Future Generation Computer Systems 117 (2021) 439–452

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Akka framework based on the Actor model for executing distributed Fog Computing applications

Satish Narayana Srirama [a,*], Freddy Marcelo Surriabre Dick [b], Mainak Adhikari [b,*]

[a] *School of Computer and Information Sciences, University of Hyderabad, India*
[b] *Mobile & Cloud Lab, Institute of Computer Science, University of Tartu, Estonia*

A R T I C L E I N F O

Article history:
Received 19 November 2019
Received in revised form 18 October 2020
Accepted 10 December 2020
Available online 22 December 2020

Keywords:
Internet of Things
Fog Computing
Actor programming model
Akka toolkit
Distributed processing
Docker

A B S T R A C T

Future Internet of Things (IoT)-driven applications will move from the cloud-centric IoT model to the hybrid distributed processing model, known as Fog computing, where some of the involved computational tasks (e.g. real-time data analytics) are partially moved to the edge of the network to reduce latency and improve the network efficiency. In recent times, Fog computing has generated significant research interest for IoT applications, however, there is still a lack of ideal approach and framework for supporting parallel and fault-tolerant execution of the tasks while collectively utilizing the resource-constrained Fog devices. To address this issue, in this paper, we propose an Akka framework based on the Actor Model for designing and executing the distributed Fog applications. The Actor Model was conceived as a universal paradigm for concurrent computation with additional requirements such as resiliency and scalability, whereas, the Akka toolkit is a reference implementation of the model. Further, to dynamically deploy the distributed applications on the Fog networks, a Docker containerization approach is used. To validate the proposed actor-based framework, a wireless sensor network case study is designed and implemented for demonstrating the feasibility of conceiving applications on the Fog networks. Besides that, a detailed analysis is produced for showing the performance and parallelization efficiency of the proposed model on the resource-constrained gateway and Fog devices.

1. Introduction

The Internet of Things (IoT) represents a comprehensive environment that interconnects a large number of heterogeneous physical objects with sensing and actuating capabilities through the Internet to enhance the efficiency of the smart applications in the real-time domains such as ambient assisted living, domotics, smart cities, logistics, manufacturing, agriculture, etc. In the IoT domain, the common methodology is Cloud-centric IoT (CIoT) [1], where the sensor data is collected from the physical objects through the local gateways, which is later pushed/pulled, stored and processed on the centralized cloud servers. Further, the cloud servers raise alarms and generate the control signals for the actuators according to the analyzed sensor data and the scenario. However, for processing/analyzing the low-latency and high resilience applications, CIoT has significant issues due to relying on communications with the far-away cloud data centers [2]. Consequently, the researchers have proposed different ubiquitous computing models such as Mobile Edge Computing (MEC) [3], mobile cloud [4] and Fog computing [5] for resolving the communication issue between the local IoT devices and remote computing devices.

Nowadays, Fog/Edge computing enables distributed application components/tasks from the centralized cloud servers to execute in the intermediate local Fog/Edge devices (e.g. routers, switches, gateways/hubs, proximal computational resources including laptops and private clouds, etc.), to reduce the latency by minimizing the data transmission time between the front-end IoT devices and the back-end cloud servers. Today, the Fog and Edge computing are being used interchangeably and this paper takes a similar stance [5]. Tasks such as data pre-processing, aggregation, and real-time feedback loops can take place at network Edge devices and local Fog nodes, while the cloud servers handle storage and offline big data processing. Fig. 1 shows the Fog computing architecture with the edge analytics performed closer to the Edge network and the large-scale distributed data analytics performed on the centralized cloud servers. Specifically, the Fog provides five basic mechanisms: storage, compute, acceleration, networking, and control towards enhancing the IoT systems in five subjects: security, cognition, agility, low latency, and efficiency [5].

Fog computing allows two types of applications at the high level [6]. (A) Applications that are managed by a cloud provider, in

* Corresponding author.
E-mail addresses: satish.srirama@uohyd.ac.in (S.N. Srirama), mainak.adhikari@ut.ee (M. Adhikari).

https://doi.org/10.1016/j.future.2020.12.011
0167-739X/© 2020 Elsevier B.V. All rights reserved.

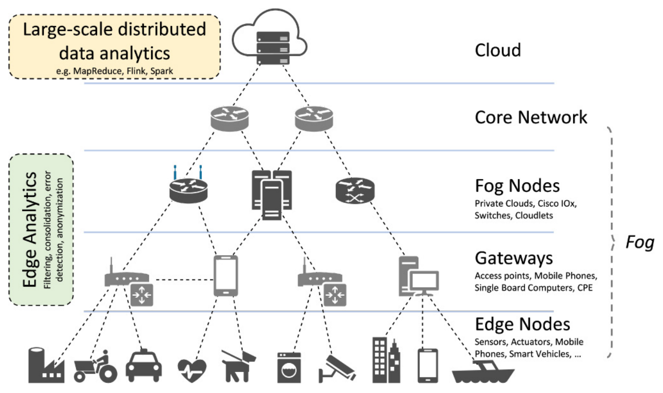
Large-scale distributed data analytics
e.g. MapReduce, Flink, Spark
Cloud
Core Network
Edge Analytics
Filtering, consolidation, error detection, anonymization
Fog Nodes
Private Clouds, Cisco IOx, Switches, Cloudlets
Gateways
Access points, Mobile Phones, Single Board Computers, CPE
Edge Nodes
Sensors, Actuators, Mobile Phones, Smart Vehicles, ...
Fog

Fig. 1. Fog topology with main activities at the layers.

a top-down approach, where the cloud provider deploys the necessary geo-distributed resources and manages the IoT application execution across the topology by considering proximity and QoS parameters such as user load, latency, cost models, etc. (B) Applications that are managed by the individual service providers, where the Fog resources are provided by different vendors such as general public, private clouds, network operators, etc. In this case, the gateways decide how to schedule the components across the Fog topology (bottom-up approach). Both application types face major challenges with the QoS-aware resource provisioning for the Fog providers and optimal placement of the modules for the IoT applications. Al-Khafajiy et al. [7] have proposed different types of resource provisioning policies in Fog network by optimizing the power control and latency, respectively. Mahmud et al. [8] and Adhikari et al. [9] have discussed various types of data offloading policies in Fog network by considering Quality-of-Experience and multi-objective optimization, respectively. Further, Wu et al. [10], and Alam et al. [11] have introduced different types of Fog placement policies using various heuristic strategies in Fog and MEC environment.

1.1. Motivation

Most of the existing developed IoT solutions fully/partially fall back to the centralized cloud servers due to resource unavailability or failure to guarantee the QoS objectives of the local Fog devices in the network. This raises the complexity of designing and deploying efficient IoT-driven applications harnessing the Fog networks. As a result, there are two significant challenges for processing IoT-driven applications efficiently on the Fog network. Firstly, ***how to develop a distributed framework in Fog network for efficiently processing/analyzing IoT applications?*** This faces a major issue to develop a QoS-aware resource provisioning mechanism in Fog networks for optimal placement of modules for the IoT applications, and fully/partially fall back to the centralized cloud, under resource unavailability or failure to guarantee QoS at fog network. Thus, a suitable resource provisioning mechanism can help to process/analyze the IoT applications at the edge of the network instead of the centralized cloud servers with minimum processing delay. Secondly, ***how to distribute the applications on the local Fog devices (also known as worker nodes) with a suitable load balancing policy?*** This faces another issue to develop lightweight containerization approaches in the Fog networks for deploying the IoT applications on the local Fog devices with efficient resource utilization. This can help to achieve parallel and fault-tolerant execution of the tasks while utilizing the resource-constrained Fog devices.

1.2. Contributions

The Actor Model, which defines the actors as its basic unit of computation, addresses the need of working in a distributed environment with the requirements of concurrency, resiliency, and scalability among others [12,13]. The Akka toolkit[1] is an implementation of the Actor Model, which offers a series of modules and libraries through its platform that can be used to build concurrent and distributed applications [14]. By motivating from the distributed nature of the Actor Model and Akka framework, we first design an Akka framework based on the Actor Model for Fog computing. Further, we develop a policy for dynamically deploying the IoT applications on the local Fog devices. The main goal of this work is to support the parallel and fault-tolerant execution of the tasks on the local resource-constrained Fog devices for minimizing the processing time. Further, a lightweight Docker containerization approach is used to deploy the application on a distributed Fog devices with a suitable load balancing policy for utilizing the computing resources efficiently. To demonstrate the effectiveness of the proposed Akka framework based on the Actor Model for Fog computing in terms of processing time and load evaluation, we consider a cloud-centric Wireless Sensor Network (WSN)-aware IoT scenario for analyzing the IoT applications. The major contributions of the paper are summarized as follows.

1. We develop a framework for IoT applications using Actor Programming Model with Akka toolkit in Fog network for parallel and fault-tolerant execution of the tasks on the local Fog devices with minimum processing time.
2. We introduce a lightweight Docker containerization approach in the Fog network for deploying the application on distributed Fog devices with efficient utilization of the computing resources.
3. Finally, we demonstrate the efficiency of the proposed Akka framework based on the Actor Model for Fog computing in terms of processing time and load evaluation using a use case study, i.e. WSN-aware IoT scenario.

1.3. Paper organization

The rest of the paper is structured as follows: Section 2 discusses the related works on different models and strategies of Fog computing for processing real-time applications. Section 3 describes the overview of the Actor programming model and the Akka framework. Section 4 discusses the architecture and working methodology of an existing wireless sensor network for centralized data processing. Section 5 discusses the proposed architecture by applying the Akka framework based on the Actor Model for Fog computing through a WSN-aware case study. Further, this section discusses the task distribution and dynamical deployments of the tasks on the Fog networks. Section 6 discusses the experimental evaluation along with various performance metrics. Finally, Section 7 concludes the paper with future research directions.

2. Related work

Nowadays, it is interesting to note that there is significant attention in the Fog computing for processing the real-time applications [5,15] and the terminologies used in this context is very broad. Traditionally, if the processing was performed on the Edge devices at the edge of the network then it was called Edge computing, however, in the context of IoT, it was called mist

1 https://doc.akka.io/docs/akka/current/typed/cluster-sharding.html

computing [16]. Similarly, offloading the data from the resource-constrained device to a powerful computing device in proximity was called as Cloudlet [17]. This scenario also got extended to the mobile cloud scenario as well, where the process-intensive tasks were offloaded to the private or public cloud servers [4]. On the other hand, if the devices were using the mobile operator network, it was called as Multi-access Edge computing (MEC) [18].

Several works have been dealing with the computation at the network edge, developing the frameworks and platforms that can be applied to different domains. Feng et al. [19] have proposed a framework in Edge computing on the road for vehicles by efficiently utilizing the available resources on the Edge devices. Chang et al. [20] have proposed the idea of the Indie Fog infrastructure, in which the user's Edge devices such as routers are used for providing a computational service platform on the network edge. Long et al. [21] have proposed an Edge computing framework for video data processing based on the availability of mobile devices and their computation power.

Fürst et al. [22] have proposed the actor based execution framework for distributed IoT applications, called as Nandu. This framework is designed based on the Actor Model that can adapt and migrate the tasks dynamically based on the information provided by the developers. Elgamal et al. have designed a scalable dynamic programming algorithm in the Edge network [23]. The main goal of this work is to partition the real-time applications across the local Edge devices and the centralized cloud servers for minimizing the completion time of the end-to-end operations. Sanchez et al. have designed an actor-based model with Akka toolkit for crowdsensing and IoT applications while enabling new features in the multi-cloud environment with low resource consumption and robustness [14]. Aske et al. have developed an actor-based distributed framework in the Edge environment, namely ActorEdge [12]. The main objective of this framework is to utilize the computing resources efficiently with minimum latency by distributing real-time applications on the local Edge devices dynamically.

Aiello et al. have designed a mobile agent platform for wireless sensor networks based on SPOT technology using a Java-based framework for supporting agent-oriented programming of wireless sensor applications [24]. Fortino et al. have introduced an agent-based computing paradigm for supporting the design, implementation, and analysis of the IoT applications [25]. The agent-oriented approach is developed based on the concept of the agent-based cooperating smart object methodology that provides effective programming models and actor design. Spencer et al. have developed an accurate and inexpensive framework for next-generation wireless smart sensors with efficient structural health monitoring systems by providing extensible, reusable, and modular middleware services for minimizing data loss and efficient resource utilization [26]. Lohstroh et al. have proposed the actor-based framework with interface automata for processing the IoT applications, called as accessors [27]. This framework receives the real-time input stimuli and generates the final output in each time-frame using different actors. Hiesgen et al. have designed a C++ actor framework for native development of the IoT applications. The main objectives of the work are to minimize the memory footprint and maximize the efficiency of the environment [28]. Another important framework to develop the decentralized, cooperating, dynamic, and heterogeneous IoT ecosystems jointly with other well-established/emerging computing paradigms is Agent-based Computing (ABC) [29]. The ABC provides the idea, methods, techniques, and tools for programming, simulation, and systematic conceptualization for interacting with heterogeneous IoT applications in a distributed environment.

Most of the related works have dealt with the computation at the edge using the typical distributed Edge model without further considering the importance of the models and the relationships behind the connectivity of the nodes in the network. More specifically, the Actor Model is helpful for designing an application model to build an efficient framework for processing the IoT applications efficiently. In addition, the Akka toolkit is used as the main framework for providing a robust Edge architecture, which can be used to conceive applications on the network edge by making use of all the available power of the different nodes to compose the network. In current years, there are several actor based frameworks that are designed for processing the IoT application by providing a higher level of abstraction. However, most of the frameworks fail to support the parallel and fault-tolerant processing of the IoT applications with efficient utilization of the resource constraint devices in Fog network. To address the above-mentioned challenges, in this paper, we have designed a distributed Fog computing framework with the Actor programming Model and Akka toolkit. Besides that, the Docker containerization approach is used for deploying the IoT applications efficiently on a dynamic Fog environment with minimum processing time.

3. Akka Toolkit based on the Actor Programming Model

In this section, we discuss the basic overview of the Actor Programming Model and the Actor Model of computation followed by the overview of the Akka toolkit along with their advantages.

3.1. Actor programming model

The Actor Model consists of a set of actors, which are isolated, concurrent, and solely interacted through a network with a transparent message-passing technique [30]. The Actor Model with the artificial intelligence technique was developed by Carl Hewitt in 1973 [31]. The model was conceived as a universal paradigm for concurrent computation in a highly concurrent and parallelizable distributed environment. At a higher level, the model is simple and supports a high degree of parallelism. The basic unit of the computation model is an Actor. An Actor is an entity that can communicate with other actors through messages in a network [13]. An actor can also create other actors in the network as per the requirements. In this case, the creator actor will be the "parent actor" and the created actors will be the "child actors". The main advantages of the Actor Model are listed as follows.

1. An Actor Model extends the benefits of object-oriented language by splitting the business logic and control flow.
2. An Actor Model allows to decompose a system into independent, autonomous, and interactive components, that can work simultaneously.

An actor cannot corrupt the state of the remaining actors in the network and avoids the race condition. Each actor calls *spawn* operation for creating a new actor in the network. This operation is often used to distribute the workloads using a divide and conquer technique. For faster processing and improving the response time of the IoT applications, an actor divides the incoming tasks into a set of sub-tasks and distributes them concurrently among the newly created actors using *spawn* operation. Each actor monitors the remaining actors in the network during processing or analyzing an IoT application for detecting and propagating errors in the distributed environment. Stronger coupling between the actors in a network can be expressed by the bi-directional links for better message transmission and prohibits invalid states during run-time. An actor embodies the following three properties, including:

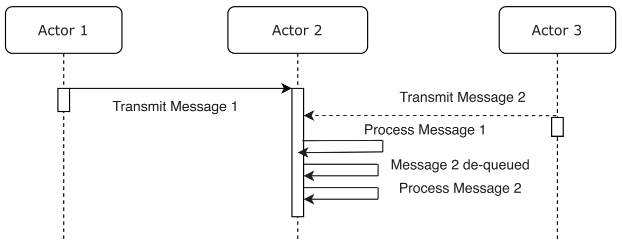

Fig. 2. The message passing and processing flow of multiple actors.

1. **Information processing:** Processing the applications partially/fully for its behavior.
2. **Storage:** Store the current state of an actor.
3. **Data Communication:** Transmit the message to the remaining actors in the network through the message passing technique.

In the current decades, several actor-based languages have been designed [32]. Armstrong has developed the Erlang programming language to build a distributed system without considering the downtime [33]. The first de-facto implementation of an Actor Model is provided by the Erlang language. Scala is another programming language that includes the actors for the standard distribution with the Akka framework.[2] Scala language runs on Java Virtual Machine and offers functional programming as well as to object-oriented programming (OOP). Over the last decades, with the advent of cloud computing and Fog computing, the Actor programming model has gained more momentum. Scalability and fault tolerance are two important traits for processing or analyzing the IoT applications on the distributed Fog environment.

3.2. Actor model of computation

An actor has state and behavior, much like an object in the OOP Paradigm. However, in the Actor Model, there are some restrictions that bring guarantees at the time of carrying out computations [13]. The state is owned completely by the actor and cannot be shared or accessed by other actors in the system. This means that there is no necessity for locks or other types of synchronization mechanisms in a multi-threaded environment. An actor can change its state in response to a message or can perform some computation depending on the message. The computation capabilities of the actor constitute its behavior.

Message passing is one of the key concepts of the Actor Model. The message passing processing flow of multiple actors is shown in Fig. 2. An actor can react to the sensed messages sequentially and processes each message independently, *i.e.* one at a time. However, while each actor processes the incoming messages sequentially, the other actors can work concurrently with each other, which helps to process multiple messages simultaneously in the network. It is the only allowed mechanism to communicate between actors in a network. The steps of processing a message by an actor are as follows.

- The actor adds the current incoming message at the end of the queue.
- If the actor is busy and was not scheduled for processing, then the actor marks the message as ready to process.
- A hidden scheduler takes the immediate ready message from the queue and starts processing.
- The actor modifies the current state information and transmits the message to another actor.
- Finally, the actor unschedules the message from the queue.

To accomplish the above-mentioned procedure, the following properties need to be available in an actor.

- **A mailbox** stores the incoming messages in the queue as first-come-first-serve basis.
- **A behavior** contains the internal variables, state of an actor, etc.
- **Incoming messages** contain pieces of data for representing single or multiple methods with their parameters.
- **A processing environment** takes the actors which have some messages to react and invokes their message handling codes.
- **An address** identifies an actor for assigning the incoming messages.

In concrete, an actor can do one of the following things in response to a message from another actor.

- Send messages to other actors for load distribution and further processing in the Fog network.
- Create new actors as per the availability and the requirement of the network.
- Designate the behavior of the actor, which is used to receive the next message for further processing.

Since there is always at most one message being processed per actor, the invariant of an actor can be kept without synchronization. This happens automatically without using locks. One of the main achievements of this model is the decoupling of the actor from the process of sending messages, which can be done asynchronously. An actor can only communicate with other connected actors in a network. Connections between the actors can be done through a direct physical attachment or memory/disk access or network address or E-mail address.

Depending on the type of connections, the addresses of the actors will vary. It could be a MAC address in the case of a physical connection or a simple memory address. Messages are delivered on the best efforts basis. Once an actor has sent a message, it is the responsibility of the receiver to handle it. This is the key element that allows decoupling a message from the sender actor. This type of communication is also referred to as *"fire and forget"*.

The Actor Model is thus an abstract concept based on some axioms, that defines the behavior and the structure of the model. There are several properties and mechanisms working behind scenes. Implementations of the model should obey those rules and may use other concepts on the top of it to expose the behavior of the model in a practical way.

3.3. Overview of Akka toolkit

Akka is a toolkit that provides a set of modules and libraries for creating distributed applications using the Actor Model as the programming model.[3] Akka provides support for Scala and Java languages with an API for each one of these programming languages. Nevertheless, it is also possible to find other implementations of the Actor Model for other programming languages, such as Akka.NET with the .NET platform, using C# and F#. The main features of the Akka toolkit are listed below.

- Akka provides a high-level and simple abstraction for parallelism and concurrent processing in a system.

[2] https://akka.io/

[3] https://doc.akka.io/docs/akka/current/typed/cluster-sharding.html

- This toolkit can provide a high-performance programming environment using non-blocking and asynchronous events.
- This can allow us to build a hierarchical actor's based network, which is suitable for fault-tolerant applications.
- This toolkit helps to deploy an actor-based networking model with different JAVA virtual machines, which can easily interconnect.
- Akka framework is more suitable for a complex distributed environment such as Fog environment.

Akka documentation is extensive and provides a complete overview of the different concepts and tools it offers. Most of the implementations of the toolkit have their origin in the research on the field and industry experience. The company behind the development of Akka, Lightbend,[4] is a commercial company, offering enterprise software solutions for the distributed systems and the cloud environment. This company is also behind the development of the multiple frameworks and platforms such as the Play framework and the Lagom framework,[5] which use Akka as an underlying technology.

In order to start building an application with Akka, it is important to have a clear understanding of the basic concepts of the toolkit. In this case, the core concept is the concept of an Actor and how it is implemented by Akka. On top of this concept, other more elaborated concepts are built such as remoting, clustering and sharding, which Akka makes available through different modules with their respective names. These are some of the main modules, which are used to build the proposed Fog based IoT application. The list of modules and libraries in Akka is quite extensive. The use cases of each of these depend on the specific characteristics of the application to be developed. Most of the modules can work independently of others, and there is no need to include all of them to have a functional application.

4. Case study: WSN-aware IoT scenario

In this section, we describe a case study related to Wireless Sensor Network (WSN)-aware IoT scenario for testing the feasibility of the proposed architecture of the Actor Programming Model for Fog Computing (as described in Section 5) [34]. The reference architecture of the WSN-aware IoT use case is shown in Fig. 3. The architecture fits with the principle of data science such as storage, description, analysis, maintenance, discovery, etc. Thus, the main purpose of this architecture is to visualize the measurement with various sensed data and produces different IoT scenarios for optimizing data acquisition and further analysis.

4.1. Basic components

The WSN-aware IoT scenario is composed of various WSNs with a set of ordinary *wireless sensor nodes* that can transmit the data to the cloud-centric *Data Analytics for Sensor Dashboard (DAS-Dashboard)* for analysis through local *gateway* device. Further, the DAS-Dashboard is interconnected with various external trusted entities of different domains in the network for transmitting the analytical results or received the state information. The basic components of the WSN-aware IoT scenario are discussed as follows.

1. **Cluster of wireless sensor nodes:** The cluster contains a group of wireless sensor nodes that perform default sensing tasks and transmit the tasks to the outside world via a gateway device in the network.

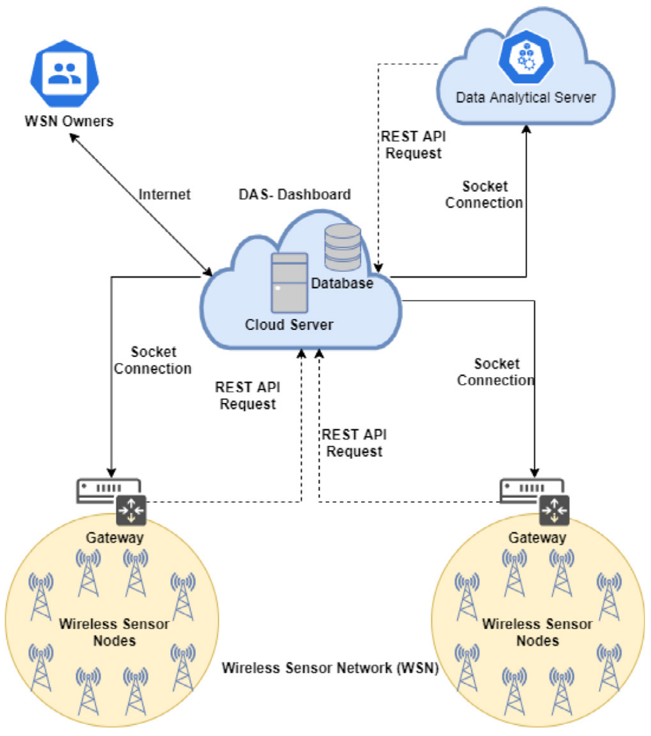

Fig. 3. Architecture of the WSN-aware IoT scenario [34].

2. **Gateways:** A gateway works as a mediator between the wireless sensor nodes and the centralized cloud servers with a point-to-point connection over the local network or Internet. Each gateway forwards the collected data from the sensors to the outside and also receives the instructions and updates the targeted sensor nodes.
3. **Data Analytic Server (DAS):** This is an important cloud component, which is used for real-time data processing/ analyzing. It acquires the data from the wireless sensor nodes through the DAS-Dashboard and communicates with other services on the network for gathering more data for further processing via the Internet.
4. **DAS-Dashboard:** This is the main component of the WSN architecture, which consists of a cloud service in charge of collecting, storing, and publishing data, transmitted by the wireless sensor nodes. The main two responsibilities of the DAS-Dashboard are to store the collected sensed data in a database for data visualization to the WSN owners and perform outsource data processing such as filter and analyzing the sensed data (mostly for predicting the future measurements) while communicating with the other entities of different domains of the network.

4.2. Working principle of WSN components

The main goal of the WSN-aware IoT scenario is to exploit the trusted links between the wireless sensor nodes and the DAS-Dashboard through gateways while analyzing the sensed data and adjusting the wireless sensor nodes based on the analytical outcomes. The data transmission between the external entities and the DAS-Dashboard follows two standards: (1) *External Data*- the data coming from the external entities such as gateways or DAS is received via Representational State Transfer (REST) API; (2) *Outbound Data*- the DAS-Dashboard transmits the data to the outside entities via a socket connection. The DAS-Dashboard is deployed on a well-dimensioned cloud server with a reliable

[4] https://www.lightbend.com/white-papers-and-reports
[5] https://www.lightbend.com/lagom-framework-part-of-lightbend-platform

Internet connection to the external entities without considering the energy and performance constraints. The external WSN users can visualize the data such as wireless sensor node location, measured analytical results (such as weather forecasting), the performance of external servers, failure report of the servers, etc. through a Web User Interface. The sequence diagram of the flow of communication between the DAS-Dashboard and external entities in the network is shown in Fig. 4.

In terms of data analysis, the WSN-aware IoT scenario consists of the computing forecast models using the ARIMA (Auto Regressive Integrated Moving Average) method [35], which is a commonly used technique for weather forecasting with the sensed data, received from the wireless sensor nodes. Later, all the models are computed and compared with different indicators including the AIC (Akaike information criterion). Thus, the main challenge of this model is to produce a better forecast model in order to reduce the frequency of readings and transmission by the sensors. Further, this can be applied in real-time IoT applications, which can have a significant impact on the energy consumption of the sensors nodes and prolong their lifetime [36].

The main limitation of this WSN-aware IoT scenario is that it relies on centralized cloud services for real-time data processing. The DAS and the DAS-Dashboard components are defined as the cloud services that are located in distant locations, which most probably would increase the round-trip time of data processing, which is one of the crucial aspects for the real-time applications. In addition, this architecture does not consider the computation power available on the proximity of the data, namely the Fog network, that can process the data on the edge of the network without transmitting the data to the centralized cloud.

5. Proposed architecture of Akka framework for Fog computing

By motivating from the existing WSN-aware IoT scenario and to overcome the limitation of this model, we have designed an Actor Programming Model for Fog computing with the Akka framework. The proposed architecture for the Fog transformation of the WSN application[6] is illustrated in Fig. 5. The proposed model is designed for processing the sensed data, received from the wireless sensor nodes with the Actor Model for concurrent computation with Akka toolkit for the reference implementation of the proposed model. Further, a Docker containerization approach is used for dynamically distributing the sensed data on the multiple computing devices (also known as worker nodes) through gateways (also known as master nodes) in the Fog network.

5.1. Basic components of proposed architecture

The proposed Akka framework based on the Actor Model for Fog Computing is composed of a set of *wireless sensor nodes* and various computing nodes as a form of clusters in terms of *IoT manager*, *master nodes* and *worker nodes*. The IoT manager receives the sensed data from the nearby wireless sensor nodes of the IoT cluster, which are transmitted to the suitable worker node on the worker cluster for further processing/analysis by taking a decision at the master node in the master cluster of the Fog network. The basic components of the proposed Akka framework based on the Actor Model for Fog Computing are discussed as follows.

[6] https://github.com/marcelo-s/akka-framework-test

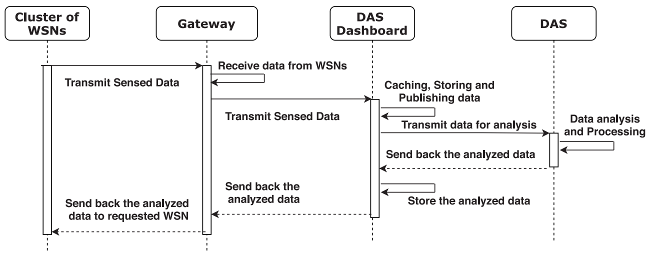
Cluster of WSNs
Gateway
DAS Dashboard
DAS
Transmit Sensed Data
Receive data from WSNs
Caching, Storing and Publishing data
Transmit Sensed Data
Transmit data for analysis
Data analysis and Processing
Send back the analyzed data
Send back the analyzed data
Store the analyzed data
Send back the analyzed data to requested WSN

Fig. 4. Sequence diagram of WSN with DAS.

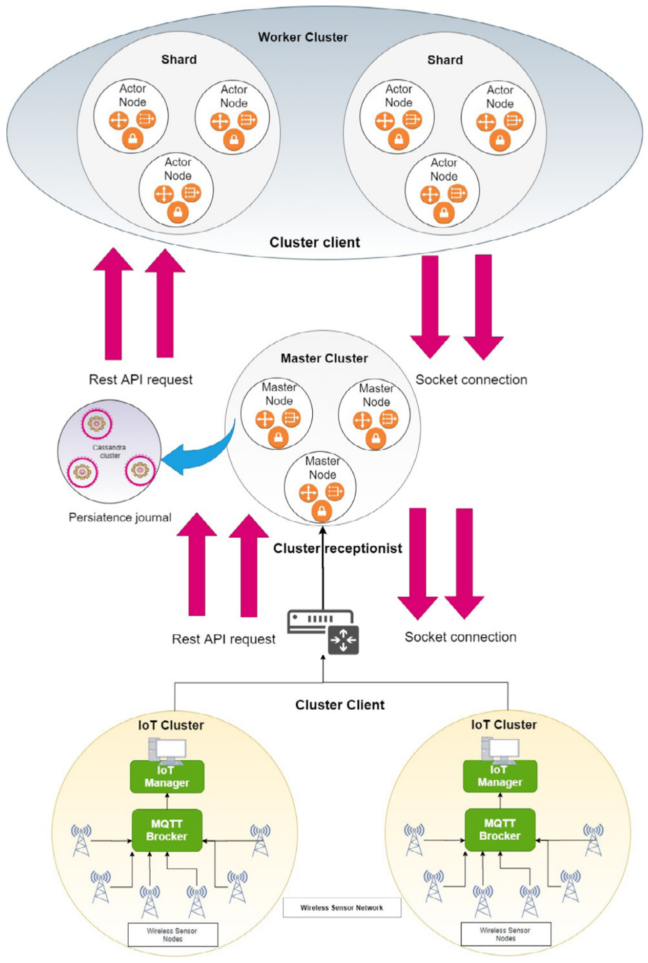

Fig. 5. Proposed Architecture of Akka framework based on the Actor Model for Fog Computing.

1. **IoT Cluster:** The IoT cluster consists of a set of wireless sensor devices that can generate the tasks/data for further processing or analysis. This cluster has a manager or Cluster Head (CH), in charge of receiving all the data from the sensor nodes and passes the sensed data to the master cluster. The IoT cluster manager can be perceived as the gateway device in the Fog network, as per Fig. 5.
2. **Master Cluster:** The master cluster acts as the main controller of the proposed Akka framework based on the Actor Model for Fog Computing. The main functions of the master cluster are to receive data from the IoT cluster, send data to the worker cluster for processing, and send back the processed results to the IoT cluster. It also maintains its state in a distributed journal.

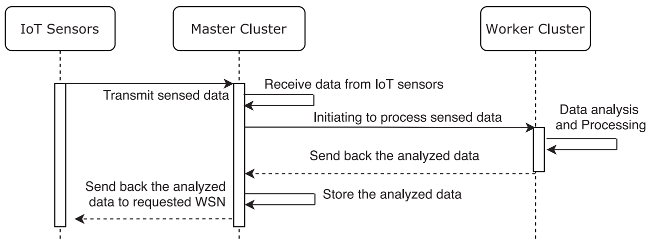
IoT Sensors
Master Cluster
Worker Cluster
Transmit sensed data
Receive data from IoT sensors
Initiating to process sensed data
Data analysis and Processing
Send back the analyzed data
Send back the analyzed data to requested WSN
Store the analyzed data

Fig. 6. Sequence diagram of communication links between the components of the Proposed Architecture.

3. **Worker Cluster:** The worker cluster is mainly used for data processing/analyzing in a Fog network. This cluster distributes the workloads among all the active worker nodes in the worker cluster for minimizing the processing delay.

The sequence diagram of the proposed Akka framework based on the Actor Model for Fog Computing is shown in Fig. 6.

5.2. Working principle of the components of proposed model

The working principles of the components of the proposed Akka framework based on the Actor Model for Fog Computing architecture are discussed as follows.

IoT Cluster: The IoT cluster is modeled with the idea of grouping all the wireless sensor nodes of the network under a cluster. There are two types of devices on this cluster. The first one is wireless sensors, which can measure the different properties of the environment. Sensors are modeled as actors within this IoT cluster. The IoT manager is the second type of device of this cluster, which is represented as a more powerful device (such as a router) with the capability of receiving and sending the data from the external entities through the Internet. The IoT manager works as a CH within the cluster and acts as a gateway device between the computing nodes of the IoT cluster and the external entities.

The communication unit on this cluster is named as *sensor data*. The sensor data consists of a set of information from a specific sensor. The data includes information such as the sensor Id, and an array of values, which are read by the sensors. This sensed data is published using the MQTT (Message Queuing Telemetry Transport) protocol [37], which is a lightweight messaging protocol in the IoT cluster that works efficiently in the resource-constrained devices. This protocol uses a publish–subscribe model that allows for communication between the multiple devices in a network. In the cluster, a sensor publishes its respective sensor data to a specific topic, which the IoT manager is subscribed to, represented in Fig. 7. Once a message is received by the IoT manager, it is sent to the master cluster for taking a decision for further processing. In order to support the publish–subscribe model, an MQTT broker has to be present for handling all the reception and delivery of the messages. The open-source broker server, namely Mosquitto, is used by the MQTT protocol for the application. Note that, the IoT cluster is designed in such a way that is generic enough, and the gateway can support different types of applications by connecting to the IoT devices. In a simple Fog-based application, a specific type of sensor data is produced and the data can directly initiate processing at the master cluster. In such a scenario, the MQTT broker is not required at the gateway devices.

Master Cluster: The master cluster acts as a controller of the proposed Akka framework based on the Actor Model for Fog Computing. It is in charge of receiving all the sensor data, which are coming from the IoT manager and transmit the data

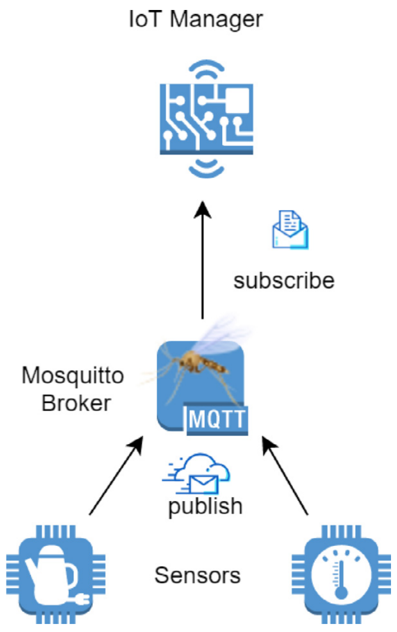

Fig. 7. IoT Cluster communication.

for further processing to the worker cluster. It also handles the passing of data on the other way, from the Worker cluster to the IoT cluster. Thus, this functionality is also placed on the gateway devices along with the IoT Cluster Manager based on the bottom-up approach.

A *cluster Singleton* is used by the master cluster, which is a mechanism provided by the Akka framework to enable one specific instance of an actor among all the nodes of the cluster. This is done in order to have only one source of truth with respect to handling the data. While this approach can be seen as a limitation, it actually serves the purpose of keeping the state of the application within a single actor instead of having a shared state between different actors, which is harder to manage and in most cases not advisable. Moreover, this cluster is used as a controller and does not perform any type of computation that can cause bottlenecks or other issues related to having only one instance of an actor.

Further, more nodes can be added to this cluster in order to increase the availability of the singleton actor. For instance, in a case, where the node with the singleton actor lives is taken down, the singleton actor will migrate to another available node in the cluster. With the persistence facility enabled using a persistence journal, the state of the singleton actor can be recovered by replaying all the stored events or using snapshots, which will bring back the new singleton actor state to its previous state before the failing of the original one. While the discussed functionality is extensive, most of the Fog-based applications need a single gateway, which initiates the execution on the Fog network. As a result, both IoT and master clusters can remain on a gateway device.

As previously mentioned, the master cluster has a persistent state. This state is persisted in a distributed journal so that it can keep track of all the sensor data that it handles. The journal stores the state of this cluster in a Cassandra database. The Cassandra database can be modeled as a multi-node data center in almost the same fashion as the Akka cluster. The resemblance of the configuration and the concepts behind the creation of a Cassandra cluster makes it ideal to use in the context of a distributed application, as it follows the same philosophy of working in a distributed environment. The Cassandra cluster can be at a much

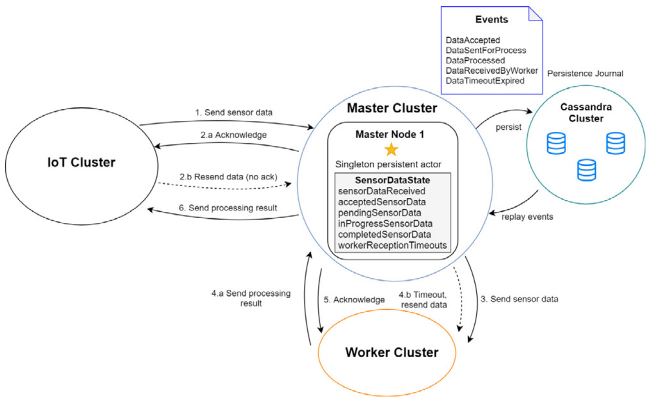

Fig. 8. Master cluster data handling.

powerful node in the Fog network or a third-party manages the state of the nodes in a public Fog setup such as Indie Fog [20].

The persisted state can include different types of data. For instance, specific data about the sensors and their readings can be persisted, so that later this information can be sent to the cloud for further analysis. In this sense, the master cluster acts as a gateway of the whole application for communicating with other applications or external entities. The simplified model of handling the sensor data in the master cluster is illustrated in Fig. 8.

The whole process starts when the master cluster receives sensor data from the IoT manager. The reception of the data is confirmed with an acknowledgment and the data acceptance is persisted on the state of the cluster. Further, the data is sent to an available/reachable worker node on the worker cluster. When a sensor data is sent to the worker cluster, the master cluster sets a timeout for the reception of the data by the worker cluster. This is done in order to guarantee the processing of the data. In this case, if no acknowledgment is received, then the data is sent once again until an acknowledgment is received from the Worker cluster. Once the processing of the data is completed by the Worker cluster, the result of the processed data is sent back to the master cluster, and finally, the master cluster forwards these results back to the IoT cluster.

Throughout the handling of the sensor data, the singleton actor of the master cluster keeps its state in a state object called SensorDataState, which is persisted in the distributed journal. Five types of events that are persisted in the Cassandra cluster are listed as follows.

- **DataAccepted:** When a sensor data has reached to the master cluster.
- **DataSentForProcess:** When the data has been sent to the worker cluster.
- **DataReceivedByWorker:** When the data has been received by the worker cluster.
- **DataProcessed:** When a result is received from the worker cluster.
- **DataTimeoutExpired:** When no acknowledgment of the reception of the data is received in timestamp from the worker cluster.

These events are persisted when the singleton actor receives a message corresponding to the mentioned actions, either from the IoT cluster or from the worker cluster. The timeout event is a special case of the singleton actor. For this event, there is no response from the worker cluster, i.e. no message needs to be received for the event to happen. To handle the timeouts of the data, a specific map is maintained based on the data identifier and the timeout for each data message. A special task is scheduled to go through this map over a certain period of time to make sure that the data has not overdue its timeout. In such a case, it will persist the event DataTimeoutExpired and will proceed to send the data again to a newly available worker.

In order to keep track of all the data, i.e. SensorDataState, it makes use of different collections: sensorDataReceived, pendingSensorData, inProgressSensorData, acceptedSensorData, completedSensorData and workerReceptionTimeouts, which are self-descriptive. The data state is updated whenever an event has happened. These events are persisted so that they can be replayed in the case of recovery of the master singleton in the master cluster.

A simple case of how the update process works are discussed as follows: when a particular sensor data has been accepted, it is registered in the sensorDataReceived, acceptedSensorData, and pendingSensorData collections. Further, when the work has been started, it is removed from pendingSensorData and added to inProgressSensorData collection. Finally, when the data processing is completed, it is removed from the inProgressSensorData and adds to the completedSensorData collection. The sensorDataReceived collection stores all the sensor data, received by the master cluster. This collection serves as historical data and can be used as a source of information for further analysis. The other collections are used as control mechanisms for data handling.

Cassandra cluster can be completely eliminated, where the gateway does not have to store the data more than just initiating the edge analytics in the Fog network. This is a lot more common approach to developing Fog based applications. In the reference architecture, presented in this paper, the Cassandra cluster was considered, since the main WSN-aware IoT scenario (presented in Section 4) has the DAS-Dashboard to provide the data when required, and we have discussed this as a means to provide similar storage facility in the Fog setup. Moreover, by placing the distributed journal in the Fog network, we can also ensure the privacy-related design decisions to stop moving the data beyond the gateways, i.e. up to a certain level of the Fog network.

Worker Cluster: The Worker cluster is the processing cluster in the Actor Programming Model for Fog Computing. It receives sensor data from the master cluster, processes it, and finally, returns the result back to the master cluster. Worker nodes in this cluster are registered to the master cluster so that the master cluster can proceed to send the data to the workers in the near future. The worker cluster is configured as a Cluster Client in the same way as the IoT cluster so that it can send messages to the external master cluster.

The Worker cluster is modeled using the concept of Cluster Sharding, which is provided by the Akka framework. The idea is to distribute the work processing evenly among all the nodes in the cluster. In order to accomplish this task, all the worker nodes have a worker region. This is the shard region to which all the sensor data must go through before arriving at the corresponding work entity actor that lives within a specific shard (actors within a shard are also called entities). These entity actors in the shard represent sensors of the system. This means that each worker entity in the shard is in charge of processing the data to the corresponding sensor from the IoT cluster. The series of steps of how the sensor data is handled by the Worker cluster is illustrated in Fig. 9. Some communication details including acknowledgment of workers and shard communications are omitted to maintain the simplicity of the diagram.

Firstly, a worker node is started and sends a message to the master cluster with an actor reference of the worker system that lives in the node. The Master cluster keeps a list of Worker nodes so that it can later send works to them. Secondly, once a working system is registered, the master cluster sends the sensor

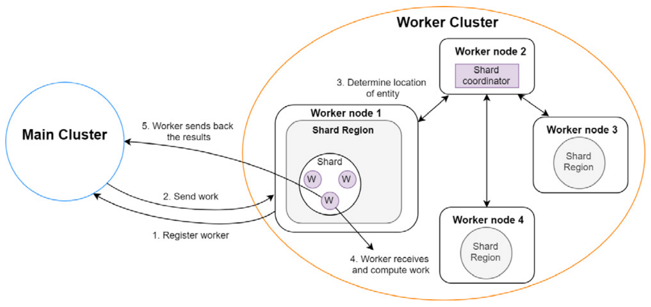

Fig. 9. Worker cluster data handling.

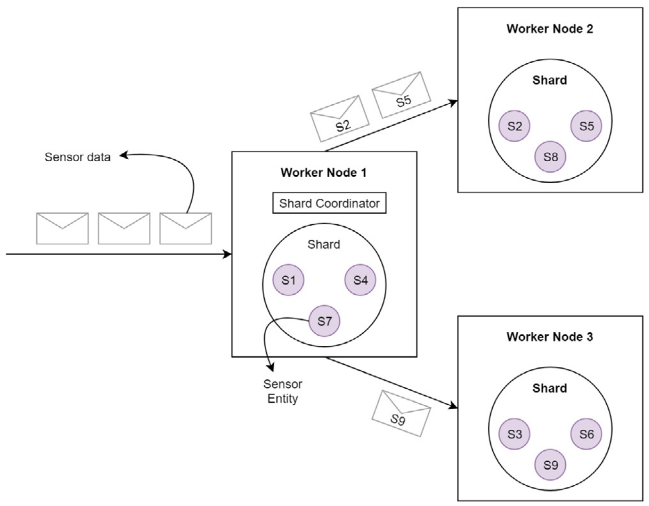

Fig. 10. Distribution of actors in the Worker cluster.

data to a registered worker system through its cluster singleton instance. The third step deals with the distribution of the work between the worker nodes. This aspect is handled by the Akka sharding mechanism with the help of a message extractor, which is described in more detail in Section 5.3. On the fourth step, a Worker entity receives the sensor data and performs the required computation with a WorkProcessor that uses the Routing mechanism to distribute the processing among all the nodes of the cluster. The final step is carried out once the processing is finished by the worker and the result is sent back to the master cluster. It is also important to mention that the mechanisms to handle the failures are also implemented using a supervision strategy on the child WorkProcessor, which is discussed further in Section 5.4.

A worker message extractor is defined in order to extract the messages as well as the entities and shard ids. The entityId corresponds to the sensorId of the data that is going to be processed. This means that an entity within the shard corresponds or represents a sensor from the IoT cluster. The shardId is determined based on the hash code of the entityId modulo of the total number of shards. This is done in order to properly distribute the work among the shards. The hash modulo operation is a safe bet in general cases and is recommended by the Akka. As per the message, it is just passed through the shard regions without further modification.

As mentioned earlier, the worker actor uses Akka's Routing mechanism to distribute the processing of the data. This is one of the key features of this processing cluster. The remote routing enables to perform computations in different nodes in parallel order. In order to do so, a Router actor is created in each worker node. The WorkProcessor is a self-contained router actor that contains the logic for processing the incoming task. This Router actor proceeds to create routes on every node of the cluster. These routes are the actual processing actors. The WorkProcessor is created as a child of the ShardRegion actor. As a parent, the ShardRegion actor allows handling the exceptions produced during the processing of a data through a supervision strategy. Once the processing of the data is completed, the result is sent back to the master cluster using the cluster client mechanism. The master cluster must send an acknowledgment about the received result. However, if the acknowledgment is not received in a given period of time, a new message with the result is sent to the master cluster until an acknowledges is received about the results within the mentioned timestamp.

5.3. Distributed computation for Fog application

As already discussed, one of the challenging tasks in the Fog network is to deal with the distribution of the incoming tasks on the local Fog devices based on the available computation power of those devices. The Actor Model helps to think in terms of partitioned applications, where entities of the domain can be conceived as actors within the application. In the IoT domain, these entities are most likely wireless sensors. The sensors can be modeled as a form of actors, which are distributed across the different nodes of a cluster. This idea is mainly used for distributing the incoming tasks on the worker cluster.

We can handle the distribution of the tasks in different contexts within the cluster using different mechanisms, provided by the Akka framework. Firstly, the Worker cluster uses the concept of *sharding* to distribute the actors across different nodes on the cluster as illustrated in Fig. 10. Here, the Actors in the worker cluster represent the sensors from the IoT cluster. Each one is in charge of processing the task coming from its respective sensor. Further, the Akka framework handles the distribution of the actors using the sharding mechanism and balances the loads in the cluster using a specific routing strategy without any further manual intervention.

Akka sharding distributes the sensor entities among all the nodes on the cluster, however, it is also possible to take the distribution process one step further. We can use all the computing power of the cluster by distributing the computation tasks among the nodes on the cluster. This is done using the cluster-aware routing mechanism of the Akka framework, as shown in Fig. 11. The idea in this context is to distribute all the different tasks that need to be carried out for sensor data, among the nodes of a cluster. In order to take advantage of this functionality, the work needs to be partitioned into smaller tasks that are processed in parallel order inside the cluster. This aspect entails the independence of the processing of these tasks. The final result of the processing is aggregated using a specific aggregator actor, which receives the results, coming from the different nodes of the cluster.

As discussed in the WSN-aware IoT scenario case study, reading of the sensor data is represented as a set of numeric values. Different forecast models must be constructed and compared against each other in order to fit the data better way. The computation process of a forecast model with the ARIMA technique requires a significant amount of computation power [34]. As a result, the work is divided into several sub-tasks and each sub-task is processed by one actor. The sub-task and the actor are responsible for one of the forecast models. The tasks are distributed on the cluster using the cluster-aware routing mechanism. The final results are compared to the aggregator actor. The distribution of processing tasks aims to make efficient use of the processing capabilities of the nodes within the cluster. Thus,

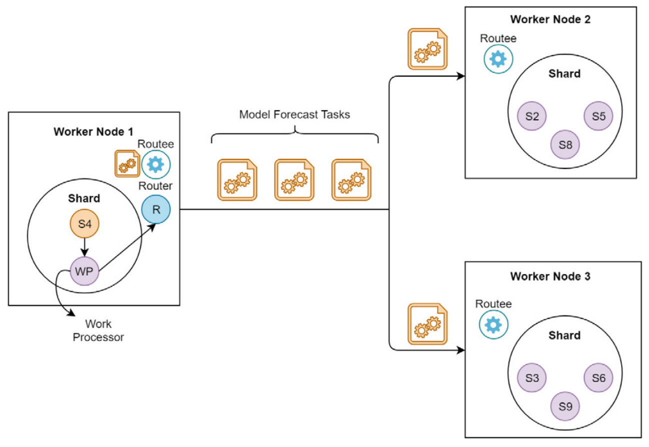

Fig. 11. Distribution of computation tasks of a sensor entity in the Worker cluster.

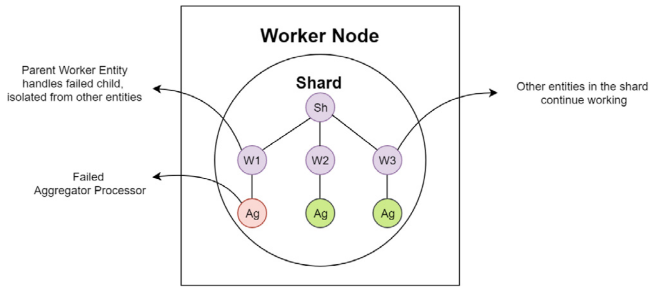
Worker Node
Shard
Sh
W1
W2
W3
Ag
Ag
Ag
Parent Worker Entity handles failed child, isolated from other entities
Failed Aggregator Processor
Other entities in the shard continue working

Fig. 12. Isolation of failures on a shard.

instead of having one node, multiple nodes in the cluster are used to execute the tasks for a specific sensor data in parallel order. This can minimize the processing time as the processed data is carried out at the same time by different nodes.

5.4. Fault-tolerance using Akka framework

One of the most common approaches for handling the fault-tolerance in the Fog network is to use the replicas of a service. This type of approach can also be used within an Akka cluster, in which more nodes can be added to maintain the availability of the system in case a failure occurs in one of the nodes. Notably, other types of methods can be used to deal with the fault-tolerance of the Fog network by considering the nature of the framework with the Actor Model. Concretely, the actor system hierarchy can be used to isolate the failures within an actor system, which is illustrated in Fig. 12. The main idea of this approach is to isolate the different areas or segments of the network. If there is a problem within a particular branch on the actor hierarchy, the problem can be handled by the parent actor of that branch while other branches can continue to work normally. This mechanism with the failures emphasizes designing the decisions within an application.

For example, on the worker cluster, workers are modeled as entity actors living in the shards. Each worker entity has a child actor, which is in charge of doing the aggregation. If there is a problem with the child actor, then its corresponding parent worker actor should resolve the issue using a specific supervision strategy. Other worker actors are not affected by this failing actor and can continue their normal processing. In the case of a major failing scenario, the node itself becomes unreachable, then the whole shard is migrated to another node in the cluster, providing high availability with respect to the working actors. In this case, the sharding mechanism is handling this migration process. This is the only way of achieving high availability. Besides that, the Akka framework offers other types of features and modules including cluster-aware routers, which is used for distributing the tasks among the worker cluster, which not only deals with the issues regarding the availability but also helps in terms of scalability.

It is also important to notice that the failure handling process is done without the intervention of any other entities, other than the Fog network itself. This is a desired feature on Fog infrastructure, where maintainability can be hard to accomplish. Applications using this framework, and more specifically, the actors/entities of an application need to be designed in a way that the actor hierarchy reflects the domain entities, such as bounded contexts. This could potentially help to better visualize and address particular problems in specific areas of the application. For instance, in the context of the Fog network, a Fog device can define specific actors for different services. As a result, if one of those actors has a problem, then only that service would become unavailable while the remaining services would still be available.

6. Empirical evaluation

A series of experiments have been conducted to evaluate the performance of the proposed Akka framework based on the Actor Model for Fog Computing in terms of processing time and load evaluation. To demonstrate the effectiveness of the proposed model, we enhance the existing cloud-centric WSN-aware IoT scenario to the Actor Model with Akka toolkit in the Fog network for processing/analyzing IoT applications with a small scale real-time setup. The main idea behind the experimental evaluations is to demonstrate the advantages of the Actor Model on the local Edge/Fog devices for executing the tasks with minimum processing time.

6.1. Simulation environment and setup

The IoT applications on the network edge require a distributed environment with a set of connected local Fog/Edge devices that can constantly send and receive data with minimum processing delay. These type of requirements are quite challenging for the OOPs-based Actor Models for processing the applications [22]. However, an actor-based Fog framework with multiple independent actors are distributed on the Fog network and can process or deliver *at most one message* at a time. The multiple actors within a network can process an application concurrently, at the same time without any synchronization or lock.

The overall deployment process of the proposed Akka framework based on the Actor Model for Fog Computing is relatively simplified using Docker in a Docker Swarm environment. The developed and properly configured Akka applications are defined in a *Dockerfile* that needs to be copied on a container in Raspberry Pi (considered as Fog device) for further execution. The *Dockerfile* allows to create images according to the hardware structure of the Raspberry Pi and is also useful for the dynamic deployment of multiple services on the Raspberry Pi by updating some parameters. Further, a *Dockerfile* can build the images and publish them in a Docker Hub. As a result, the images are available on the remote Fog devices in the network. Next, an *Application Stack* is deployed on top of the Docker Swarm for assigning the swarm nodes, which are defined in *docker-compose file*. Finally, each Fog node starts its corresponding Akka system to form the corresponding Akka clusters with proper functionality. The main benefit of Docker is to create and manage the Swarm environment based on several distinct actions including adding nodes, scaling specific services, restarting the services, etc. In addition,

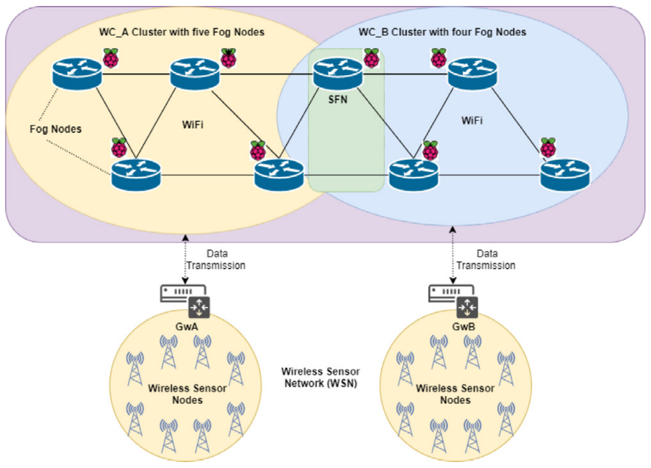

Fig. 13. Evaluation setup of Proposed Model.

the Docker handles to deploy and start each system individually using the Application Stack.

To evaluate the performance and applicability of the proposed Akka framework based on the Actor Model for Fog Computing, a Fog hierarchy is established with the real set of devices, as shown in Fig. 13. The proposed real-time setup consists of 10 standard *Raspberry pi 3* devices with CPU capacity 1.2 GHz (or 2451 MIPS) and 1GB LPDDR2 RAM. The devices are connected through the WiFi channel. Two of these devices act as the gateways (GwA and GwB), which help to transmit or receive the data to/from the Fog devices in the network. The IoT cluster and the Master cluster functionalities, discussed in Section 5, are deployed together on the gateway devices. Here, we consider two worker clusters with a set of five nodes and four nodes respectively. The first worker cluster with five nodes is connected with gateway GwA and the second worker cluster with 4 nodes is connected through gateway GwB. Each gateway device sends the sensed data to its own worker cluster and both the worker clusters share a worker node together, also known as Shared Fog Node (SFN). Anyone of the gateway can assign the tasks to the SFN for further processing as per its availability. The simulation parameters for the proposed Akka framework based on the Actor Model for Fog Computing are listed in Table 1. Considering usual Fog computing scenarios with Fog nodes [22] in the range of 5–10 in proximity, and the number of sensors connected per Fog node in the order of 10, which is a good representative of the general Fog based IoT application.

For the first experiment, a different number of sensors are simulated and produce data values at regular intervals. The sensed data are transmitted through the gateway device GwA, which pushes the data to the worker cluster for further processing. The sensor data values are produced using the normal distribution with mean 25 and standard deviation 3. The data values are generated uniformly within the time interval of 10 to 30 s. The processing of the sensor data is initiated once the gateway has 100 different values from all the sensor nodes to which it is connected or after every 60 secs, whichever was faster. This procedure of collecting the sensor data was continued for about 15 min. The number of ARIMA models to compute per sensor data is fixed at 6. Then, the distributed sensor data processing tasks are performed on the worker cluster (WC_A). During this experiment, the SFN is ignored and no tasks are allocated there. This can be adjusted through the Docker configuration for conducting the experiments. The configuration of the experiments can also be obtained along with the source code from the GitHub repository.[7]

[7] https://github.com/marcelo-s/akka-framework-test

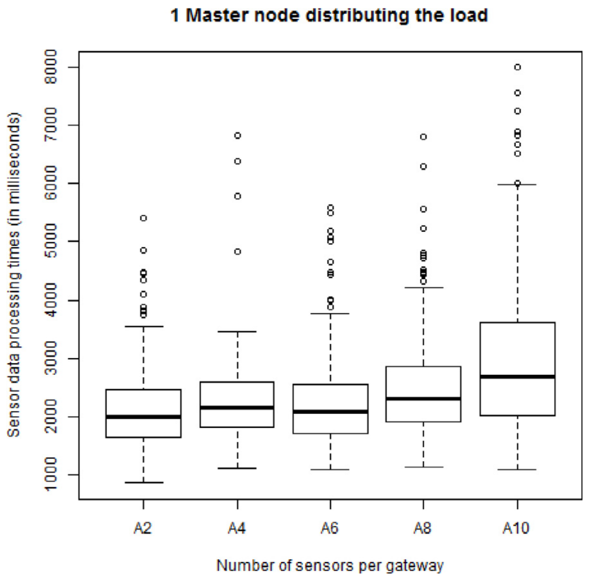

Fig. 14. Sensor data processing times on WC_A.

Table 1
Parameters used for simulation.

Parameters	Values
Number of simulated sensors	2–30
Distribution of generated sensor data:	Normal, mean 25, s.d. 3
Number of routees per node	10
Number of worker nodes:	4, 5
Number of gateway nodes:	2
Number of ARIMA models to compute per sensor data	6
CPU capacity of gateway/worker nodes:	1.2 GHz (or 2451 MIPS)
Memory capacity of gateway/worker nodes:	1 GB LPDDR2 RAM
Frequency of data generation	10–30 [data/sec]

Later, we simulate the sensor data from both the gateways GwA and GwB with multiple worker clusters.

6.2. Evaluation of Fog migration using case study

The evaluation of the proposed Akka framework based on the Actor Model for Fog Computing with the Akka toolkit based on two performance matrices, *i.e.* processing time and load evaluation is discussed in the following sub-sections.

6.2.1. Processing time

The processing time of a real-time data is the summation of the amount of time required to transmit the data from the source sensor to a local Fog device of a worker cluster through a gateway node and the total computation time on the selected Fog node (also known as worker node). Thus, the total time is the difference between the moment the sensor data is pushed to the gateway until the response is written back in the MQTT topic of the IoT manager. For the first experiment, all the real-time data from the sensor nodes are transmitted to the Fog cluster for further processing through a gateway node GwA. Fig. 14 represents the processing time of the sensor data on a Fog cluster, which is received by the Fog nodes concurrently. In this scenario, the gateway (which also works as a master cluster) does not have to store the data more than just initiating the edge analytics at the Fog nodes in the network. Thus, the processing times of the sensor data are mostly reflected by the current load of the Fog devices. From Fig. 14, it can be clearly observed that the processing times of the sensor data have increased as the number

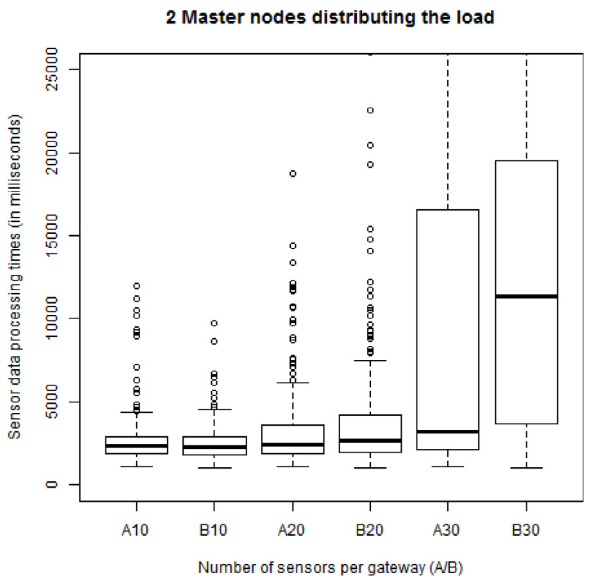

Fig. 15. Sensor data processing times on both the worker clusters WC_A and WC_B.

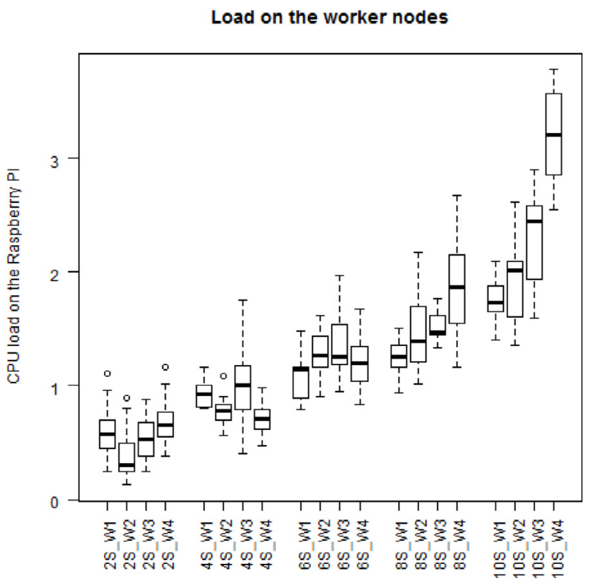

Fig. 16. Load on worker nodes in WC_A during the sensor data processing.

of requests increased (due to more number of sensors) on the gateway nodes.

In the second scenario, the sensor data are passed through multiple gateway nodes (*GwA* and *GwB*) for further processing on multiple worker clusters (WC_A and WC_B) with a different set of Fog nodes. Each gateway device sends the sensed data to its own worker cluster, *i.e. GwA* sends the data to the worker cluster WC_A, and *GwB* sends the data to the worker cluster WC_B. Each of the gateway nodes distributes the incoming sensor data on the Fog devices of its own worker cluster based on the current loads of the clusters and the resource availability. Further, the SFN is shared by both the worker clusters such as WC_A and WC_B for further processing and any one of the gateways (*i.e. GwA* and *GwB*) can assign the tasks to the SFN as per its availability and workload. Fig. 15 shows the efficiency of distributed data processing on the Fog hierarchy. We can observe that as the number of worker nodes increases, the sensor data processing can be performed much faster. This is also evident as the worker cluster WC_A with 5 Fog nodes took significantly lesser times when compared to cluster WC_B for processing the sensor data.

6.2.2. Load evaluation

In load evaluation, we observed that how efficiently the senor data are distributed among the Fog devices of a worker cluster or multiple worker clusters in a Fog network. In the first scenario, the gateway *GwA* distributes the incoming data from 10 sensor nodes to a worker cluster with 4 Fog devices. Fig. 16 shows the load on the respective Fog nodes of a worker cluster during processing the sensor data. The x-axis of the figure represents the experiment with n number of sensors, on one of the four worker nodes (nS_Wx). From Fig. 16, it is clearly observed that loads of the Fog devices have increased while increasing the incoming sensor data to the worker cluster through the gateway device. The performance analysis shows that most of the cases the incoming sensor data are nicely distributed on the Fog nodes of the worker cluster. We also observe some of the nodes being a bit overloaded when compared to the others. This is due to the variations in the simulation matrices (e.g. number of data points sent by each sensor, processing load of the considered ARIMA model, the frequency at which the processing is initiated on the Fog network, etc.).

In the second scenario, the multiple gateway devices including *GwA* and *GwB* distribute the incoming sensor data on the Fog nodes of multiple worker clusters efficiently. Fig. 17 shows the load distribution on the Fog devices of their own worker cluster. The figure contains the details of one of the sample nodes from WC_A and one from WC_B and the SFN. Since both the gateways pushed the data the tasks to the SFN for processing, thus its load is significantly higher. With 30 concurrent sensors on both the gateways, the load on the joint worker cluster is significantly high with a mean 13.97 and a median 15.10. This means that on a 4 core Raspberry PI, the CPU is 100% busy all the time, there were a significant number of processing tasks waiting in the queues. This resulted in the higher processing time, which is observed in Fig. 15. So, increasing the load further is not logical anymore.

This is the point where we need to consider other approaches for data processing on the Fog hierarchy. Firstly, process the sensor data on the hierarchical Fog-cloud environment instead of local Fog devices only, where the delay-sensitive applications are mostly processed on the resource-constrained Fog nodes and remaining tasks are offloaded to the centralized resource-rich cloud servers for further processing. The decisions of the Fog resource provisioning and cloud usage can be based on different QoS parameters such as delay, cost, energy usage [38], etc. Secondly, scale-up the worker cluster by adding more number of Fog nodes dynamically, such that one can process or analyze the data at the network edge efficiently with minimum delay.

From the analysis, we can observe the feasibility of the Actor Programming Model for the Fog Computing. The detailed results also validated the applicability of the Akka framework and dynamic deployment with Docker through the WSN case study. In addition, the proposed Akka framework based on the Actor Model for Fog computing shows the performance and parallelization efficiency on the resource-constrained gateway and Fog devices.

7. Conclusion and future works

In this paper, we have developed an Actor programming model with Akka toolkit in the Fog network for executing the Fog based distributed applications. The proposed Actor Programming Model for Fog Computing and the Akka toolkit with a comprehensive set of tools are useful for processing the IoT applications at the edge of the network using a set of self-managing Fog

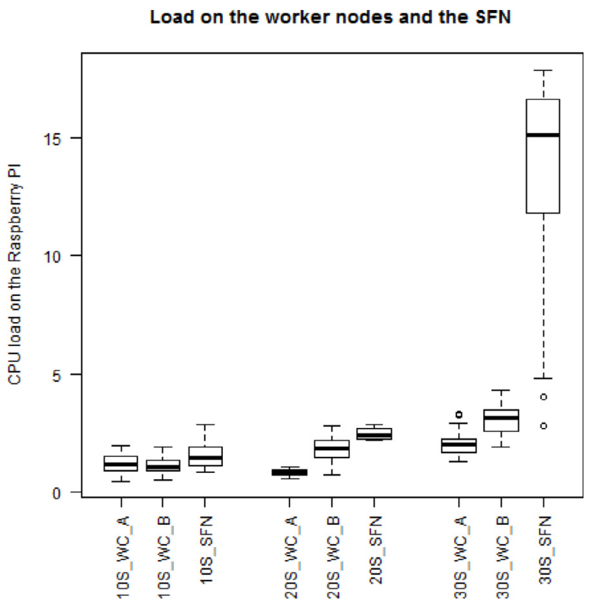

Fig. 17. Load on one worker node each in WC_A and WC_B during the sensor data processing.

nodes with minimum delay. Further, the Docker containerization approach is used to dynamically deploy the applications on the Fog network. To validate the proposed actor-based framework, a WSN case study is designed and implemented for demonstrating the feasibility of conceiving applications on the Fog networks. A detailed analysis demonstrates the performance and parallelization efficiency of the proposed approach on the resource constrained gateways and Fog devices such as Raspberry Pi.

As part of the future work, we would like to extend the current study by considering additional features of the Akka toolkit such as the Akka Streams. These can be used to efficiently transfer the data across the processing worker nodes. Besides that, we are also interested in the mobility aspects of Fog devices. For dealing with the mobility aspects of the Fog nodes and gateway devices, we are developing a simulated testbed for Fog computing based on an opportunistic network emulator (STEP-ONE [39]), where different mobility models can be applied and studied. Next, we will try to introduce the Actor programming model and the Akka toolkit as the execution frameworks of the STEP-ONE testbed along with the existing process execution based on Flowable Business Process Management Systems. Further, we also deal with the interoperable issue of the large numbers of heterogeneous IoT devices/sensors and processing components in the Fog framework using the help of the local gateway devices to make interactions with the non-interoperable IoT devices/sensors for granting interoperability [40].

CRediT authorship contribution statement

Satish Narayana Srirama: Conceptualization, Methodology, Funding acquisition, Supervision, Validation, Formal analysis, Investigation, Data curation, Resources, Writing - original draft, Writing - review & editing, Visualization. **Freddy Marcelo Surriabre Dick:** Software, Conceptualization, Methodology, Validation, Formal analysis, Investigation, Data curation, Writing - original draft. **Mainak Adhikari:** Validation, Formal analysis, Investigation, Data curation, Writing - review & editing, Visualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S.N. Srirama, Mobile web and cloud services enabling Internet of Things, CSI Trans. ICT 5 (1) (2017) 109–117.
[2] C. Mouradian, D. Naboulsi, S. Yangui, R.H. Glitho, M.J. Morrow, P.A. Polakos, A comprehensive survey on fog computing: State-of-the-art and research challenges, IEEE Commun. Surv. Tutor. 20 (1) (2017) 416–464.
[3] R. Roman, J. Lopez, M. Mambo, Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges, Future Gener. Comput. Syst. 78 (2018) 680–698.
[4] H. Flores, S.N. Srirama, Mobile cloud middleware, J. Syst. Softw. 92 (2014) 82–94.
[5] C. Chang, S.N. Srirama, R. Buyya, Internet of things (IoT) and new computing paradigms, in: R. Buyya, S.N. Srirama (Eds.), Fog and Edge Computing: Principles and Paradigms, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2019, pp. 3–23.
[6] M. Aazam, S. Zeadally, K.A. Harras, Offloading in fog computing for IoT: Review, enabling technologies, and research opportunities, Future Gener. Comput. Syst. 87 (2018) 278–289.
[7] M. Al-khafajiy, T. Baker, H. Al-Libawy, Z. Maamar, M. Aloqaily, Y. Jararweh, Improving fog computing performance via fog-2-fog collaboration, Future Gener. Comput. Syst. 100 (2019) 266–280.
[8] R. Mahmud, S.N. Srirama, K. Ramamohanarao, R. Buyya, Quality of Experience (QoE)-aware placement of applications in Fog computing environments, J. Parallel Distrib. Comput. 132 (2019) 190–203, http://dx.doi.org/10.1016/j.jpdc.2018.03.004.
[9] M. Adhikari, S.N. Srirama, T. Amgoth, Application offloading strategy for hierarchical fog environment through swarm optimization, IEEE Internet Things J. (2019).
[10] C. Wu, W. Li, L. Wang, A. Zomaya, Hybrid evolutionary scheduling for energy-efficient fog-enhanced internet of things, IEEE Trans. Cloud Comput. (2018) 1.
[11] M.G.R. Alam, M.M. Hassan, M.Z. Uddin, A. Almogren, G. Fortino, Autonomic computation offloading in mobile edge for IoT applications, Future Gener. Comput. Syst. 90 (2019) 149–157.
[12] A. Aske, X. Zhao, An actor-based framework for edge computing, in: Proceedings of the 10th International Conference on Utility and Cloud Computing, 2017, pp. 199–200.
[13] C. Hewitt, Actor model of computation: scalable robust information systems, 2010, arXiv preprint arXiv:1008.1459.
[14] D.D. Sanchez, R.S. Sherratt, P. Arias, F. Almenarez, A. Marin, Enabling actor model for crowd sensing and IoT, in: 2015 International Symposium on Consumer Electronics, ISCE, IEEE, 2015, pp. 1–2.
[15] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16, http://dx.doi.org/10.1145/2342509.2342513, URL: http://doi.acm.org/10.1145/2342509.2342513.
[16] J.S. Preden, K. Tammemäe, A. Jantsch, M. Leier, A. Riid, E. Calis, The benefits of self-awareness and attention in fog and mist computing, Computer 48 (7) (2015) 37–45.
[17] M. Satyanarayanan, P. Bahl, R. Cáceres, N. Davies, The case for VM-based cloudlets in mobile computing, IEEE Pervasive Comput. (4) (2009) 14–23.
[18] M. ETSI, Mobile Edge Computing (MEC); framework and reference architecture, 2016, ETSI, DGS MEC 3.
[19] J. Feng, Z. Liu, C. Wu, Y. Ji, AVE: Autonomous vehicular edge computing framework with ACO-based scheduling, IEEE Trans. Veh. Technol. 66 (12) (2017) 10660–10675.
[20] C. Chang, S.N. Srirama, R. Buyya, Indie fog: An efficient fog-computing infrastructure for the Internet of Things, Computer 50 (9) (2017) 92–98.
[21] C. Long, Y. Cao, T. Jiang, Q. Zhang, Edge computing framework for cooperative video processing in multimedia IoT systems, IEEE Trans. Multimed. 20 (5) (2017) 1126–1139.
[22] J. Fürst, M.F. Argerich, K. Chen, E. Kovacs, Towards adaptive actors for scalable iot applications at the edge, Open J. Internet Things 4 (1) (2018) 70–86.
[23] T. Elgamal, A. Sandur, P. Nguyen, K. Nahrstedt, G. Agha, Droplet: Distributed operator placement for iot applications spanning edge and cloud resources, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, IEEE, 2018, pp. 1–8.
[24] F. Aiello, G. Fortino, A. Guerrieri, R. Gravina, Maps: A mobile agent platform for wsns based on java sun spots, Proc. ATSN (2009).
[25] G. Fortino, W. Russo, C. Savaglio, W. Shen, M. Zhou, Agent-oriented cooperative smart objects: From IoT system design to implementation, IEEE Trans. Syst. Man Cybern.: Syst. 48 (11) (2017) 1939–1956.
[26] B. Spencer Jr., J.-W. Park, K. Mechitov, H. Jo, G. Agha, Next generation wireless smart sensors toward sustainable civil infrastructure, Procedia Eng. 171 (2017) 5–13.
[27] M. Lohstroh, E.A. Lee, An interface theory for the internet of things, in: SEFM 2015 Collocated Workshops, Springer, 2015, pp. 20–34.

[28] R. Hiesgen, Embedded actors–A better abstraction for distributed messaging in the IoT project, 2015.
[29] C. Savaglio, M. Ganzha, M. Paprzycki, C. Bădică, M. Ivanović, G. Fortino, Agent-based Internet of Things: State-of-the-art and research challenges, Future Gener. Comput. Syst. 102 (2020) 1038–1053.
[30] G.A. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, Technical Report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.
[31] C. Hewitt, P. Bishop, R. Steiger, A universal modular ACTOR formalism for artificial intelligence, in: Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973, pp. 235–245.
[32] G. Agha, Concurrent object-oriented programming, Commun. ACM 33 (9) (1990) 125–141.
[33] J. Armstrong, A history of Erlang, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, 2007, pp. 6-1–6-26.
[34] G.M. Dias, T. Adame, B. Bellalta, S. Oechsner, A self-managed architecture for sensor networks based on real time data analysis, in: 2016 Future Technologies Conference, FTC, IEEE, 2016, pp. 1297–1299.
[35] G.E. Box, G.M. Jenkins, G.C. Reinsel, G.M. Ljung, Time Series Analysis: Forecasting and Control, John Wiley & Sons, 2015.
[36] C. Liu, K. Wu, M. Tsao, Energy efficient information collection with the ARIMA model in wireless sensor networks, in: IEEE Global Telecommunications Conference, 2005, Vol. 5, GLOBECOM'05, IEEE, 2005, p. 5.
[37] U. Hunkeler, H.L. Truong, A. Stanford-Clark, MQTT-S–A publish/subscribe protocol for Wireless Sensor Networks, in: 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops, COMSWARE'08, IEEE, 2008, pp. 791–798.
[38] R.K. Naha, S. Garg, A. Chan, S.K. Battula, Deadline-based dynamic resource allocation and provisioning algorithms in Fog-Cloud environment, Future Gener. Comput. Syst. 104 (2020) 131–141.
[39] J. Mass, S.N. Srirama, C. Chang, STEP-ONE: Simulated testbed for edge-fog processes based on the opportunistic network environment simulator, J. Syst. Softw. (2020) 110587.
[40] G. Aloi, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, C. Savaglio, Enabling IoT interoperability through opportunistic smartphone-based mobile gateways, J. Netw. Comput. Appl. 81 (2017) 74–84.

Satish Narayana Srirama is an Associate Professor at School of Computer and Information Sciences, University of Hyderabad, India. He is also a Visiting Professor and the honorary head of the Mobile & Cloud Lab at the Institute of Computer Science, University of Tartu, Estonia, which he led as a Research Professor until June 2020. His current research focuses on cloud computing, mobile cloud, IoT, Fog computing, migrating scientific computing and large scale data analytics to the cloud. He received his Ph.D. in computer science from RWTH Aachen University in 2008. He is an IEEE Senior Member and an Editor of Wiley Software: Practice and Experience journal and works as a program committee member of several international conferences and workshops. Dr. Srirama has successfully managed several national, international and enterprise collaborative research grants and projects.

Freddy Marcelo Surriabre Dick is a Master Student at University of Tartu, Estonia. His area of research includes Internet of Things, Fog Computing and Cloud Computing.

Mainak Adhikari is currently working as a Post Doctorate Research Fellow at University of Tartu, Estonia. He has completed his Ph.D in Cloud Computing from IIT(ISM) Dhanbad, India in 2019. He has obtained his M.Tech. From Kalyani University in the year 2013. He earned his B.E. Degree from West Bengal University of Technology in the year of 2011. His area of research includes Internet of Things, Fog Computing, Cloud Computing, Serverless Computing and Evolutionary algorithm. He has Contributed numerous research Articles in various national and inter-national journal and Conference. He servers an Associate Editor of Cluster Computing Journal and Internet of Things Magazine.