorization. After the successful registration of the edge device with the IoT server, the agent application running on the edge device periodically communicates with the server.

2.9 Android Essentials

As we have designed a mobile device-based CEP system, thus some of the essential Android concepts are presented next.

1. Gradle: It is the build tool which can install and compile various dependencies in an Android system [75].

2. API version: Each Android release comes with a new API version [76]. The latest API version is 27 (codename Oreo) at the time of writing this thesis. The support for Java 8 has been added from API version 24 (codename Nougat) onwards. This is an important feature as most of the big data API's (such as those associated with Apache Flink and Apache Siddhi) are written in Java 8 using lambda expressions [50] [52]. As shown in Figure 2.19, Java 8 support is provided by using the *desugar* tool to perform *bytecode* transformations to generate a *.dex* file from a *.java* file. The *.dex* file represents an Android executable file similar to *.class* files in Java. Android application is represented by an Android Package Kit (APK) file (similar to the *jar* file in Java) which can consist of one or multiple *.dex* files.

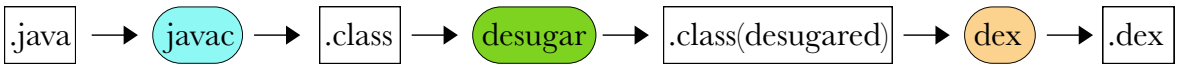


Figure 2.19: Java 8 language feature support [77]

3. Activity: An activity is the Graphical User Interface (GUI) based component in Android which represents a single screen [78]. When the user goes from one activity to another, the data in the previous activity gets lost. An example of an activity is a login page or a registration page.

4. Service: Service is a non-GUI-based component which can run in the background [78]. An email service running on the mobile device is an example of an Android service.

5. Multi-dex feature: An APK file represents an application in Android. The Android compiler has a 64K limit which means that an APK file can contain a maximum of 65,536 methods [79]. But when we import multiple dependencies, the number of methods exceeds 64K. The multi-dex feature overcomes this issue by creating multiple *.dex* files for an application (represented by an APK file).

6. Android Debug Bridge (ADB) shell: Android is a Linux-based system with each application representing a different user in the Linux system [80]. The ADB shell provides the root level access similar to a terminal in Linux and Macintosh systems.

2.10 Real-Time Dashboards

Monitoring is a crucial step in real-time analytics as it provides insights into the system behavior and performance. A real-time dashboard is tuned for low latency and high-performance needs. It automatically refreshes the visualizations in the form of charts, graphs, and tables after some refresh interval period. The WSO_2 IoT server provides the various built-in dashboard plugins which are easy to configure and deploy. However, for

larger systems, some of the enterprise scale real-time dashboards such as ElasticCompute Logstash Kibana (ELK) stack [81], Ganglia [82], Graphite [83], and Datadog [84] can also be configured with the WSO_2 IoT server.

Chapter 3

Related Work

This survey of related work is divided into three sections. Section 3.1 provides insights into different CEP engines and their key features. Section 3.2 discusses the related work in the context of smart buildings and homes. Section 3.3 describes the work related to RPM domain which is the main use case discussed in the thesis. Finally, Section 3.4 provides the concluding remarks for this chapter.

3.1 CEP and Stream Processing systems

The term CEP has its roots in discrete event systems and active databases [85]. In contrast to a passive database which can store, retrieve and update the persisted data, an active database system can respond automatically to the continuous events arriving on the system using Event Condition Action (ECA) rules. An ECA rule performs the desired action when a certain logical condition is met as raw events arrive on an active database system. Such ECA rules are suitable for simple business process management as well as big data management scenarios [86]. *Ode* [87], *Samos* [88] and *Snoop* [89] are a few examples of active database systems. A study conducted in [85] discusses the various limitations of active databases in comparison to CEP systems, such as lack of standardization, difficulty in optimizing the large applications and difficulty in performing various distributed and parallel operations. For addressing these limitations, a CEP system was required for

handling complex scenarios.

The initial work on CEP was done by David Lukham in the early 1990's during the development of the *Rapide* project [90]. The *Rapide* project is an event-driven simulation language which can be used to model a multi-layered event architecture. During this work, Lukham coined the term CEP in his book entitled '*The Power of Events*' [22]. Another set of work related to CEP was done during the *Stream* project at the Standford University in the late 1990's [91] [92]. This work was further continued by a collaborative project at Brandeis University, Brown University and Massachusetts Institute of Technology (MIT) during the early 2000's which led to the development of the Aurora system discussed later in this chapter. In recapitulation, as a repercussion of the aforementioned research works, various CEP and SP systems were devised, which are shown in a time-series order in Figure 3.1. During the earlier development of these systems, not much distinction was made between SP and CEP systems. As the development nurtured, the distinction between the two systems became more apparent. A summarized overview of the various open-source, enterprise and cloud-based CEP/SP systems is presented in the following subsections. The open-source CEP engines are discussed first and the Commercial off-the-shelf (COTS) CEP engines are discussed next.

3.1.1 Open-Source CEP engines

In 2003, the TelegraphCQ project introduced a general purpose CQL named *StreaQuel* [93]. The *StreaQuel* inherited all the SQL constructs and supports continuous stream processing by using a *WindowIs* operator to provide various window specific features. Later, this work was followed by the development of a NiagaraCQ engine which can receive data from various distributed active databases over the Internet [94]. In other words, NiagaraCQ is an extension of an active database system with an additional functionality of distributing the information sources over a wide geographical area. Due to this feature,

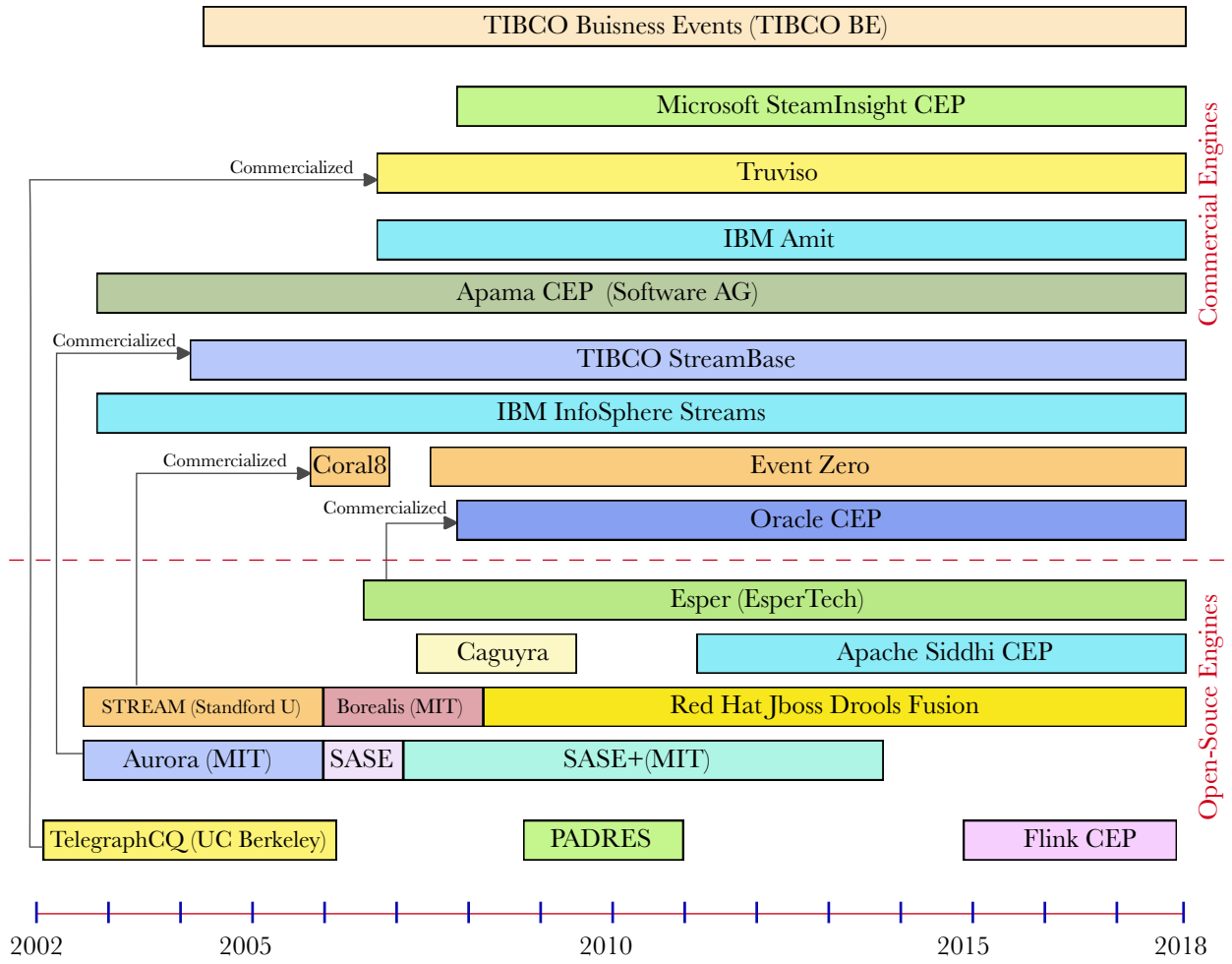


Figure 3.1: Evolution of CEP systems [5]

the TelegraphCQ system is more scalable than a regular active database system.

Another set of works led to the development of the Aurora system which was the first SP system to have DSMS capabilities [23]. The authors of Aurora have also introduced an imperative CQL language called Stream Query Algebra (SQuAl). SQuAl uses a box-and-arrow approach to connect various CQL operators. The authors also introduced a mechanism to persist various data streams into an intermediate storage, so that different

ad-hoc queries can be executed on the persisted data streams. An ad-hoc query is created on-the-fly whereas a continuous query needs to be installed before the CEP system starts ingesting the sensor data streams. In this paper, the support for specifying various types of QoS requirements for each operator was also provided. Such a QoS parameter is used by the query scheduler to optimize the query processing. The Aurora system was further evolved as the Medusa system which provided support to scale the stream processing over multiple nodes leading to its high-availability features [95]. On top of Medusa and Aurora, Borealis was built to provide a distributed stream processing engine having load shedding and fault tolerance capabilities [26]. It also avails some of the advanced features such as a graphical query editor and a stream visualizer. Later on, the Gigascope system was built to focus on the networking type applications involving high data transfer rates [96]. In this paper, the authors have also proposed a CQL language named GQSL.

The *Stream* project introduced a well-defined CQL consisting of three different types of operators: stream-to-relation, relation-to-relation, and relation-to-stream [97]. The stream-to-relation operator is responsible for converting a data stream to a relational table by using various time-based and count-based windows each having either sliding or tumbling/batch expiration policy. The relation-to-relation operator uses various SQL constructs for performing the data transformation operations on the relational tables. Furthermore, various relation-to-stream operators such as IStream, DStream, and RStream are used to generate a data stream from a relational table. The various load shedding techniques were also devised to handle the data streams arriving at higher arrival rates.

In [98], Demers *et al.* have proposed a Cayuga CEP engine which uses the Cayuga Event Language (CEL) for expressing CEP patterns. CEL is based on the Cayuga query algebra. The Cayuga engine receives various data streams using event receivers running

on separate threads. The incoming data streams are first deserialized; this was followed by event arrival time-stamping and then enqueued to a priority queue. A single threaded Cayuga CEP engine dequeues the data stream events to detect complex events. The detected complex events are then forwarded to a client notifier thread which publishes the complex events to the client. Another work was done by Daniel *et al.* in [29] proposes a SASE CEP system which is primarily focused on processing RFID data. The prototype system simulated a retail-store shoplifting scenario using CQL queries. A new CQL language SASE is also proposed in this paper. The SASE query language uses a tree-based approach to perform complex event processing. The SASE language provides support for the sequence operators using Non-deterministic Finite Automata (NFA) constructs. In conclusion, this system laid a foundation upon which various other systems were built. The SASE system was further extended in SASE+ which allows the user to define various event selection policies [54]. An important contribution of this paper is that it provided the methodology to compare the query complexity of SASE+ with other CQL languages such as CEL. However, this comparison model is not applicable to a wide range of CQL's available today [24].

Esper is the most widely used CEP engine written in Java, C#, and .NET (NEsper) [56]. The Esper engine is also used as a core in the Oracle CEP. It provides a declarative CQL called EPL. The support for the clustered deployment using Esper HA mode is also available. A major limitation of Esper is that it is a centralized CEP system. However, it has been integrated with Apache Storm to provide distributed CEP capabilities by some researchers [99]. JBoss Drools Fusion [55] is an extension of the Drools rules engine having additional event processing capabilities. It supports two modes: cloud mode (default mode) and stream mode. In the cloud mode, there is no notion of flow of time, which means the engine is not able to determine the age of an event. Due to this reason, there is

no support for sliding windows and event ordering in this mode. The stream mode, on the other hand, requires the data stream to be time-ordered. In this mode, the engine uses a session clock to force the synchronization among different data streams.

Apache Flink is a recent real-time stream processing engine developed by DataArtisans which provide exactly-once processing [17]. The CEP library is written on top of the DataStream API as discussed in Section 2.4.1. It supports the cluster deployment and the HA mode. The support for Amazon Elastic MapReduce (EMR), Docker [100], YARN [14], Mesos [101] and Google Compute Engine (GCE) [102] is also available. Various patterns can be deployed using various data stream API lambda expressions. It introduced a mechanism to tackle out-of-order events by using time-stamps and watermarks. The various time-stamps such as event time, ingestion time and processing time are used by the engine. The watermarks are the time-stamps which are emitted at the sources in the Flink topology. These watermarks are sent to the operators which are then forwarded to other downstream operators. The operator only processes the events which have a greater time-stamp than the watermark time-stamp. The streams and transformations are basic elements in Flink. Additionally, savepoints can be added to help in resuming the execution of a cluster from a previously saved state. The savepoints take a snapshot of the Flink program and save it to a state back-end. Also, the back-pressure handling mechanism is supported by the Flink CEP which lets the Flink CEP to gracefully deal with the load spike such as high arrival rates. Every operator in the CEP system has a queue to receive events. The back-pressure mechanism lets the predecessor operators know that the queue in the successor operators of the query tree is full. Apache Siddhi CEP is an advanced and lightweight CEP engine which supports a large number of CEP operators. More details about Siddhi CEP are given in Section 2.4.2.

3.1.2 Commercial off-the-shelf (COTS) CEP engines

As shown in Figure 3.1, in addition to open source engines, various COTS engines have been devised. COTS CEP engines are not discussed in detail as the open-source systems are more preferable in the current research scenario due to the high usage cost of COTS CEP engines. A survey conducted by Gartner in 2009 reported that COTS CEP costs between $100,000 and $250,000 [34]. However, the methodology proposed in this dissertation is extensible such that it can be extended to support COTS/enterprise engines with a modest effort. Some of the COTS engines have their roots in open-source products. For example, the Aurora system has been commercialized as the TIBCO StreamBase CEP (see Figure 3.1) which is nowadays used in algorithmic trading, foreign exchange trading, real-time patient monitoring, airline industry, etc. [103] [19]. The Esper CEP engine has been commercialized by Oracle with an addition of the Oracle CQL support to form Oracle CEP [104]. The Truviso was founded by a UC Berkley Professor who was involved in the development of the TelegraphCQ system. This company is known to provide various business intelligence and algorithmic trading solutions using CEP. Other COTS CEP systems such as Microsoft SteamInsight CEP [57] and IBM InfoSphere Streams [105] are also available in the market. The Stream project has been extended to form the Coral8 CEP which was later merged with Aleri CEP. The Coral8 system also provides a graphical environment for developing CEP applications using the Coral8 Studio. Both the Coral8 and Aleri CEP support centralized as well as clustered deployment. The Apama CEP [106] has been ranked as the best CEP in a 25 criteria (such as architecture, stream handling, pricing and market presence) survey conducted by Forrester Wave [107].

Table 3.1 shows various CEP operators which are supported by some of the systems that have been discussed earlier. This table reuses some of the results presented by Cugola *et al.* in [24]. Please note that this table does not include all the available operators

supported by the CEP systems. The ✓ symbol indicates that the respective operator support is present while the ✗ symbol indicates that the operator support is absent. From this table, it is evident that Apache Flink and Apache Siddhi support the most type of CEP operators.

3.1.3 CEP, Cloud, and IoT

In 2016, Higashino proposed the idea of CEP-as-a-Service (CEPaaS) in his Ph.D. dissertation [112]. The goal is to leverage the advantages of SaaS to provide CEPaaS so that there is no-upfront charges and maintenance cost is low. He proposed Attributed Graph Rewriting for Complex Event Processing (AGeCEP) as a language agnostic technique to model the CQL queries. To support his proposition for CEPaaS, Higashino designed a simulator called CEPSim that runs on top of the CloudSim simulator [113] [114]. CloudSim is a popular cloud simulator written in Java which can effectively model a public, private or hybrid cloud. It allows the users to create a data-center, cloudlet, and broker in addition to defining different policies as shown in Figure 3.2. The CEPSim module cre-

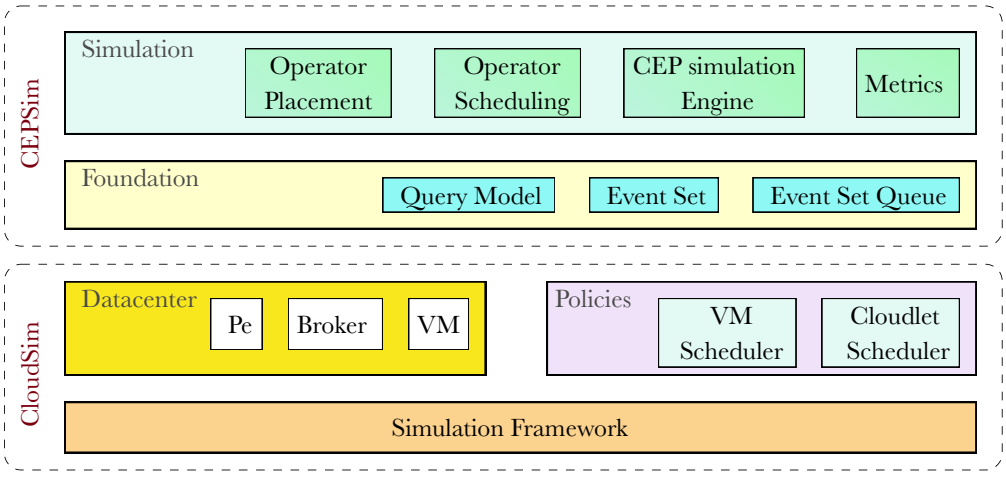


Figure 3.2: The architecture of CEPSim [111]

Table 3.1: Type of operators supported in various systems

Engine Name	Selection	Projection	Renaming	Conjunction	Disjunction	Negation	Sequence	Sliding window	Batch Window	Join	Union	Intersect
TelegraphCQ [93]	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	✓
Ode [87]	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗
Samos [88]	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗
Snoop [89]	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗
Stream [97]	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓
Aurora [23]	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✗
Gigascope [96]	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗
Cayuga [98]	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗
Amit [108]	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓	✗	✗
SASE [29]	✓	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗	✗
SASE+ [54]	✓	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗	✗
TESLA [109]	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
Aleri [106]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗
StreamBase [110]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Oracle CEP [104]	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓
Esper [56]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
TIBCO BE [19]	✓	✗	✗	✓	✓	✓	✓	✓	✗	✓	✓	✗
Flink [17]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Siddhi [10]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

ates a query model and supports the operator placement and the operator scheduling for performing the CEP simulation. It also provides the mechanism to compute various CEP specific metrics for the performance evaluation. A major limitation of CEPSim is that it does not have single and multiple query optimization mechanisms and assumes that a submitted query is already optimized. Another limitation is that it only supports the scenarios in which the query does not fail at runtime. It is important to mention that our work compares the performance of the edge-based mobile CEP with state-of-the-art CEPaaS system considered as a baseline system.

A few cloud providers have begun to provide support for complex event processing as-a-service. However, multiple systems need to be combined with one another to achieve this service. For example, the Amazon Kinesis [115] can be used to collect the health sensor data streams, which can be forwarded to Amazon Lambda [11] and can be processed using Apache Flink CEP deployed on the cloud [17]. Recently, IBM has also started to provide a CEP service on the cloud using IBM Watson IoT Server [70] using IBM Bluemix IoT server [116]. The CEP toolkit can be used by IBM InfoSphere [105] or the Node-RED CEP [117] running on top of IBM Bluemix. Node-RED CEP is designed using the Node-RED platform which is a programming tool having a browser-based editor to create flows. The Node-RED is built using *Node.js* and it is used by IBM Bluemix as a boilerplate application. Many cloud-based SP platforms such as TimeStream [118] and StreamCloud [119] are also being considered for the CEP services. To the best of our knowledge, the WSO_2 IoT server (discussed earlier in Section 2.7.1) is the only open-source IoT platform which provides an out-of-the-box support for CEPaaS [68]. For this reason, we have used the WSO_2 IoT server in our prototype system implementation.

The Mobile Edge Computing (MEC) concept leverage the mobile edge device to provide the cloud computing capabilities [120]. Similar to the cloudlet [121], the resources

on the edge device such as computing, networking, and storage are used to perform the delay sensitive tasks on mobile edge and the heavyweight tasks on the back-end server [122]. The proposed edge based CEP technique is similar to the mobile edge computing methodology.

3.2 CEP for Smart Buildings and Smart Homes

Complex Event Processing is often used in sensor-based smart buildings and homes. A representative set of existing work is presented. Chandrashekar *et al.* [123] have used a CEP engine to detect complex events for a smart home. The proposed architecture consists of various smart home sensors, a web interface, and a centralized CEP engine. Only the general architecture is given in this paper and no prototype implementation has been reported. Juan *et al.* [8] have integrated Service Oriented Architecture (SOA), IoT platform and CEP engine to provide a smart home solution. As shown in Figure 3.3, the system architecture consists of various sensors installed at home, an IoT server, an Enterprise Service Bus (ESB), a CEP adapter and an Esper CEP engine [56].

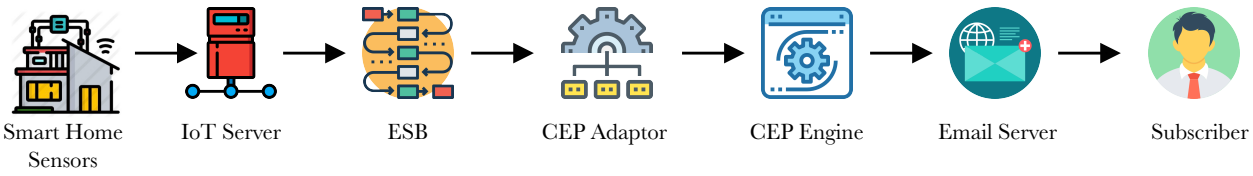


Figure 3.3: The architecture of a smart home system as proposed in [111]

The authors have used *Xively* [124] as an IoT server and *Mule* [125] as an ESB. As shown in Figure 3.3, various sensors in a smart home send the data streams to the *Xively* IoT server which forwards them to a *Mule* ESB using HTTP. The ESB normalizes the sensor data streams to a common format used by the CEP engine. Then, these events are forwarded to an Esper CEP engine where complex events are detected using various CEP

patterns written using Esper EPL. As soon as the complex events are detected, alerts are sent to an event receiver (such as an email server) which sends the alert to all subscribers. The various use cases such as *Fire Pattern*, *Irresponsible Stove*, *Power Failure* and *Irresponsible Television* are considered. Each of these use cases has a pattern query intended for finding a complex event.

3.3 CEP for Remote Patient Monitoring

A remote patient monitoring system is a smart health care solution the popularity of which is growing rapidly. It provides an economical solution for patient monitoring by attaching sensors to a remote patient and processing the sensor data to determine impending health problems and informing the healthcare professionals. A representative set of existing research on remote patient monitoring is presented in this section. Kamel and George have proposed a Remote Patient Tracking and Monitoring (RPTM) system in [126]. In their system, medical sensors send health sensor data streams to an Android-based mobile device, which are then encrypted using the Advanced Encryption Standard (AES) mechanism and forwarded to a General Packet Radio Services (GPRS) server for detecting complex events. The health signals are sent periodically to the server. The mobile device and the hospital server communicate with one another using a third party. The various vital signs of the patient are persisted in an SQLite database in an Android device. The major disadvantage of this method is that it does not work in a real-time fashion, as initially the data streams are persisted to an SQLite database and then SQL queries are applied to find the anomalies. Stipkovic *et al.* [127] developed a mobile CEP prototype system using an unofficial port of the Esper CEP engine on Android called *Esper-Android* [128]. Their system consists of 3 layers: an event source layer, an event handling layer and an event processing layer. The event source layer receives various

health data streams and forwards them to an event processing layer where the events are pre-processed, filtered and co-related to generate complex events. The detected complex events are forwarded to the event handling layer for visualization. Unfortunately, the project is incomplete as support for *Esper-Android* has been discontinued due to third-party library dependencies not supported by Android [128]. Banos *et al.* have built a ubiquitous RPM system called *PhysioDroid* [129]. Their system consists of wearable monitoring devices, a mobile device such as a smartphone and a remote persistent storage system for analytics. The mobile device acts as an edge gateway to collect and upload sensor data to the remote persistent storage. The authors used a static interval-based approach which works on interval bounds for each physiological parameter. No CEP engine or any other real-time analytic approach is used for detection of various health alerts. This paper also lacks a performance analysis for the proposed approach.

Vaidehi *et al.* in [131] and [132] have described a CEP-based Remote Health Monitoring System (CRHMS), that can receive various biological parameters such as heart rate, respiration rate, and blood pressure using *Zephyr Bio Harness* sensors. The location of the patient is accessed by using the GPS sensor. These sensor data streams are sent to an Android phone which forwards them to the hospital CEP server which consists of a JBoss Drools Fusion CEP Engine [55] for complex event detection. The proposed system has been tested using synthetic data at an arrival rate of 25 events/second which is not inline

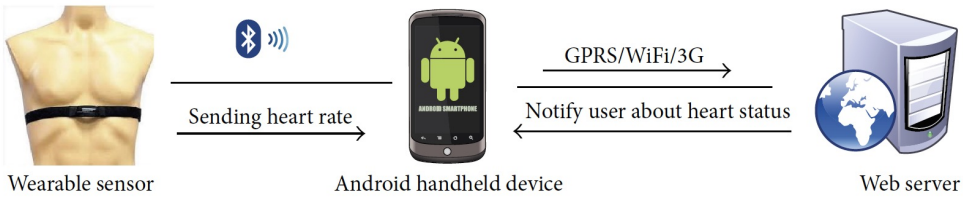


Figure 3.4: System architecture for RPM system [130]

which the most of health sensors. Kakria *et al.* [130] have developed a server-based RPM system for monitoring cardiac issues such as Tachycardia and Bradycardia. As shown in Figure 3.4, the authors have used a 3-tier architecture consisting of physiological sensors, an Android mobile device, and a web portal. An event forwarding gateway application runs on the Android device, which receives sensor data using the Bluetooth Low Energy (BLE) technology. This data is then transferred to a web server using the Extensible Messaging and Presence Protocol (XMPP) and alerts are predicted using fuzzy logic. No complex event processing system is used in this research and the various alerts are generated using the threshold-based approach in which the upper and lower thresholds for various physiological parameters are defined.

Lam and Haugen, in [133], have designed a state-machine-based CEP model using the ThingML modeling language. They reported that a CEP model built using the ThingML [134] requires a small memory and could be efficient for embedded devices like Arduino and Raspberry-Pi. However, the proposed model was only tested on a computer workstation having a large resource pool. Also, during experimentation, it was revealed that the model built using ThingML does not outperform other well-known CEP engines. However, the authors claim that the proposed model is expressive in writing state machine-based CQL queries. As the CEP system requires a full-fledged DSMS and CQL support, much work needs to be done for this model in order to be considered appropriate as an embedded CEP for mobile devices.

In 2018, Rodriguez *et al.* have made a Complex Event Processing for Heart Failure Prediction (CEP4HFP) system which uses predictive analytics on patient's historical database for predicting a possible heart stroke [135]. As shown in Figure 3.5, the authors used a 3-tier architecture consisting of monitoring, analysis, and visualization module. The health sensor data streams are collected using MySignals wearable sensors [136] and are sent to

an Arduino mega micro-controller. These health sensor data streams are then forwarded to a Raspberry-Pi device where data is formatted to a common format used at the server. The Raspberry-Pi device forwards the formatted health sensor data streams to a MongoDB NoSQL database [60] and to a Siddhi CEP engine running inside a WSO_2 Data Analytics Server (DAS) [68] employed on the server-side. Triggered alarms are sent to cardiologists using a web-based console and to caregivers using a mobile-based visualization module. It is important to mention here that WSO_2 DAS is another solution provided by WSO_2 which provides data analytics capabilities. In this research, we have used a WSO_2 IoT server which also contains a DAS server as the analytics tier as explained in Chapter 2. Their system can be considered as a CEPaaS system as the complex event processing is done on the server side as the Raspberry-Pi device is used as a gateway agent. This approach does not involve the patient's device enrollment which is required to uniquely

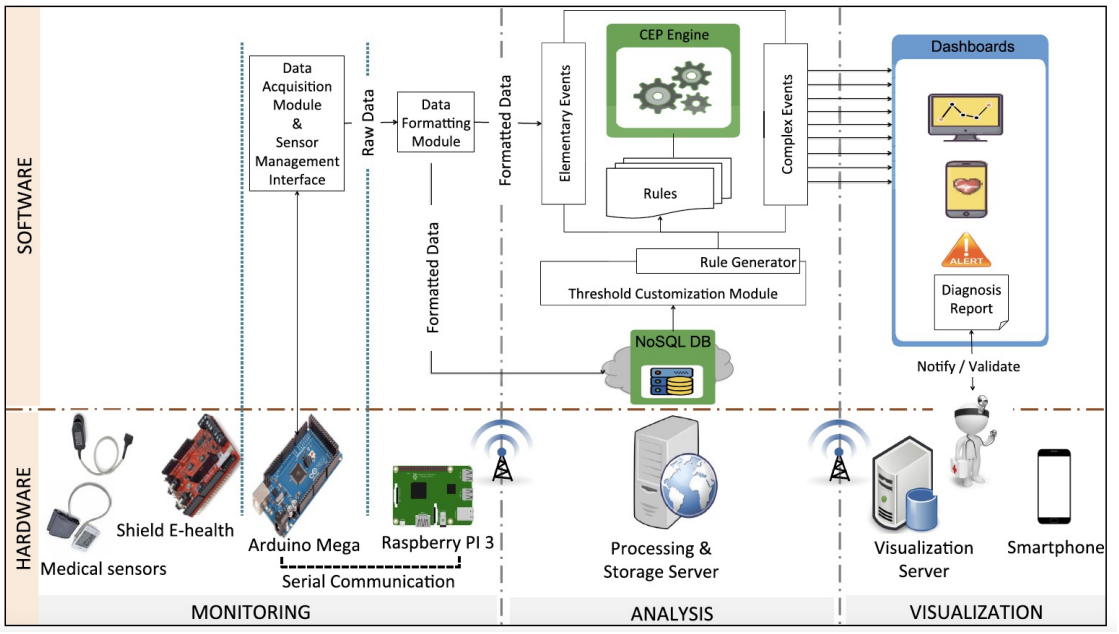


Figure 3.5: System architecture for RPM system [135]

identify the device at the server side for security purposes. Also, no authentication and authorization protocols have been followed to ensure the safety of the patient's personal data. Another common criticism is the fact that patients would have to carry two extra devices i.e. an Arduino board and a Raspberry-Pi for using this service. However, our approach involves a single mobile device which is relatively easy to carry by the patient and does not require any wired connections.

In another work done by Mohammed et al. in [137], the authors have proposed an RPM system by building an Android-based Electrocardiogram (ECG) monitoring application which uses an IOIO-OTG micro-controller [138] to collect the ECG signals and upload them on the cloud using a Filezilla server [139]. The system architecture consist of three layers: hardware layer, application layer, and cloud layer. The hardware layer contains IOIO micro-controller and various health sensors to collect the sensor data streams. Further, these sensor data streams are sent to the application layer on the Android device. The authors have used sensor sampling methods to minimize the power consumption of the mobile device. This application is made using Model View Controller (MVC) design pattern [140]. In the proposed methodology, the data is not sent in a real-time fashion as the sensor data streams are first stored into a Secure Digital (SD) card and then uploaded to a cloud server. Also, the signal processing of the ECG signal is done on the hospital server side to find the impending health issues. However, for the edge computing-based proposed system described in this dissertation the complex event processing is performed continuously and the complex event signaling an impending health problem for an RPM system is immediately detected and sent to the back-end hospital server.

In [141] and [142], Kharel *et al.* have proposed a smart healthcare monitoring system using the fog computing technique. As shown in Figure 3.6, the system architecture consists of three layers: an edge layer, a fog layer, and a cloud layer. The edge layer fur-

ther consists of multiple health sensor devices connected to a Raspberry-Pi board using a ZigBee, bluetooth, or 2G/3G connection. Further, the Raspberry-Pi board is connected to a Long Range Wide Area Network (LoRaWAN) [143] gateway which is connected to another LoRaWAN gateway at the fog layer.

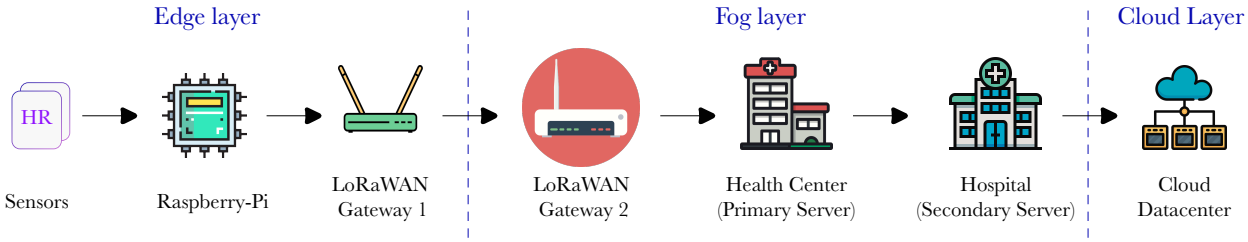


Figure 3.6: A simplified view of a system architecture used in [141]

The authors have leveraged a hierarchal fog layer approach in which the health center is considered as a primary fog server and the hospital is considered as a secondary fog server as shown in Figure 3.6. Hence, from the edge device's perspective, the health center can be considered as a fog node, albeit the hospital server can be considered as a fog node form the health center's perspective. The advantage of this approach is that the Internet connectivity is not required for remote patient monitoring, as the data transfer is done by the local LoRaWAN gateway. The authors have conducted various experiments for both indoor and outdoor scenarios to test the effectiveness of their prototype. Figure 3.7 shows the Google map view of the buildings involved in the outdoor experiment. The health sensors send the sensor data streams to the Raspberry-Pi present in first building which is 2 Km away from the LoRaWAN gateway deployed in the second building. It was found that the health sensor data was transferred at a Signal to Noise Ratio (SNR) of -13.9 Decibel milliwatts (dBm) in this case. Thus, the prototype system can be deployed in remote areas where the Internet connectivity is not available. However, this approach restrains the patient to stay within the LoRaWAN connectivity zone of 2 Km, which is

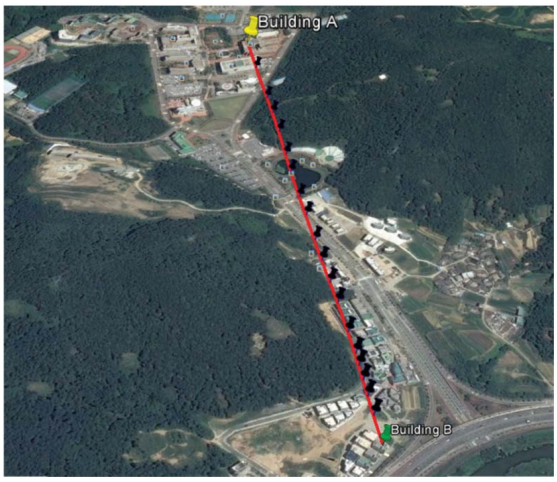


Figure 3.7: An outdoor experiment conducted in [141]

a major limitation. The approach proposed in this thesis does not have this issue, as the complete CEP is performed in the mobile device and the data transfer from health sensors is done by a local bluetooth connection.

Another work reported in [144] describes a pulse monitoring system which also used the Android application as an edge gateway and sends data to a web portal for analysis and visualization. A similar approach is described in [145] which uses an Android device as a gateway agent. Another research in [146] and [147] employed an IoT-based approach to process the health sensor data streams on the cloud. The authors have used an *Intel Galileo Gen 2* IoT agent to collect the sensor data streams from the mobile device and forward these to an IoT server deployed on the cloud. However, the authors have not used any real-time analytics system as the computation is done by a batch processing based Hadoop system. Further, no performance analysis is done in any of these two papers to demonstrate the effectiveness of the technique.

Woodbridge *et al.* have proposed an RPM system for congestive heart failure named as *Wanda* [148]. The *Wanda* has a three-tier architecture in which the first tier consists

of various health sensors that transmit the health sensor data streams to the second tier consisting of a web server. The third tier uses database servers to persist the health sensor data streams and perform the analysis using linear regression. Further, this system is not a real-time system and does not involve any CEP engine. However, as the authors are predicting a heart stroke, performing batch analysis seems to be appropriate. In 2017, Naddeo *el. al* [149] have proposed a real-time m-health monitoring system. Their system consists of an Android application which receives various physiological sensors using the *Zephyr Bioharness BH3* sensor and performs noise filtering using various high-pass and low-pass filters. This filtered data is sent by an Android application to a remote Personal Health Record (PHR) server for analysis and visualization. A major shortcoming of this paper is that it does not describe the real-time analysis technique required for this system. Another similar work is reported in [150] where the authors proposed to integrate the CEP engine and the IoT server for smart healthcare. This paper is primarily focused on the key benefits of using CEP on the cloud. However, no actual system is designed and no performance analysis is done.

In 2018, Chisanga *et al.* proposed a telemonitoring system for remotely monitoring cardiac patients in his Ph.D. dissertation [151]. As shown in Figure 3.8 the proposed system consists of 5 modules: health sensors, mobile application, Machine-to-Machine (M2M) gateway, M2M server and Electronic Health Record (EHR) [152]. Initially, the

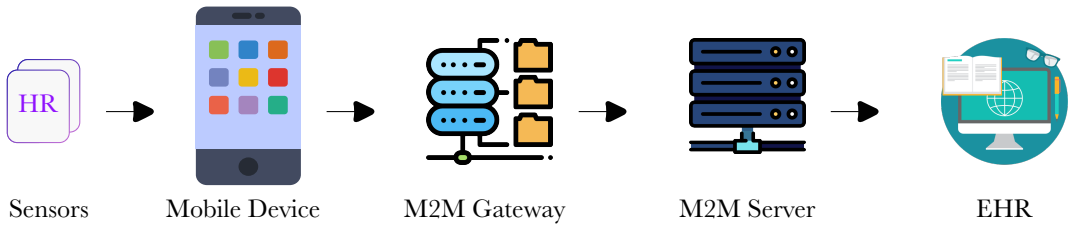


Figure 3.8: System architecture used in [151] and [152]

health sensor data streams are collected using the *Zephyr HxM Heart Rate Monitor* and sent to an Android smartphone using bluetooth. The mobile device has a gateway application built using Android API version 15 (Ice-Cream Sandwich). The M2M mobile application forwards the health sensor data streams to the M2M gateway which are then forwarded to the M2M server. Both the M2M gateway and the M2M server are separate instances of the OpenMTC platform [153], which provides the distributed middle-ware capabilities. The EHR consists of a web-based application built using PHP and a MySQL database for persisting the records. The patient enrollment is required to use this system and RBAC is used to provide the authorization. The disadvantage of the proposed approach is that no real-time monitoring technique is used and the complete event detection is done on a remote server using various SQL queries.

3.4 Concluding Remarks

The major limitation of the aforementioned techniques described in Section 3.2 and Section 3.3 is that the mobile device is simply used as a forwarding agent while CEP is done on a remote server and thus the techniques rely on the availability of network connectivity at all times.

In smart home and smart buildings, none of the existing works performs any alert generation on an embedded device using the CEP technique. In the case of RPM, none of these existing works can raise an alarm notifying a health problem for the patient when the network becomes unavailable. This problem is effectively addressed by the methodology and system proposed in Chapter 4.

Chapter 4

System Architecture and Prototype

This chapter discusses the two different system architectures used for performing CEP for the remote patient monitoring use case. These system architectures are Server Complex Event Processing (SCEP) and MCEP which are discussed in Section 4.1 and Section 4.2 respectively. Section 4.3 provides the system prototype implementation and experimental setup details. Then Section 4.4 and Section 4.5 discuss the implementation details for the sensor simulator and the timekeeper respectively.

4.1 Server CEP System

As shown in Figure 4.1, the SCEP system architecture is three-tiered consisting of multiple sensors, a Mobile Device (MD), and an IoT Hospital Server (IHS). The mobile device along with the sensors comprise the edge system that communicates with the centralized back-end server. Multiple bluetooth and WiFi enabled wireless sensors can be used by the sensor-based system which can forward the sensor data to an Android or iOS device. For example, in a remote patient monitoring system, the sensors can be wearable health sensors worn by the patient. Such cheap and efficient sensors are provided by Cooking Hacks [136]. Some other commercial health monitoring sensors can be used such as *Zeo Sleep Monitor* [154] which monitors sleep disorders and *ViSiMobile* [155] which can measure ECG, HR, Arterial Oxygen Saturation (SpO2), skin temperature, etc. As shown in

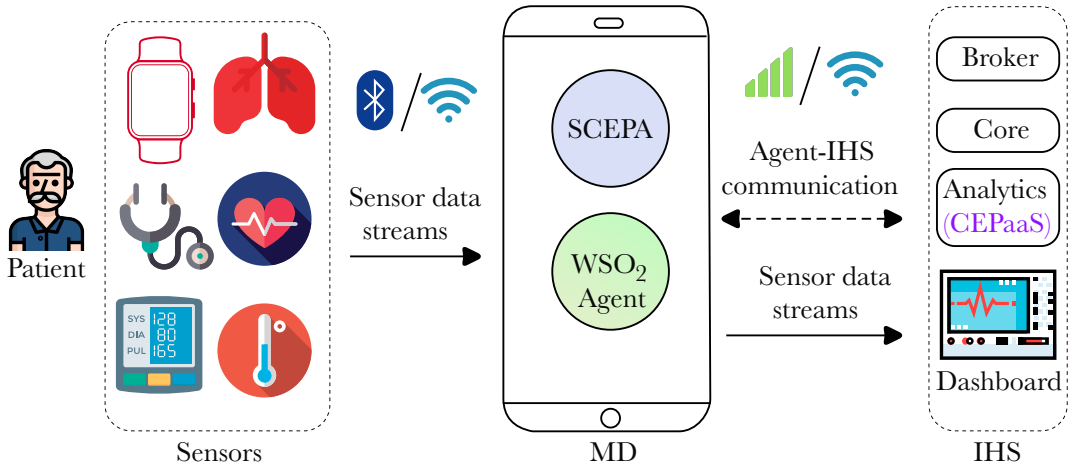


Figure 4.1: Server CEP system architecture

Figure 4.1, the multiple sensors send the sensor data streams to a mobile device which consists of a SCEP Application (SCEPA) and a WSO_2 agent gateway application. The server complex event processing application forwards the health sensor data streams to the IHS. Communication between the sensors and the mobile device is done using bluetooth or WiFi whereas data transmission between the mobile device and the IHS is performed using either a cellular or a WiFi connection. The architecture shown in Figure 4.1 can be used in other use cases such as smart buildings and smart homes. In the smart building use case, the wearable health sensors can be replaced by wired/wireless sensors deployed in a smart building such as room temperature sensors and light intensity sensors. In such a case, the mobile device can be replaced by a local server or a Raspberry-Pi board depending upon the workload.

4.1.1 Interaction Among Various Components

Figure 4.2 shows a high-level sequence diagram for various components in the SCEP system. As shown in the strict fragment, initially the mobile device is registered with the IoT server using the WSO_2 agent application. Once device registration is successful, a

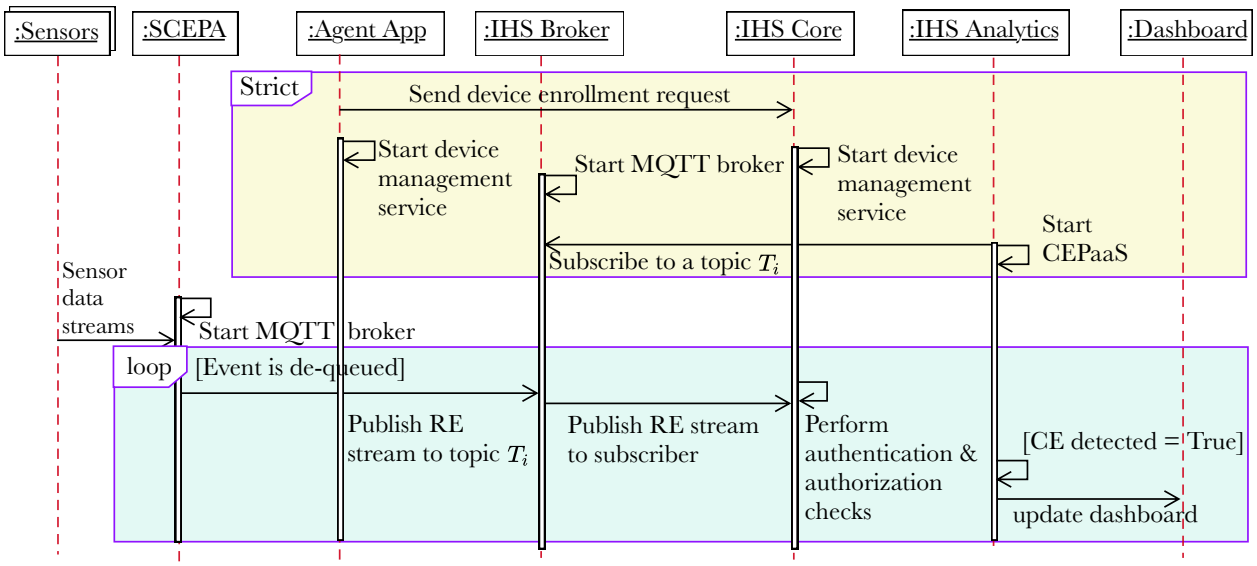


Figure 4.2: Sequence diagram showing a high-level interaction of the components in the SCEP system

device management service is started on the IHS core and the WSO_2 agent application. This service is responsible for various functions such as authenticating the device periodically, managing phone calls, sharing device information, sharing device location and sending the collected information to the IHS in an encrypted manner. The MQTT broker is started on the IHS broker so that RE data streams can be sent from the SCEPA running on MD (publisher) to CEPaaS running on the hospital server (subscriber). The Event Listening Service (ELS) that starts running inside the IHS analytics tier subscribes to a topic: T_i where i represents a unique patient id in an RPM system. Device registration is a one-time process. If the device is rebooted, it will be authenticated using device credential certificates on the server consisting of device information such as International Mobile Equipment Identity (IMEI) number. After this one-time setup is complete (as shown in the strict fragment), the SCEP application starts the MQTT service. Further, SCEPA opens an input TCP sockets (one for each sensor) to receive sensor data streams.

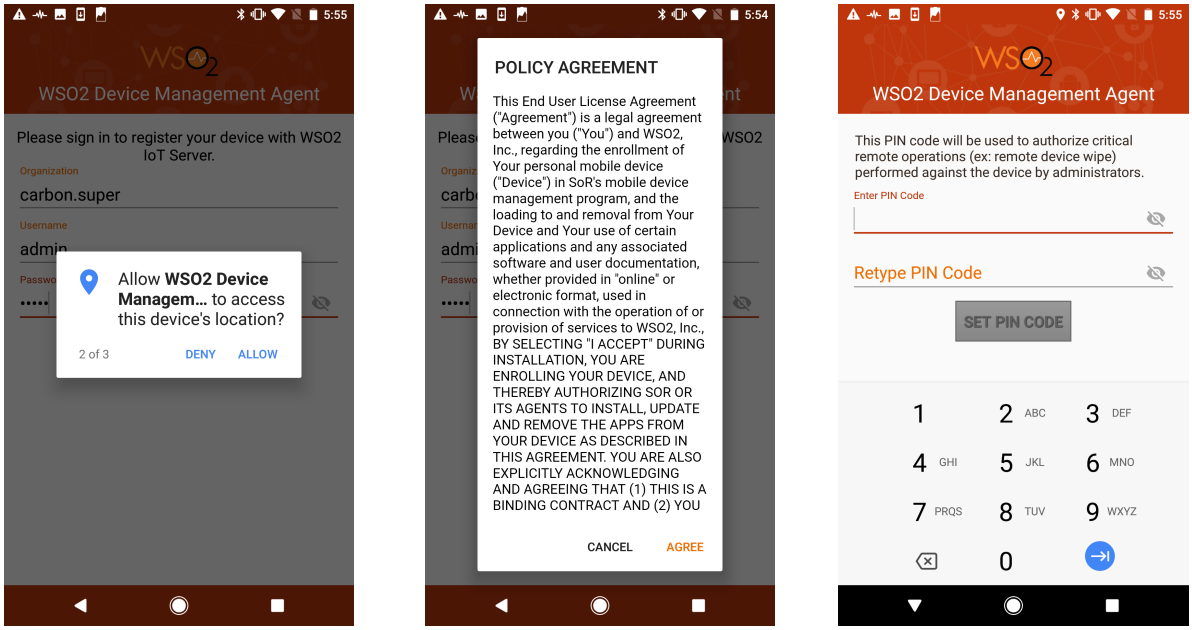
Multiple worker threads running in parallel append the received sensor data streams to a thread-safe linked-blocking queue [41] from which sensor data streams are published to an MQTT broker in a particular topic: T_i. These sensor data streams are received by the CEPaaS running at the IHS analytics tier and the detected complex events are updated to the dashboard using a Java Management eXtensions (JMX) agent [156] to notify the hospital staff. These complex events can also be persisted on a database for long-term historical analytics. The device enrollment which is an initial step for mobile device and IoT server communication (as shown in the strict fragment of Figure 4.2) is discussed next.

4.1.2 Device Enrollment/Registration Process

A mobile device has to be first registered with the back-end server in order to use CEPaaS. For a successful registration, the IoT server deployed at the back-end should be up and running and the server's IP and all relevant port numbers need to be known apriori. Figure 4.3(a) shows the registration page where the user enters his/her organization name, username, and password for the login process. Figure 4.3(b) shows a retrieved policy agreement from the IoT server and Figure 4.3(c) shows the user PIN code setup screen which is used to secure the user from critical remote operations performed by the IoT server administrator. After successful enrollment, the initial login is done to use the service which is discussed next.

4.1.3 An Initial Login into SCEP Application

After successful enrollment, an agent (on the mobile device) periodically communicates with the IoT server in the background. In our case, as the IP of the IoT server is not static, thus a login functionality has been added to the server CEP application. After the login is done, the user can start using the SCEP application (on the mobile device) which will further use CEPaaS running on the IoT server. The essential components of the SCEP and



(a) (b) (c)

Figure 4.3: (a) Registration page (b) Accepting policy agreement (c) Setting secure PIN

CEPaaS applications are discussed in Section 4.1.4 and Section 4.1.6 respectively.

4.1.4 Components of the SCEP Application

Figure 4.4 shows the components of SCEPA which is used to forward the raw sensor data streams from the mobile device to the IoT server. The various components that are stacked over one another represent multiple parallel instances of that component and a solid line represents multiple parallel sensor data streams. The various data streams are received by the TCP socket objects (one socket for each sensor) and appended to a thread-safe linked-blocking queue by a producer thread (Worker 1). A dedicated thread-safe queue is used for each sensor data stream. Further, the dequeue worker (Worker 2) retrieves the sensor data stream from a queue and sends it to the IHS using the MQTT service running on the

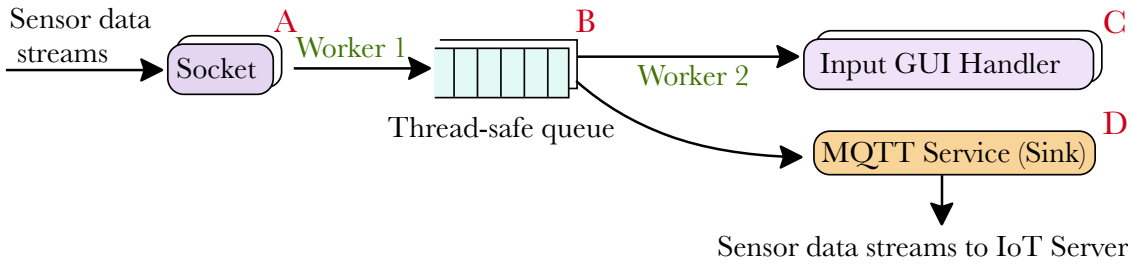


Figure 4.4: Various components of the SCEP application

mobile device. This MQTT service forwards the sensor data streams to the back-end IoT server as per the selected QoS. Please note that the MQTT service also has its own queues for enabling the persistent session, and if the QoS ≥ 1 is selected, the sensor data stream tuples are temporarily persisted in case the back-end server goes offline.

4.1.5 IoT Hospital Server

The hospital server consists of three components as shown in Figure 4.5.

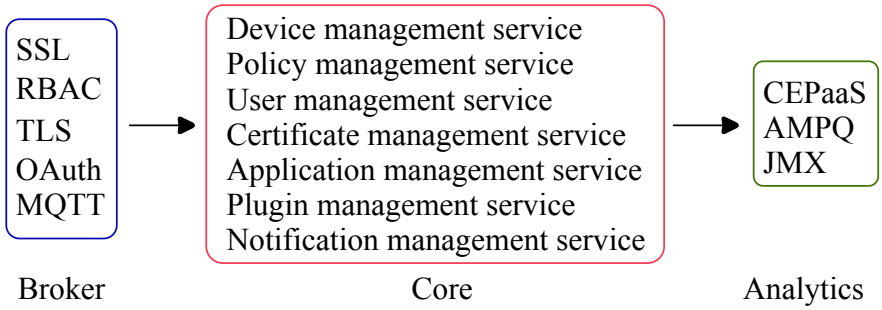


Figure 4.5: Key components of IHS

1. Broker: The broker is responsible for providing authorization, authentication, and security features as discussed in Section 2.7.1.

2. Core: The core provides various services (see Figure 4.5) which have already been discussed in Section 2.7.1.

3. Analytics: CEPaaS runs inside the analytics tier and detects the complex events. ActiveMQ is used as the JMS queue for queuing the raw sensor events before sending them to the CEP engine [64]. The JMX agent [156] is responsible to publish various system specific and CEP specific metrics on the *JConsole* [157].

4.1.6 Components of CEP-as-a-Service

Figure 4.6 shows the various components of CEPaaS which is running on the IoT server. The different components have been labeled with a superscript from *A* to *M*, in the order of event processing. A solid line represents multiple parallel sensor data streams whereas a dashed line represents a single sensor data stream. Each component which is shown as a box in Figure 4.6 receives an input data stream and emits an output data stream as a result of the operation performed by that component. Thus, various output streams must be defined before starting the service such that an output stream contains all the attributes which have been emitted by its predecessor component. When an attribute is added or removed from an input data stream (RE.v.1 for example) as a result of an operation done

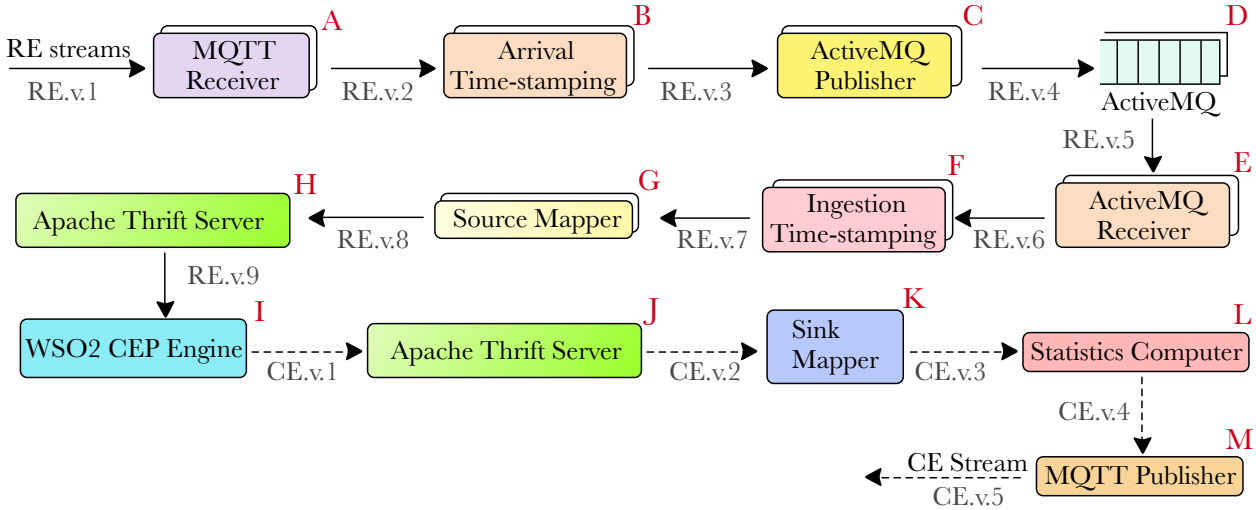


Figure 4.6: Components of the CEPaaS module deployed on the IoT server

by a component (MQTT receiver in this case), then the output stream can be referred to as a stream having a different version (RE.v.2 in this case). As shown in Figure 4.6, a raw stream has 9 versions (RE.v.1 to RE.v.9) whereas a complex event stream has 5 versions (CE.v.1 to CE.v.5). A brief discussion of each component is provided next in the order of event processing.

(A) MQTT Receiver: It receives a raw sensor stream on a particular topic after validating the content using the default/custom content validator. Multiple instances of the MQTT receivers (one for each sensor stream) receive raw sensor data streams in parallel.

(B) Arrival time-stamping: Multiple arrival time-stamping components run in parallel. Each component receives a particular stream and appends a system generated nanosecond precision time-stamps to indicate the arrival time.

(C) ActiveMQ publisher: An ActiveMQ [64] is used as a JMS queue [38]. The ActiveMQ publisher is responsible for sending the messages to a particular brokered-queue managed by an ActiveMQ broker. ActiveMQ supports both topics and brokered-queues to transfer messages, but we are using the brokered-queue in this implementation. For setting a JMS publisher, the various adapter properties such as JMS destination type, JMS destination name, JMS factory name, JMS provider Uniform Resource Locator (URL), JMS Connection Factory name, Java Naming and Directory Interface (JNDI) name, a username and a password need to be defined as per ActiveMQ server configurations which is running on the IoT server.

(D) ActiveMQ: The ApacheMQ provides support for Advanced Message Queuing Protocol (AMQP), Streaming Text Oriented Message Protocol (STOMP), MQTT, OpenWire [158] and other protocols. The size of each ActiveMQ queue is set to a maxi-

mum of 2 GB (restrained by the maximum value of an integer). As shown in Figure 4.7, a web-based GUI can be used to view the list of all ActiveMQ queues, topics and a number of messages enqueued/de-queued in each of the queue/topic.

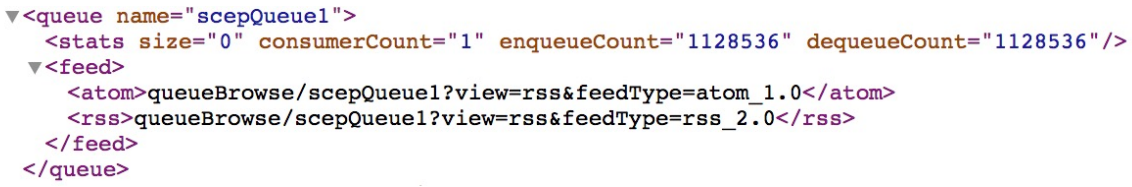


Figure 4.7: A screen-shot of the web GUI for ActiveMQ

(E) ActiveMQ subscriber: It is used to receive the sensor data stream events from a particular ActiveMQ queue. A subscriber subscribes to a particular queue using a unique queue name identifier and then forward the received sensor tuples as an output sensor data stream (RE.v.6 in this case).

(F) Ingestion time-stamping: This module is used to append the CEP engine ingestion time-stamps using a nanosecond precision system clock, before sending the sensor data streams to the CEP engine. Multiple ingestion time components work in parallel to time-stamp each sensor stream.

(G) Source mapper: A CEP system supports various events formats such as XML, JSON, key-value pairs and HL7. The role of the source mapper is to convert the type of the sensor data stream event to the format required by the CEP engine.

(H) Apache thrift server: it is the binary communication protocol originally developed by Facebook [159]. It provides a Remote Procedure Call (RPC) framework to build the cross-platform services written in different frameworks and languages [160]. WSO_2 Data Analytics Server (DAS) running inside the analytics tier provides real-time, batch and predictive analytics by using the other services such as the CEP

engine and Apache Spark. Thus, the Apache thrift acts as a mediator to perform RPC on the CEP engine using the data bridge agent as shown in Figure 4.8. The various events received by the thrift server are temporarily stored in a queue. Setting a large queue size is recommended for high throughput, whereas a small queue size is recommended for low latency. Further, the marshaling and un-marshaling are done in batches by using the provided batch size. Data Bridging Agent (DBA) receives data on TCP port # 7611 and uses port # 7711 for SSL. Core pool can be configured to use the system cores to perform the computation. Other parameters such as *SocketTimeout*, *KeepAliveTimeInPool*, *BatchSize*, and *EvictionTimePeriod* should be configured appropriately as per performance requirements. The transport layer provides the abstraction to read and write to the network and the protocol layer provides a mapping of a data structure to a wire-format.

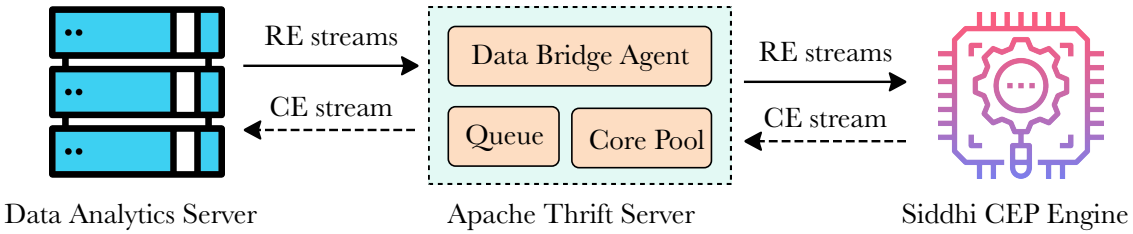


Figure 4.8: Thrift server

(I) CEP engine: It receives multiple sensor data streams and finds the complex events according to the CQL query which has been deployed. A single complex event stream, as shown by a dashed line, is sent to the sink mapper. The complex event detection time-stamping is done in the CEP engine.

(J) Apache thrift server: The detected complex events are sent back to the thrift server which are then sent back to the DAS for further processing.

(K) Sink mapper: The sink mapper converts the data type of the events in CEP stream to the type required by the event publisher.

(L) Statistics computer: It computes various CEP specific metrics such as average CEP latency and average CEP queuing latency by using the time-stamps taken by the IoT server.

(M) MQTT publisher: The MQTT broker component publishes the various streams to the event listener such as a dashboard, email, or a database.

4.1.7 Performance tuning for ActiveMQ

The WSO_2 IoT server provides a web GUI-based console for core and analytics tiers. The web console of the analytics tier can be used to configure various types of publishers and subscribers with one another to generate a streaming analytics pipeline [68]. It means that only the supported event publishers or event subscribers can be used and we have selected the ActiveMQ as a queue. The optimal case would be to use a thread-safe queue data structure such as a linked-blocking queue due to its broker-less nature and less latency. Unfortunately, the broker-less queue such as ZeroMQ [161] or a queue data-structure is not yet available in WSO_2 IoT server. Both WSO_2 IoT server and ActiveMQ use the OSGi framework to provide asynchronous messaging. The ActiveMQ uses a message broker Message Oriented Middleware (MOM) to manage, host, acknowledge and transfer messages. ActiveMQ has more latency compared to broker-less queues as a broker-less queue does not employ an additional component such as a broker, but instead uses a queue data structure to transfer messages [162]. In order to reduce the queuing latency incurred by ActiveMQ, some performance tuning steps have been performed, as explained next.

1. By default, the ActiveMQ broker persists all the incoming messages in a file-based database called as *KahaDB* [163] using a file-based cursor object which leads to additional delays. As ActiveMQ is based on the TCP protocol, so for every message, the acknowledgment is sent by the JMS receiver after which the message is deleted from *KahaDB*. For performance optimization, the persistence mode of ActiveMQ is turned off.

2. The in-memory storage of the messages is turned on, to reduce queuing latency. This is achieved by using a *VM cursor* object in which a reference to the message is held in-memory and passed to the dispatch queue to publish a message to the ActiveMQ listener [64].

3. The flow control mechanism has been turned off as it leads to additional queuing delays inside ActiveMQ.

Further, the WSO_2 IoT server uses OpenWire [158] as a default wire protocol to transfer data between the ActiveMQ publisher, ActiveMQ receiver and ActiveMQ broker. In our current implementation, the ActiveMQ uses Asynchronous JavaScript and XML (AJAX) [164] as a REpresentational State Transfer (REST) API connector [165] to send the messages to the *Jetty* server [166] at a specific Uniform Resource Identifier (URI) (tcp://0.0.0.0:61616 in this case). The JMS receiver adapter and the JMS publisher adapter running on the IoT server read and publish data respectively from this URI.

4.1.8 Device Management Dashboard

Figure 4.9 shows the IHS dashboard where an administrator can perform various operations like calling the patient, sending a text message using Google Cloud Messaging (GCM) [167] and viewing live location.

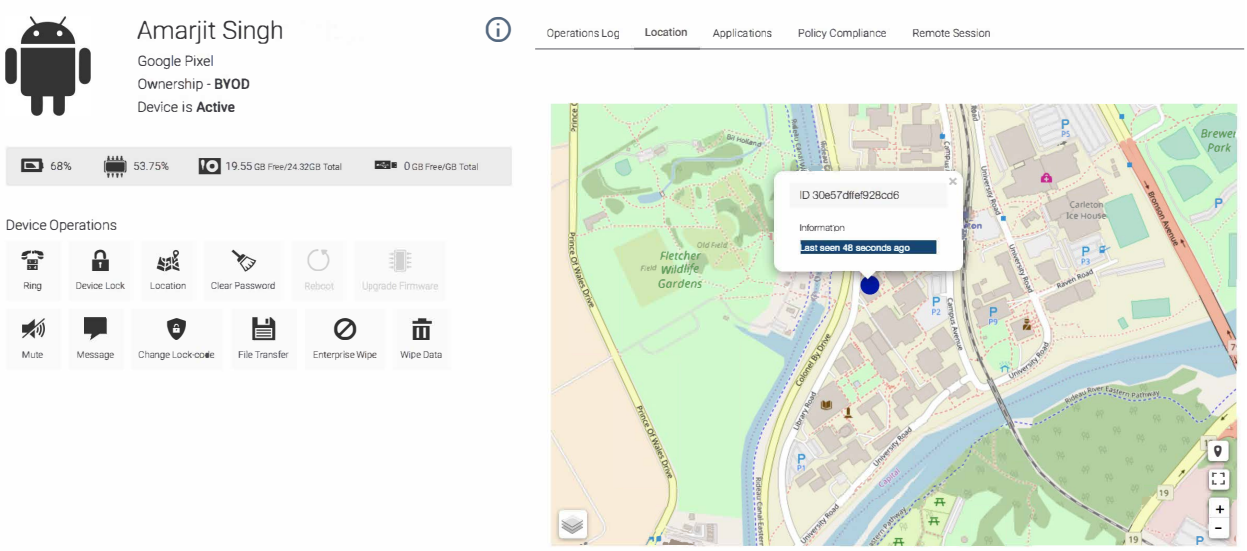


Figure 4.9: Administrator dashboard for a particular user

4.1.9 Setting up an Analytics Dashboard

We have used the DataDog dashboard for visualization which is used by a number of key companies such as Samsung, AT&T and FedEx [84]. Figure 4.10 shows the various components which need to be configured with one another to publish the system and CEP specific metrics to a web-based DataDog dashboard. The JMX agent [156] col-

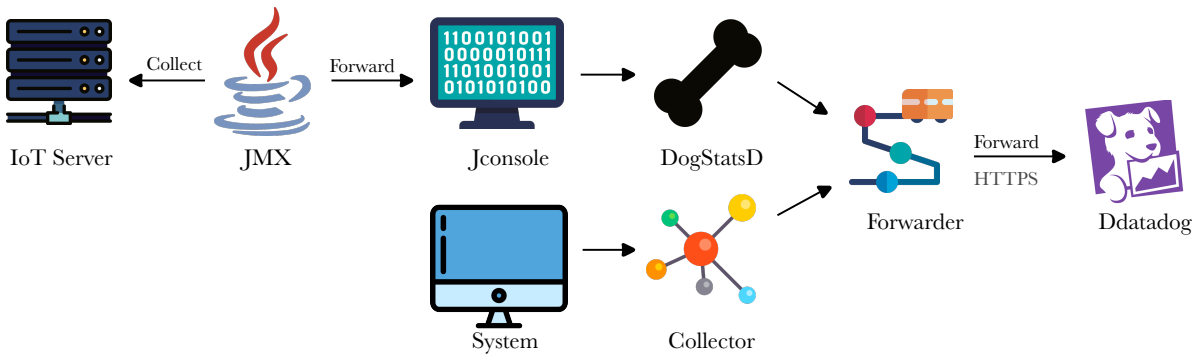


Figure 4.10: Dashboard setup

lects the CEP specific metrics from the IoT server and forwards them to the *JConslole* [157]. A valid API registration key is required to configure the *DogStatsd* agent with the DataDog web console. The *DogStatsd* agent [168] running on the IoT server collects the application-level metrics from the *JConslole* Managed Beans (MBeans) [169] using the *.yml* configuration file (a separate file for each application). The *Collector* component is responsible for collecting the system-level metrics such as system Central Processing Unit (CPU) usage, system memory usage, and system network consumption. Both types of metrics fetched by the *Collector* and *DogStatsD* components are sent to a *Forwarder* component using the User Datagram Protocol (UDP). The *Forwarder* component sends these metrics to a web-based DataDog dashboard using the Hypertext Transfer Protocol Secure (HTTPS) protocol. Figure 4.11 shows a screen-shot of the dashboard running on the IoT server displaying metrics such as average CPU utilization, average memory consumption, and average network usage. The dashboard setup process for the MCEP system is similar to the one for the SCEP system with each system having different configurations files for MBeans.

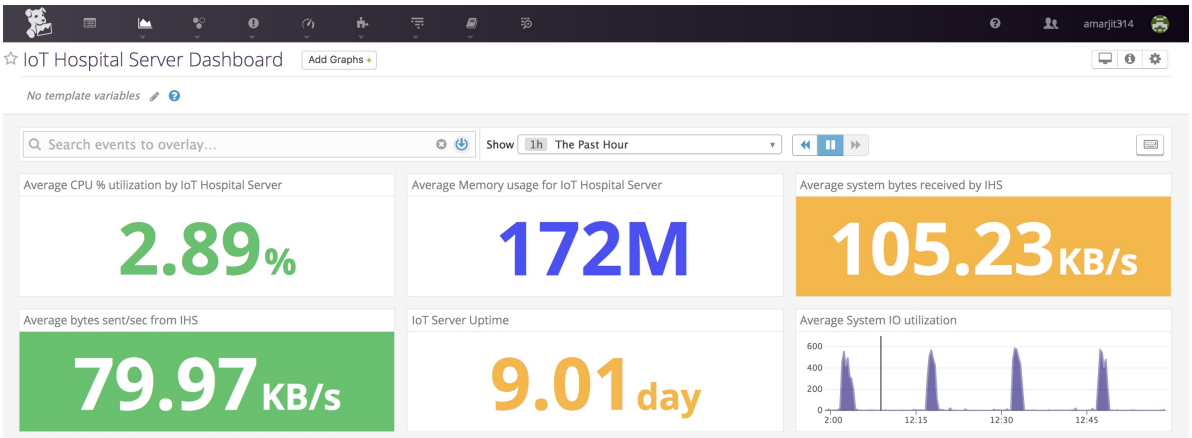


Figure 4.11: Screen-shot on the DataDog dashboard running at the IHS

4.2 Mobile CEP System

The mobile CEP system has been designed to perform complex event detection on the edge device using an embedded CEP engine that forwards the complex events to an IHS. Although the following discussion refers to the RPM use case, the MCEP architecture can be used in the context of other use cases as well. As shown in Figure 4.12, similar to the SCEP architecture, the MCEP architecture also consists of three components:

1. Sensors: For the RPM use case, various wearable health sensors such as an Apple watch, Glucometer sensor, and Pulse-oximeter are used.

2. Mobile device: The mobile device is used to perform complex event detection using the mobile CEP application and sends the detected complex events to the back-end hospital server.

3. IHS: An ELS running on the analytics tier receives the complex event alerts which are further sent to a DataDog dashboard to notify the hospital staff.

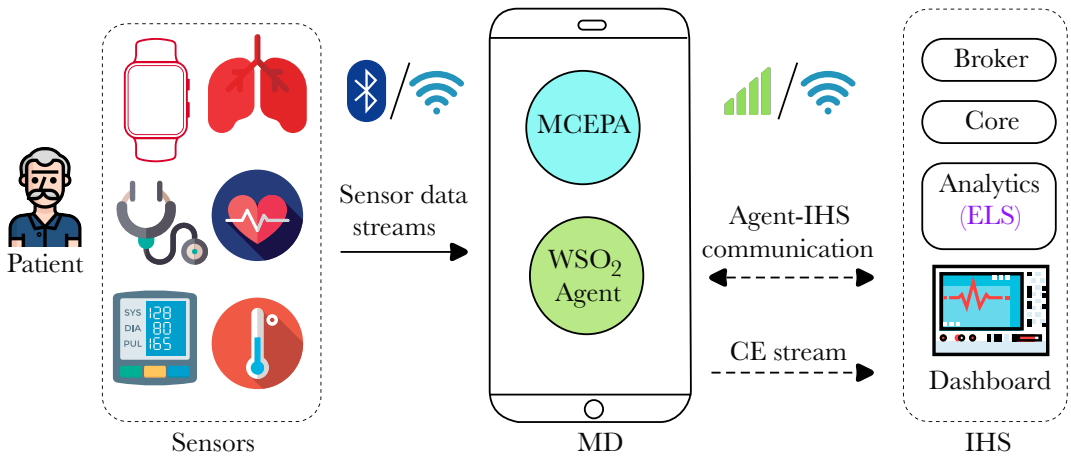


Figure 4.12: Mobile CEP system Architecture

The main difference between the MCEP system and the SCEP system is that all the complex events are detected on the mobile device instead of processing them on a centralized IoT server. Unlike the SCEP system, which has a CEPaaS running on the IHS, the MCEP system has an ELS running on the hospital server which is subscribed to listen to the complex events sent by the MCEP Application (MCEPA) running inside the mobile device. The sequence diagram in Figure 4.13 shows a high-level interaction among the various components shown in Figure 4.12. The device registration process, as shown in the strict fragment, is the same as the one described for the SCEP system in Section 4.1.1. Once this one-time configuration is complete, the patient can start the CEP service on his/her device which continues to run indefinitely. This service consumes raw sensor data streams from both wearable health sensors and embedded mobile sensors. Complex events are detected based on CQL queries programmed apriori by a health professional. As soon as a complex event is detected by the CEP engine running inside MCEPA, it is sent to

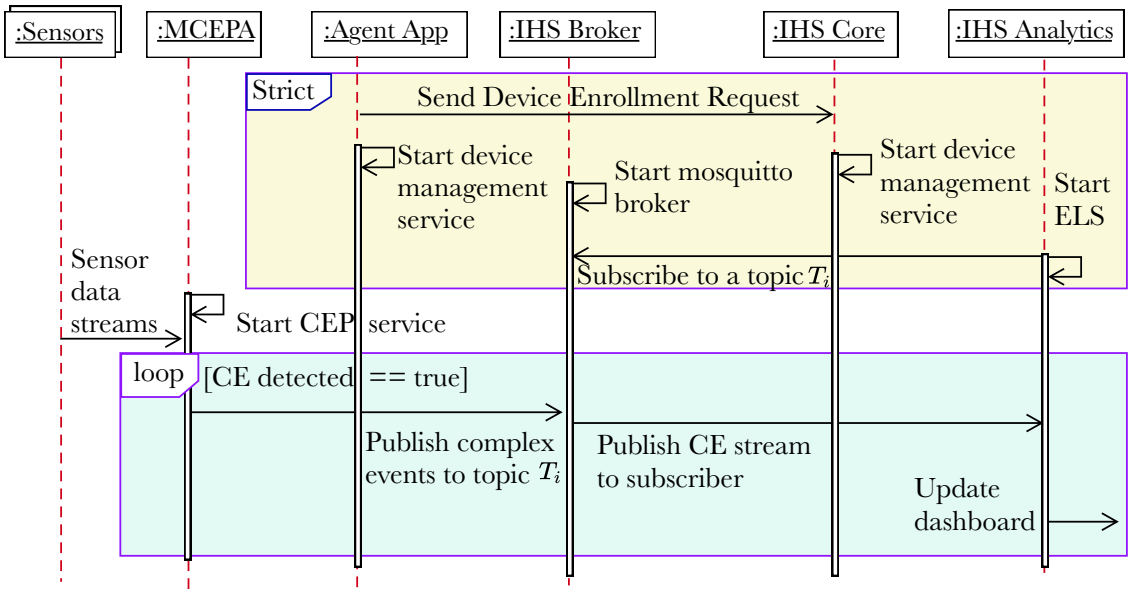


Figure 4.13: The sequence diagram for mobile CEP system

the MQTT broker (running inside the IHS broker) on a particular topic. The broker forwards the complex event stream to the subscriber: an ELS running in the IHS analytics. The event listening service sends the CE stream to a dashboard using a JMX agent for visualization.

4.2.1 Key Features of Mobile CEP

The key features of the proposed mobile CEP system include:

1. User enrollment and device management service: The patient can either enroll using Bring Your Own Device (BYOD) [170] or a Corporate Owned Personally Enabled (COPE) feature [171]. In BYOD, users provide their own device to receive the service, while COPE is an enterprise-based feature in which the enterprise (IoT server administrator in this case) has root level access to the device. Further using a dashboard, the IHS administrator can manage multiple mobile devices, enforce various policies, black-list, and white-list applications, encrypt device storage, configure WiFi, Virtual Private Network (VPN), and work-profile configurations.

2. Interactive GUI: Mobile CEP provides a GUI to enroll in IHS and view live data generated by the various sensors. It also shows application statistics such as the total number of complex events detected, average CEP latency in addition to local alarms generated by the engine. As the CEP system runs continuously, the GUI also provides a button to shut-down the CEP engine.

3. CEP and MQTT as Android-service: Both CEP and MQTT have been implemented as an Android-service so that they can run in the background even if the component that created it is destroyed. For the remote patient monitoring use case, it allows the patient to perform other operations such as playing games, dialing/receiving

phone calls and surfing the Internet along with the MCEP application running in the background.

4. Multi-tenant architecture: Users having different roles such as an administrator, doctor, or health assistant (each having different authorization level) can be added/removed in the IoT server.

4.2.2 Components of the Mobile CEP Application

Figure 4.14 shows the various components of the mobile CEP application. The components have been labeled with a superscript ranging from *A* to *J*, in the order of event flow. A solid line represents multiple parallel sensor data streams whereas a dashed line represents a single sensor data stream. Various components that have been shown as stacked over one another represent multiple parallel instances. The basic functionality of each component is explained next.

(A) Query repository: Prior to running the CEP engine, the appropriate query needs to be selected. Each query is written by a health professional and corresponds to a

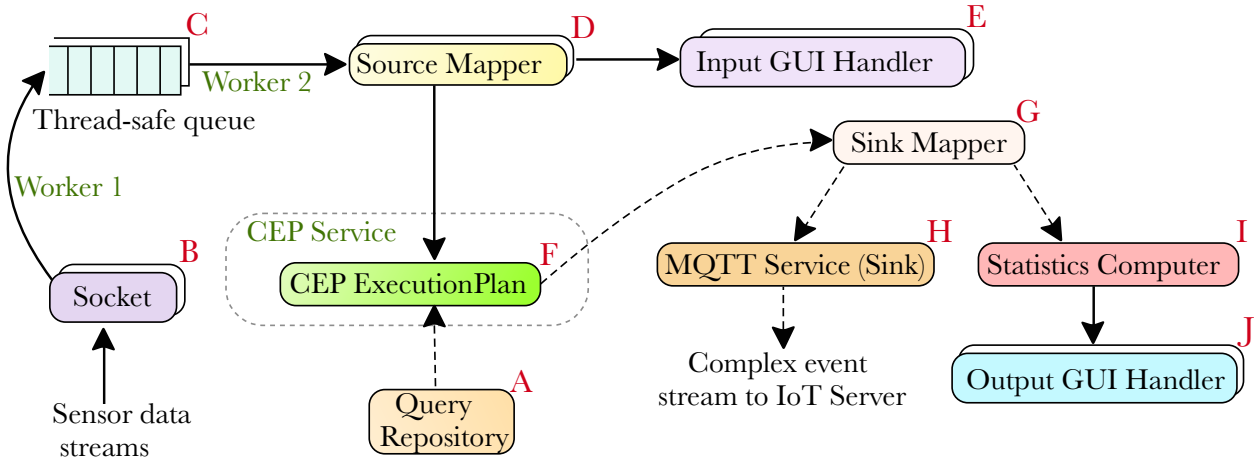


Figure 4.14: Components of mobile CEP application

particular disease such as CHF or Arrhythmia.

(B) Socket objects: Multiple TCP socket objects receive input data streams concurrently, de-serialize streams and parse them as JSON tuples. Further, an event arrival time is added to each tuple and these tuples are then appended to a thread-safe queue by the *Worker 1* thread in a First-In-First-Out (FIFO) order.

(C) Thread-safe queue: *Worker 2* is another thread which continuously checks if there is a tuple present in the queue. If so, it fetches the tuple from the head of the queue, then appends the event ingestion time in nanosecond precision using the system clock and sends it to the source mapper.

(D) Source mapper: The mobile CEP application can receive streams in multiple formats which are mapped to a common Siddhi event format using the source mapper. Multiple instances of the source mapper perform mapping in parallel.

(E) Input GUI handler: In Android, a handler provides communication between the main GUI thread and the other background threads. Multiple handler instances (one for each sensor) update live raw sensor data to the GUI in a parallel fashion.

(F) CEP execution plan: It consumes raw events from various source mappers in parallel and executes the CEP query to generate complex events in Siddhi event format which are forwarded to a sink mapper.

(G) Sink mapper: The sink mapper converts Siddhi events to JSON events which are then forwarded to a statistic computer and a sink such as the MQTT service.

(H) MQTT service: This service publishes the JSON complex event stream on a particular topic to the MQTT broker running on the IHS broker.

(I) Statistics computer: It computes various CEP statistics and sends them to the output GUI handler.

(J) Output GUI handler: Multiple parallel instances of the output GUI handler update various CEP statistics to the GUI on the mobile device.

4.2.3 Screenshot of Mobile CEP Application

Figure 4.15 is a sample screenshot of the mobile CEP application running on Google Pixel [172]. Initially, a query is selected from a drop-down menu, after which the mobile CEP application starts consuming the sensor data streams from 3 sensors as shown in Figure 4.15. The real world *slp01a/slpdb* dataset (consisting of sensors such as electrocardio-

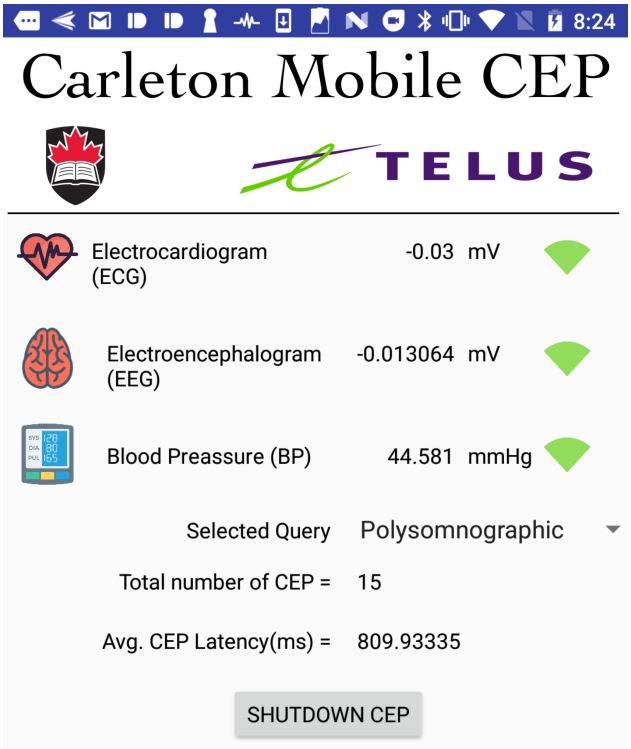


Figure 4.15: Screenshot of mobile CEP application

gram, electroencephalogram and blood pressure) from the MIT-BIH polysomnographic database [173] is used in this case. Various metrics such as the number of complex events and the average CEP latency are also shown on the mobile device GUI.

4.2.4 Mobile CEP Algorithm

Algorithm 4.1 shows a high-level logic used for devising the mobile CEP application running on the mobile device. Here, it is assumed that before Algorithm 4.1 runs, the user has

Algorithm 4.1: Mobile CEP Algorithm

```
1  if Network is available then
2      Login into the server using valid credentials
3      if Login is successful then
4          Start Device management service
5          Start MQTT service
6      end
7  else
8      Check network availability periodically
9      if Network is available then
10         Start MQTT service
11         Start Device management service
12     end
13 end
14 Start CEP service
15 Query_y ← selected query
16 AppRuntime ← createSiddhiAppRuntime(Query_y)
17 IsAppRunning ← true
18 AppRuntime.start()
19 Call sensorHandler()
```

already registered his/her mobile device with the IHS. As soon as the MCEP application is started, the availability of the wireless network on the device is checked automatically. If the network is available (Line 1), then login to the IoT server is performed (Line 2). If the login is successful (Line 3), then three services are started: device management service (Line 4), MQTT service (Line 5), and CEP service (Line 14). The device management service is responsible for authentication, authorization, and encryption of data exchanged between the mobile device and the IoT server. The MQTT service forwards complex events corresponding to a particular topic to the hospital server using the MQTT broker. When the network is not available, only the CEP service is started (Line 14) and network availability is checked periodically (Line 8) such that device management service (Line 11) and MQTT service are started (Line 10) if the network becomes available (Line 9). Once the CEP service is started, then the user (patient in the RPM use case) is prompted to select a particular type of query related to his/her problem (Line 15). This query is pre-programmed in the application and can be changed by a medical professional from the IoT server. An application runtime is created based on the selected query (Line 16). The selected query is responsible for detecting the complex event that depends on the condition of the patient being monitored. The CEP service is started (Line 18) after which the sensor handler function is called (Line 19) which is explained in Algorithm 4.2. There is a separate handler function for each sensor data stream, which is responsible to receive the sensor data streams from the respective sensor.

4.2.5 MCEP Sensor Handler Algorithm

Algorithm 4.2 is used to receive the data streams from a single sensor and then append them to a thread-safe queue. The same logic is used by the SCEP application. For receiving the input sensor data stream from $sensor_y$, a socket object ($socket_y$) is created on a new thread (Line 3). Further, an $InputHandler_y$ is created to receive a stream and to

Algorithm 4.2: MCEP Sensor Handler Algorithm

```
1  while Application is running do
2      foreach Sensor_y do
3          Socket_y ← Create socket object                    ▷ start a new thread
4          InputHandler_y ← GetInputHandler('Stream_y')
5          while IsAppRunning do
6              Tuple ← Socket_y.receive()
7              foreach Tuple do
8                  Perform de-serialization
9                  ArrivalTime ← System.nanotime()
10                 Append Tuple to a thread-safe FIFO queue    ▷ en-queue to a queue
11                 Update GUI using InputHandler_y
12             end
13         end
14     end
15     foreach Queue_y do
16         while AppRunning do
17             if Queue_y.isEmpty == false then
18                 IngestionTime ← System.nanotime()
19                 InputHandler_y.send()                       ▷ send to CEP engine
20                 Queue_y.remove()                            ▷ de-queue from a queue
21             end
22         end
23     end
24     foreach Complex event detected do
25         Use sink mapper to change event format
26         Send an event to statistic computer(Payload)
27         Call MQTT service(QoS, Payload, TopicName)
28     end
29 end
```

send it to the CEP engine (Line 4). Various sockets receive the sensor streams in parallel and perform de-serialization of stream data from binary to JSON event tuple for each received tuple (Line 8). Event arrival time is determined using a nanosecond precision system clock and appended to the event tuple (Line 9). Then, the tuple is en-queued to a thread-safe linked-blocking queue following a FIFO order (Line 10) and the GUI is updated (Line 11). A thread-safe queue is preferable in a producer-consumer scenario as it provides concurrency control mechanisms such as *mutex* to avoid problems due to race conditions. For each queue (Line 15), another thread runs in parallel which checks whether the queue contains any tuples (Line 17) and if so, it appends the event ingestion time (Line 18) before sending the tuple to the engine using the respective input handler: $InputHandler_y$ (Line 19). Finally, that event is de-queued from the LinkedBlocking queue (Line 20). As soon as the complex event is detected (Line 24), it is sent using a streaming callback to the statistics computer which computes the necessary metrics and updates the GUI (Line 26). Also, the detected complex event is sent to the MQTT service which forwards it to a Mosquito broker on a specific topic in order to forward it to the IHS (Line 27).

4.2.6 MCEP MQTT Service Algorithm

Algorithm 4.3 is responsible for enabling the MQTT pub-sub system as an Android service. For registration of the mobile device with the MQTT broker running on the server, the device enrollment is required (Line 1). The MQTT endpoint is required to connect to the IoT Server (Line 3) using the device registry information stored on the mobile device (Line 1). Various other parameters are set such as the required QoS type, will topic (Line 5), will message (Line 5) and MQTT timeout (Line 6). If all the parameters are set (Line 7), then MQTT client object is created (Line 8) using the MQTT endpoint (Line 3) and a client id (Line 2). The *SubscribeToQueue* (Line 9) and *PublishToQueue* (Line 14) functions are over-ridden to enable subscribing and publishing to a particular topic in MQTT bro-

Algorithm 4.3: MQTT Service Algorithm

```
1  DeviceRegistry ← Get device registry from the local registry
2  ClientID ← Generate the client id using the DeviceRegistry
3  MQTTEndPoint ← IP address of IoT server
4  QoS ← Get the selected Quality of Service
5  Set WillMessage and WillTopic
6  MQTTTimeout ← Timeout to resend the packet if QoS ≥ 1
7  if ClientID, MQTTEndPoint, QoS, WillMessage and WillTopic are set then
8      MQTTClient ← MqttClient(MQTTEndPoint, ClientID)
9      MQTTClient.SubscribeToQueue(QoS)
10     MQTTClient.SetDisconnectionWillForClient(QoS, WillMessage, WillTopic)
11 end
12 while Application is running do
13     MQTTClient.MessageArrived(TopicName, Payload)
14     MQTTClient.PublishToQueue(QoS, Payload, TopicName)
15     MQTTClient.DeliveryComplete()
16 end
```

ker deployed at the back-end server. Other methods can be over-ridden accordingly so as to provide the last will message in case of connection error (Line 10). While the application is running (Line 12), the processing each message on its arrival for type checking and other actions can also be done in *messageArrived* method (Line 13) and then published using the *PublishToQueue* method (Line 14). The publisher can perform the required action when delivery is complete by over-riding the *deliveryComplete* method (Line 15).

4.2.7 MCEP Average Network Consumption

As shown in Figure 4.16, the average data rate at which the sensor data streams are received by the mobile device is denoted by RX (expressed in MB/s). The average data rate at which sensor data streams are sent by the mobile device to the IoT server is denoted by TX(also in MB/s).

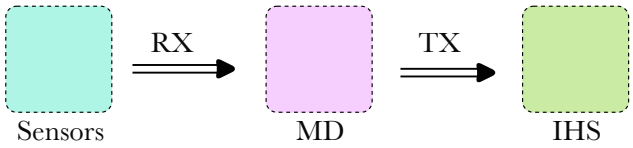


Figure 4.16: Data sent and received by the mobile device

Assuming that a user (patient) is using bluetooth or WiFi for connecting the sensors with the mobile device, TX can be used to compute the user cost. Here, we assume that a patient is using the mobile network for the transfer of data between the mobile device and the back-end IoT server. The user cost can be computed by as:

$$\text{User Cost } (\$/hour) = TX * \text{cost per MB} * 3600 \tag{4.1}$$

4.3 System Prototype Implementation

The system prototypes and the experimental set up (referred to simply as the system prototypes in the following discussion) used for analyzing their performance are discussed in this section. As shown in Figure 4.17, the system prototype for both the server CEP and mobile CEP systems consist of five components: a timekeeper, a Sensor Simulator (SS), an IoT hospital server, a mobile device, and a wireless router. Note that the timekeeper used in the experimental setup for performance measurement is not needed in a production system in which sensor simulator is replaced by the actual senor devices. The

timekeeper is used to perform global time-stamping to compute the end-to-end latency. This module is required as various components (with different un-synchronized clocks) are involved in computing the end-to-end latency. Therefore, a timekeeper is required to provide a global time-stamping for raw event streams (from the sensor simulator) and the complex event stream (from IHS) using a single clock as explained later in Section 4.5. Please note that the latency of the system can increase with the increase in the complexity of the query and the software used for detecting the complex event. The sensor simulator is used to generate the sensor workload which is further explained in Section 4.4. The various components which are stacked over one another inside the sensor simulator and the timekeeper represent multiple instances of that respective component running in parallel. The solid line represents multiple parallel data streams while a dashed line represents a single sensor data stream. As shown in Figure 4.17, the data streams generated by the sensor simulator are sent in parallel to both the mobile device and the timekeeper (for global generation time-stamping). In the SCEP system, raw event streams are sent from the mobile device to the IoT server whereas only a complex event stream is sent from the mobile device to the IoT server for the MCEP system. For both architectures, a single complex event stream is sent from the IHS to the timekeeper for global notification time-stamping. The system configuration for the aforementioned components is provided next.

1. Timekeeper: The timekeeper module is written in Java and deployed on a computer workstation having 16 GB of Random Access Memory (RAM), a 2.8 GHz Intel Core i7 processor and a 1 TB Hard Drive (HD) running on Ubuntu 14.04 Long Term Support (LTS).

2. Sensor Simulator: The Java-based sensor simulator program is running on a workstation equipped with 8 GB of RAM, a 2.8 GHz Intel Core i7 processor and a 1 TB

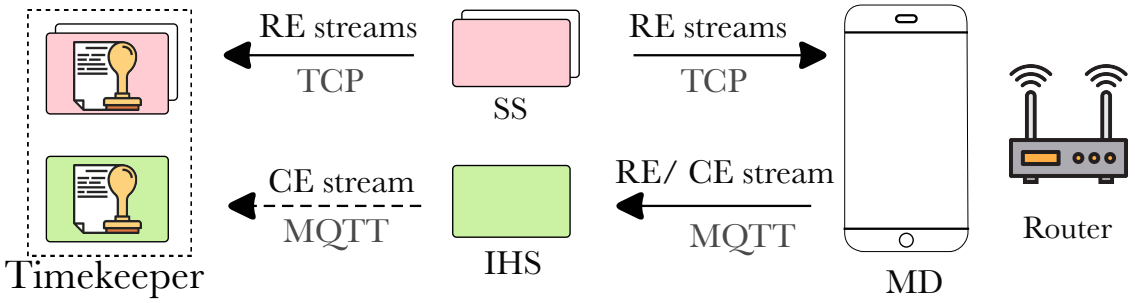


Figure 4.17: System prototype setup

HD using Ubuntu 14.04 LTS.

3. IHS: An IoT server is deployed on a system having 16 GB of RAM, a 3.5 GHz Intel Core i7 Processor and a 1 TB Solid State Drive (SSD) running under High Sierra MacOS. The MQTT broker and the MQTT subscriber is deployed on the broker and the analytics tiers of the IoT server respectively. The Java Virtual Machine (JVM) configurations for the broker, core, and analytics components of the IHS used in the prototype are given in Table 4.1. It is useful to dedicate the CPU resources to each component such as broker, core, and analytics. Here, -Xmx represents the maximum size of the JVM heap (4 GB in this case) which can be allocated to the respective tier.

Table 4.1: Java Memory Configurations

Parameter	**Broker**	**Core**	**Analytics**
-Xmx	4096 MB	4096 MB	4096 MB

4. Mobile Device: A Google Pixel smartphone [172] having 4 GB of RAM, 1.6 GHz processor, 32 GB of storage, and an AArch64 quad-core processor running Android Nougat is used as the mobile device. WSO_2 IoT server version 3.0 is deployed on IHS along with its compatible Android agent version 3.1.27 running on the mobile

device. Both the mobile CEP and server CEP applications that are written using Java are built on Android Studio 3.0.1 IDE using Gradle build tools version 26.0.2 [75]. For the mobile CEP application, due to the large size of the Siddhi CEP libraries, the multidex feature has to be enabled to overcome the 64K limit of the Android Dalvik compiler. Relevant Internet, WiFi, and network permissions must be enabled for the MCEP and SCEP applications. The MQTT publisher is deployed on the mobile device.

5. Router: A 5 GHz AC1750 Tp-Link dual-band wireless router with a maximum bandwidth of 1350 Mbps is used to transfer data between the various components.

4.3.1 Dependencies used in Mobile CEP

This subsection lists some of the CEP specific, MQTT related and testing related dependencies used in the mobile CEP application as shown in Table A.1 of Appendix A. Different scopes such as *implementation*, *test implementation*, *compile only*, *annotation processor*, and *android test implementation* are used as per requirements [174]. It is important to note that as the grade build system does not take care of the transitive dependencies, thus the transitivity for some of the dependencies is excluded to make sure the duplicate and conflicting dependencies does not create any issue.

4.4 Sensor Simulator

A multi-threaded sensor simulator program is used to simulate multiple sensors based on a given input rate and simulation time. A nanosecond sleep time is used to generate a constant inter-arrival rate for each sensor. As shown in Figure 4.17, the data streams generated by the sensor simulator are sent simultaneously to the mobile device and the timekeeper using TCP sockets. All the sensor simulator daemons send data streams con-

currently on separate threads, where each thread generates a stream of JSON tuples. A JSON tuple consists of both metadata and payload data as shown in Table 4.2. A detailed description of each field is provided in Table 4.3. The metadata includes information such as patient id, sensor id and tuple id whereas the payload data includes the respective sensor value(s) and an event generation time-stamp (Tg). In certain cases, the sensor data stream tuple may consist of an array of data values instead of a single value, but in our experimentation, a single value is used. Patient_{id} is required at the IHS in order to uniquely identify a patient when multiple patients are enrolled with the RPM service.

Table 4.2: A sample tuple generated by the sensor simulator

```
{'Metadata':{'PatientId': 1500,'SensorId':'HR','TupleId': 1324},
 'Payload':{'Value': 66,'GenerationTime': 255073580723571}}
```

Also, a combination of Patient_{id}, Sensor_{id}, and Tuple_{id} can be used to uniquely identify an event received at the IHS when multiple patients are enrolled.

Table 4.3: Tuple format

Field	**Type**	**Description**
Patient_{id}	Metadata	Unique to each patient
Sensor_{id}	Metadata	Unique for each sensor for a patient
Tuple_{id}	Metadata	Auto-incrementing integer for each tuple
Value	Payload	A uniformly distributed integer between 1 and 100
T_g	Payload	A nanosecond precision time-stamp

The sensor simulator can generate both synthetic and real-data using synthetic and real datasets respectively. For simulating the real-data, the sensor simulator uses sensor data available at the *slp01a/slpdb* dataset from the MIT-BIH polysomnographic database [173].

This dataset consists of 2-hour duration data of 4 health signals recorded at 250 Hz. In a synthetic dataset, the tuple values are uniformly distributed integers ranging from 1 to 100. The real dataset is used to test the functional correctness of the proof-of-concept prototype, whereas all the other experiments were performed using a synthetic dataset. The synthetic dataset is preferred over a real dataset because of the ability to control the various workload parameters including tuple values and tuple inter-arrival times.

4.4.1 Sensor Simulator Algorithm

Algorithm 4.4 shows the sensor simulator algorithm which can generate both the synthetic and real-time sensor data streams (by reading the stored data from a file) for X sensors. Initially, the user selects the type of architecture that is currently deployed (Line 1). This step is required because different port numbers are used to transfer data for the two architectures. Then, various parameters such as the timekeeper IP address (Line 2), the mobile device IP address (Line 3), the sensor data stream arrival rates (Line 4), and the simulation time are initialized (Line 5). With a particular architecture type selected, a port number is set for each sensor (Line 7) and the sensor simulator is started on a new thread

Algorithm 4.4: Sensor Simulator Algorithm

```
1 ArchitectureType ← MCEP || SCEP
2 IP_TK ← IP address of the timekeeper
3 IP_MD ← IP address of the mobile device
4 λ_X ← Arrival rate for Sensor_X                    ▷ Set for each sensor
5 Runtime_X ← Simulation runtime in seconds
6 foreach Sensor_X do
7 |   Port_X ← Set port# based on selected ArchitectureType
8 |   Start Sensor_X Generator(Port_X, IP_TK, IP_MD, λ_X, Runtime_X )
9 end
```

by passing the various parameters to the $Sensor_X$ generator function (Line 8). The sensor $Sensor_X$ generator algorithm used to generate data for a single sensor is discussed in Section B.1 of Appendix B.

4.5 Timekeeper

A timekeeper daemon has been devised in order to measure the end-to-end latency of complex events from the sensor simulator to the IoT hospital server. Since the sensor simulator and the IHS run on separate workstations, time measurements need to be made by a third independent machine using a single clock. A timekeeper module appends global time-stamps to all raw event streams coming from SS and the complex event stream coming from IHS. The generation time of a tuple at the sensor simulator (T_g) is time-stamped as the global generation time (T_{gg}) and the notification time at the IHS (T_n) is time-stamped as the global notification time at the timekeeper (T_{gn}).

Algorithm 4.5: Timekeeper Algorithm

```
1 ArchitectureType ← MCEP || SCEP
2 foreach Sensor_X do
3 |   Start listener for Sensor_X                    ▷ on a new thread
4 end
5 if ArchitectureType == MCEP then
6 |   Start Complex event listener (MCEP)            ▷ on a new thread
7 else
8 |   Start Complex event listener (SCEP)            ▷ on a new thread
9 end
```

4.5.1 Timekeeper Algorithm

As shown in Algorithm 4.5, initially a particular type of architecture (MCEP or SCEP) needs to be selected (Line 1), as the sensor simulator can be used with different architectures. For each $Sensor_X$, a listener daemon is started on a new thread. The sensor listener daemon (described in Algorithm C.1) is responsible for performing the time-stamping of the respective sensor. Once this step is done, the timekeeper opens a server socket to start listening to raw sensor data streams generated by the sensor simulator. Depending upon the selected type of architecture, the respective complex event listener is started on a new thread (Line 6, Line 8). The $Sensor_X$ listener algorithm which is used to receive and time-stamp the raw event sensor stream tuples for a single sensor is discussed in Section C.1 of Appendix C. The $Sensor_X$ listener further uses $DequeThreadSensor_X$ which is discussed in Section C.2 of Appendix C. Further, the complex event listener algorithm which is used to receive and time-stamp the complex event stream data tuples is discussed in Section C.3 of Appendix C

Chapter 5

Performance Analysis

This chapter first describes the remote patient monitoring use case model that has been considered in the performance analysis (see Section 5.1). The various types of event time-stamping are discussed in Section 5.2. Section 5.3 provides the description of the workload and system parameters used in the experiments. Then, various performance metrics are described in Section 5.4 followed by the discussion of a thorough performance analysis of the SCEP system and the MCEP system in Section 5.5 and Section 5.6 respectively. Finally, a comparison between the two systems is discussed in Section 5.7.

5.1 The Remote Patient Monitoring Use Case Modeling

This section focuses on the RPM use case used in fall detection for elderly people. A survey conducted by the World Health Organization (WHO) reported that the occurrence of fall is common among elderly people and seems to increase with age and frailty level [175]. As per this survey, each year approximately 28-35% people more than 65 years of age fall, whereas this number reaches 32-42% for 70 years old. Falls lead to 20-30% of mild to severe injuries and are the underlying cause of 10-15% of all emergency department visits. However, if a fall event is notified to hospital staff as soon as possible, the further loss can be circumvented.

Fall detection can be monitored remotely using a CEP engine ingesting real-time sen-

sor data streams from several mobile sensors and various physiological sensors. A fall complex event can be identified with more certainty if certain events happen in a specific order. For example, an increase in heart rate event (occurring for one or more times) followed by an increase in the breath rate event (occurring for one or more times) within some specific time interval may indicate that the elderly person has fallen. The complex event may not necessarily gave rise to a critical event, but an alert is sent nevertheless to the health services professional who can further look into weather the situation is critical. The pattern query presented in Table 5.1 is used for the performance analysis for all the experiments performed in this chapter. This pattern query detects the complex events from two streams of events (Streams A and B) generated by the sensor simulator.

Table 5.1: CEP Query

Every A$\left[value_A \geq Th_A\right]\left\langle min_A : max_A\right\rangle \rightarrow$ B$\left[value_B \geq Th_B\right]\left\langle min_B : max_B\right\rangle$ within T_{win}

Note that an event from stream A and stream B is referred to as an A and B in the following discussion. Event A represents the heart rate and event B refers to the breath rate. The query leads to the detection of a complex event if $Count_A$ instances of event A with tuple values larger than or equal to Th_A are followed by $Count_B$ instances of event B with tuple values larger than equal to Th_B. $Count_A$ and $Count_B$ are integers lying in the range bounded by min_A to max_A and min_B to max_B respectively. Special cases such as $\left\langle min_A:\right\rangle$ and $\left\langle :max_A\right\rangle$ correspond to the situations in which event A must occur at least min_A times or at most max_A times respectively. The *within* T_{win} construct used at the end of the query (as shown in Table 5.1) signifies that the computations described earlier are made for events occurring within a time window size of T_{win}. *Every A* condition means that a parallel state machine instance is triggered for every occurrence of A event which waits for B event to occur until time T_{win}. Various parallel instances of event A will wait

for corresponding event B. A timer is started with the occurrence of event A for that state machine instance. A complex event that corresponds to both the heart rate and the respiration rate going above their respective thresholds specified by Th_A and Th_B can be detected by specifying the appropriate threshold values for A and B. Such a complex event may signal an impending heart problem requiring a notification for the health service provider. More complex queries are required for handling more complex situations (e.g. a heart stroke occurrence) can be modeled with the help of multiple sensors [176].

5.2 Various types of Event Time-stamping

The various event times captured during the experiments for the SCEP and MCEP systems are shown in Figure 5.1 and Figure 5.2 respectively. A brief description of these event times is provided in Table 5.2.

Table 5.2: Various times associated with an event

Time	Description	Time-stamped at
T_g	Event generation time	Sensor simulator
T_{gg}	Event global generation time	Timekeeper
T_a	Event arrival time at the queue	MD for MCEP & IHS for SCEP system
T_i	Event ingestion time	MD for MCEP & IHS for SCEP system
T_d	Event detection time	MD for MCEP & IHS for SCEP system
T_n	Event notification time	IHS
T_{gn}	Event global notification time	Timekeeper

Figure 5.1 and Figure 5.2 are extended versions of Figure 4.17 which has been discussed earlier in Section 4.3. It is important to note that T_a, T_i, and T_d are time-stamped in the CEP engine when it is running in the mobile device (for MCEP system as shown in Figure 5.1) or in IHS (for SCEP system as shown in Figure 5.2). T_g and T_n are always time-stamped at the sensor simulator and IHS respectively for both systems. For both MCEP and SCEP

systems, the various sensor data stream generators in the sensor simulator also forward the sensor data streams to the timekeeper (in addition to the mobile device) for global time-stamping as shown in Figure 5.1 and Figure 5.2. For MCEP system (see Figure 5.1), the multiple sensor data streams (shown by a solid line) are sent by the sensor simulator to the mobile device where the complex events are detected and a complex event stream is forwarded to the IHS (shown by a dashed line), which further forwards the complex events to a timekeeper for global timestamping. For the SCEP system (see Figure 5.2), the multiple streams are sent by the sensor simulator (shown by a solid line) to the mobile device which acts as the gateway agent and forwards the streams to the IHS. In the IHS, the multiple streams are ingested by a CEP engine to generate a complex event stream which is forwarded to the timekeeper for global timestamping (shown by a dashed line).

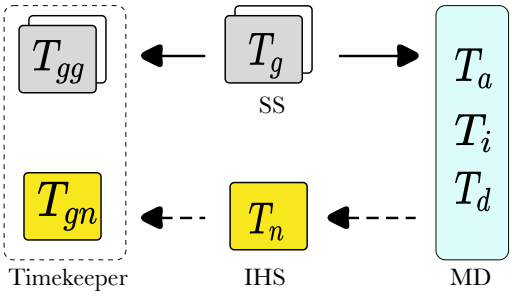


Figure 5.1: Time-stamping in MCEP

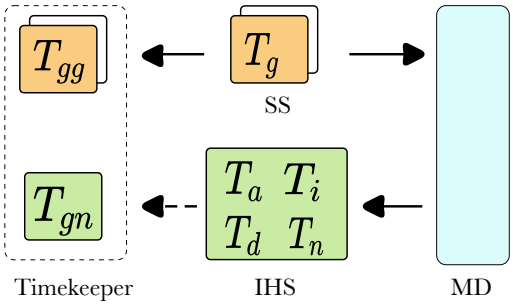


Figure 5.2: Time-stamping in SCEP

5.3 Workload and System Parameters

The various workload and system parameters are described next.

- Average raw event arrival rate (λ_{RE}): It is the average rate of the raw events generated by the sensor simulator.
- Threshold for sensor stream x (Th_x): The value of the threshold parameter is used by the selection predicate (π) to filter the sensor data streams tuples which are greater

than Th_x.

- $Count_x$: The count is used to specify the number of times a particular event has to occur to accept it as an intermediate event. An exact number of occurrences can also be specified through the count parameter. We have used the $\langle min{:}max \rangle$ specifier for $Count_x$ where $\langle min{:} \rangle$ means that an event has to happen at-least min times while no upper bound is specified. In other words, the notation, $\langle$min:max$\rangle$, which means that the event should happen at-least min times but less than max times.
- Time window (T_{win}): The time window specifies the maximum time until which event A will wait for event B to occur. Please note that this time will be different for each state machine instance. The time window starts as soon as event A arrives at the CEP system. Then a separate instance of the state machine is started and it waits for event B for a time less than T_{win}.
- Number of sensors (N_s): This parameter represents the number of sensors that have been simulated by the sensor simulator.
- Simulation runtime in minutes (T_{run}): It is the length of the simulation runtime period in minutes.
- Quality of Service (QoS): It is the QoS type (0, 1 or 2) used by the MQTT publisher to transmit the sensor data streams to the MQTT broker. These QoS parameters were discussed earlier in Item 4 of Section 2.5.

The various values for the workload and system parameters used in the experiments are presented in Table 5.3. Factor-at-a-time experiments were performed on the system in which one parameter was varied in a given experiment while others were held at their default values. The value in bold for each parameter presented in Table 5.3 corresponds to the default value of the parameter.

Table 5.3: Workload and system parameters

Parameter	Description	Units
λ_{RE}	100, 300, **500**, 1000, 2000	events/second
Th_x	**50**, 60, 70, 99	-
$Count_x$	1, 5, **10**	-
T_{win}	0.005, 0.035, 0.06, 0.1, 0.2, **10**	seconds
T_{run}	**5**, 60	minutes
QoS	**0**, 1, 2	-

5.4 Performance Metrics

Let T_a^x and T_i^x be the arrival time and ingestion time respectively for the earliest arriving event, among all the events from the different sensor data streams that led to the complex event. Moreover, let T_g^x and T_{gg}^x be the generation time and global generation time respectively for the earliest arriving event that corresponded to the complex event. Also, let T_d^x, T_n^x and T_{gn}^x represent the complex event detection time, complex event notification time and complex event global notification time respectively. The various CEP specific metrics and how they are computed from the various measured times are discussed next.

- Average CEP latency (L): A complex event is generated when a CQL pattern match occurs by ingesting data from multiple sensor data streams. The latency of a complex event is measured from the time of ingestion (T_i) for the first event (from any sensor data stream) that leads to the complex event to the time at which the complex event gets detected (T_d). If the total number of complex events detected during an experiment is N, then the average CEP latency is given by Equation (5.1).

$$L = \frac{\sum_{x=1}^{N} T_d^x - T_i^x}{N} \tag{5.1}$$

The average CEP latencies for the MCEP and SCEP systems are represented by L_{MCEP}, and L_{SCEP} respectively.

- Average complex event queuing delay (Q): In the MCEP system, a raw event upon arrival at the mobile device waits in a thread-safe FIFO queue before it gets ingested by the CEP engine deployed on the mobile device, so both T_a and T_i are measured at the mobile device as shown in Figure 5.1. For the SCEP system, the queuing for raw events is done at the ActiveMQ deployed at the IoT server, so both T_a and T_i are measured at the IoT server as shown in Figure 5.2. Q is defined as the average of the queuing delay corresponding to all the N complex events as given by Equation (5.2). The expression within the summation term of the numerator is the queuing delay for the x^{th} complex event and is determined as the difference between the ingestion time (T_i^x) for the earliest event that led to the x^{th} complex event and the arrival time (T_a^x) for the same event.

$$Q = \frac{\sum_{x=1}^{N} T_i^x - T_a^x}{N} \tag{5.2}$$

The average CEP queuing latencies for the MCEP and SCEP systems are represented by Q_{MCEP} and Q_{SCEP} respectively.

- Average complex event End-to-End (E2E) latency (E): It is the average time taken by an event (which corresponds to the earliest raw event leading to a complex event) from the time it is generated by the sensor simulator (T_g) to the time it is notified at

the IoT server (T_n). However, as discussed earlier, T_g and T_n are time-stamped in the sensor simulator and the IoT server respectively using clocks that are not synchronized. Thus, E is computed using T_{gg} and T_{gn} (instead of T_g and T_n) both of which are time-stamped on the timekeeper module. E is computed using Equation (5.3), where T^x_{gg} and T^x_{gn} represent the global generation time for the x^{th} raw event that corresponds to a complex event and the global notification time for the x^{th} complex event, both time-stamped at the timekeeper.

$$E = \frac{\sum_{x=1}^{N} T^x_{gn} - T^x_{gg}}{N} \tag{5.3}$$

Further, the average E2E latency for the MCEP and SCEP systems is represented by E_{MCEP}, and E_{SCEP} respectively. A diagram showing the relationship among CEP specific metrics L, Q, and E is presented in Figure 5.3. In this figure, the multiple instances of input sensor data streams (one for each sensor) are shown in parallel such that tuples in the n^{th} sensor data stream (where $n \in 1 \ldots y$) are denoted by T^n_a and T^n_i as arrival time and ingestion time respectively. However, as the complex

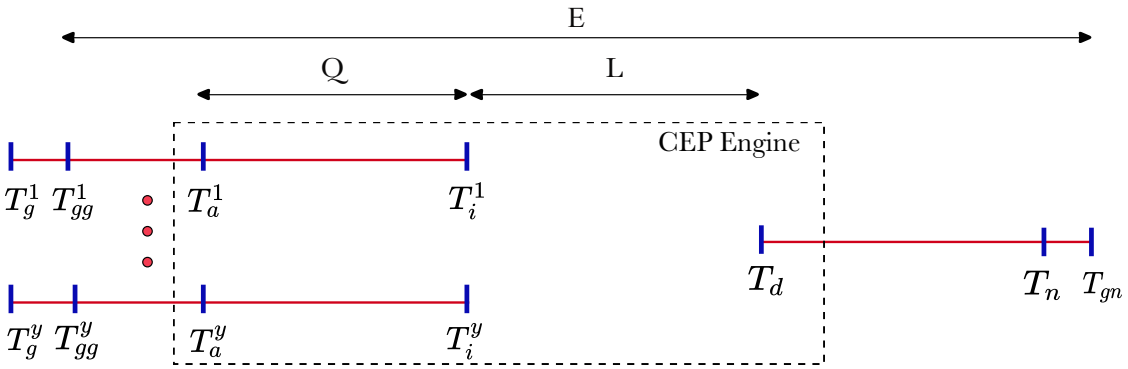


Figure 5.3: CEP specific metrics

event is generated from a pattern which ingests multiple sensor data events, only one complex event is shown on the right-hand side of Figure 5.3.

- Average raw event throughput for the server CEP application ($\mu_{RE-SCEPA}$): This parameter represents the average rate of raw events emitted by the server CEP application to the IoT server. The $\mu_{RE-SCEPA}$ for x number of sensor data streams is calculated using Equation (5.4), where μ_{RE_x} represents the average raw event throughput for x^{th} sensor data stream.

$$\mu_{RE-SCEPA} = \frac{\mu_{RE_1} + \mu_{RE_2} + \cdots + \mu_{RE_x}}{x} \tag{5.4}$$

- Average complex event throughput (μ_{CE}): It is the rate at which complex events are emitted by the CEP engine. $\mu_{CE-SCEP}$ and $\mu_{CE-MCEP}$ represent the average complex event throughput for SCEP and MCEP respectively.
- Average memory usage (MU): MU is the average memory consumed by the mobile application (in MB) during an experiment. MU_{SCEPA} and MU_{MCEPA} represent the average memory usage for the SCEP application and the MCEP application respectively. MU_{MCEPA} is computed using Algorithm A.2.
- Average CPU utilization (CU): It is the average CPU utilization by the mobile application during its experiment runtime. CU_{SCEPA} and CU_{MCEPA} represent the average CPU utilization for the SCEP application and the MCEP application respectively. The CU_{MCEPA} is computed using Algorithm A.1. The application is un-installed and installed again for each experiment for computing the MU and CU.
- Average CPU utilization by IHS (CU_{IHS}): CU_{IHS} represents the average CPU utilization of the IoT server. $CU_{IHS-SCEP}$ and $CU_{IHS-MCEP}$ represent the average CPU utilization by the IHS for the SCEP system and the MCEP system respectively.

- Average number of bytes received per second by the mobile device (RX): RX is the average data reception rate (bytes/second) by a mobile device application. RX_{SCEPA} and RX_{MCEPA} represent RX for the SCEP application and the MCEP application respectively.
- Average number of bytes transmitted per second from a mobile device (TX): TX is the average data transmission rate (bytes/second) sent by a mobile device application to the IoT server. TX_{SCEPA} and TX_{MCEPA} represent TX for the SCEP application and the MCEP application respectively.
- User cost (UC): The UC is the average cost (in $/hour) paid by the user for using the CEP service as discussed in Section 4.2.7. UC_{SCEP} and UC_{MCEP} represent the user cost for using the SCEP service and the MCEP service respectively.
- Remaining Battery Life (RBL): It is the amount of battery power remaining (in %) on the mobile device during an experiment. It is an important metric representing the power consumption of an application. The different types of RBL used in the experimentation are provided next.

  - $RBL_{SCEPA-FG}$ and $RBL_{MCEPA-FG}$ represent the battery usage for the server CEP and mobile CEP applications respectively when these applications are running in the foreground on the mobile device and no other service is running in the background.
  - $RBL_{SCEPA-BG}$ and $RBL_{MCEPA-BG}$ represent the battery usage for the server CEP and mobile CEP applications respectively when these applications are running on the background of the mobile device and no other application is running on the foreground.
  - $RBL_{YouTube-FG+SCEPA-BG}$ represent the battery usage when a YouTube 720P

video is played on the mobile device in the foreground and the SCEP application is running in the background on the mobile device.

- $RBL_{YouTube-FG+MCEPA-BG}$ represent the battery usage when a YouTube 720P video is played on the mobile device in the foreground and the MCEP application is running in the background on the mobile device.

Application profilers such as *Trepn*, *Power Tutor* or *Intel Performance Viewer* can be used to perform system-level and application-level performance profiling in a mobile device [177]. However, the accuracy of these applications is a concern, thus these applications are not used to profile the mobile device in our experimentation. To solve this issue, the various application metrics such as *CU* and *MU* have been calculated using a *bash* script which reads *dumpsys* information using ADB shell. As shown in Algorithm A.1 and Algorithm A.2, the script reads various application and system specific metrics and parses this information using a combination of various *grep* commands, regular expressions, *awk* scripts and *sed* expressions.

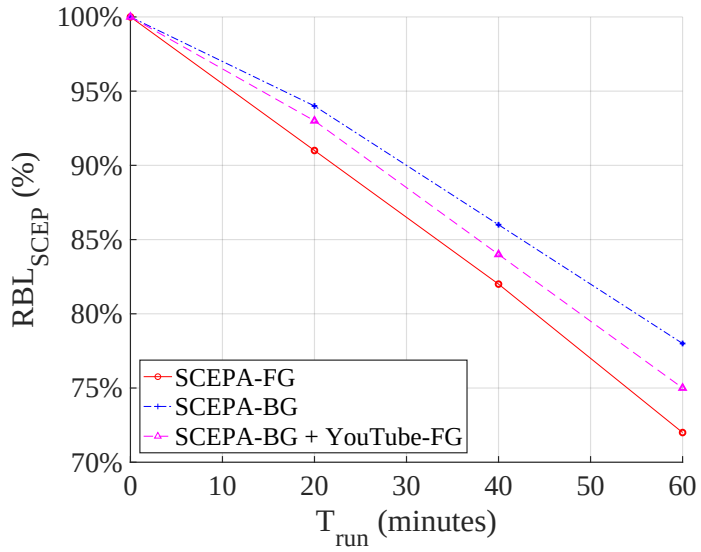
A synthetic workload comprising two streams A and B is used in the performance analysis. Each experiment for the SCEP system and the MCEP system was repeated 3 times and the resulting values of a given performance metric were found to be close to one another. The values of the various performance metrics used for generating the graphs are averages over the three runs and are provided in a table beside the respective graph. The results of the experiments that capture the relationship between the system/workload parameters and various performance metrics for the SCEP system and the MCEP system are discussed in Section 5.5 and Section 5.6 respectively. A performance comparison of the two systems is done in Section 5.7.

5.5 Experiments for Server CEP system

This section describes all the experiments that were performed on the server CEP system.

5.5.1 Impact of T_{run} on RBL_{SCEP}

The experiment was performed for 60 minutes with an initial battery power of 100%. During the length of the experiment, the value of the battery level on the mobile device was noted every 20-minute interval, as shown by T_{run} in Figure 5.4. The battery consumption for the SCEP application was computed in three scenarios. (1) the SCEP application running in the Foreground (FG) and no other application running in the background; (2) the SCEP application running in the Background (BG) with no other application running in the FG on the mobile device; and (3) the SCEP application running in the background and the YouTube application running in the foreground playing the video at 720P (HD resolution). As shown in Figure 5.4, RBL decreases with an increase in T_{run} in each of



T_{run}	RBL		
	FG	BG	$BG + FG$
0	100%	100%	100%
20	91%	94%	93%
40	82%	86%	84%
60	72%	78%	75%

Figure 5.4: Impact of application usage time on the battery consumption in SCEP system

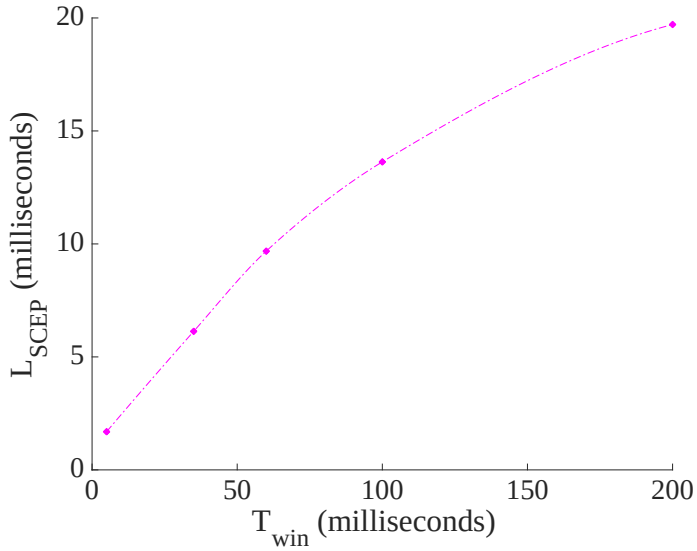
the three scenarios. For any given T_{run}, the highest drop of 28% in battery power is observed in the case of the SCEP gateway application running in the foreground (scenario 1). The least battery consumption was reported in the case of the SCEP gateway application running in the background (scenario 2) while no other application was running in the foreground. This led to a 22% drop in batter in power at the end of an hour. This is because when an application is running in the background, the Android OS does not refresh the user interface leading to lesser power usage. Scenario 3 (that corresponds to the SCEP gateway application running on the mobile device as an android-service and the YouTube application running in the foreground) gave rise to a drop in battery power that lies in between the drops in battery powers achieved in the two other scenarios for any given T_{run}. Overall, the drops in RBL observed for all the three scenarios are within 6% of one another.

5.5.2 Impact of T_{win} on L_{SCEP}

The impact of the size of the time window (T_{win}) on the average CEP latency in the SCEP system is presented in Figure 5.5. With an increase in the time window length, L_{SCEP} increases as a larger T_{win} means that A events can wait longer for B events. Thus, a higher number of B events arriving late within the T_{win} duration that lead to a complex event result in a larger value of L_{SCEP}.

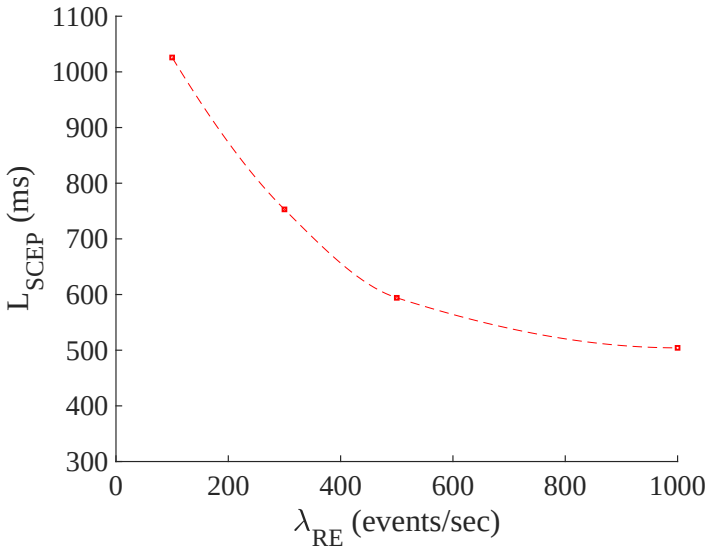
5.5.3 Impact of λ_{RE} on L_{SCEP}

As shown in Figure 5.6, the average CEP latency decreases with an increase in the average raw event arrival rate. This is because, with an increase in λ_{RE}, the inter-arrival time of the event B is reduced. This led to a decrease in the waiting time of the A events in the CEP engine, resulting in the lower values of L_{SCEP}.



T_{win}	L_{SCEP}
5	1.69
35	6.13
60	9.68
100	13.63
200	19.71

Figure 5.5: Impact of time window length on the average CEP latency in SCEP system

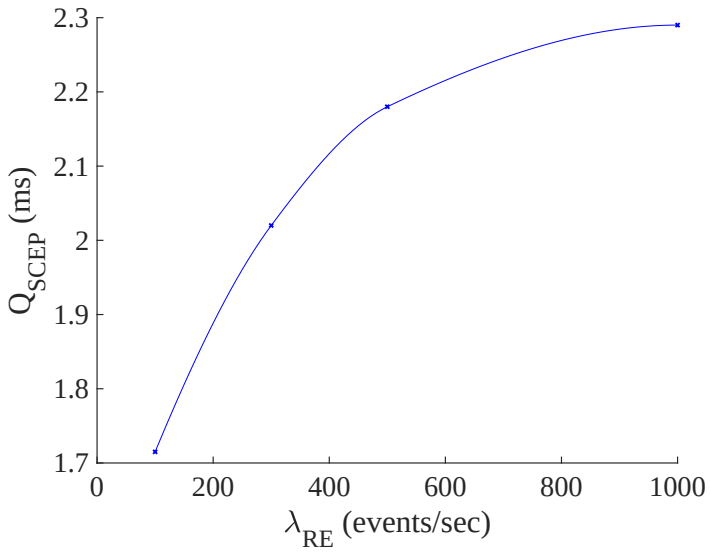


λ_{RE}	L_{SCEP}
100	1025.87
300	725.96
500	594.12
1000	504.23

Figure 5.6: Impact of raw event arrival rate on the average CEP latency in SCEP system

5.5.4 Impact of λ_{RE} on Q_{SCEP}

As shown in Figure 5.7, as λ_{RE} is increased, the average complex event queuing latency at the IoT server increases. This is because, as more raw events are waiting in the CEP system due to an increase in the arrival rate.

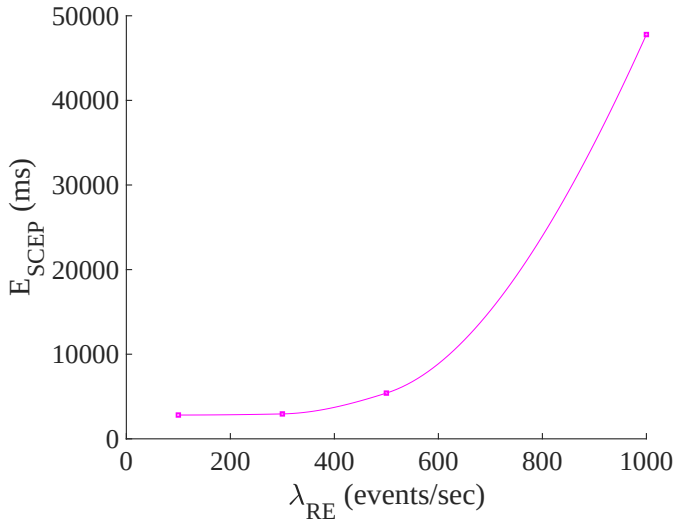


λ_{RE}	Q_{SCEP}
100	1.71
300	2.02
500	2.18
1000	2.29

Figure 5.7: Impact of λ_{RE} on the average complex event queuing delay in SCEP system

5.5.5 Impact of λ_{RE} on E_{SCEP}

The end-to-end latency depends upon various factors such as the sum of various transmission times, queuing delays and event processing latencies. As shown in Figure 5.8, as λ_{RE} is increased, more complex events are detected per unit time. This seems to increase the resource contention resulting in an increase in the transmission delay (as more complex events will be sent to the timekeeper) and the queuing delay (see Figure 5.8) leading to an increase in the average end-to-end delay.



λ_{RE}	E_{SCEP}
100	2811.88
300	2943.21
500	5404.33
1000	47782.39

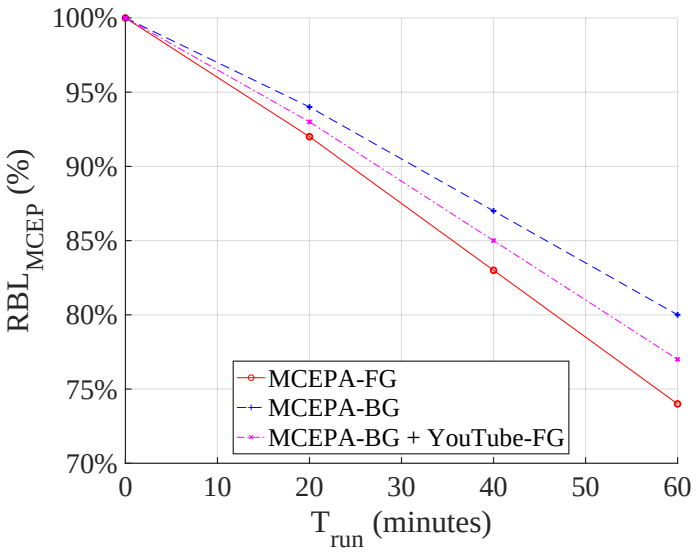
Figure 5.8: Impact of λ_{RE} on the average end-to-end delay in SCEP system

5.6 Experiments for Mobile CEP system

This section shows all the experiments that were performed on the mobile CEP system.

5.6.1 Impact of T_{run} on RBL_{MCEP}

The battery consumption for the MCEP system was computed with the same three scenarios as described in Section 5.5.1. As shown in Figure 5.9, for the MCEP system, it is observed that with the passage of time (shown by T_{run}), the largest battery drop of 26% was observed when the MCEP application was running in the foreground. However, when the MCEP application was running in the background (as an android-service) and the user was watching a 720P video with the YouTube application running in the foreground, a 23% battery drop within an hour was observed. The least battery consumption of 20% was found when the MCEP application was running as-a-service in the background while no other application was running in the foreground.

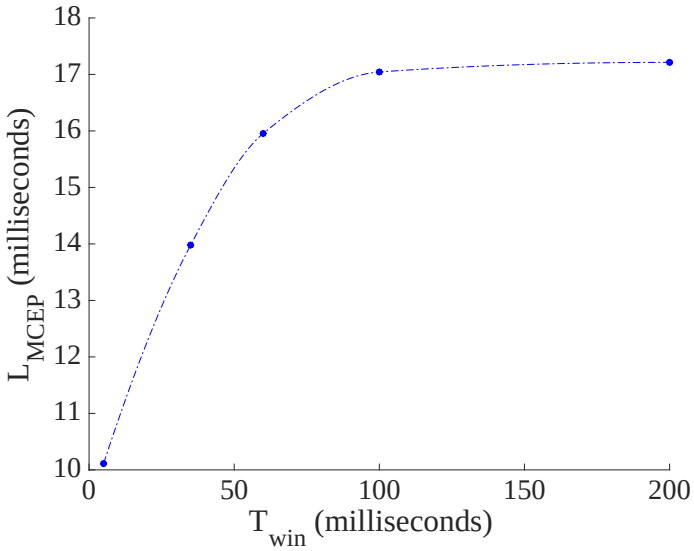


T_{run}	*RBL*		
	FG	*BG*	*BG + FG*
0	100%	100%	100%
20	92%	94%	93%
40	83%	87%	85%
60	74%	80%	77%

Figure 5.9: Impact of experiment runtime on the battery power in MCEP system

5.6.2 Impact of T_{win} on L_{MCEP}

The impact of the size of the time window on the average CEP latency in the MCEP system is captured in Figure 5.10. Please note that, as soon as a time window expires, the events accumulated in the previous time window are flushed out and the CEP engine starts looking for complex events using the new raw events arriving on the MCEP application. With an initial increase in the size of the time window up to 100 ms, L_{MCEP} is observed to increase. Increasing the size of T_{win} allows complex events with higher latencies to occur on the MCEP application. Hence, with an increase in T_{win}, a larger number of complex events with higher latencies seem to occur on the MCEP application resulting in higher values of L_{MCEP}. Further increase in the time window greater than 100 ms led to a small increase in L_{MCEP} due to temporal queuing of incoming events outside the CEP engine, as the Android OS dynamically allocates more memory to the application which takes some time.

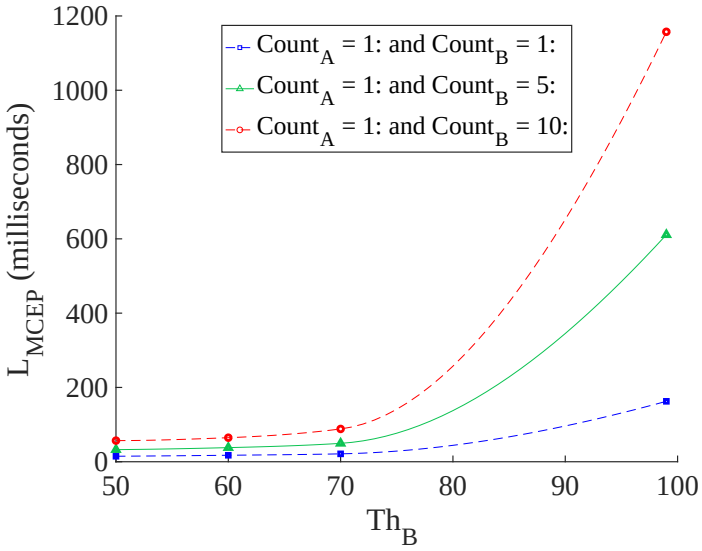


T_{win}	L_{MCEP}
5	10.11
35	13.97
60	15.95
100	17.04
200	17.21

Figure 5.10: Impact of time window on the average CEP latency in MCEP system

5.6.3 Impact of $Count_B$ and Th_B on L_{MCEP}

The effect of the threshold for event B on the average CEP latency is presented in Figure 5.11. Note that in the graph legend, the minimum and maximum values of $Count_A$ and $Count_B$ are presented. Given a specific value of $Count_B$, L_{MCEP} is observed to increase with an increase in Th_B. In the synthetic workload, the sensor data values corresponding to the stream B are uniformly distributed between 1 and 100. Thus, as Th_B increases, the time it takes to receive tuples with values higher than or equal to Th_B increases as a result of which L_{MCEP} increases. For a given value of Th_B, L_{MCEP} increases with an increase in the value of $Count_B$. This is because, as $Count_B$ increases, it takes more time for the complex event to occur as a result of which L_{MCEP} increases.



Th_B	L_{MCEP}		
	1:	**5:**	**10:**
50	14.80	32.61	56.89
60	17.44	38.00	64.74
70	21.24	49.53	88.28
99	162.65	611.04	1157.35

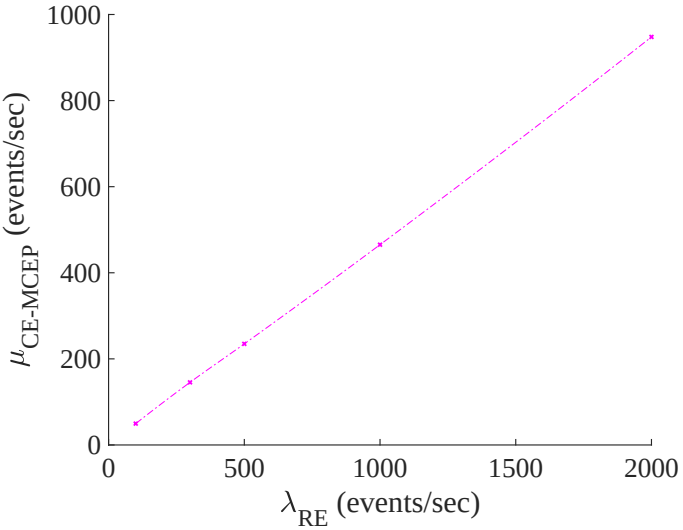
Figure 5.11: Impact of Th_B and $Count_B$ on average CEP latency in MCEP system

5.6.4 Impact of λ_{RE} on $\mu_{CE-MCEP}$

As shown in Figure 5.12, with an increase in λ_{RE}, the number of complex events detected per second by the MCEP application ($\mu_{CE-MCEP}$) increases. This is because as a higher λ_{RE} means that more possible combinations of event A can be followed by event B resulting in a larger complex event throughput.

5.6.5 Impact of λ_{RE} on L_{MCEP}

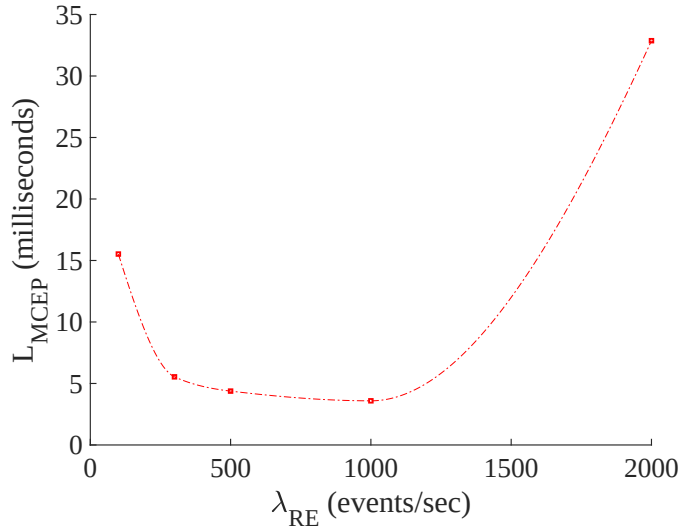
As shown in Figure 5.13, the variation in L_{MCEP} with respect to λ_{RE} displays a non-monotonic behavior. There seem to be two counteracting factors that determine the value of L_{MCEP}. The first factor is the time for the required number of tuples that can trigger a complex event to arrive in the system and the second is the queuing delay experienced by the tuples before they can be processed by the CEP engine in the mobile device. At low values of λ_{RE} ($\lambda_{RE} \leq 1000$ events/second), there is very little queuing and the first factor



λ_{RE}	$\mu_{CE-MCEP}$
100	49.6
300	145.4
500	234.9
1000	465.2
2000	948.2

Figure 5.12: Impact of raw event arrival rate on $\mu_{CE-MCEP}$

dominates the system performance. Thus, as λ_{RE} increases, the complex events are detected faster and L_{MCEP} decreases. At higher values of λ_{RE} ($\lambda_{RE} \geq 1000$ events/second),



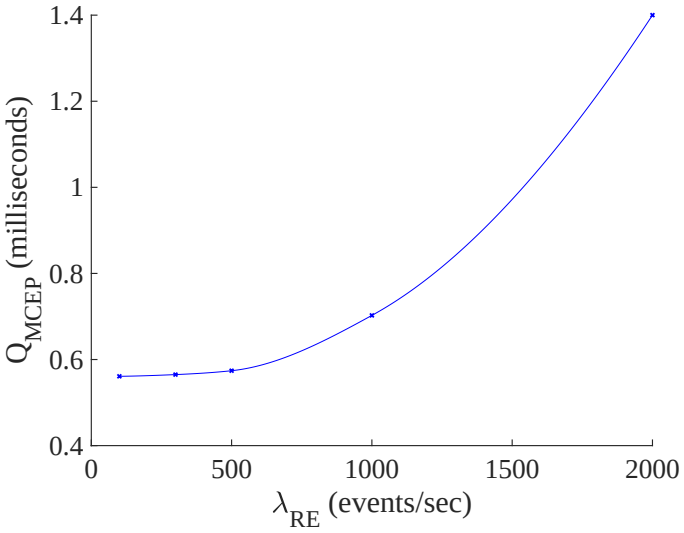
λ_{RE}	L_{MCEP}
100	15.52
300	5.53
500	4.38
1000	3.59
2000	32.87

Figure 5.13: Impact of raw event arrival rate on average CEP latency in MCEP system

the system contention increases and more and more queuing starts occurring in the system. In this case, the second factor starts dominating the system performance and L_{MCEP} starts increasing with respect to λ_{RE}.

5.6.6 Impact of λ_{RE} on Q_{MCEP}

Figure 5.14 shows the effect of the raw event arrival rate on the average complex event queuing delay. As λ_{RE} increases, system contention increases and as a result, incoming events need to wait for a longer period of time in the queue leading to higher Q_{MCEP}.

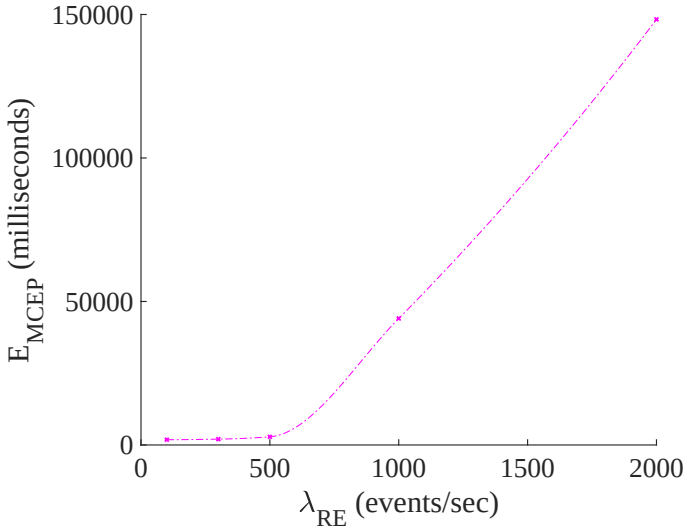


λ_{RE}	Q_{MCEP}
100	0.561
300	0.565
500	0.574
1000	0.703
2000	1.400

Figure 5.14: Impact of raw event arrival rate on average queuing latency in MCEP system

5.6.7 Impact of λ_{RE} on E_{MCEP}

Figure 5.15 shows the effect of the raw event arrival rate on the average complex event end-to-end delay. The end-to-end latency depends upon delays occurred due to various types of queuing delays (in CEP engine and MQTT broker), CEP latency and other transmission delays. With the increase of λ_{RE}, E_{MCEP} is observed to increase. This is due to an



λ_{RE}	E_{MCEP}
100	1831.00
300	2032.54
500	2837.03
1000	44068.94
2000	148282.71

Figure 5.15: Impact of λ_{RE} on average end-to-end latency in MCEP system

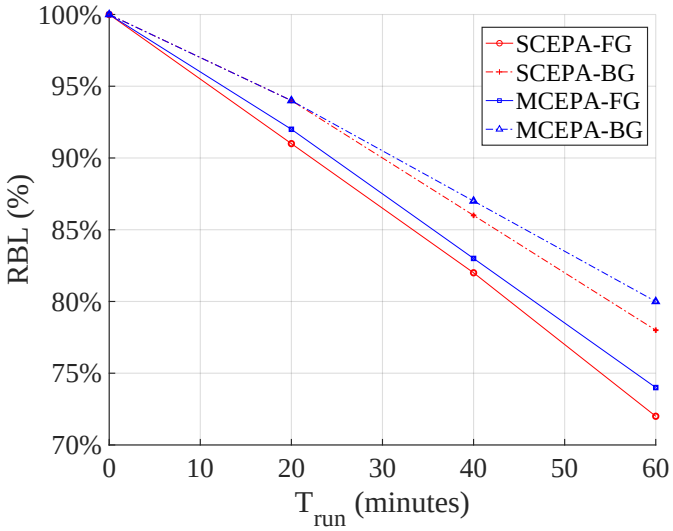
increase in L_{MCEP} (see Figure 5.14) and Q_{MCEP} (see Figure 5.13) with an increase in λ_{RE}.

5.7 Performance Comparison of SCEP and MCEP Systems

In this section, the performance comparison between the MCEP and SCEP systems is presented.

5.7.1 Comparison of RBL_{SCEP} and RBL_{MCEP} when T_{run} is increased

The impact of the T_{run} on the power consumption of the MCEP and SCEP applications is presented in Figure 5.16. The experiment was performed for 60 minutes with an initial battery level of 100%. During the experiment, the values of the battery level on the mobile device were noted every 20-minute interval, as shown by T_{run} in Figure 5.16. Recall that scenario 1 corresponds to the SCEP/MCEP application running in the FG and no other application running in the background. In scenario 2, the SCEP/MCEP application is running in the BG with no other application running in FG on the mobile device. It is



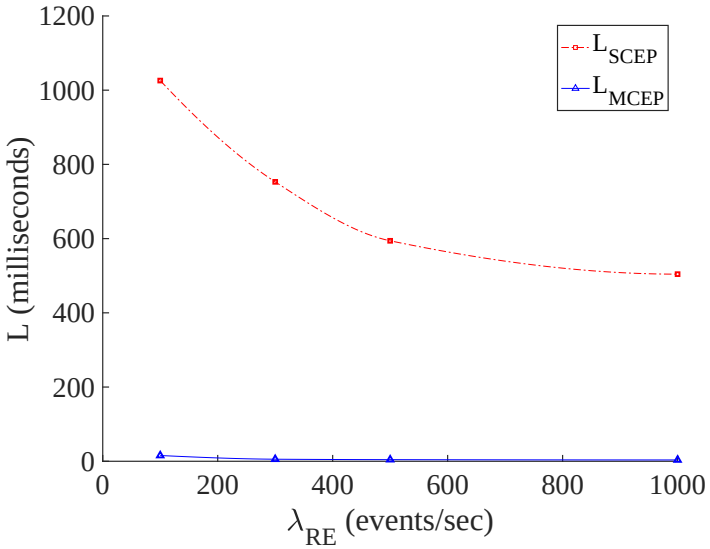
T_{run}	*MCEP*		*SCEP*	
	FG	*BG*	*FG*	*BG*
0	100%	100%	100%	100%
20	92%	94%	91%	94%
40	83%	87%	82%	86%
60	74%	80%	72%	78%

Figure 5.16: Impact of runtime on battery usage in MCEP and SCEP systems

found that the battery usage of an application for scenario 1 is always lower in comparison to scenario 2. Also, for a given scenario the battery usage for the MCEP application is lower than that for the SCEP. This is due to the fact that only complex events are transferred to the IoT server when the MCEP application is used. On the other side, all the raw events (from multiple sensors) are forwarded to the IoT server when the SCEP application is used, causing an increase in the battery consumption. This experiment shows that the proposed MCEP system provides 2% power savings (both in background and foreground), in comparison to the SCEP system.

5.7.2 Comparison of L_{MCEP} and L_{SCEP}

As shown in Figure 5.17, for a particular λ_{RE}, the average CEP latency for the SCEP system is higher than the average CEP latency for the MCEP system. This is because, in the case of server CEP, the data analytics server uses Apache thrift as a middle-ware to send the requests to the CEP engine using remote method invocations, causing the additional



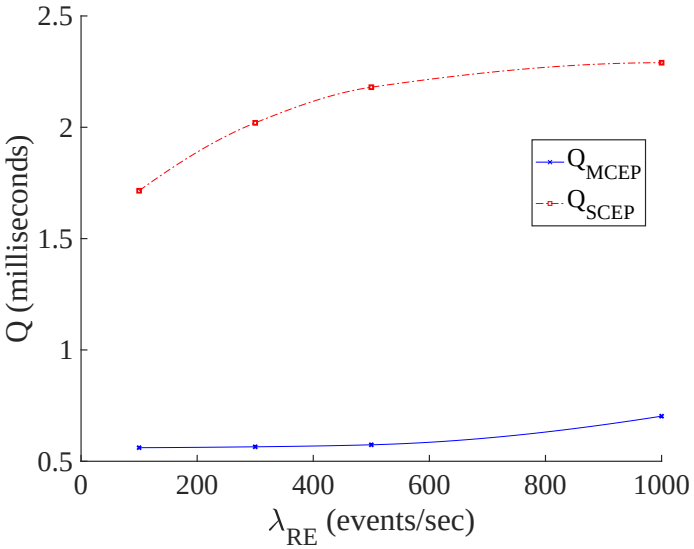
λ_{RE}	L_{SCEP}	L_{MCEP}
100	1025.87	15.52
300	725.96	5.53
500	594.12	4.38
1000	504.23	3.59

Figure 5.17: Comparison of average CEP latency in SCEP and MCEP systems

delays. This results in a higher CEP latency for SCEP in comparison to the MCEP system which does not use a middleware system. This leads to the important conclusion that there is a trade-off between security and latency for the SCEP system. Although enabling additional features in the IoT server provides more security, it also leads to an increase in CEP processing latency.

5.7.3 Comparison of Q_{MCEP} and Q_{SCEP}

As shown in Figure 5.18, for a given λ_{RE}, the average complex event queuing latency is more for the SCEP system, as a brokered queue (ActiveMQ) is used instead of a thread-safe queue employed in the case of the MCEP application. This is because the ActiveMQ system is a computationally heavy system (relative to the thread-safe queue based MCEP system) which comprises of three sub-components: ActiveMQ publisher, ActiveMQ receiver, and ActiveMQ broker. The ActiveMQ broker acknowledges every received senor stream tuple sent by the ActiveMQ publisher so as to remove it from memory when it



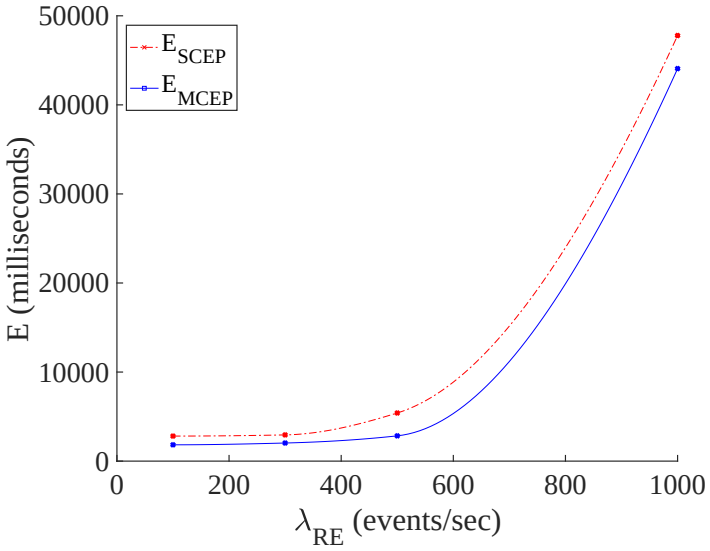
λ_{RE}	Q_{SCEP}	Q_{MCEP}
100	1.71	0.561
300	2.02	0.565
500	2.18	0.574
1000	2.29	0.703

Figure 5.18: Impact of average raw event arrival rate on Q_{MCEP} and Q_{SCEP}

has been received by an ActiveMQ receiver. However, this acknowledgment is not sent in the case of a thread-safe queue is implemented in the mobile CEP system. For the server CEP, the various performance tuning configurations have been experimented with for reducing the overhead caused due to the several reasons explained in Section 4.1.7. If a broker-less queue such a ZeroMQ is available in the future for IoT server, the queuing latency can be further reduced [161]. It is interesting to note that for higher values of λ_{RE}, the queuing latency in the MCEP increases more sharply in comparison to the server CEP queuing latency which is increasing at a much smaller rate (see Figure 5.14). This is due to the large dedicated memory pool available in the server CEP that is not present in the mobile CEP. From this experiment, we can conclude that even if the server CEP system is powerful, yet it leads to queuing delay due to various overheads due to content validation and other features.

5.7.4 Comparison of E_{MCEP} and E_{SCEP}

For a particular value of λ_{RE}, the end-to-end delay for the SCEP system is higher than the one for the MCEP system, as all the raw sensors streams are forwarded to the IoT server leading to larger transmission delays. From Figure 5.19, we can conclude that, in spite of using the large time window of 10 seconds (default time window) that leads to additional queuing delays on the mobile device (due to a lower memory availability), E_{MCEP} achieved on MCEP system with a given λ_{RE} is less than E_{SCEP} achieved on the SCEP system.

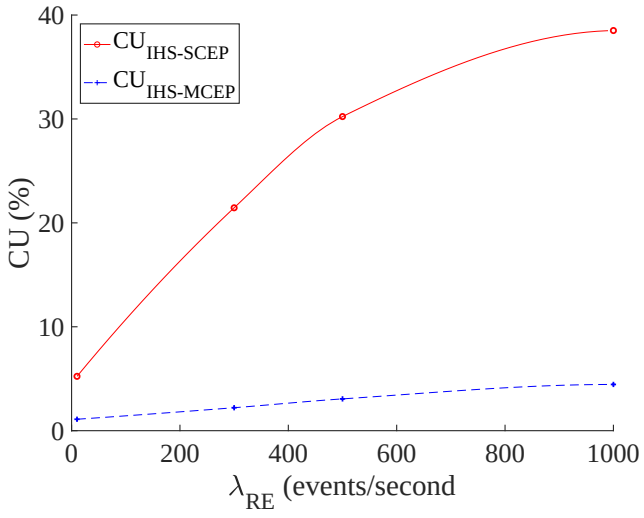


λ_{RE}	E_{SCEP}	E_{MCEP}
100	2811.88	1831.00
300	2943.21	2032.54
500	5404.33	2837.03
1000	47782.39	44068.94

Figure 5.19: Impact of average raw event arrival rate on E_{MCEP} and E_{SCEP}

5.7.5 Comparison of $CU_{IHS-MCEP}$ and $CU_{IHS-SCEP}$

Figure 5.20 shows the impact of λ_{RE} on the CPU utilization of the IoT server in for the MCEP system ($CU_{IHS-MCEP}$) and SCEP system ($CU_{IHS-SCEP}$). For a particular value of λ_{RE}, $CU_{IHS-MCEP}$ is lower than $CU_{IHS-SCEP}$. This is because, in the case of the SCEP



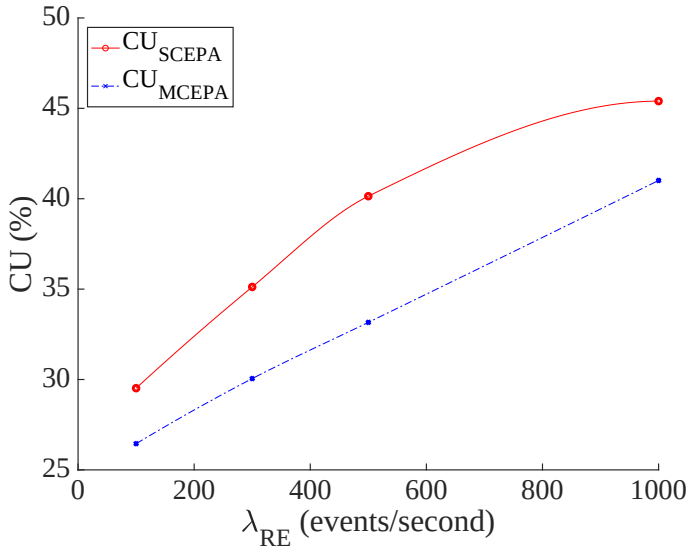
λ_{RE}	$CU_{IHS-SCEP}$	$CU_{IHS-MCEP}$
100	5.23	1.10
300	21.45	2.21
500	30.15	3.06
1000	38.51	4.45

Figure 5.20: Impact of average raw event arrival rate on the IHS CPU utilization

system, all the raw sensor data streams are received, parsed, type converted, enqueued, dequeued and processed in the IoT server and then complex events are forwarded to the timekeeper by using the MQTT broker and metrics are sent to the DataDog dashboard by the JMX agent. However, in the case of the MCEP system, only CEP alerts are received by the IoT server and no further processing has to be done which leads to lower CPU utilization. From this graph, we can conclude that the MCEP system leads to a smaller load on the IoT server, which is one of the advantages of the MCEP system.

5.7.6 Comparison of CU_{SCEPA} and CU_{MCEPA}

Figure 5.21 shows the CPU utilization observed for the MCEP application and the SCEP gateway application. In case of the MCEP application, CU_{MCEPA} seems to increase linearly with the increase of λ_{RE} as more processing is done inside the CEP engine for the higher raw event arrival rates. However, in the case of the SCEP gateway application, CU_{SCEPA} increases at a much smaller rate as the events have to be forwarded to the IoT



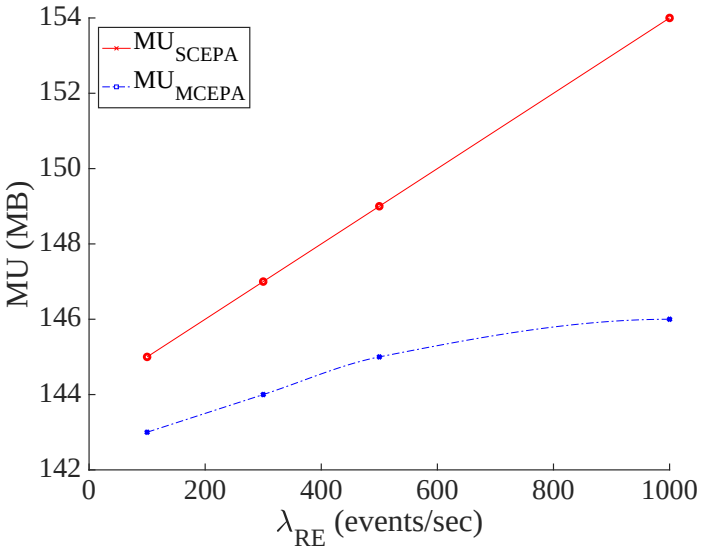
λ_{RE}	CU_{SCEPA}	CU_{MCEPA}
100	29.52	26.45
300	35.12	30.05
500	40.14	33.16
1000	45.40	41.01

Figure 5.21: Impact of average raw event arrival rate on CU_{SCEPA} and CU_{MCEPA}

server. Also, it is interesting to note that the CPU utilization of the MCEP application is 41.01% when 2 sensors are sending data streams at 1000 Hz, which is lower in comparison to the 45.40% CPU utilization reported in the case of SCEP application.

5.7.7 Comparison of MU_{SCEPA} and MU_{MCEPA}

Please note that both the MCEP and SCEP applications are running on the mobile device. As shown in Figure 5.22, for λ_{RE} of 1000 events/second, the memory used by MCEP application (146 MB) is lower than that the memory used by the SCEP application (154 MB). This is due to the fact that the memory usage is dependent upon the number of events which are received by the mobile device as well as the number of events sent by the mobile device. In the case of the SCEP application, all the incoming raw events need to be enqueued into an MQTT queue before sending them to the IoT server which leads to the higher memory usage. An important conclusion from this experiment is that for the sensors clocked at a higher sampling rate of 1000 events/second, the SCEP system



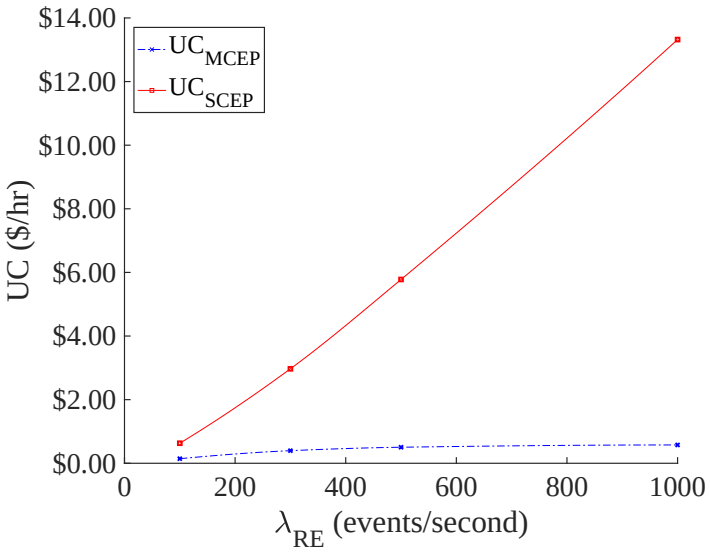
λ_{RE}	MU_{SCEPA}	MU_{MCEPA}
100	145	143
300	147	144
500	149	145
1000	154	146

Figure 5.22: Impact of average raw event arrival rate on MU_{SCEPA} and MU_{MCEPA}

needs more memory which can be a limiting factor for such a system. This is because the Android OS has an upper bound on the amount of memory that can be used by an application which is dependent on factors such as the Android API version and the mobile device configuration. Thus, the proposed mobile CEP system seems to a better option for the mobile devices having a constrained memory.

5.7.8 Comparison of UC_{SCEP} and UC_{MCEP}

As shown in Figure 5.23, for any value of raw event arrival rate, the amount of data transferred per second (TX) is more for the server CEP in comparison to the mobile CEP. This is because the SCEP application forwards all the raw events to IoT server. Equation (4.1) (which was discussed in Chapter 4) is used to compute the data transfer cost incurred by the user for using the MCEP service and SCEP service. The current rate of \$0.05/MB, offered by a major telecommunication company in Canada (Bell [178]) is used. For any given λ_{RE}, a significantly lower data transfer cost is observed for the MCEP system in



λ_{RE}	UC_{MCEP}	UC_{SCEP}
100	$0.14	$0.63
300	$0.40	$2.97
500	$0.50	$5.78
1000	$0.58	$13.32

Figure 5.23: Impact of average raw event arrival rate on UC_{SCEP} and UC_{MCEP}

comparison to the SCEP system as in case of MCEP system only the complex events are sent while in case of the SCEP application the complete raw event streams are forwarded. It is interesting to note that at an arrival rate of 1000 events/second, the MCEP system provides a savings of $12.74/hour ($13.32/hour - $0.58/hour) which is a significant amount for a user.

Chapter 6

Conclusions and Future Work

This chapter first provides a synopsis of the proposed approach for detecting complex events in sensor-based systems (see Section 6.1). Then, the key characteristics of the MCEP and the SCEP system are summarized in Section 6.2. An overview of the performance comparison of the MCEP system and the SCEP systems is provided in Section 6.3. Finally, Section 6.4 discusses the directions for future research.

6.1 Synopsis of the Proposed Research

The primary goal of the proposed approach was to investigate two different architectures for performing complex event processing in sensor-based systems using two approaches: SCEP (centralized server-based approach) and MCEP (edge device-based approach). In 2017, a high-level simulation-based approach, for the SCEP has been presented in [65], which describes three main components for server CEP: Container Management System (CMS), multi-cloud environment and multi-tenant design. We have described a SCEP architecture and implementation of its prototype (in Chapter 4) which have more features than the CEPaaS proposed in [65]. The proof-of-the-concept prototype for the SCEP system is achieved by using the WSO_2 IoT server running as the back-end server and embedding Siddhi CEP on a Google Pixel mobile device [172]. Our SCEP system provides many features such as device enrollment, device authentication, device authorization and

multi-tenancy support. However, such SCEP system has some disadvantages that include the necessity of a persistent network connectivity, high data transfer cost for the user and a larger mobile device power consumption as discussed in Section 5.5.

In order to address these issues, we have devised a MCEP system which can effectively handle the network unavailability problem by performing CEP on the edge device instead of processing the sensor data streams on a remote cloud. This system has been realized by successfully embedding a CEP engine on the mobile device to perform the complete complex event detection on the edge device and send various complex events (alerts) to a remote back-end server to notify the concerned personnel. The proof-of-concept prototype for the proposed technique has been built successfully and tested using a synthetic dataset on a Google Pixel mobile device running on the *Android Nougat*. This system leads to a reduction in user cost, lower mobile device energy consumption, reduction in various latencies (such as processing latency, queuing delay, end-to-end latency) and improvement of the overall performance of the system. To the best of our knowledge, this is the first mobile CEP system which can perform the complete CEP on the mobile edge device [179] [180].

The SCEP and MCEP systems have been tested for a remote patient monitoring use case by varying various system/workload parameters as described in Section 5.5 and Section 5.6 respectively. A comparison of the proposed system with the centralized approach is also provided in Section 5.7.

6.2 Characteristics of SCEP and MCEP systems

A comparison between the key characteristics of the MCEP system and the SCEP system is presented next.

- Network connectivity requirement: The MCEP system does not mandate a persis-

tent Internet connection with the back-end IoT server. Thus, if the network is not available temporarily, the user can still receive the local alarms generated on the mobile device.

- User cost: As shown in Section 5.7.8, the user cost for the MCEP system is significantly lower compared to the cost of the centralized server CEP system. For the typical pricing data available at [178] that was used in the experiment (as discussed in Section 5.7.8), the MCEP system provides savings of approximately $13/hour, over the central server based CEPaaS system. This is because the data transfer is reduced in the MCEP system, as only complex events are sent to the IoT server.
- Security and data privacy: As the mobile CEP system processes the sensor data streams locally, thus the user has better data privacy in comparison to the SCEP system. In order to ensure the data privacy and security of the SCEP system, various authentication and authorization methods have to be employed on the IoT server which can lead to additional delays. Ensuring data privacy and security of a centralized server comes at the expense of processing latency. Thus, the MCEP system has an advantage over the SCEP system as it requires relatively lesser security mechanism to be imposed on the system for ensuring data privacy.
- Out-of-order message delivery: As the SCEP gateway application forwards all the sensor data streams, this can lead to synchronization issues among various sensor streams at the back-end server. This issue is less evident in the MCEP system as the sensor devices are locally connected to the edge device using Wi-Fi or bluetooth connections.

These characteristics lead to the conclusion that the MCEP system has a significant number of benefits over the SCEP system. However, the SCEP system also has a few benefits

over the MCEP system as described below.

1. Predictive analytics: In the MCEP system, only the complex events are sent to the IoT server. This means that the historical data of the patient is not available. However, in the case of the SCEP system, the historical data can be further used by various predictive analytics algorithms using machine learning to predict future alerts.

2. Easier to deploy security mechanisms: The IoT server comes with off-the-shelf authentication and authorization features which are easy to configure. However, in the case of mobile CEP, such features have to be manually added and customized.

6.3 Performance Comparison of SCEP and MCEP systems

A rigorous performance analysis of the prototypes systems for SCEP and MCEP is described in Chapter 5. The key observations from the experiments described in the chapter are summarized next.

- Energy efficiency: For the experiment described in Section 5.7.1, the MCEP system provides up to 2% power savings in comparison to the SCEP system.
- Support for data rate: For the experiments described in this thesis, the mobile CEP application saturates at the arrival rate of 2000 events/second in comparison to the SCEP application which saturates at 1000 events/second. Thus, the MCEP system is able to handle sensors with higher sampling frequencies.
- Delays: The MCEP system has a lower average CEP latency, average queuing delay and average end-to-end delay in comparison to the SCEP system as shown in Section 5.7.2, Section 5.7.3, and Section 5.7.4 respectively.
- Memory utilization of the mobile device: As shown in Figure 5.22, the MCEP application results in a lower average memory usage, in comparison to the SCEP appli-

cation, as in the case of MCEP only the complex events need to be sent to the IoT server.

- CPU utilization: For an arrival rate lower than 1000 events/second, the average CPU utilization of the MCEP application is less than that of the SCEP application as shown in Figure 5.21.
- Load on the back-end server: As shown in the Figure 5.20, at an arrival rate of 1000 events/second, the SCEP system consumes 34% more CPU in comparison to the MCEP system.

6.4 Future Work

Directions for further research include the following:

- The sensor simulator module in the MCEP/SCEP system can be replaced with MySignal wireless/wired sensors [136] which provide real health sensor data. The MySignals system provides an Android-based API to receive and process the sensor data streams sent by the various health sensors. Devising such a system can lead to the development of a commercial prototype of an MCEP system.
- The MCEP system can be extended to form a hybrid CEP system such that real-time analytics is performed on the mobile device and the predictive analytics is being performed on the IoT server using the stored historical data. Investigation of such a system forms an important direction for future research.
- The performance of the current system can be analyzed when multiple devices (one device per user) are enrolled with the IoT server. This would test the scalability of the system as the number of users using sensor-based systems is expected to grow.
- As shown in Figure 5.18, the problem of high queuing latency in the server CEP can

be effectively handled by using a hybrid CEP system having a pre-processing component on the mobile application, which could send the streaming average values computed over a batch time window of some time unit (such as 1 second). This hybrid CEP system can also reduce user cost as the data transfer to the IoT server is reduced. Whether or not such an averaging approach that reduces user cost gives rise to inaccuracies in complex event detection needs investigation.

- Investigation of systems that use a query partitioning methodology can be employed to partition the CEP query by following the principles of distributed CEP [181]. This can effectively distribute the workload between the edge-device and the IoT server.
- To the best of our knowledge, currently, the support for mobile CEP is only available in Android and Raspberry-Pi devices. However, in future, other lightweight CEP engines may be successfully embedded on other platforms such as iOS or Blackberry. It will be interesting to extend the techniques described in this thesis to systems using such devices/platforms.
- A user-friendly feature for managing sensor devices (adding/removing sensor devices) will be a good addition to the commercial version of the system.
- Using multiple mobile devices with one serving as the primary device and the other(s) serving as backups may be helpful when the system is continuously used without an opportunity for recharging the battery of the mobile device. The secondary device can replace the primary device when it runs out of battery power. The investigation of such a system focusing on how to perform an effective "handoff" from one device to the other devices forms an interesting direction for future research.

Appendix A

Dependencies & Algorithms

A.1 Dependencies

Table A.1: Dependencies for the MCEP application

implementation(‘org.wso2.siddhi:siddhi-core:4.1.11’) transitive = false
implementation(‘org.wso2.siddhi:siddhi-query-api:4.1.11’) transitive = false
implementation(‘org.wso2.siddhi:siddhi-query-compiler:4.1.11’) transitive = false
implementation(‘org.wso2.siddhi:siddhi-annotations:4.1.11’) transitive = false
implementation(‘org.apache.log4j.wso2:log4j:1.2.17.wso2v1’) transitive = false
implementation ‘org.osgi:org.osgi.core:6.0.0’
implementation ‘org.eclipse.osgi:org.eclipse.osgi.services:3.3.100.v20120522-1822’
implementation ‘org.wso2.orbit.com.lmax:disruptor:3.3.2.wso2v2’
implementation ‘io.dropwizard.metrics:metrics-core:3.1.0’
implementation ‘org.slf4j:slf4j-api:1.7.21’
implementation(‘org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.0.2’) transitive = false
implementation ‘org.eclipse.paho:org.eclipse.paho.android.service:1.1.1’
compileOnly ‘com.android.support:support-annotations:24.2.0’
annotationProcessor(’org.wso2.siddhi:siddhi-annotations:4.1.11’) transitive = false
testImplementation ‘junit:junit:4.12’
androidTestImplementation ‘com.android.support.test.espresso:espresso-core:3.0.1’

A.2 Algorithms

Algorithm A.1: Algorithm for computing average CPU utilization in MCEP

```
Counter=1
while [$Counter -ge 1]
do
  CurrentCPU=$(adb shell top -n 1 | grep mcep | awk '{print $5}')
  TotalCPU=$((TotalCPU + CurrentCPU))
  AverageCPU=$(echo ''scale=2;$TotalCPU/$Counter'' | bc)
  echo $AverageCPU
  ((Counter++))
done
```

Algorithm A.2: Algorithm for computing average memory usage in MCEP

```
Counter=1
while [$Counter -ge 1]
do
  Data=$(adb shell dumpsys meminfo | grep mcep | sed 's/:.*//')
  List=$(echo $data | tr '' '' ''\n'')
  Current=($List)
  Tuple=${Current[0]}
  CurrentMemory=$(sed 's|[K,]||g' <<< $Tuple)
  TotalMemory=$((TotalMemory + CurrentMemory))
  AverageMemory=$(((TotalMemory / Counter)/ 1000))
  echo $AverageMemory
  ((counter++))
done
```

Appendix B

Sensor Simulator Algorithm

B.1 $Sensor_X$ Generator Algorithm

The $sensor_X$ generator, shown in Algorithm B.1, generates the sensor data stream for a single sensor. Initially, the *StartTime* is recorded with the nanosecond precision current system time (Line 1). Further, the *EndTime* is set by adding the simulation runtime ($Runtime_X$) for $Sensor_X$ in nanoseconds (Line 2). As we need to send sensor data streams to both the mobile device and the timekeeper, various objects such as a *socket*, *server-socket*, and *print-writer* are initialized for both the mobile device (Line 3) and the timekeeper (Line 4). As we are using TCP connections to send the sensor data streams, we need to set the *SoTimeout* value to represent the re-transmission timeout. If the tuple acknowledgment does not come in before the *SoTimeout*, then the tuple will be re-transmitted (Line 5). This *SoTimeout* should be larger than the Round Trip Time (RTT). Setting a low value of *SoTimeout*, will result in duplicate data stream tuples. To make sure that all the data stream tuples are sent, the socket buffer size is increased to 2 GB which is limited by the highest values of integer (2^{32}). Then, a nanosecond sleep time is computed using the arrival rate of the stream (Line 6). Before sending the sensor data streams to the timekeeper and the mobile device, a connection needs to be set up from the sensor simulator to the mobile device and the timekeeper. If both connections are successful (Line 7), then

Algorithm B.1: $Sensor_X$ Generator Algorithm

```
1  StartTime ← System.nanotime()
2  EndTime ← StartTime + 10^9 * Runtime_X
3  Initialize Socket_MD, ServerSocket_MD, PrintWriter_MD
4  Initialize Socket_TK, ServerSocket_TK, PrintWriter_TK
5  Set SoTimeout() and SendBufferSize(2^32) for Socket_MD and Socket_TK
6  SleepTime ← 10^9 / λ_X
7  if Socket_TK.isConnected() and Socket_MD.isConnected() then
8      while System.nanotime() ≤ EndTime do
9          Read the next tuple from file and parse it
10         Time ← System.nanotime()
11         MobileTuple ← new MDEvent(Patient_id, Sensor_id, Tuple_id, Value, Time)
12         MobileTupleString ← MobileTuple.toString()
13         TimeKeeperTuple ← new TKEvent(Tuple_id)          ▷ a light-weight tuple
14         TimeKeeperTupleString ← TimeKeeperTuple.toString()
15         PrintWriter_MD.write(MobileTupleString)
16         PrintWriter_TK.write(TimeKeeperTupleString)
17         Tuple_id ← Tuple_id + 1
18         Thread.Sleep(SleepTime)
19     end
20 end
```

the sensor data streams are sent to the mobile device and the timekeeper until the current system time is less than or equal to the simulation end time (Line 8). The synthetic data is read from a file and parsed in order to generate a tuple (Line 9). The current time is computed using the system nanosecond time (Line 10) and appended with the values read from the file in order to generate a tuple for the mobile device (Line 11). Further, this tuple is converted to a JSON tuple using the *toString* function (Line 12). Similarly, a

light-weight tuple for the timekeeper is generated consisting of a tuple id only (Line 13). Two different tuples (for mobile device and the timekeeper) are generated as the timekeeper is employed just to compute the end-to-end latency and thus does not require the other tuple elements. Finally, a data stream tuple is sent to the mobile device (Line 15) and timekeeper (Line 16) using the appropriate print writer objects. The tuple id is incremented (Line 17) and then the thread sleeps for *SleepTime* nanoseconds (Line 18) after which the next tuple can be generated.

Appendix C

Timekeeper Algorithms

C.1 $Sensor_X$ Listener Algorithm

Algorithm C.1 is responsible for listening to a particular sensor ($Sensor_X$) and appending global time-stamps on it. The sensor listener daemon listens on a port P_X (Line 1) using a server socket object (Line 2). The buffer size of the socket object is set as 2 GB (Line 3). The received raw events are saved to a CSV file located at $Path_X$ (Line 4) using a print writer object (Line 5). The server socket object waits for a connection request from the client socket (Line 10). A linked blocking queue is initialized for the specified buffer size (Line 7). Once the connection is successful (Line 11), then the incoming sensor data stream tuple is read using a buffered reader (Line 13). As the raw event tuple is received, it is appended to a linked blocking queue by an enqueue thread and then written to the CSV file by a dequeue thread (Line 8) which is running on a separate thread. A separate thread for writing to file is used to reduce the TCP acknowledgment latency (as MQTT protocol is used to send the data to Timekeeper). If the write operation to file is performed on the same thread (for both en-queuing and de-queuing), the effective arrival rate will be reduced due to the TCP acknowledgment delay. The dequeue thread for $Sensor_X$ ($dequeThreadSensor_X$) is started on a new thread as discussed in Algorithm C.2. The current system time is fetched (Line 15) and appended as the event

Algorithm C.1: $Sensor_X$ Listener Algorithm

```
1  ListeningPort ← P_X
2  SeverSocket ← new ServerSocket(P_X)
3  SeverSocket.setReceiveBufferSize(2^32)
4  FilePath ← Path_X
5  PrintWriter ← new PrintWriter(Path_X)
6  Running ← true
7  LinkedBlockingQueue ← new LinkedBlockingQueue(2^32)
8  Start DequeThreadSensor_X()                          ▷ on a new thread
9  while Running do
10     SeverSocket.accept()
11     if Socket.isConnected == true then
12         while true do
13             ReceivedData ← BufferedReader.readLine()
14             if ReceivedData ≠ null then
15                 T_gg ← System.nanotime()
16                 FileDataPacket ← ReceivedData + "\t" + T_gg + "\n"
17                 LinkedBlockingQueue.add(FileDataPacket)
18             end
19         end
20     end
21 end
```

global generation time (T_{gg}) to the received sensor data stream tuple (Line 16). Further, this tuple is appended to a linked blocking queue (Line 17) which is then de-queued by an $DequeThreadSensor_X$ algorithm as explained next.

C.2 $DequeThreadSensor_X$ Algorithm

Algorithm C.2 dequeues the global timestamped raw event tuples for $Sensor_X$ from the linked blocking queue (Line 2) and then writes them to a file using a print writer object (Line 3).

Algorithm C.2: $DequeThreadSensor_X$ Algorithm

```
1 while true do
2 |   Tuple ← LinkedBlockingQueue.take()
3 |   PrintWriter.write(Tuple)
4 |   PrintWriter.flush()
5 end
```

C.3 *ComplexEventListener* Algorithm

Algorithm C.3 is a generic algorithm for receiving the complex event stream from various event publishers such as IoT hospital server. For a particular architecture (MCEP or SCEP), the topic name variable represents the topic on which the CE stream is published (Line 1). The MQTT client (publisher in this case) is initialized on the localhost machine by specifying the port number (Line 4). After the connection between the MQTT client and the MQTT broker is successful (Line 5), the MQTT client subscribes to a topic T_{Arch} on which the complex event stream is published by the IHS (Line 6). While the MQTT client is connected to the MQTT broker (Line 9), the payload which consists of a complex event data stream tuple is received (Line 10). Further, this tuple is parsed using regular expressions (Line 12) and then the current system time is computed as global event notification time (Line 13). The file tuple is generated as tab-separated (Line 14) values and further added to a linked blocking queue (Line 15). Another thread reads from this queue

and writes the complex events to a file using the logic which is similar to Algorithm C.2.

Algorithm C.3: *ComplexEventListener* Algorithm

```
1  TopicName ← T_Arch
2  FilePath ← Path_Arch
3  PrintWriter ← new PrintWriter(Path_Arch)
4  MqttClient ← new MqttClient("tcp:\\localhost:1883")
5  MqttClient.connect()
6  MqttClient.subscribe(T_Arch)
7  LinkedBlockingQueue ← new LinkedBlockingQueue(2^32)
8  Start DequeThreadCEP()                          ▷ on a new thread
9  while MqttClient.isConnected == true do
10     ReceivedData ← MqttClient.getPayload()
11     if ReceivedData ≠ null then
12         Parse the input using regular expressions
13         T_gn ← System.nanotime()
14         FileTuple ← ReceivedData + "\t" + T_gn + "\n"
15         LinkedBlockingQueue.add(FileTuple)
16     end
17 end
```

References

[1] P. Zikopoulos and C. Eaton, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media Publications, 2011.

[2] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *The International Journal on Very Large Data Bases (VLDB) Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm@ twitter," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data*, pp. 147–156, 2014.

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[5] L. J. Fülöp, G. Tóth, R. Rácz, J. Pánczél, T. Gergely, A. Beszédes, and L. Farkas, "Survey on Complex Event Processing and Predictive Analytics," in *Proceedings of the 5th Balkan Conference in Informatics*, pp. 26–31, 2010.

[6] Berg Insight, "mHealth and Home Monitoring." [Online available at]: http://www.berginsight.com/reportpdf/productsheet/bi-mhealth6-ps.pdf, [Accessed: 15-March-2018].

[7] Spyglass Consulting Group, "Trends in Remote Patient Monitoring 2015." [Online available at]: http://www.spyglass-consulting.com/wp_RPM_2015.html, [Accessed: 21-March-2018].

[8] J. Boubeta-Puig, G. Ortiz, and I. Medina-Bulo, *Approaching the Internet of Things through integrating SOA and Complex Event Processing*. IGI Global Publishers, 2014.

[9] R. Raj, R. K. Sahu, B. Chaudhary, B. R. Prasad, and S. Agarwal, "Real-time Complex Event Processing and Analytics for Smart Building," in *Proceedings of the Conference on Information and Communication Technology (CICT)*, pp. 1–6, 2017.

[10] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "Siddhi: A Second Look at Complex Event Processing Architectures," in *Proceedings of the ACM Workshop on Gateway Computing Environments (GCE)*, pp. 43–50, 2011.

[11] Amazon, "Amazon Lambda Basics." [Online available at]: https://aws.amazon.com/lambda/, [Accessed: 15-Feb-2018].

[12] IBM, "Big Data & Analytics Hub: Extracting Business Value from the 4 V's of Big Data." [Online available at]: http://www.ibmbigdatahub.com/infographic/extracting-business-value-4-vs-big-data, [Accessed: 21-Mar-2018].

[13] Reinsel, David and Gantz, John and Rydning, John, "Data Age 2025: The Evolution of Data to Life-Critical." [Online available at]: https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia,

B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC)*, pp. 1–16, 2013.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[16] I. Verbitskiy, L. Thamsen, and O. Kao, "When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink," in *Proceedings of the Smart World Congress International IEEE Conferences*, pp. 698–705, 2016.

[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, pp. 28–38, 2015.

[18] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful Scalable Stream Processing at LinkedIn," *International Journal on Very Large Data Bases (VLDB) Journal Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[19] P. C. Brown, *Architecting Complex-Event Processing Solutions with TIBCO*. Addison-Wesley Professional Publications, 2013.

[20] WSO_2, "Stream Processor Documentation." [Online available at]: https://docs.wso2.com/display/SP400/Stream+Processor+Documentation, [Accessed: 24-Feb-2018].

[21] Apache Software Foundation, "Spark Programming Guide." [Online available at]: https://spark.apache.org/docs/latest/streaming-programming-guide.html, [Accessed: 27-July-2018].

[22] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Company, 2001.

[23] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A New Model and Architecture for Data Stream Management," *The International Journal on Very Large Data Bases (VLDB) Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[24] A. Margara and G. Cugola, "Processing Flows of Information: From Data Stream to Complex Event Processing," in *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, pp. 359–360, 2011.

[25] The University of Oslo, "Department of Informatics: Data Stream Management Systems." [Online available at]: https://www.uio.no/studier/emner/matnat/ifi/INF5100/h17/teaching-material/inf5100-dsms-2017.pdf, [Accessed: 27-July-2018].

[26] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, *et al.*, "The Design of the Borealis Stream Processing Engine.," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pp. 277–289, 2005.

[27] M. Cammert, C. Heinz, J. Krämer, and T. Riemenschneider, "CEP Engine and Method for Processing CEP Queries," June 28 2012. US Patent App. 12/929, 539.

[28] G. Cugola and A. Margara, "TESLA: A Formally Defined Event Specification Language," in *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, pp. 50–61, 2010.

[29] D. Gyllstrom, E. Wu, H. J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex Event Processing over Streams," *in Proceedings of the Clinical Orthopaedics and Related Research (CORR)*, vol. abs/cs/0612128, 2006.

[30] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik, "Towards a streaming SQL standard," *International Journal on Very Large Data Bases (VLDB) Journal Endowment*, vol. 1, no. 2, pp. 1379–1390, 2008.

[31] A. Paschke, P. Vincent, and F. Springer, "Standards for Complex Event Processing and Reaction Rules," in *Proceedings of the Rule-based modeling and computing on the semantic web*, pp. 128–139, 2011.

[32] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proceedings of the IEEE International Conference on Data Mining Workshops*, pp. 170–177, 2010.

[33] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache storm perspective," *International Journal of Computer Trends and Technology*, vol. 19, no. 1, pp. 9–14, 2015.

[34] N. Leavitt, "Complex-Event Processing Poised for Growth." [Online available at]: https://www.computer.org/cms/ComputingNow/homepage/news/CO_1209-ComplexEventProcessing.pdf. [Accessed: 28-July-2018].

[35] M. Saboor and R. Rengasamy, *Designing and developing Complex Event Processing Applications*. Sapient Global Markets Publications, 2013.

[36] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S — A Publish/Subscribe Protocol for Wireless Sensor Networks," in *Proceedings of the 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE)*, pp. 791–798, 2008.

[37] Apache Software Foundation, "Kafka: A Distributed Streaming Platform." [Online available at]: https://kafka.apache.org/, [Accessed: 23-April-2018].

[38] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, *Java™ Message Service API Tutorial and Reference: Messaging for the J2EE™ Platform*. Addison-Wesley Professional Publications, 2002.

[39] Apache Software Foundation, "RabbitMQ is the Most Widely Deployed Open-Source Message Broker." [Online available at]: https://www.rabbitmq.com/, [Accessed: 13-April-2018].

[40] Health Level Seven® International, "Health Level Seven Standard, Version 2.3: An Application Protocol for Electronic Data Exchange in Healthcare Environments." [Online available at]: http://www.hl7.org/implement/standards/product_brief.cfm?product_id=140. [Accessed: 27-May-2018].

[41] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of Distributed Computing*, pp. 267–275, 1996.

[42] Oracle Corporation, "Array Blocking Queue Documentation." [Online available at]: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html, [Accessed: 25-May-2018].

[43] Oracle Corporation, "Priority Blocking Queue." [Online available at]: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html, [Accessed: 26-May-2018].

[44] Oracle Corporation, "Synchronous Queue Documentation." [Online available at]: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html, [Accessed: 27-May-2018].

[45] Jenkov Aps, "Blocking Queues." [Online available at]: http://tutorials.jenkov.com/java-concurrency/blocking-queues.html, [Accessed: 27-July-2018].

[46] Eclipse Foundation, "H2 Database Engine." [Online available at]: http://www.h2database.com/html/main.html, [Accessed: 26-Feb-2018].

[47] Apache Software Foundation, "Apache Flink Table API and SQL." [Online available at]: https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/table/tableApi.html#table-api, [Accessed: 23-July-2018].

[48] Y. N. Law, "Models and Operators for Continuous Queries on Data Streams," *Ph.D. Thesis, Department of Computer Science, University of California*, pp. 1–136, 2006.

[49] S. Chakravarthy and Q. Jiang, *Stream data processing: A Quality of Service Perspective: Modeling, Scheduling, Load shedding, and Complex Event Processing*. Springer Science & Business Media, 2009.

[50] Apache Software Foundation, "FlinkCEP - Complex Event Processing for Flink." [Online available at]: https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/libs/cep.html, [Accessed: 27-May-2018].

[51] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over streams," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data*, pp. 407–418, 2006.

[52] WSO_2, "Siddhi Streaming SQL Guide 4.0." [Online available at]: https://wso2.github.io/siddhi/documentation/siddhi-4.0/#query, [Accessed: 27-May-2018].

[53] WSO_2, "Complex Event Processor Documentation." [Online available at]: https://docs.wso2.com/display/CEP300/Windows, [Accessed: 27-July-2018].

[54] Y. Diao, N. Immerman, and D. Gyllstrom, "SASE+: An Agile Language for Kleene Closure over Event Streams," *University of Massachusetts Amherst (UMassAmherst) Technical Report*, pp. 1–13, 2007.

[55] M. Salatino, M. De Maio, and E. Aliverti, *Mastering JBoss Drools 6*. Packt Publishing Ltd., 2016.

[56] EsperTech, "Esper CEP Documentation." [Online available at]: http://www.espertech.com/esper/, [Accessed: 04-May-2018].

[57] Microsoft Corporation, "Microsoft StreamInsight™ is a Powerful Platform to deploy Complex Event Processing (CEP) Applications." [Online available at]: https://docs.microsoft.com/en-us/previous-versions/sql/streaminsight/ee362541(v=sql.105). [Accessed: 8-May-2018].

[58] WSO_2, "The WSO_2 CEP Architecture." [Online available at]: https://github.com/wso2/siddhi, [Accessed: 27-March-2018].

[59] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM Special Interest Group on Operating Systems (SIGOPS) Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[60] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media Inc., 2013.

[61] Google Inc., "What Is Cloud Pub/Sub?." [Online available at]: https://cloud.google.com/pubsub/docs/overview, [Accessed: 27-July-2018].

[62] HiveMQ, "Clustering MQTT – Introduction and Benefits." [Online available at]: https://www.hivemq.com/blog/clustering-mqtt-introduction-benefits/, [Accessed: 21-April-2018].

[63] HiveMQ, "MQTT Essentials." [Online available at]: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels, [Accessed: 27-July-2018].

[64] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in Action*. Manning Publications, 2011.

[65] W. A. Higashino, M. A. M. Capretz, and L. F. Bittencourt, "CEPaaS: Complex Event Processing as a Service," in *Proceedings of the IEEE International Congress on Big Data (BigData Congress)*, pp. 169–176, 2017.

[66] S. Barrett, *Arduino Microcontroller Processing for Everyone: Part I*. Morgan & Claypool Publishers, 2010.

[67] E. Upton and G. Halfacree, *Raspberry Pi® User Guide, 4th Edition*. John Wiley & Sons Publishers, 2017.

[68] P. Fremantle, "WSO_2 Whitepaper: A Reference Architecture for the Internet of Things." [Online available at]: https://wso2.com/download/getfile/wso2_whitepaper_a-reference-architecture-for-the-internet-of-things.pdf. [Accessed: 28-July-2018].

[69] J. Murty, *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media Inc., 2008.

[70] Y. Chen, J. E. Argentinis, and G. Weber, "IBM Watson: how cognitive computing can be applied to big data challenges in life sciences research," *Clinical Therapeutics Journal*, vol. 38, no. 4, pp. 688–701, 2016.

[71] WSO_2, "IoT Server Documentation." [Online available at]: https://docs.wso2.com/display/IoTS310/Architecture, [Accessed: 27-July-2018].

[72] Apple Inc., "Local and Remote Notification Programming Guide : APN's Overview." [Online available at]: https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/APNSOverview.html#//apple_ref/doc/uid/TP40008194-CH8-SW1. [Accessed: 7-August-2018].

[73] L. Moroney, *The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform*. Apress Inc., 2017.

[74] I. Warren, A. Meads, S. Srirama, T. Weerasinghe, and C. Paniagua, "Push notification mechanisms for pervasive smartphone applications," *IEEE Pervasive Computing*, vol. 13, no. 2, pp. 61–71, 2014.

[75] K. Kousen, *Gradle Recipes for Android: Master the New Build System for Android*. O'Reilly Media Inc., 2016.

[76] Google Inc., "Platform Codenames, Versions, API Levels, and NDK Releases." [Online available at]: https://source.android.com/setup/start/build-numbers, [Accessed: 29-March-2018].

[77] Google Inc., "Use Java 8 language Features." [Online available at]: https://developer.android.com/studio/write/java8-support, [Accessed: 29-March-2018].

[78] Google Inc., "Application Fundamentals." [Online available at]: https://developer.android.com/guide/components/fundamentals#Components, [Accessed: 01-April-2018].

[79] Google Inc., "Enable Multidex for Apps with over 64K Methods." [Online available at]: https://developer.android.com/studio/build/multidex, [Accessed: 15-April-2018].

[80] Google Inc., "Android Debug Bridge (ADB)." [Online available at]: https://developer.android.com/studio/command-line/adb, [Accessed: 17-April-2018].

[81] S. Chhajed, *Learning ELK Stack*. Packt Publishing Ltd., 2015.

[82] M. Massie, B. Li, B. Nicholes, V. Vuksan, R. Alexander, J. Buchbinder, F. Costa, A. Dean, D. Josephsen, P. Phaal, *et al.*, *Monitoring with Ganglia: Tracking Dynamic Host and Application Metrics at Scale*. O'Reilly Media Inc., 2012.

[83] V. Sharma, *Beginning Elastic Stack: Graphite monitoring and graphs*. Springer Publications, 2016.

[84] DataDog, "Modern Monitoring and Analytics." [Online available at]: https://www.datadoghq.com/, [Accessed: 28-Feb-2018].

[85] S. Rizvi, "Complex Event Processing Beyond Active Databases: Streams and Uncertainties." [Online available at]: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-26.pdf. [Accessed: 8-August-2018].

[86] F. Bry, M. Eckert, P.-L. Pătrânjan, and I. Romanenko, "Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits," in *Proceedings of the International Workshop on Principles and Practice of Semantic Web Reasoning*, pp. 48–62, 2006.

[87] N. H. Gehani and H. V. Jagadish, "Ode as an Active Database: Constraints and Triggers," in *Proceedings of the International Journal on Very Large Data Bases (VLDB) Journal*, pp. 327–336, 1991.

[88] S. Gatziu, A. Geppert, and K. R. Dittrich, "The SAMOS Active DBMS prototype," in *Proceedings of the Special Interest Group on Management of Data (SIGMOD) Conference*, pp. 1–11, 1995.

[89] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *Data & Knowledge Engineering*, vol. 14, no. 1, pp. 1–26, 1994.

[90] D. C. Luckham, "Rapide: A Language and toolset for Causal Event Modelling of Distributed System Architectures," in *Proceedings of the Worldwide Computing and Its Applications (WWCA)*, pp. 88–96, 1998.

[91] M. Gualtieri and J. R. Rymer, "The Forrester Wave™: Complex Event Processing (CEP) platforms, Q3 2009." [Online available at]: http://www.reinsa.co.cr/2009wave_complex_event_processing_cep_platforms_q3.pdf. [Accessed: 5-August-2018].

[92] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query Processing, Resource Management, and Approximation in a Data Stream Management System," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR) Stanford InfoLab*, pp. 1–12, 2002.

[93] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data*, pp. 668–668, 2003.

[94] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, pp. 379–390, 2000.

[95] S. B. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The Aurora and Medusa Projects," *IEEE Database Engineering Bulletin*, vol. 26, no. 1, pp. 3–10, 2003.

[96] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a Stream Database for Network Applications," in *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data*, pp. 647–651, 2003.

[97] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, *STREAM: The Stanford Data Stream Management System*. Springer Publications, 2016.

[98] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. M. White, *et al.*, "Cayuga: A General Purpose Event Monitoring System," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pp. 412–422, 2007.

[99] A. S. Al-Haboobi, A. F. Alharan, and R. H. Alsagheer, "Experimenting with Storm/Esper Integration and Programmatic Generation of Storm Topologies," *International Journal of Computer Science and Information Security*, vol. 14, no. 8, pp. 169–178, 2016.

[100] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, pp. 1–1, 2014.

[101] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pp. 295–308, 2011.

[102] S. Krishnan and J. L. U. Gonzalez, *Google Compute Engine*. Springer Publications, 2015.

[103] TIBCO Software Inc., "StreamBase Whitepaper." [Online available at]: https://www.tibco.com/sites/tibco/files/resources/DS-TIBCO-StreamBase-final.pdf, [Accessed: 05-Feb-2018].

[104] Oracle Corporation, "Complex Event Processing Getting Started." [Online available at]: https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/overview.htm#CEPGS106, [Accessed: 01-Feb-2018].

[105] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM Infosphere Streams for Scalable, Real-time, Intelli-

gent Transportation Services," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1093–1104, 2010.

[106] Software AG, "Apama Complex Event Processing Platform." [Online available at]: https://www.softwareag.com/corporate/products/apama_webmethods/analytics/default, [Accessed: 03-Feb-2018].

[107] Forrester Wave, "The Forrester Wave - Streaming Analytics, Q3 2017." [Online available at]: https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=IML14592USEN, [Accessed: 07-Feb-2018].

[108] A. Adi and O. Etzion, "Amit-The Situation Manager," *The International Journal on Very Large Data Bases (VLDB) Journal*, vol. 13, no. 2, pp. 177–203, 2004.

[109] G. Cugola and A. Margara, "Complex Event Processing with T-REX," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1709–1728, 2012.

[110] TIBCO Software Inc., "Introduction to TIBCO StreamBase® Complex Event Processing." [Online available at]: https://www.tibco.com/resources/demand-webinar/introduction-tibco-streambase-complex-event-processing, [Accessed: 10-Feb-2018].

[111] W. A. Higashino, M. A. M. Capretz, and L. F. Bittencourt, "CEPSim: A Simulator for Cloud-Based Complex Event Processing," in *Proceedings of the IEEE International Congress on Big Data*, pp. 182–190, 2015.

[112] W. A. Higashino, "Complex Event Processing as a Service in Multi-Cloud Environments," *Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Western Ontario (UWO)*, pp. 1–207, 2016.

[113] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of Scalable Cloud Computing Environments and the CloudSim toolkit: Challenges and Opportunities," in *Proceedings of the International Conference on High Performance Computing Simulation*, pp. 1–11, 2009.

[114] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Journal of Software—Practice & Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[115] Amazon, "Get started with Amazon Kinesis." [Online available at]: https://aws.amazon.com/kinesis/, [Accessed: 11-Feb-2018].

[116] IBM, "Bluemix Homepage." [Online available at]: https://www.ibm.com/cloud/, [Accessed: 03-April-2018].

[117] Github, "Node Red CEP." [Online available at]: https://github.com/CeZL/node-red-contrib-cep#readme, [Accessed: 19-Feb-2018].

[118] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable Stream Computation in the Cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 1–14, 2013.

[119] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An Elastic and Scalable Data Streaming System," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[120] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proceedings of the Sixth International Conference on Advances in Future Internet*, pp. 48–55, Citeseer, 2014.

[121] Y. Jararweh, F. Ababneh, A. Khreishah, F. Dosari, *et al.*, "Scalable cloudlet-based Mobile Computing Model," *Procedia Computer Science*, vol. 34, pp. 434–441, 2014.

[122] R. Roman, J. Lopez, and M. Mambo, "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges," *Future Generation Computer Systems*, vol. 78, no. 1, pp. 680–698, 2018.

[123] K. Pooja, K. Chandrashekar, M. Thungamani, and C. Gireesh Babu, "Complex Event Processing In Smart Homes," *International Journal of Scientific Engineering and Applied Science (IJSEAS)*, vol. 1, no. 2, pp. 544–550, 2015.

[124] N. Sinha, K. E. Pujitha, and J. S. R. Alex, "Xively based Sensing and Monitoring System for IoT," in *Proceedings of the International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1–6, 2015.

[125] D. Dossot, J. d'Emic, and V. Romero, *Mule in Action*. Manning Publications, 2014.

[126] M. Kamel and L. George, "Remote Patient Tracking and Monitoring System," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 12, pp. 88–94, 2013.

[127] S. Stipkovic, R. Bruns, and J. Dunkel, "Pervasive Computing by Mobile Complex Event Processing," in *Proceedings of the 10th IEEE International Conference on e-Business Engineering*, pp. 318–323, 2013.

[128] The University of Hamburg, "Esper-Android Event Stream Processing on Android." [Online available at]: https://vsis-www.informatik.uni-hamburg.de/oldServer/teaching//projects/esper-android/, [Accessed: 12-April-2018].

[129] O. Banos, C. Villalonga, M. Damas, P. Gloesekoetter, H. Pomares, and I. Rojas, "Physiodroid: Combining wearable health sensors and mobile devices for a ubiquitous, continuous, and personal monitoring," *The Scientific World Journal*, vol. 1, no. 1, pp. 1–11, 2014.

[130] P. Kakria, N. Tripathi, and P. Kitipawang, "A Real-Time Health Monitoring System for Remote Cardiac Patients Using Smartphone and Wearable Sensors," *International Journal of Telemedicine and Applications*, vol. 1, no. 23, pp. 1–11, 2015.

[131] R. Pathak and V. Vaidehi, "Complex Event Processing Based Remote Health Monitoring System," in *Proceedings of the 3rd International Conference on Eco-friendly Computing and Communication Systems*, pp. 61–66, 2014.

[132] V. Vaidehi, R. Bhargavi, K. Ganapathy, and C. S. Hemalatha, "Multi-sensor based in-home Health Monitoring using Complex Event Processing," in *Proceedings of the International Conference on Recent Trends in Information Technology*, pp. 570–575, 2012.

[133] A. N. Lam and Ø. Haugen, "Complex Event Processing in ThingML," in *Proceedings of the International Conference on System Analysis and Modeling*, pp. 20–35, 2016.

[134] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "ThingML: a Language and Code Generation Framework for Heterogeneous Targets," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pp. 125–135, 2016.

[135] A. Mdhaffar, I. B. Rodriguez, K. Charfi, L. Abid, and B. Freisleben, "CEP4HFP: Complex Event Processing for Heart Failure Prediction," *IEEE Transactions on NanoBioscience*, vol. 16, no. 8, pp. 708–717, 2017.

[136] Cooking Hacks, "MySignals changes the Future of Medical and eHealth Applications." [Online available at]: http://www.my-signals.com/, [Accessed: 09-Feb-2018].

[137] J. Mohammed, C. Lung, A. Ocneanu, A. Thakral, C. Jones, and A. Adler, "Internet of Things: Remote Patient Monitoring Using Web Services and Cloud Computing," in *Proceedings of the IEEE International Conference on Internet of Things (iThings)*, pp. 256–263, 2014.

[138] SparkFun Electronics, "IOIO-OTG." [Online available at]: https://www.sparkfun.com/products/retired/11343, [Accessed: 28-July-2018].

[139] FileZilla®, "Welcome to the Homepage of FileZilla®." [Online available at]: https://filezilla-project.org/, [Accessed: 28-July-2018].

[140] P. Dutson, *Android Development Patterns: Best Practices for Professional Developers*. Addison-Wesley Professional Publications, 2016.

[141] J. Kharel, H. T. Reda, and S. Y. Shin, "An Architecture for Smart Health Monitoring System Based on Fog Computing," in *Proceedings of the Global Wireless Summit (GWS)*, pp. 228–233, 2017.

[142] J. Kharel, H. T. Reda, and S. Y. Shin, "Fog Computing-Based Smart Health Monitoring System Deploying LoRa Wireless Communication," *IETE Technical Review*, vol. 0, no. 0, pp. 1–14, 2018.

[143] F. A. Aoudia, M. Gautier, M. Magno, M. Le Gentil, O. Berder, and L. Benini, "Long-short Range Communication Network Leveraging LoRa™ and wake-up Receiver," *Microprocessors and Microsystems*, vol. 56, no. 0, pp. 184–192, 2018.

[144] T. Reza, S. B. A. Shoilee, S. M. Akhand, and M. M. Khan, "Development of Android based Pulse Monitoring System," in *Proceedings of the 2nd International Conference on Electrical Computer and Communication Technologies (ICECCT)*, pp. 1–7, 2017.

[145] V. Wahane and P. Ingole, "An Android based wireless ECG Monitoring System for Cardiac Arrhythmia," in *Proceedings of the Healthcare Innovation Point-Of-Care Technologies Conference (HI-POCT)*, pp. 183–187, 2016.

[146] P. Dineshkumar, R. SenthilKumar, K. Sujatha, R. S. Ponmagal, and V. N. Rajavarman, "Big data Analytics of IoT based Healthcare Monitoring System," in *Proceedings of the IEEE Uttar Pradesh Section International Conference on Electrical Computer and Electronics Engineering (UPCON)*, pp. 55–60, 2016.

[147] R. Senthilkumar, R. Ponmagal, and K. Sujatha, "Efficient Health Care Monitoring and Emergency Management System using IoT," *International Journal of Control Theory and Applications*, vol. 9, no. 4, pp. 137–145, 2016.

[148] M. Suh, C. Chen, J. Woodbridge, M. K. Tu, J. I. Kim, A. Nahapetian, L. S. Evangelista, and M. Sarrafzadeh, "A Remote Patient Monitoring System for Congestive Heart Failure," *Journal of Medical Systems*, vol. 35, no. 5, pp. 1165–1179, 2011.

[149] S. Naddeo, L. Verde, M. Forastiere, G. De Pietro, and G. Sannino, "A Real-time m-Health Monitoring System: An Integrated Solution Combining the Use of Several Wearable Sensors and Mobile Devices," in *Proceedings of the International Conference on Health Informatics (HEALTHINF)*, pp. 545–552, 2017.

[150] T. Naqishbandi, C. Imthyaz Sheriff, and S. Qazi, "Big data, CEP and IoT: redefining holistic healthcare information systems and analytics," *International Conference on Advances Research in Engineering and Technology*, vol. 4, no. 1, pp. 1–6, 2015.

[151] F. Chisanga, "Medical Application of the Internet of Things (IoT): Prototyping a Telemonitoring System," *Ph.D. Thesis, Department of Electrical Engineering, University of Cape Town*, pp. 1–117, 2018.

[152] F. Chisanga, N. Ventura, and J. Mwangama, "Prototyping a Cardiac Arrest Telemonitoring System," in *Proceedings of the Global Wireless Summit (GWS)*, pp. 170–174, 2017.

[153] M. Corici, H. Coskun, A. Elmangoush, A. Kurniawan, T. Mao, T. Magedanz, and S. Wahle, "OpenMTC: Prototyping Machine type Communication in Carrier Grade Operator Networks," in *Proceedings of the IEEE Globecom Workshops*, pp. 1735–1740, 2012.

[154] Gibson Research Corporation, "Zeo Sleep Manager-Pro." [Online available at]: https://www.grc.com/zeo.htm, [Accessed: 06-March-2018].

[155] Sotera Wireless, "About Visi Mobile." [Online available at]: https://vandrico.com/wearables/device/visi-mobile, [Accessed: 25-March-2018].

[156] H. Kreger, W. Harold, and L. Williamson, *Java and JMX: Building Manageable Systems*. Addison-Wesley Longman Publishing Company, 2002.

[157] M. Chung, "Using JConsole to Monitor Applications." [Online available at]: http://www.oracle.com/technetwork/articles/java/jconsole-1564139.html. [Accessed: 28-July-2018].

[158] Apache Software Foundation, "OpenWire Protocol." [Online available at]: http://activemq.apache.org/apollo/documentation/openwire-manual.html, [Accessed: 22-May-2018].

[159] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook Whitepaper Journal*, vol. 5, no. 8, pp. 1–8, 2007.

[160] A. S. Foundation, "Apache thrift™." [Online available at]: https://thrift.apache.org/. [Accessed: 28-July-2018].

[161] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media Inc., 2013.

[162] S. Patro, M. Potey, and A. Golhani, "Comparative Study of Middleware Solutions for Control and Monitoring Systems," in *Proceedings of the 2nd International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pp. 1–10, 2017.

[163] Apache Software Foundation, "KahaDB Homepage." [Online available at]: http://activemq.apache.org/kahadb.html, [Accessed: 05-April-2018].

[164] T. Powell, *Ajax: The Complete Reference*. McGraw-Hill Inc., 2008.

[165] M. Masse, *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media Inc., 2011.

[166] Eclipse Foundation, "Jetty Server's Homepage." [Online available at]: https://www.eclipse.org/jetty/, [Accessed: 15-May-2018].

[167] Y. S. Yilmaz, B. I. Aydin, and M. Demirbas, "Google Cloud Messaging (GCM): An Evaluation," in *Proceedings of the IEEE Global Communications Conference*, pp. 2807–2812, 2014.

[168] DataDog, "Dive into DogStatsD." [Online available at]: https://docs.datadoghq.com/developers/dogstatsd/, [Accessed: 02-Feb-2018].

[169] Wikipedia, "Lesson: Introducing MBeans." [Online available at]: https://docs.oracle.com/javase/tutorial/jmx/mbeans/, [Accessed: 28-July-2018].

[170] R. Ballagas, M. Rohs, J. G. Sheridan, and J. Borchers, "BYOD: Bring Your Own Device," in *Proceedings of the Workshop on Ubiquitous Display Environments*, pp. 1–8, 2004.

[171] Citrix, "Best Practices to make BYOD, CYOD and COPE Simple and Secure." [Online available at]: https://www.citrix.com/content/dam/citrix/en_us/documents/white-paper/byod-best-practices.pdf, [Accessed: 24-July-2018].

[172] Wikipedia, "Google Pixel." [Online available at]: https://en.wikipedia.org/wiki/Google_Pixel, [Accessed: 28-July-2018].

[173] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C. K. Peng, and H. E. Stanley, "PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals," *Circulation Journal*, vol. 101, no. 23, pp. e215–e220, 2000.

[174] T. Berglund and M. McCullough, *Building and Testing with Gradle*. O'Reilly Media Inc., 2011.

[175] W. H. Organization, "WHO Global Report on Falls Prevention in Older Age." [Online available at]: http://www.who.int/ageing/publications/Falls_prevention7March.pdf, [Accessed: 8-March-2018].

[176] M. A. Habib, M. S. Mohktar, S. B. Kamaruzzaman, K. S. Lim, T. M. Pin, and F. Ibrahim, "Smartphone-based solutions for fall detection and prevention: challenges and open issues," *Sensors Journal*, vol. 14, no. 4, pp. 7181–7208, 2014.

[177] A. Bakker, "Comparing Energy Profilers for Android," in *Proceedings of the 21st Twente Student Conference on IT*, pp. 1–8, 2014.

[178] Bell Canada, "Data Charges." [Online available at]: https://support.bell.ca/Mobility/Rate_plans_features/What-are-Bell-Mobilitys-current-pay-per-use-rates, [Accessed: 05-July-2018].

[179] A. S. Dhillon, S. Majumdar, M. S. Hilaire, and A. E. Haraki, "A Mobile Complex Event Processing System for Remote Patient Monitoring," in *Proceedings of the IEEE International Conference on Internet of Things (IEEE-ICIOT)*, pp. 1–4, 2018.

[180] A. S. Dhillon, S. Majumdar, M. S. Hilaire, and A. E. Haraki, "MCEP: A Mobile device based Complex Event Processing System for Remote Healthcare," in *Proceedings of the 11th IEEE International Conference on Internet of Things (iThings)*, pp. 203–210, 2018.

[181] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel, "Distributed Heterogeneous Event Processing: enhancing Scalability and Interoperability of CEP in an Industrial Context," in *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, pp. 150–159, 2010.

[182] X. Zhang, J. Liu, B. Li, and Y. P. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 2102–2111, 2005.

[183] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing

in Spark," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394, 2015.

[184] E. Fidler, H. A. Jacobsen, G. Li, and S. Mankovski, "The PADRES Distributed Publish/Subscribe System," in *Proceedings of the Principles and Applications of Distributed Event-Based Systems*, pp. 164–205, 2005.

[185] P. Dubroy, "Memory Management for Android Apps," in *Proceedings of the Google I/O Development Conference*, 2011.

[186] J. Kharel, H. T. Reda, and S. Y. Shin, "Fog Computing-Based Smart Health Monitoring System Deploying LoRa Wireless Communication," pp. 1–14, 2018.

[187] J. Allen, *Effective Akka: Patterns and Best Practices*. O'Reilly Media Inc., 2013.

[188] K. M. Chandy and W. R. Schulte, *Event processing: Designing IT systems for Agile Companies*. McGraw Hill Publications, 2010.

[189] S. Majumder, T. Mondal, and M. J. Deen, "Wearable Sensors for Remote Health Monitoring," *Sensors Journal*, vol. 17, no. 130, pp. 1–45, 2017.

[190] C. K. Emani, N. Cullot, and C. Nicolle, "Understandable Big Data: A survey," *Computer Science Review*, vol. 17, no. 0, pp. 70–81, 2015.

[191] V. S. Cheng and P. C. Hung, "Health insurance portability and accountability act (hippa) compliant access control model for web services," *International Journal of Healthcare Information Systems and Informatics (IJHISI)*, vol. 1, no. 1, pp. 22–39, 2006.

[192] SAP, "Capture, Analyze and Act on Real-time Event Streams – with our Complex Event Processing Platform." [Online available at]: https://www.sap.com/products/complex-event-processing.html, [Accessed: 04-Feb-2018].

[193] W. M. Schutz, "Getting Started with Complex Event Processing Nodes." [Online available at]: https://www.research.ibm.com/haifa/dept/services/papers/CEP-Primer_V1_2.pdf, [Accessed: 2-July-2018].