J Grid Computing (2020) 18:57–80
https://doi.org/10.1007/s10723-019-09498-8

IoTEF: A Federated Edge-Cloud Architecture for Fault-Tolerant IoT Applications

Asad Javed · Jérémy Robert · Keijo Heljanko · Kary Främling

Received: 25 May 2019 / Accepted: 28 October 2019 / Published online: 10 January 2020

Abstract The evolution of Internet of Things (IoT) technology has led to an increased emphasis on edge computing for Cyber-Physical Systems (CPS), in which applications rely on processing data closer to the data sources, and sharing the results across heterogeneous clusters. This has simplified the data exchanges between IoT/CPS systems, the cloud, and the edge for managing low latency, minimal bandwidth, and fault-tolerant applications. Nonetheless, many of these applications administer data collection on the edge and offer data analytic and storage capabilities in the cloud. This raises the problem of separate software stacks between the edge and the cloud with no unified fault-tolerant management, hindering dynamic relocation of data processing. In such systems, the data must also be preserved from being corrupted or duplicated in the case of intermittent long-distance network connectivity issues, malicious harming of edge devices, or other hostile environments. Within this context, the contributions of this paper are threefold: (i) to propose a new Internet of Things Edge-Cloud Federation (IoTEF) architecture for multi-cluster IoT applications by adapting our earlier Cloud and Edge Fault-Tolerant IoT (CEFIoT) layered design. We address the fault tolerance issue by employing the Apache Kafka publish/subscribe platform as the unified data replication solution. We also deploy Kubernetes for fault-tolerant management, combined with the federated scheme, offering a single management interface and allowing automatic reconfiguration of the data processing pipeline, (ii) to formulate functional and non-functional requirements of our proposed solution by comparing several IoT architectures, and (iii) to implement a smart buildings use case of the ongoing Otaniemi3D project as proof-of-concept for assessing IoTEF capabilities. The experimental results conclude that the architecture minimizes latency, saves network bandwidth, and handles both hardware and network connectivity based failures.

A. Javed (✉) · K. Främling
Department of Computer Science, Aalto University, Konemiehentie 2, Espoo, Finland
e-mail: asad.javed@aalto.fi

K. Främling
e-mail: kary.framling@umu.se

J. Robert
University of Luxembourg - Interdisciplinary Centre For Security, Reliability and Trust, Luxembourg City, Luxembourg
e-mail: jeremy.robert@uni.lu

K. Heljanko
Department of Computer Science, University of Helsinki, Helsinki, Finland
e-mail: keijo.heljanko@helsinki.fi

K. Främling
Department of Computing Science, Umeå University, Umeå, Sweden

Keywords Internet of Things · Distributed systems · Edge · Cloud · Microservice · Containers · Smart buildings · Kubernetes · Kafka

1 Introduction

Ubiquitous technologies are increasingly prominent in the field of Internet of Things (IoT), in particular with the rise of edge computing. Such technologies have also flourished in industrial environments, leading to an increase of the data volume, variety, and velocity [1, 2]. As a consequence, Cyber-Physical Systems (CPS) need to extend their computing and storage capabilities by relying particularly on the cloud. However, data exchanges between IoT/CPS[1] applications and the cloud can suffer from weaknesses related to latency, bandwidth (and its cost), security, and reliability, for instance. To overcome such weaknesses, most data are processed at the edge, closer to the data sources. Edge computing, as a new computing concept, enables the envisaging of an intelligent computing infrastructure for the Internet (of Things), making physical objects smarter by connecting them with the virtual world [3, 4], while working seamlessly across heterogeneous clusters [5]. Such Edge-Cloud data processing architectures often consist of five core phases, as illustrated in Fig. 1. The *data acquisition* phase collects real-time data from IoT devices, sensors, actuators, and other information systems. The *data transmission* and *data processing* phases, also called the *data management* phase, enable either the processing of data locally on the device before sending them to the cloud, or directly transmitting them to the cloud for further computation. Afterwards, the end-users or information systems in general can consume them in the *data utilization* phase. This utilization can be in the form of a graphical user interface or various responses of user-typed commands to perform desired operations, such as controlling traffic signals or video surveillance cameras. Meanwhile, in the *data storage* phase, if needed, the application data are stored permanently for further analysis. In many IoT applications, it is often necessary to move data processing such as alarm generation from raw data streams to the edge, as it reduces network latency and saves a large amount of network bandwidth. In some other cases, the computing capabilities at the edge might be insufficient, requiring data processing to be moved towards the cloud back-end from the edge. This type of data processing placement has to be selected *at runtime* according to the available computing resources on the edge- and cloud-side. Therefore, the following challenges need to be tackled:

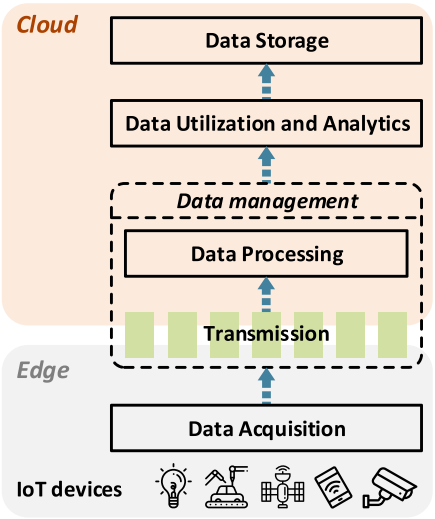


Fig. 1 An illustration of the data pipeline (five phases) in an Edge-Cloud architecture

1. **A common software stack** is required for processing, portability, and management ease, enabling flexibility of data processing placement across the clusters.
2. **(Local) fault-tolerant systems**, combined with data replication, need to be implemented to preserve the system state locally at the edge, especially in the case of a node failure or intermittent long-distance network connectivity problems. In fact, in a clustered system of many nodes in which data are transported between edge and cloud, it might be possible to permanently lose the data items due to the malfunction of edge nodes.
3. **Exactly-once data semantics** [6] should be considered to avoid data duplication or corruption. Indeed, since the processing stages acquire replicated data from any available node, the data may

[1] In this paper, IoT and CPS are used interchangeably

be repeatedly processed and delivered to another node. It causes the system to repeat unreliable actions, particularly for safety-critical environments, such as alarm generation or controlling a turbine. In other cases, the data may be unprocessed, leading the system to stall at any point.

4. **A federated management system** needs to be implemented for handling several cloud and edge clusters from a unified management interface irrespective of the physical hardware.

To address these challenges, this paper proposes the Internet of Things Edge-Cloud Federation (IoTEF), a highly dynamic and fault-tolerant architecture for IoT applications, by adapting our earlier Cloud and Edge Fault-Tolerant IoT (CEFIoT) design [7]. In contrast to the CEFIoT, the new IoTEF architecture offers federated management interface for orchestrating several heterogeneous clusters, supports transactional messaging for exactly-once data delivery, and enables fault-tolerant cluster computing on both the edge and the cloud. In addition, it adopts the same state-of-the-art cloud technologies as CEFIoT including Docker, Kubernetes, and Apache Kafka, and also deploys them for edge computing. The proposed architecture is composed of four layers: (i) Application Isolation, (ii) Data Transport, (iii) Distributed OS (Operating System), and (iv) Unified Federated Management layer. Based on this layered design, the architecture simplifies deployment and monitoring from a single management interface, and allows data processing placement on either the edge or the cloud without source code modifications. This is enabled by using the same lightweight container-based software stack on both the cloud and the edge. Furthermore, the edge also has capabilities to operate under hardware faults, such as malicious harming of edge devices or harsh environments. This requires fault tolerance using replication of data on several edge devices, reconfiguring the data processing pipeline when hardware or network failures occur, and capabilities to operate independently on the edge in a degraded manner when the edge is disconnected from the cloud back-end.

The Contributions of this Paper are Threefold: (**i**) We propose a novel Edge-Cloud architecture, IoTEF, based on the modular characteristics of microservices and lightweight virtualization. The proposed architecture supports federated management, allows computing on the edge, and simplifies fault tolerance of distributed deployments across the clusters. (**ii**) We compare several IoT architectures based on different characteristics as well as formulate the functional and non-functional requirements of our proposed architecture. (**iii**) The capabilities of IoTEF are assessed by applying them to the smart buildings use case of the ongoing Otaniemi3D project at Aalto University campus in Otaniemi [8, 9].

The rest of the article is structured as follows. Section 2 introduces the key-enabling cloud technologies combined with related work on the IoT architectures and Edge-Cloud distributed systems. Section 3 defines functional and non-functional requirements as well as proposes a federated Edge-Cloud architecture, IoTEF, for multi-cluster IoT applications. Section 4 explains the smart buildings use case along with the implementation and describes various data sensors. In Section 5, our proposed architecture capabilities are evaluated with results being presented in terms of fault tolerance, latency, and throughput. Finally, Section 6 concludes this paper with the possible future directions.

2 Background: Concepts and Related Work

There has been a substantial growth in the development of cloud technologies, communication mechanisms, and other intelligent systems. However, one fundamental principle emerges: the more benefits these systems provide for our wellbeing, the higher the potential for harm when they are unable to perform correctly. In such situations, fault tolerance is the best guarantee as it has the capability of overcoming physical, design, or human-machine interaction faults [10–12]. A large number of fault-tolerant techniques have been proposed in the literature that are mainly intended for distributed systems [13–15]. Jhawar et al. [16] present a fault tolerance approach for IoT applications, deployed in the cloud platform. This solution allows users to specify the desired level of fault tolerance through a dedicated service layer. Another fault-tolerant mechanism for intelligent IoT is implemented by Su et al. [17] which presents the

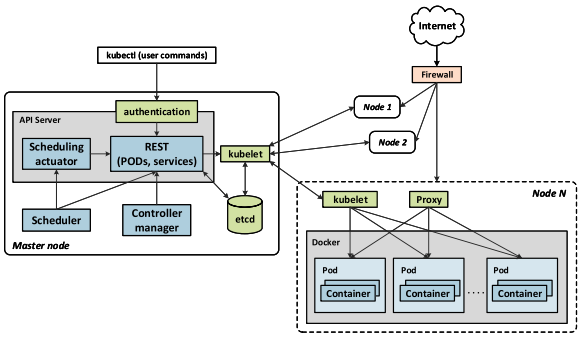


Fig. 2 High-level overview of Kubernetes cluster with one master and an *N* number of worker nodes

design of a fail recovery mechanism in WuKong middleware. Unlike our proposed fault tolerance mechanism for multi-cluster environments, the aforementioned approaches provide fault tolerance for a single cluster, and are capable of fail-over in a small network.

2.1 Theoretical Concepts

To accomplish a fault-tolerant system in the distributed IoT environment, container-based virtualization has become the apparent choice [18]. Through containers and the use of high-level languages and a common software stack, the same analytics program can be executed without source code modifications in both the edge and the cloud. Docker has been widely accepted as an open source container-based platform to create and execute distributed applications [19]. Furthermore, Docker could be by far the most reasonable contender for deploying microservices [20]. These are small, cohesive, and autonomous services that enable the concept of modular independence to structure an application as a collection of loosely coupled services, each running on its own domain [20]. To manage a bundle of containers, Google developed an open-source cluster management system called Kubernetes,[2] as the evolution of Borg [21]. With Kubernetes, it is possible to deploy high availability applications, scale and manage them during runtime, and use application-specific resources while execution [22]. The two major alternatives to Kubernetes are Docker Swarm [23] and Apache Mesos [24]. As depicted in Fig. 2, a basic Kubernetes cluster is composed of a single master and an *N* number of worker nodes. The Kubernetes master node consists of several core components: The *Kubelet* agent registers a node with the cluster, reports resource utilization, and monitors PODs. A POD is a minimal deployable unit capable of accommodating one or more containers, and has a replica set feature for enabling fault tolerance behavior; The *Proxy* service performs request forwarding across a set of back-ends. Both *Kubelet* and *Proxy* execute on each node including the worker nodes; The *API Server* services REST operations and provides interaction between all other components; The *Scheduler* assigns workloads to specific worker nodes, whereas the *controller manager* daemon watches the shared state of the cluster; A separate client called *kubectl* connects users with the cluster and assigns them control over the entire cluster; Last but not least, the replication-based configuration data storage is provided using *etcd*[3] service which notifies the system when events occur, such as data creation or deletion. In addition to the previous cloud technologies that create and manage containers, we introduce Apache Kafka[4] as a distributed, partitioned, and replicated publish/subscribe (pub/sub) data processing platform. It executes as a cluster on one or more servers spanning multiple datacenters. Figure 3 depicts a basic Kafka cluster in which a stream of messages is divided into categories called *topics*. These messages are published to *topics* using *producer* processes. The published messages are then stored in the set of servers called *brokers*. A separate set of *consumer* processes are then subscribed to one or more topics for pulling data from the brokers [25]. Additionally, Kafka uses Zookeeper service to provide coordination and synchronization within the cluster [26]. Other messaging systems, such as RabbitMQ and MQTT, could be considered alternatives to Kafka for data communication.

2.2 IoT Architectures and Edge-Cloud Distributed Frameworks

Several studies have been conducted to describe generic IoT architectures, which emphasize distributed Edge-Cloud computing and offer specifications including connectivity, scalability, and device

[2][Online]. Available: https://kubernetes.io/, accessed in May 2019

[3][Online]. Available: https://coreos.com/etcd, accessed in May 2019

[4][Online]. Available: https://kafka.apache.org/, accessed in May 2019

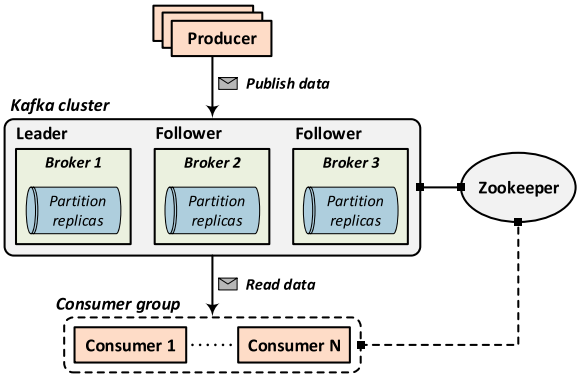


Fig. 3 Example of Kafka cluster with three brokers

management [27–32]. Krco et al. [33] provide an overview on designing IoT architecture in ETSI M2M, FI-WARE, IoT6, and IoT-A projects combined with the cloud computing capabilities. A Balena[5] platform has been developed which uses Docker-based management, instead of Kubernetes cluster orchestration, for IoT devices. The purpose of this platform is to introduce Docker containers for deploying and managing isolated applications. Although the platform is robust to sudden power failures and disk corruption, it has no functionality for managing several embedded devices as a single fault-tolerant cluster. There are other well-known platforms, such as Azure IoT Suite, Google Cloud IoT, and Amazon AWS, which deliver fully integrated cloud services and allow many systems to easily connect, manage, and ingest IoT data on a large scale. However, these platforms provide no edge fault tolerance.

In addition to the previous IoT architectures, Table 1 compares the IoTEF solution with CEFIoT and other frameworks based on various IoT characteristics. These characteristics are formalized in terms of "Yes", "No", and "Partially" availability criteria in which our proposed solution supports all the aforementioned characteristics. As seen in Table 1, Schmid et al. [5] propose an interoperable IoT architecture called BIG IoT, enabling multi-platform applications development and offering a cluster management solution. Similarly, another layered architecture called DIAT has been proposed by Sarkar et al. [37]. This approach tackles different aspects of IoT applications including interoperability, automation, scalability, and security. Unlike IoTEF architecture, these aforementioned approaches neither provide fault tolerance nor tackle computation placement on both the cloud and edge clusters. Munir et al. [36] propose a fog centric IFCIoT architecture that offers optimal performance, low latency, and high availability for IoT applications. Kelaidonis et al. [34] develop a federated IoT architecture to enable service provisioning in the distributed edge and cloud platforms, offering low latency, automation, and the unified management solution. However, unlike IoTEF architecture, the fault tolerance capabilities are understudied in these approaches. To offer edge analytics for large-scale IoT systems, Cheng et al. [38] propose GeeLytics, which can perform real-time data processing on both the edge and the cloud. This framework utilizes Docker containers, combined with pub/sub messaging, to achieve low latency analytics. Alam et al. [35] propose a microservices-based architecture by employing Docker virtualization and edge computing. This approach, combined with application management, offers fault tolerance and distributes application logic across cloud, fog, and edge devices. Similarly, Ramprasad et al. [41] describe an approach to stream IoT sensors data for smart buildings in which data analytics are distributed between the edge and the cloud. In this approach, the authors adopt Apache Kafka and Cassandra platforms for data streaming and storage respectively. As compared to our proposed architecture, the exactly-once data semantics and local fault tolerance are understudied in the aforementioned approaches. Chang et al. [39] propose a hybrid edge-cloud architecture by leveraging edge compute nodes to deliver low latency, bandwidth-efficient, and resilient end-user services. Although this solution provides fault tolerance and moves data processing closer to the users, it does not support multi-cluster application deployment. Another approach has been introduced by Elias et al. [40] for image processing, enabling automatic wildlife monitoring in remote locations. In this approach, the authors perform neural network training for animal recognition and implements fault-tolerant computation on both the edge and the cloud. As compared to the IoTEF, this solution does not offer exactly-once data delivery and is only applied to wildlife monitoring.

[5][Online]. Available: https://www.balena.io/, accessed in May 2019

Table 1 Comparison of IoTEF with other architectures

IoT Characteristics	IoTEF	CEFIoT	[34]	[35]	[36]	[37]	[5]	[38]	[39]	[40]
Data fault tolerance	Yes	Yes	No	Partially	No	No	No	No	Yes	Yes
Network fault tolerance	Yes	Yes	No	No	No	No	No	No	Yes	Yes
Node fault tolerance	Yes	Yes	No	Yes	No	No	No	No	No	Yes
High availability	Yes	Partially	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Unified cluster management	Yes	No	Yes	No	No	No	Yes	Partially	Yes	No
Automation	Partially	No	Yes	Yes	Yes	Partially	Yes	No	Yes	Yes
Layered design	Yes	Yes	No	Yes	Yes	Yes	No	No	No	No
Exactly-once data delivery	Yes	No	No	No	No	No	No	No	No	No
Container virtualization	Yes	Yes	No	Yes	No	No	No	Yes	Yes	No
Multi-cluster deployments	Yes	No	Yes	Yes	No	No	No	No	No	Yes

3 The IoTEF Architecture

The IoTEF architecture is designed to offer a unified management system for both the cloud and edge clusters in IoT systems. The proposed model, combined with the modular characteristics of microservices, simplifies both node and network fault tolerance, enables exactly-once data semantics, and overcomes fail-over in a large multi-cluster environment. The design also considers the limited resources available at the edge by employing lightweight containers instead of using traditional virtual machines. In addition, separate clusters for the edge and the cloud allow edge devices to operate independently when disconnected from the cloud back-end. Thus, the architecture performs in a degraded mode even when cloud connectivity is lost. The functional and non-functional requirements for the IoTEF architecture are listed in Table 2. These requirements are derived from the earlier related work in various domains [42–44]. This architecture is logically constructed from the underlying cloud technologies, which can also be deployed for edge computing, and has four layers of abstraction: (i) Application Isolation, (ii) Data Transport, (iii) Distributed OS, and (iv) Unified Management/Placement layer as illustrated in Fig. 4. These layers are integrated on top of each other, and together they enable the required capabilities of IoTEF. Furthermore, our proposed solution is sufficiently flexible to adapt to various hostile real-world applications including smart buildings, surveillance monitoring, and vehicle control system. The layered design of IoTEF ensures: (i) application-level software migration and portability between clusters, (ii) replication-based local data

Table 2 Functional and non-functional requirements for IoTEF

- Capability to **process data exactly once** and **replicate** them *N-times* in the cluster of an N nodes without any data duplication.
- Ability to **tolerate up to** $\frac{(N-1)}{2}$ permanent failures (node failure or disk corruption) in the cluster of an N nodes **without** interrupting the entire system, where N being an odd number greater than 1.
- Support for **local data persistence** on the edge when either a network fault occurs or a node malfunctions in the cluster of an N nodes.
- Possible to **continue data processing** from other active nodes without interrupting the entire processing pipeline.
- Ability to deploy, manage, and monitor data as well as computation placement through a **federated management system** irrespective of the underlying hardware.
- A system crash or network disconnection **should not** result in data loss or corruption.
- Able to **minimize latency peaks and additional delays** when processing stages are moved within and across the clusters.
- Capable of **scaling** the number of processing stages in order to handle more efficiently data computation.

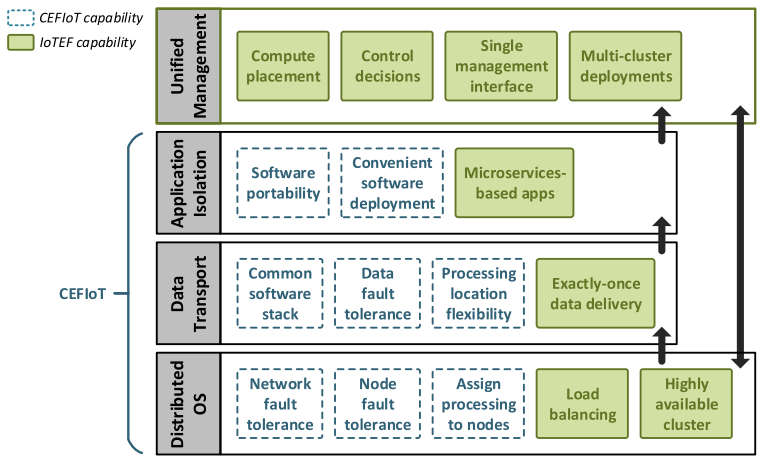


Fig. 4 A bottom-up layered architecture of IoTEF: Green (solid-line) modules correspond to the IoTEF design, whereas blue (dotted-line) modules are related to CEFIoT

buffering with exactly-once data delivery within the cluster, and (iii) fault-tolerant management for the multi-cluster data processing pipeline.

3.1 Application Isolation Layer

The Application Isolation layer of IoTEF wraps processes into separate containers and configures them to operate as a single isolated application. This wrapping is achieved by adopting Docker, which facilitates convenient software deployment and also allows the processing of data streams irrespective of the physical hardware. In the case of machines with different architecture types, containers execute the same application-level code (such as Java, Python, or Spark) without source code modifications. Docker is chosen for application isolation as it is by far the most adopted development tool, offering software portability, low CPU overhead, version control, and good network performance. It also isolates several IoT applications running on the same edge and cloud nodes from each other. Figure 5 demonstrates a running example of an IoT application on each IoTEF layer in which we have four Processing Containers/Stages (PC): *PC-1, PC-2, PC-3,* and *PC-4*. As can be seen in Fig. 5a, these processes are executed sequentially on a single machine, transporting data from source to destination.

3.2 Data Transport Layer

The Data Transport layer of IoTEF provides a pub/sub messaging framework in which streams of data are buffered and replicated across the cluster. This allows the architecture to have logical data flow in the form of containerized processes using pub/sub topics as a transport medium. In this way, data processing can be distributed efficiently, which gives us location flexibility for computation to be placed either on the edge or in the cloud. This organization of data processing pipeline maintains network fault tolerance by allowing data buffering locally at the edge, while Internet connectivity is being reconfigured. We select the Apache Kafka platform to enable the capabilities of this layer as it solves the problem of data stream fault tolerance by ensuring both online and offline messages consumption. It provides a unified high-performance data replication solution offering real-time processing, low latency, and high data rates. Figure 5b demonstrates an application viewpoint in which several dedicated Kafka topics are used to buffer data streams. Thus, allowing the data to be available at all times in the cluster even while some processing stages are being reconfigured. In addition, we use Kafka transactional API to ensure exactly-once data semantics between producers and consumers. This additional feature in Kafka allows the processing of data only once within a cluster. The current Kafka implementation does not support transactions between multiple clusters. Hence, in this paper, we limit exactly-once data delivery to a single cluster, separate for both the edge and the cloud.

This layer is further extended to two scenarios in Fig. 6. Both the edge- and cloud-side clusters consist of three Kafka topics along with four PCs. In Scenario-1, the edge-side cluster has two containers

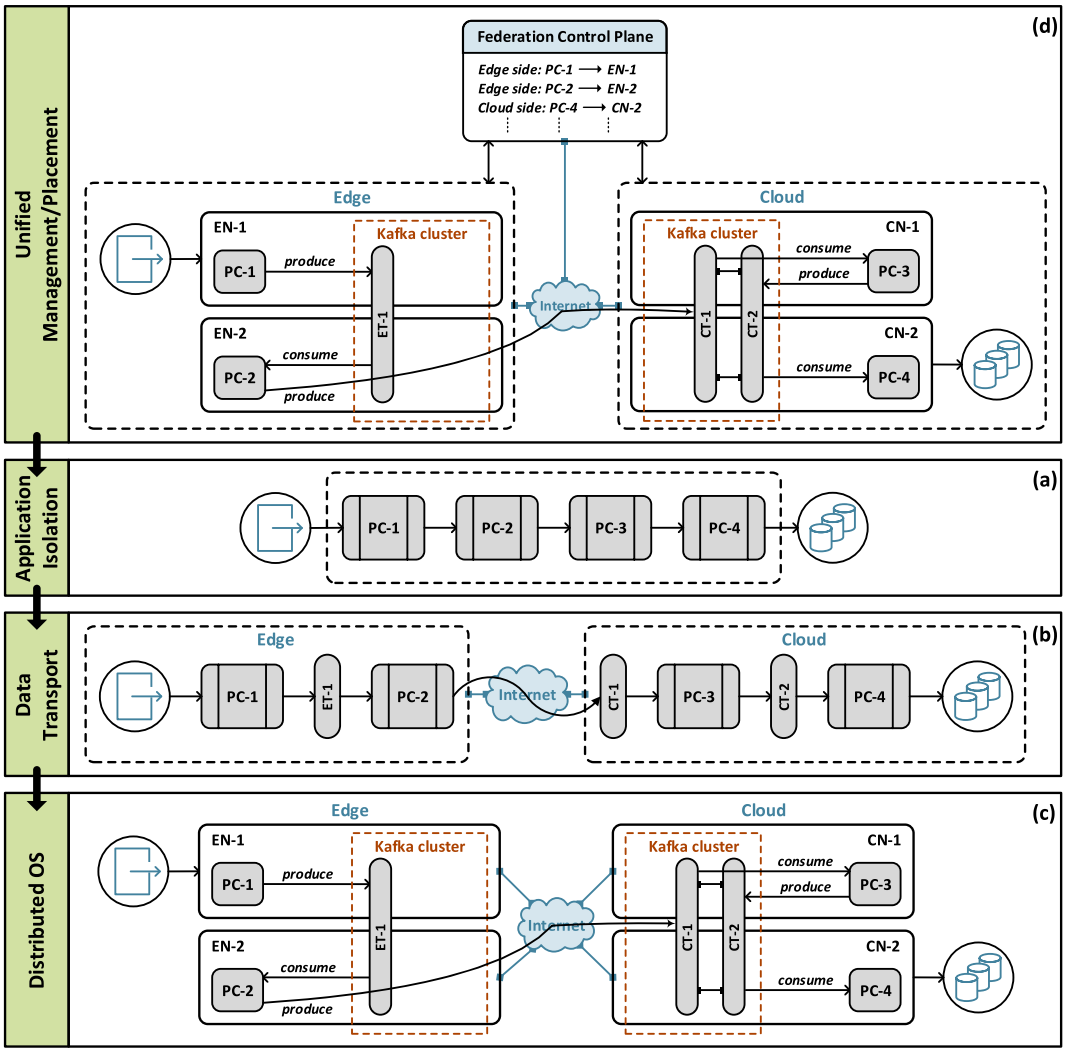


Fig. 5 Running example of an IoT application on the IoTEF layered architecture; where PC is the Processing Container, ET is the Edge Topic, CT is the Cloud Topic, EN is the Edge Node, and CN is the Cloud Node

in which PC-1 collects data from the data source, performs pre-processing (e.g., filtering or compression), and sends them to the local Edge Topic 1 (ET-1). PC-2 then consumes these data from ET-1, performs additional processing, and sends them to the cloud on Cloud Topic 1 (CT-1). Similarly, the cloud-side cluster contains PC-3 and PC-4 which consume data from CT-1 and CT-2, respectively, process them further, and deliver them to the destination. We enable a transactional feature of Kafka in these PCs to ensure exactly-once data consumption and delivery within the cluster. On the other hand, Scenario-2 depicts the behavior in which PC-3 is moved to the edge. This shows the location flexibility for data processing, since the design is based on software containers and both sides have a similar Kafka cluster. This configuration also provides benefits related to bandwidth consumption and latency. Additionally, if there is a network connectivity problem, the data will always be available on the edge-side cluster. Once the Internet outage has been resolved, for example, by using a secondary network connection, the architecture resumes processing on both the edge and the cloud. In both scenarios, the logical data flow is actively monitored through Kafka which renders the pipeline active at all times.

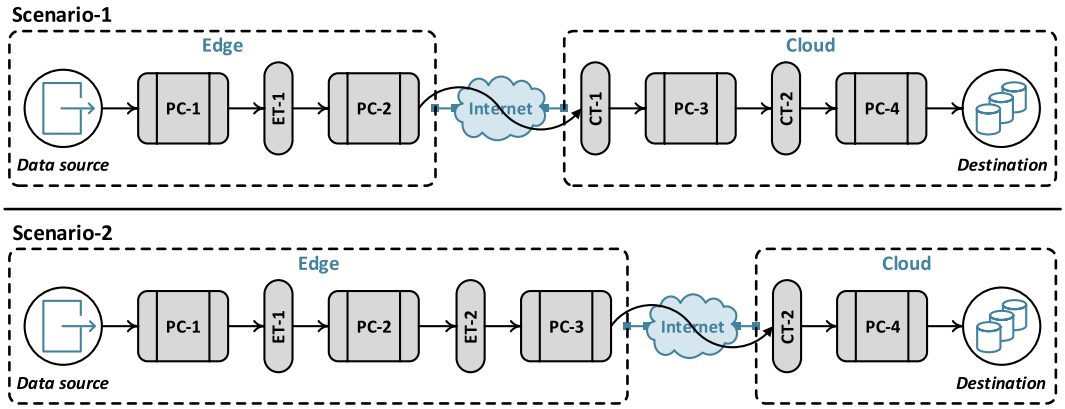


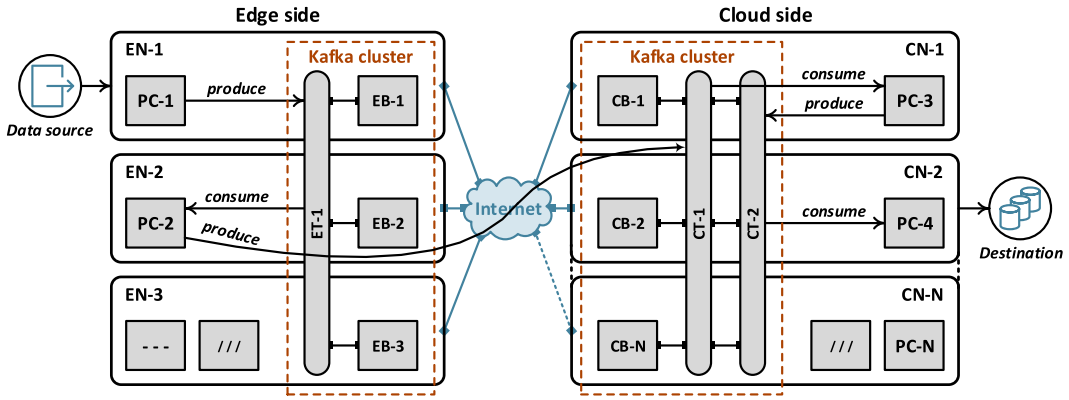
Fig. 6 Logical data processing pipeline

3.3 Distributed OS Layer

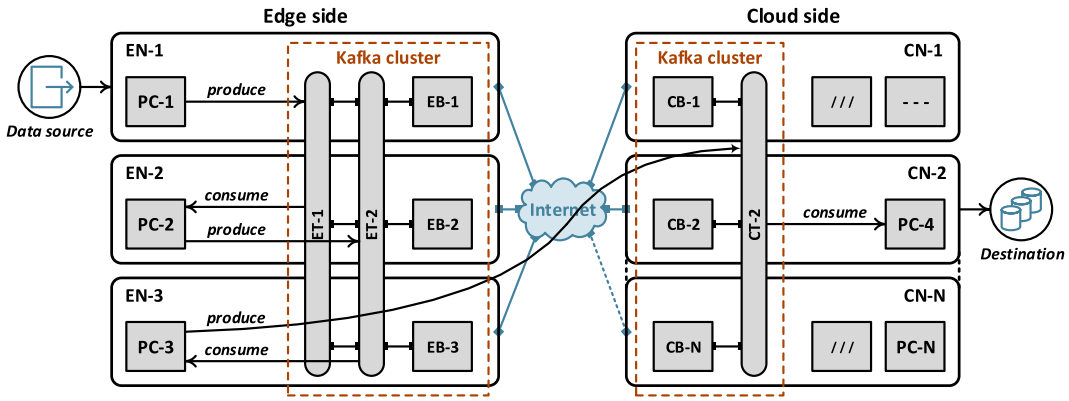
The Distributed OS layer of IoTEF is responsible for distributing processing stages in the cluster and assigning them to the physical nodes. This layer simplifies the problem of node failures by rescheduling failed processing stages on other available nodes, thus providing node and network fault tolerance on the hardware-level. In addition, it is capable of balancing the workload between different nodes. We use the Kubernetes framework to orchestrate the placement of processing stages, as it enables fault-tolerant management and provides a high availability solution. One of the key design ideas of Kubernetes is to have no single point of failure. All configuration data is stored in a replicated fashion, allowing Kubernetes to survive any single node failure in the cluster of three nodes, for instance. Figure 5c illustrates data processing placement for the same running application on the Edge Node (EN) and Cloud Node (CN), which is further extended to Fig. 7. As can be seen, the edge-side cluster contains three nodes along with the local Kafka cluster, whereas the cloud-side cluster has an N number of nodes with another Kafka cluster. This mapping provides a more detailed overview in which the edge has three Kafka brokers, thus providing a 3-way replication of data. Both sides have Kafka topics that are accessible to each physical node. In this way, if a node disconnects temporarily from either the edge- or the cloud-side cluster, it becomes inactive. Consequently, the system will not halt and the data can be consumed from any other available node. Figure 7a and b also demonstrate that (i) sensors and processing can be on different edge devices, as long as they have access to the same Kafka cluster, thus providing computing location independence, (ii) the data processing can be moved between the edge and cloud, and (iii) data are always available in the cluster, processed exactly-once, and buffered locally in the case of an Internet outage. Once the network connectivity has been resolved, the cluster continues to transmit data from local edge topic to the cloud. Figure 7b is further extended to Scenario-3 in Fig. 7c in which EN-3 becomes unresponsive on the edge-side cluster. As a consequence, Kubernetes reschedules PC-3 on EN-1. Since the data reside on the pub/sub topic, which is replicated on all the nodes, PC-3 consumes the data from ET-2 and sends them to CT-2 on the cloud-side cluster from EN-1. In this way, the architecture handles node failure by maintaining the correct system state in which the physical data flow is changed; nevertheless, the logical data flow remains the same. This shows a fault tolerance capability that is fully transparent to the application programmer, and also allows for dynamic relocation of data processing.

3.4 Unified Federated Management Layer

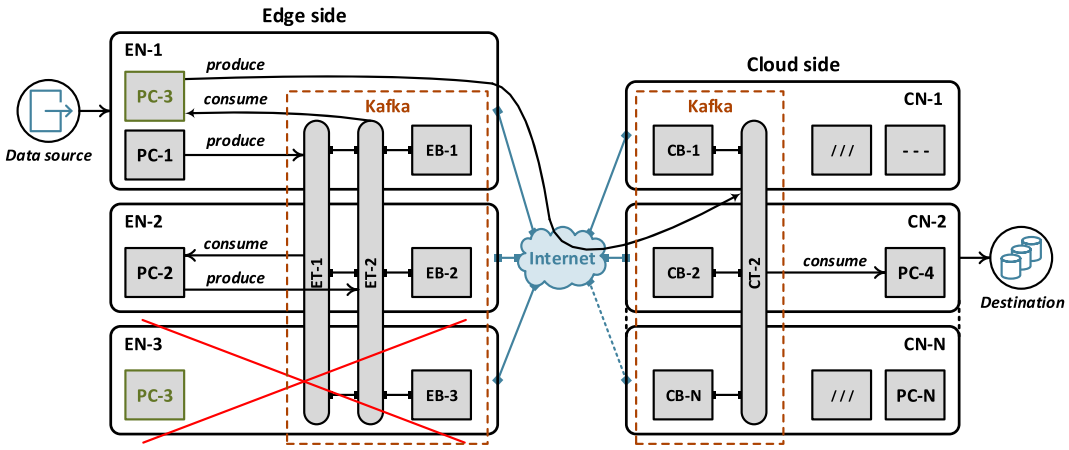
The Unified Federated Management layer of IoTEF provides a mechanism for handling several cloud and edge clusters from a single management interface. It also has an ability to monitor the data processing pipeline, ensuring that the same application deployment exists across multiple clusters. This layer has a separate control plane for creating and deploying containerized applications. In addition, the layer is capable of making control decisions for clusters and adding more clusters to the federation system. We



(a) Mapping of Scenario-1 to physical machines



(b) Mapping of Scenario-2 to physical machines



(c) Scenario-3 in which edge node EN-3 fails, PC-3 is resumed on EN-1

Fig. 7 Placement of processing stages; where EB is the Edge Broker, CB is the Cloud Broker, EN is the Edge Node, and CN is the Cloud Node

adopt the Kubernetes Federation scheme to implement the needed functionality. This federated management overcomes both hardware and network connectivity based failures by allowing on-the-fly automatic reconfiguration of the processing pipeline. As illustrated in Fig. 5d, the federation control plane manages two clusters: One on the edge and one on the cloud. Initially, this layer advises the Distributed OS layer to configure data communication stacks on the cloud and edge clusters through Kafka. It then packages processing stages of an IoT application into containers and deploys them in the multi-cluster environment. Figure 8 illustrates the six steps that are performed on our running application:

1. The federation layer employs Docker containers to configure a 3-node (three brokers) Kafka cluster. These containers with Kafka broker configuration in them are executed on each edge node as Kubernetes PODs (EB-1, EB-2, and EB-3), which are deployed through YAML [45] descriptor file.
2. Next, another Kafka cluster with three brokers is configured on the cloud by executing three Docker containers (CB-1, CB-2, and CB-3), which are also deployed as Kubernetes PODs through YAML configuration file.
3. The data processing of an IoT application is divided into PCs by placing the application code inside separate containers.
4. Once the data communication pipeline has been configured, the layer then places these PCs on the edge nodes, depending on the processing type (e.g., filtering or data compression), hardware architecture, and other resource requirements.
5. Similarly, in the case of excessive data processing (e.g., training of neural networks or prediction tasks) and data storage, the PCs can also be placed on the cloud nodes.
6. Finally, the layer manages and monitors the entire deployment through the Kubernetes high availability mechanism. In the scenario in which either the edge or cloud nodes disconnect from the system, PC will reschedule on another active node.

These steps are realized as a single application deployment inside multi-cluster environments through

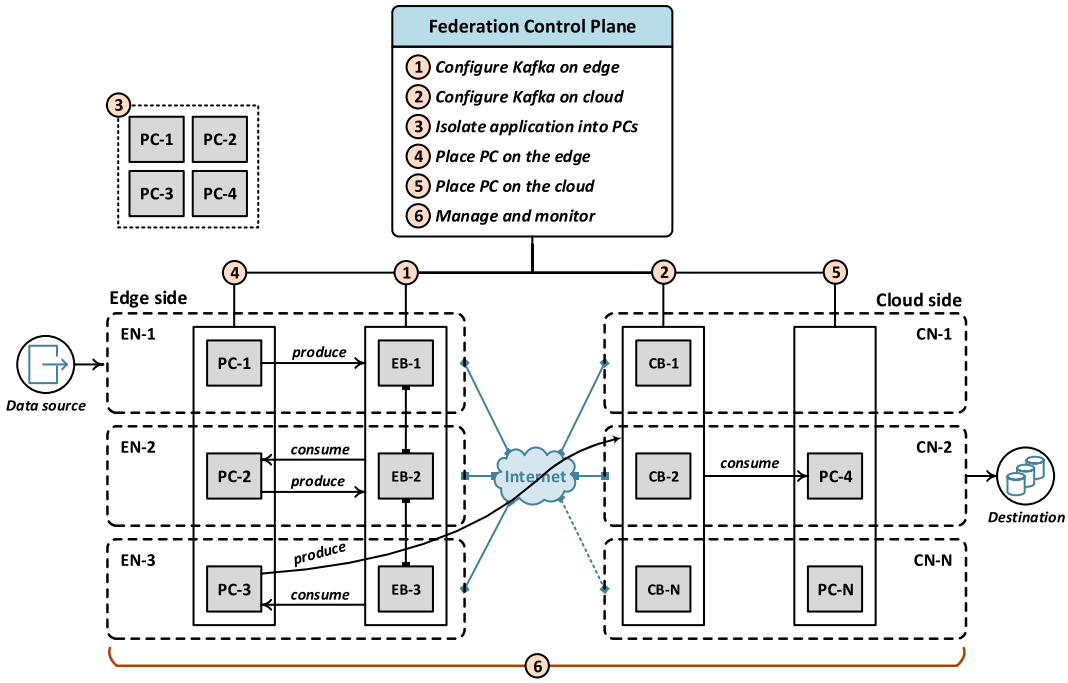


Fig. 8 Steps taken by the federated management layer

the federation control interface. Based on the aforementioned steps, the layer can perform deployment decisions for each node and have control over the entire federation. Therefore, our architecture becomes fully fault-tolerant in the sense that the failure of any single computing node on either the edge or the cloud will not disrupt the data processing pipeline.

4 Case Study: Smart Buildings

To evaluate the capabilities of our proposed architecture, we consider a smart buildings use case of the ongoing Otaniemi3D project. The proof-of-concept for Otaniemi3D has earlier been developed by the Aalto University researchers [8, 9], where Otaniemi is the name of university campus and 3D represents the dimensions in which IoT data are presented. Smart buildings are generally equipped with a Building Management System (BMS) that incorporates smart interconnected technologies to perform intelligent and responsive operations, such as fault detection, make automatic adjustments, alert management staff, and monitor performance. Such a BMS often performs data processing, data storage, and control decisions in the cloud servers. This requires longer time to access processed data and consumes a large amount of network bandwidth between the cloud and BMS. In many cases, these data may be returned to the BMS for immediate analysis, for example, controlling a ventilation system, fire alarms, or other data visualization purposes. However, in such situations, the BMS can be physically damaged by some malicious activity or the network connectivity in some of

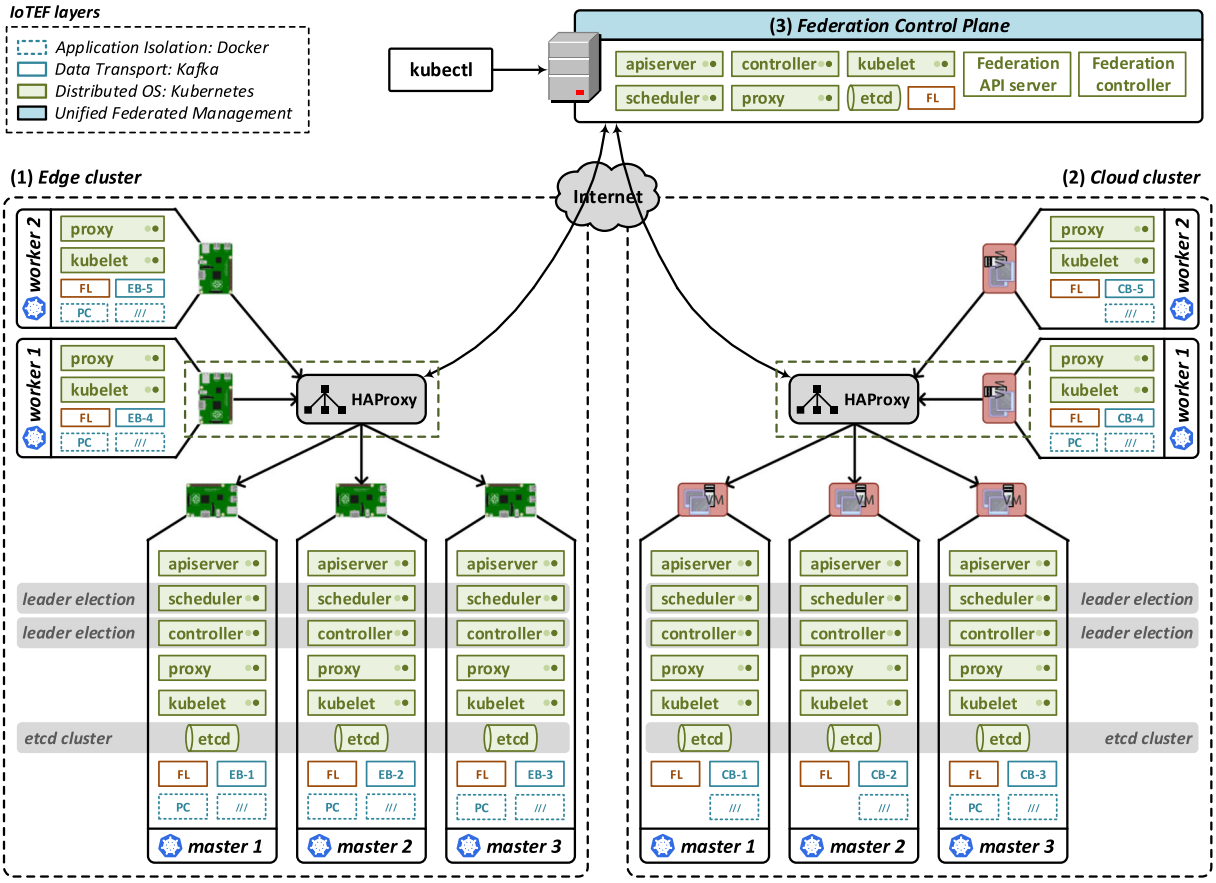


Fig. 9 An implementation model with the edge and cloud clusters, managed from the federation interface; where FL is the FLannel networking, EB is the Edge Broker, CB is the Cloud Broker, and PC is the Processing Container. HAProxy is deployed on worker node 1 for both the clusters

Table 3 Specifications of hardware/software tools

	Node type	OS	Docker	Kubernetes	Kafka	No. of nodes	Kafka brokers	etcd servers
Edge	Raspberry Pi 2 Model B+	Raspbian 9.4	v18.09	v1.9.7	v2.0	5	5	3
Cloud	Ubuntu VM	Ubuntu 16.04.5	v18.09	v1.9.7	v2.0	5	5	3
Federation	Linux Server	Ubuntu 16.04.5	v17.12	v1.13.3	–	1	–	1

the processing nodes may be temporarily cut off. This requires a unified fault-tolerant data processing capability for performing optimal deployment decisions *at runtime* on either the edge or the cloud. The system must also preserve and buffer data streams locally at the edge, especially in the case of network connectivity problems or malicious harming of edge or cloud nodes. One of the solutions is to process real-time data at the logical extremes of the network, that is, closer to the data sources of BMS, managed from a single federation system. Within this context, Section 4.1 explains our demonstrator system for the smart buildings use case and lists the hardware/software specifications. Section 4.2 describes different data sensors for obtaining real-time data.

4.1 Use Case Implementation with IoTEF

We employ the IoTEF layered design in the smart buildings use case to address all the aforementioned issues. A demonstrator system has been implemented in the Aalto University lab, which will be deployed in campus-wide area of the Otaniemi3D project. Figure 9 shows a running implementation of the selected use case in which the Application Isolation layer is realized in the form of all encapsulated processes, executing inside Docker containers. If needed, other application-specific processes, such as data collection and data compression, can also be encapsulated in this layer. The Distributed OS layer is logically observed by adopting the Kubernetes high availability framework. As seen in Fig. 9, the processes in green *solid-line* containers (apiserver, scheduler, controller, etcd, proxy, kubelet) configure the Kubernetes cluster. These components have previously been described in Section 2.1. Similarly, the Data Transport layer is modelled by configuring the Kafka pub/sub cluster for both the edge and cloud (processes in blue *solid-line* containers: EB-1 to EB-5 and CB-1 to CB-5). The federation control plane in Fig. 9 is the part of Unified Federated Management layer deployed through the Kubernetes federation scheme. All the IoTEF layers work together to enable the required capabilities in this use case. In our implementation, the communication between containers is provided by a Flannel[6] overlay network. It runs a small binary agent called "flanneld" on each node, which is responsible for allocating a unique subnet lease (/24 by default) out of a larger, preconfigured address space. Flannel directly utilizes either the Kubernetes API or etcd to store the network configuration, allocated subnets, and any other system data. Further, Table 3 lists the system parameters as well as specifications of hardware/software tools. Our implementation consists of three sub-systems:

1. **Edge cluster:** The edge-side cluster is composed of five Raspberry Pi (RPi) nodes. We use RPi 2 with 1GB RAM and a 900MHz 4-core ARM Cortex-A7 CPU. Each RPi executes Kafka and Kubernetes inside Docker containers, ensuring the same software stack which communicates through Kafka and are managed by Kubernetes. We use *kubeadm* tool to spread Kubernetes on the edge nodes. It performs the actions necessary to configure a minimum viable cluster in a user-friendly manner. This tool bootstraps the initial Kubernetes control-plane by executing *kubeadm init* command on all the three master nodes. We provide our own "config.yaml" file as an input to this command for setting up highly-available cluster. This file mainly consists of configuration parameters including etcd server endpoints, IP addresses of API server and load balancer, and network subnet for Flannel. As a result, Kubernetes initiates all the required components on three separate master nodes. Similarly, we execute *kubeadm join* on the other two worker nodes to join the cluster. The three master nodes of kubernetes, combined

[6][Online]. Available: https://coreos.com/flannel/, accessed in May 2019

with a 3-node etcd cluster and leader election feature, offers a single node fault tolerance. Thus, if one master fails, another node begins to operate as a new master node. As seen in Fig. 9, all five edge nodes are connected through HAProxy load balancer, which is deployed on *worker 1* node, enabling high availability by balancing API requests between the master nodes. More master and worker nodes can be integrated with the cluster. In addition, a 5-node Kafka cluster is also configured by executing one Kafka broker on each edge node, thus enabling data fault tolerance while tolerating two Kafka nodes failure.

2. **Cloud cluster:** We consider five Virtual Machines (VMs) for the cloud-side cluster in which each VM has 8GB RAM and a 2.40GHz Intel(R) dual core CPU. As seen in Fig. 9, this cluster also consists of three Kubernetes master and two worker nodes that are initialized through *kubeadm* tool. The implementation is similar to the edge-side cluster in which each VM runs Kafka and Kubernetes inside Docker containers, providing the same software stack. Thus, tolerating a single master node failure and two Kafka nodes failure in the cluster of five nodes.
3. **Federation:** The federation control plane executes on a separate server containing 16GB RAM and a 3.20GHz Intel(R) 4-core CPU. Both the cloud and edge clusters are added to this federation system, which are then managed from the unified control interface. Although we implement the federation plane on a single node and assume that it is available at all times, it can also be configured as a fault-tolerant system similar to the edge and cloud clusters.

4.2 Data Sensors for Smart Buildings

In our use case, we collect three types of sensor data from each building: (i) the images with motion detected in them are captured from the camera sensors, (ii) burglar alarm data are also recorded from each of these images, and (iii) the ventilation system data for each building are collected from 137 temperature, 96 CO2, and 55 central heating unit sensors. Figure 10 illustrates the three aforementioned sensors along with the map of Otaniemi area in which five RPi boards correspond to five separate buildings of the university campus. Although a single RPi sufficiently handles a smaller amount of data for several buildings, it may be unable to manage the data of all five buildings and perform excessive data computation on them. To tackle such situations, one RPi (edge node) is dedicated to process and buffer the real-time data for each building.

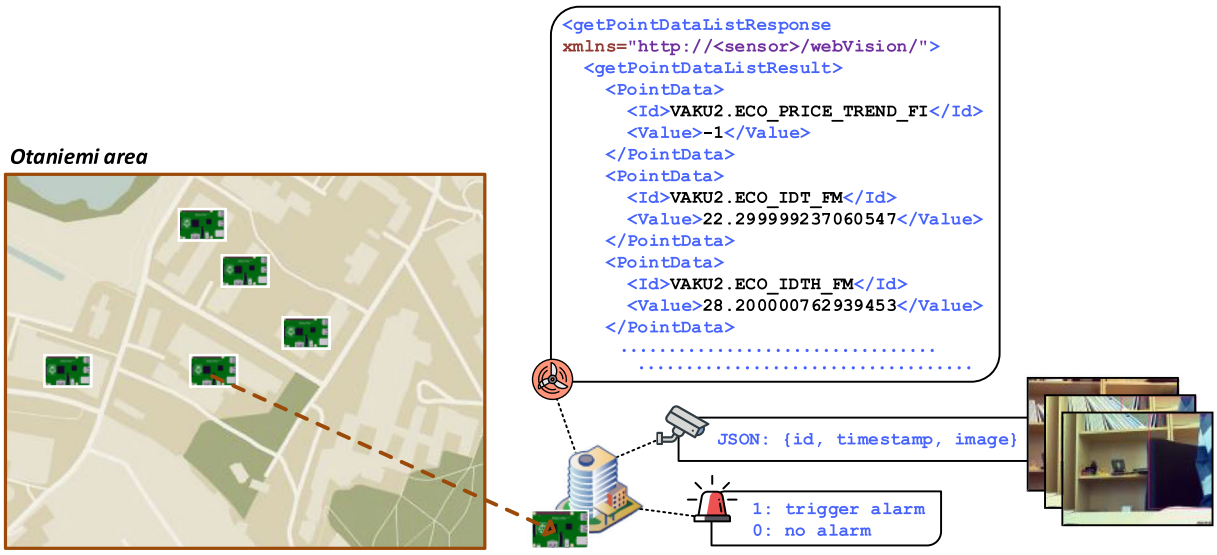


Fig. 10 The map of Otaniemi area with five RPi and three types of sensor data for each building

Table 4 Summary of various latency notations; where SP is the Special Process and PC is the Processing Container

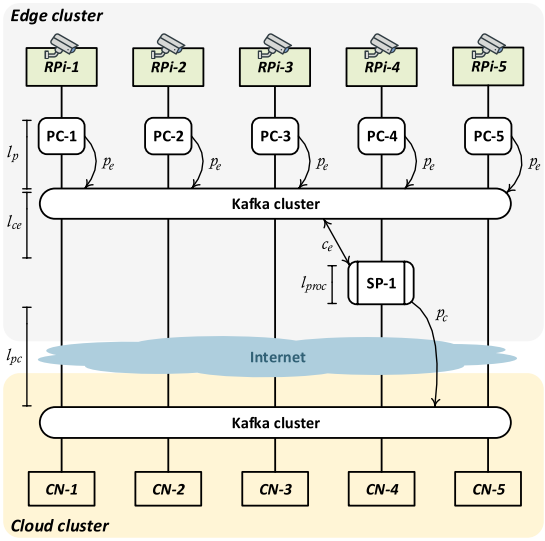
Symbol	Description	Definition
p_e	Produce to edge	Send single image from PC to edge cluster
p_c	Produce to cloud	Send single image from SP-1 to cloud cluster
c_e	Consume from edge	Consume single image from edge topic inside edge cluster
c_c	Consume from cloud	Consume single image from edge topic inside cloud cluster
ℓ_p	Latency of producing data on edge	Time taken to send a single image from camera sensor to the local edge topic
ℓ_{ce}	Latency of consuming data on edge	Time taken to retrieve a single image on edge from the edge topic
ℓ_{cc}	Latency of consuming data on cloud	Time taken to retrieve a single image in the cloud from the edge topic
ℓ_{pc}	Latency of producing data on cloud	Time taken to send a single image to the cloud topic after processing in SP-1
ℓ_{proc}	Latency of data processing	Time taken to process a single image on either the edge or the cloud
d_e	Delay in data processing on edge	Time taken to start another container on the edge after previous one fails
ℓ_m	Latency of moving data compute	Time taken to move the data processing container from edge to cloud

5 Performance Evaluation and Discussions

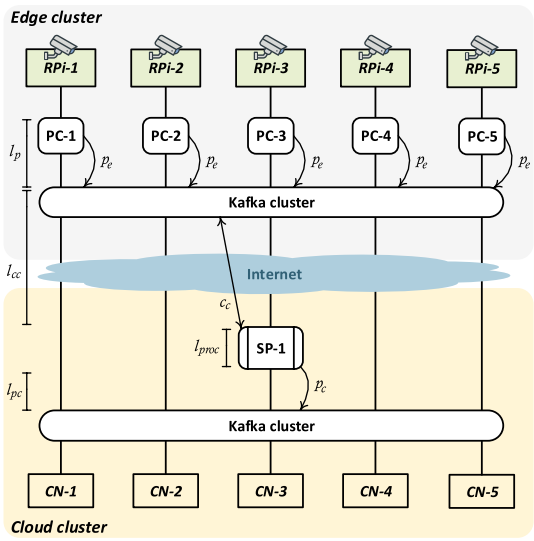
To assess the performance of IoTEF architecture implemented in the smart buildings use case, we consider the three aforementioned sensors and measure latency, throughput, and fault tolerance values. All the data are processed, replicated, and transmitted to the cloud for fault tolerance purposes, permanent storage, and further analytic. However, the results are only plotted for image data as the camera sensors generate a large amount of data and corresponds to the worst case in terms of our performance metrics. Other sensor types will lead to similar or better results. Table 4 defines the notations that are used for latency measurements on both the edge- and the cloud-side clusters. Likewise, Fig. 11 illustrates real-time scenarios in which a single local Processing Container (PC) runs on each RPi. These PCs collect images in jpeg format from the camera sensors attached to each RPi (each building), convert them into JSON string using Base64 encoder, and send them to the local Kafka cluster. The entire process in PC is measured as latency ℓ_p for each image. A Special Process (SP-1), also termed as Consume-Produce process, then consumes these images, processes them by applying image compression, and sends them to the cloud Kafka cluster. The retrieval of images in SP-1 from the edge cluster is measured as latency ℓ_{ce} and ℓ_{cc}, depending on the SP-1 location. Similarly, the same is the case for transmitting images to the cloud topic from SP-1 with latency ℓ_{pc}. We consider the image compression technique in SP-1 for processing as it is one of the approaches to decrease the payload on the network and simulate another processing time latency ℓ_{proc}. In fact, we can also reduce the amount of storage space for each image, thereby, buffering many of them inside the edge and cloud topics. Furthermore, the producer and consumer processes in Fig. 11 enable Kafka transactions API in Java to process images by initializing the transactions and commit them after the operation has been completed. Both the PC and SP-1 are created using the replication-based workload of Kubernetes and monitored through Kubernetes federation, ensuring the availability of these processes at all times. This SP-1 is also able to execute on any of the five edge or cloud nodes without source code modifications.

5.1 Performance in Terms of Latency

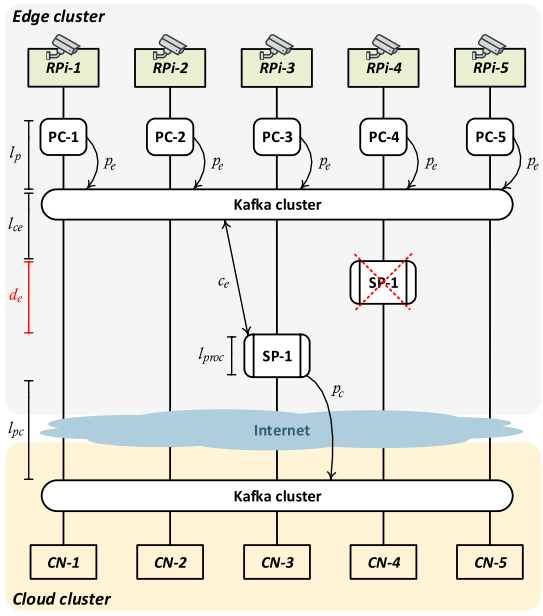
We measure various latency values and throughput by considering the scenarios illustrated in Fig. 11. The experimental results for processing image data on either the edge or the cloud are presented in Fig. 12. These graphs represent the latency distribution providing image latency as "latency" bins on the horizontal axis, and the frequency as number of images obtained in each bin on the vertical axis for three different image sizes. We consider images with dimensions 640x480, 1024x768, and 1280x1024 that occupy around 40kB, 87kB, and 146kB respectively. As seen in Fig. 12g, when the image size increases, the latency ℓ_p also increases since more time is required to prepare higher resolution images for transmission. As a consequence, a smaller number of images is sent to the local edge cluster. Compared



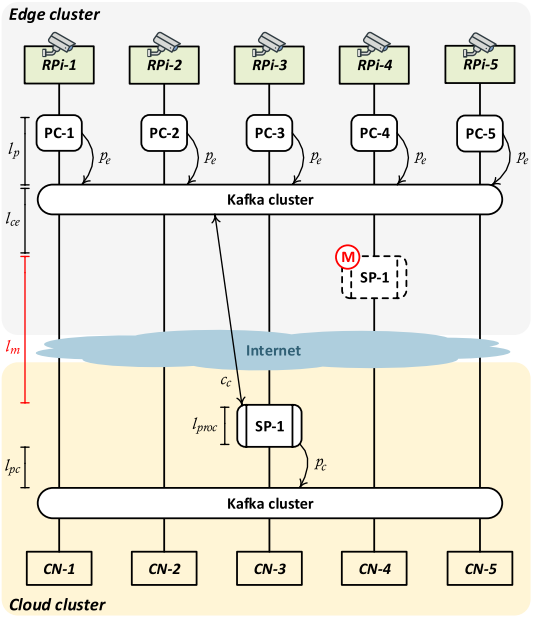
(a) A normal scenario with five PCs and one SP-1 in the edge cluster



(b) A scenario with five PCs in the edge and one SP-1 in the cloud



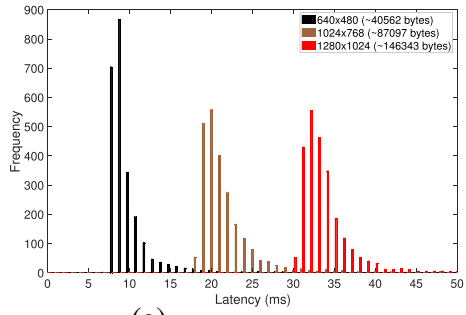
(c) A scenario when SP-1 fails in the edge cluster



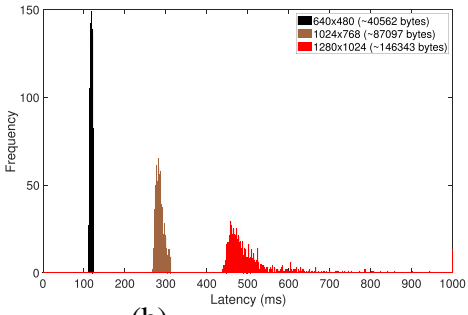
(d) A scenario when SP-1 moves to the cloud. Ⓜ shows data processing movement

Fig. 11 Real-case scenarios for assessing fault tolerance and measuring latency values

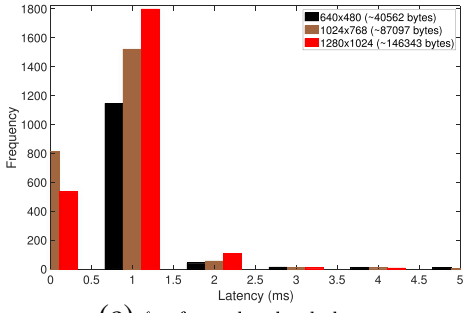
to 1280x1024 image in which the highest bin (around 180 images) has 40ms/image latency, 640x480 and 1024x768 images show better response at their highest bins. Thus, around 200 images of size 640x480 and 220 images of size 1024x768 are sent to the edge cluster with ℓ_p of 15ms/image and 25ms/image respectively. Figure 12a and b show the effect of data processing time (image compression in this case) on the cloud and edge respectively. As compared to the cloud-side cluster which has more processing power, there is a significant increase in ℓ_{proc} values on the edge-side cluster. For instance, with the image of



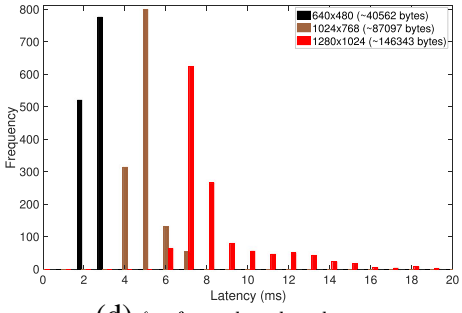
(a) ℓ_{proc} on the cloud



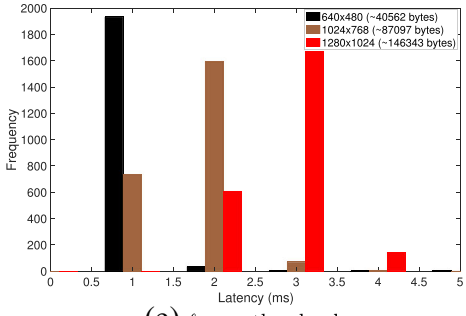
(b) ℓ_{proc} on the edge



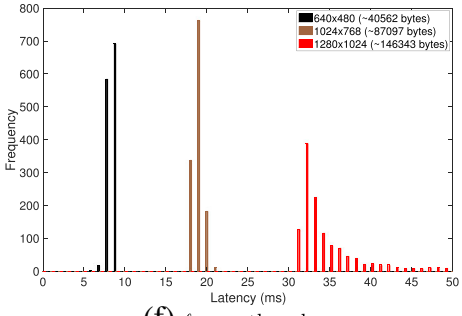
(c) ℓ_{pc} from the cloud cluster



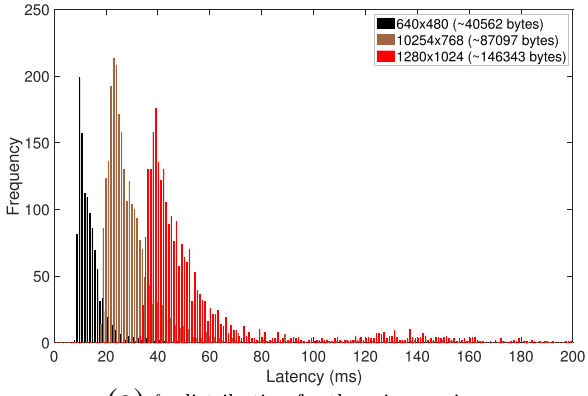
(d) ℓ_{pc} from the edge cluster



(e) ℓ_{cc} on the cloud

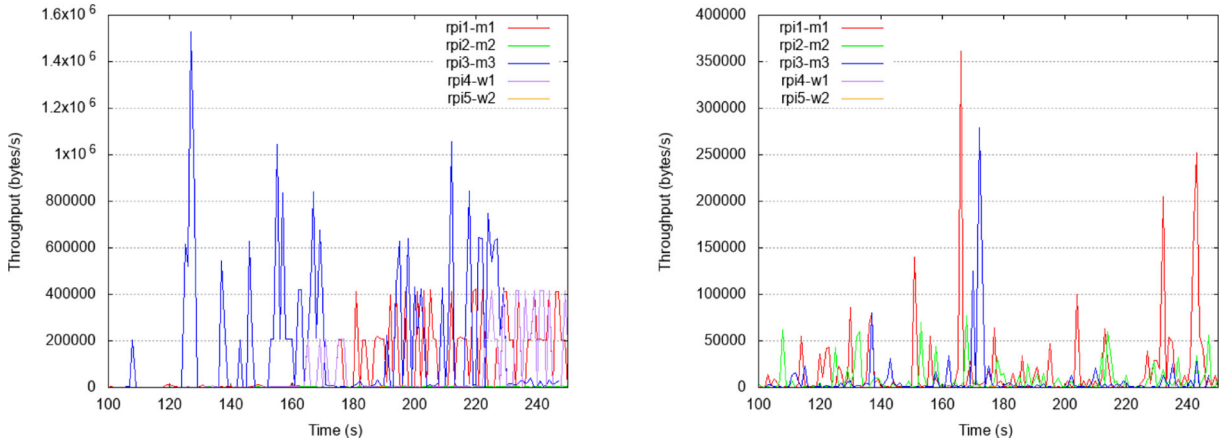


(f) ℓ_{ce} on the edge



(g) ℓ_p distribution for three image sizes

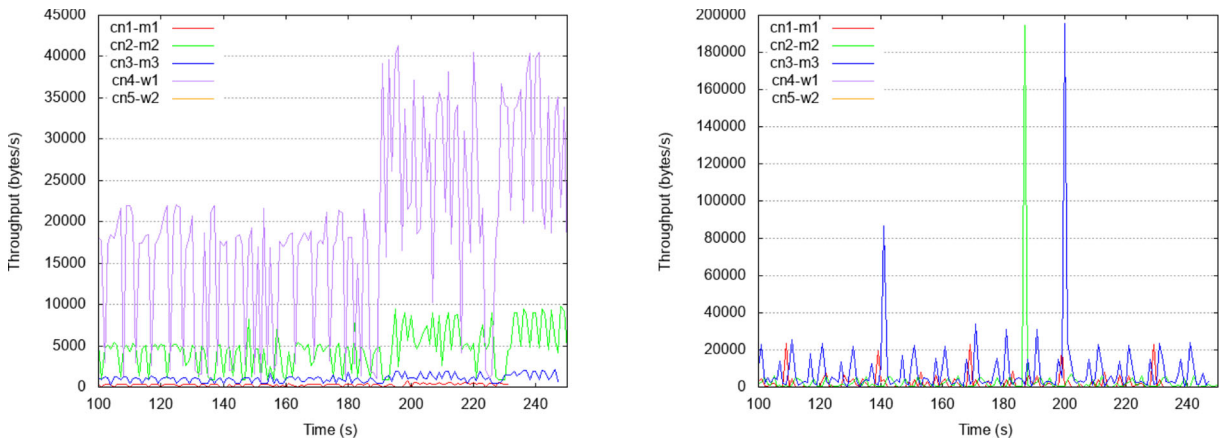
Fig. 12 Latency measurements distribution for three different image sizes (note that frequency is the number of cases/images in each "latency" bins obtained in our experiment)



(a) Throughput for Kafka traffic (b) Throughput for Kubernetes and etcd traffic

Fig. 13 Throughput in the edge cluster

size 1024x768, around 600 images are processed on the cloud with ℓ_{proc} of 22ms/image. However, the edge only processes around 70 of these images with 290ms/image ℓ_{proc} latency. When images are sent to the cloud for further processing, the ℓ_{pc} values are relatively low for both the cloud and edge, as shown in Fig. 12c and d respectively. In contrast to Fig. 12d, the ℓ_{pc} values in Fig. 12c are even more smaller when data transmission to cloud topic (from SP-1) happens on the cloud. This is because the load is at minimum and no other Kafka producers or consumers run on the cloud-side cluster. Thus, around 1200, 1500, and 1800 images of all three sizes are sent to the cloud topic with 0.75ms/image, 1.0ms/image, and 1.25ms/image ℓ_{pc} bins respectively. On the other hand, if SP-1 executes on the edge in Fig. 12d, a smaller number of images (around 790, 800, and 650 for all three image sizes) is sent to the cloud with ℓ_{pc} bins of 3ms/image, 5ms/image, and 7ms/image respectively. This is due to the limited amount of computing resources available at the edge and each RPi already has a single local Kafka producer active at all times, causing a smaller amount of delay. Similarly, the image data consumption (in SP-1) on the edge-side cluster is shown in Fig. 12f in which the ℓ_{ce} values are relatively higher than the ℓ_{cc} values on the cloud-side cluster (see Fig. 12e). As an example, with the image of size 1024x768, around 1600 images with 2ms/image ℓ_{cc}



(a) Throughput for Kafka traffic (b) Throughput for Kubernetes and etcd traffic

Fig. 14 Throughput in the cloud cluster

and 780 images with 19ms/image ℓ_{ce} are consumed on the cloud and edge respectively. Indeed, in this consumption case, the edge is able to retrieve, process, and send many images to the cloud with low latency, thus minimizing the network bandwidth. Further, in the scenario in which SP-1 fails on RPi-4 and resumes processing on RPi-3 (see Fig. 11c), we measure the delay d_e of 4 seconds. Similarly, the latency ℓ_m of moving data processing from RPi-4 to CN-3 node in Fig. 11d is noticed to be around 6 seconds.

In addition to the latency measurements, Fig. 13 and Fig. 14, respectively, show the throughput in the edge- and cloud-side clusters. These graphs represent the network traffic for Kafka and Kubernetes providing throughput in bytes/s over the period of time. We execute *tcpdump* on each node to capture and filter the network traffic. As seen in Fig. 13a, the Kafka traffic begins at 100s. It increases gradually since the producers and consumers start to produce, replicate, and consume data in the edge cluster. The network traffic is at maximum ($1.5x10^6$ bytes/s) around 125s. At this time, all the requests are sent to the Kafka topic partition leader which resides at `rpi3-m3` node. Once the data processing pipeline has been fully up and running, the throughput starts to increase after 180s for every edge node. On the other hand, Fig. 13b shows the administration traffic including Kubernetes and etcd for all five nodes. On average, this traffic represents around 10% of the overhead. The throughput for `rpi1-m1` node increases significantly to around 350000 bytes/s at 170s. This is due to the etcd leader that executes on `rpi1-m1` node and synchronizes the entire edge cluster state with the Kafka data processing pipeline. Similarly, as illustrated in Fig. 14a, the throughput in the cloud cluster is at maximum for `cn4-w1` node, since SP-1 executes on this cloud node. At around 190s, this SP-1 begins to consume image data from the edge Kafka cluster and the throughput increases to around 40000 bytes/s. The administration traffic including Kubernetes and etcd on the cloud is also plotted in Fig. 14b. On average, this traffic represents around 47% of the overhead as compared to the Kafka traffic in Fig. 14a. The throughput in Fig. 14b significantly increases to 200000 bytes/s at around 190s and 200s for both `cn2-m2` and `cn3-m3` respectively. This is due to the initiation of SP-1 container on `cn4-w1` node at around 190s which forces Kubernetes and etcd to synchronize the cluster state with other nodes. Overall, compared to the Kafka throughput in the edge cluster, the cloud cluster has relatively less throughput for all the nodes.

5.2 Fault Tolerance Assessment

We consider iptables utility as an administration tool for testing fault tolerance behaviour. This tool allows to configure, maintain, and inspect a set of IP packet rules for the network. Such rules enable the system to accept or reject specific network traffic. The following command is used to disable network connectivity:

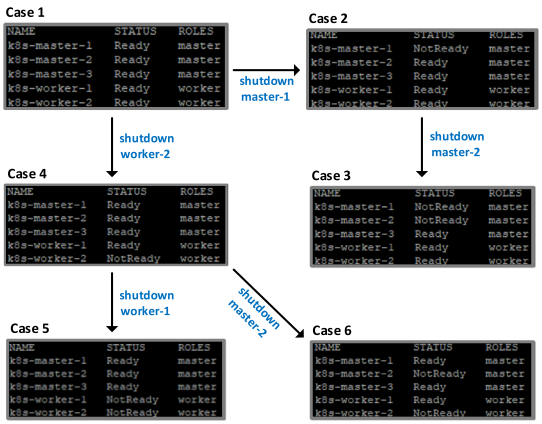
$$iptables - A\ INPUT - j\ DROP$$

As seen in Fig. 15a, this command appends a DROP rule to the end of an INPUT table. It is executed on one of

```
pi@k8s-master-1:~ $ sudo iptables -A INPUT -j DROP
pi@k8s-master-1:~ $ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
KUBE-SERVICES  all  --  anywhere             anywhere
KUBE-FIREWALL  all  --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere

Chain FORWARD (policy DROP)
target     prot opt source               destination
KUBE-FORWARD  all  --  anywhere             anywhere
DOCKER-USER  all  --  anywhere             anywhere
DOCKER-ISOLATION  all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
DOCKER     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
ACCEPT     all  --  anywhere             anywhere
```

(a) Example of iptables command on RPi



(b) Execution of *kubectl* for showing different cases

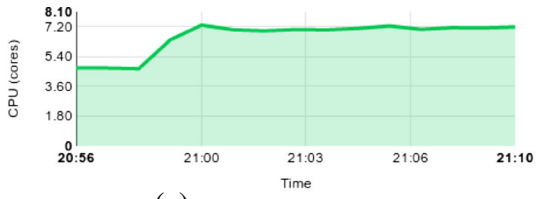
Fig. 15 Screenshots for fault tolerance assessment on the edge cluster

Table 5 Fault tolerance behavior on the edge-side cluster: ✓means active, × means inactive, M corresponds to the Kubernetes master node, W corresponds to worker node

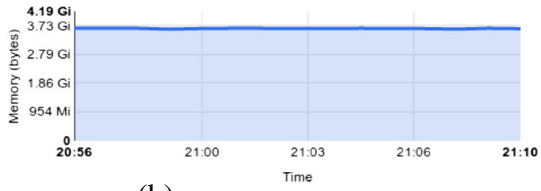
Case	M-1	M-2	M-3	W-1	W-2	Observations
1	✓	✓	✓	✓	✓	The default condition (best-case scenario) in which all nodes are up and running.
2	×	✓	✓	✓	✓	The cluster tolerates one node failure. Other master nodes handle the API requests.
3	×	×	✓	✓	✓	The data still process on other active nodes. However, the cluster is unable to handle new requests.
4	✓	✓	✓	✓	×	No effect on the high availability mechanism. The edge cluster only looses one computing node.
5	✓	✓	✓	×	×	No effect on the high availability mechanism. However, the cluster reduces to three nodes.
6	✓	×	✓	✓	×	The cluster continues to operate since two master nodes are still available for handling API requests.
7	×	×	×	✓	✓	The worst-case scenario in which we have no fault tolerance and no control over the cluster.

the RPi nodes and the DROP rule is highlighted with red box. Besides, Table 5 lists the observations for seven different cases in which the network connectivity between edge nodes is temporarily cut-off through iptables. These cases are obtained by running *kubectl* on the federated management server, and outputs are displayed in the form of `Ready` and `NotReady` status, as demonstrated in Fig. 15b. For each case, a scenario of Fig. 11a is considered and the results of CPU and memory usage for the edge cluster are collected in Fig. 16. Since both the edge and cloud have similar fault tolerance implementation, the behaviour is only analyzed for the edge-side cluster. The CPU usage graphs in Fig. 16 provide the time in MM:SS format on the horizontal axis and the aggregated CPU cores, which are dedicated for the entire Kubernetes edge cluster, on the vertical axis. We have in total 20 CPU cores in which each RPi has 4 cores. Similarly, the memory usage graphs provide the similar time duration on the horizontal axis and the aggregated memory in-use (out of total 5 Gbytes) of the edge cluster on the vertical axis. These results are plotted on the Kubernetes Dashboard (web-based UI) through Heapster metrics tool, which enable cluster monitoring and performance analysis for Kubernetes. Heapster collects various signals, such as compute resource usage and lifecycle events, and exports them via REST endpoints to the dashboard. As can be seen, Fig. 16a and b show the normal behaviour when all five edge nodes are active. In the beginning (at around 20:58), the nodes initialize data communication pipeline through Kafka cluster, thus increasing CPU usage from 5.0 cores to around 7.2 cores (at time 21:00). This value shows the active cores for our use case scenario. Similarly, the cluster occupies around 4.19 Gbytes out of 5 Gbytes. When one master node fails, the CPU and memory usage decreases to around 6 cores and 2.8 Gbytes (at time 21:12) in Fig. 16c and d respectively. Figure 16e and f show the behaviour when two master nodes become inactive. Both the CPU and memory usage decrease significantly to around 3 cores and 1.9 Gbytes respectively. Moreover, the cluster is unable to handle new requests as our implementation tolerates a single master node failure. Similarly, Fig. 16g–j correspond to the cases when one and two worker nodes become inactive. The CPU and memory usage slightly reduce to 3.5 cores and 2.25 Gbytes, respectively, for a short interval (from time 15:25 - 15:30) in Fig. 16i and j. During that time, the two nodes are unavailable for processing; nevertheless, other three nodes are active. The cluster continues to perform data processing tasks accordingly by self-adapting itself to the failures. The same is the case in Fig. 16k and l in which one master and one worker node become inactive at the time duration from 18:59 - 19:07. The CPU usage slightly decreases from 7 cores to around 5.25 cores; nevertheless, the cluster continues to operate and manages data processing pipeline. Overall, disconnecting two physical nodes from the cluster will not disrupt the data processing pipeline. The architecture self-adapts and reconfigures around two nodes failure (including at most one master node).

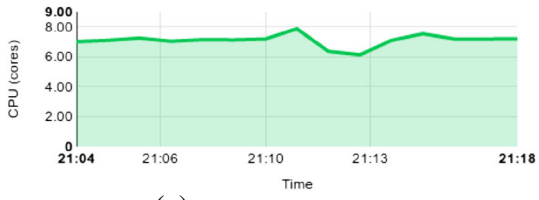
Based on the experimental results, it has been observed that the proposed Edge-Cloud architecture,



(a) Case 1: CPU usage

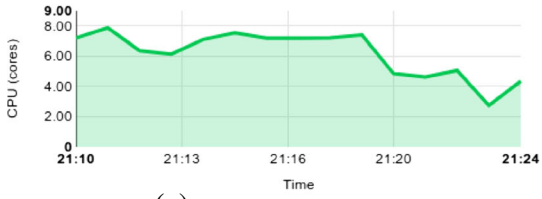


(b) Case 1: Memory usage

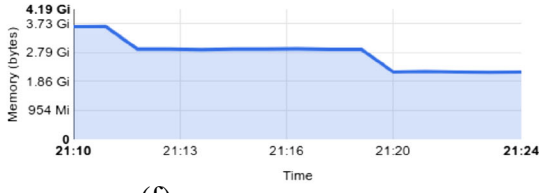


(c) Case 2: CPU usage

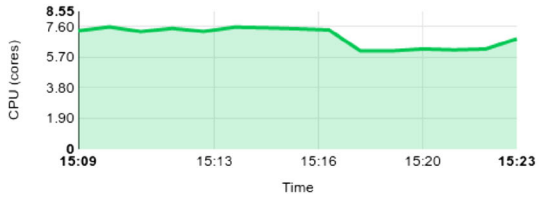
(d) Case 2: Memory usage



(e) Case 3: CPU usage



(f) Case 3: Memory usage

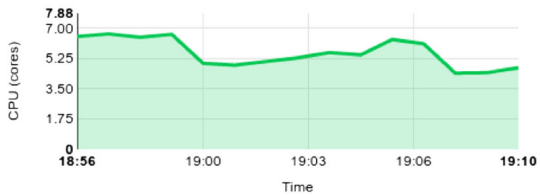


(g) Case 4: CPU usage

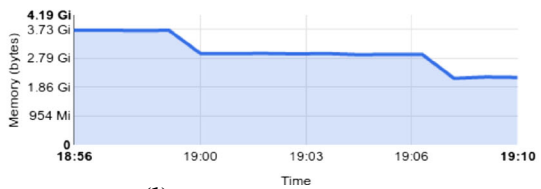
(h) Case 4: Memory usage

(i) Case 5: CPU usage

(j) Case 5: Memory usage



(k) Case 6: CPU usage



(l) Case 6: Memory usage

Fig. 16 CPU and Memory usage for fault tolerance cases. CPU (cores): the aggregated sum of active cores for the entire Kubernetes edge cluster, Memory (bytes): the aggregated memory in-use of the edge cluster, and Time: the time duration in MM:SS (minutes:seconds) for analyzing fault-tolerant behaviour

IoTEF, is capable of minimizing latency and consuming a smaller amount of network bandwidth by processing data items closer to the data sources. In addition, unified implementation for both the edge and the cloud, combined with federated management, enables multi-cluster fault tolerance and high availability in the IoT/CPS systems. Hence, it can handle hardware-based failures on the Distributed OS layer and network connectivity based failures on the Data Transport layer.

6 Conclusion

We conclude this paper in the following two sections. Section 6.1 describes the summary and implications of this work. Section 6.2 explains the limitations of our proposed solution along with the possible future directions.

6.1 Summary and Implications

This paper proposes a federated Edge-Cloud architecture, IoTEF, for IoT/CPS applications by adapting our earlier CEFIoT layered design. It uses the same state-of-the-art cloud technologies as CEFIoT including Docker, Kubernetes, and Apache Kafka, and also deploys them for edge computing. This new architecture has four layers: (i) Application Isolation, (ii) Data Transport, (iii) Distributed OS, and (iv) Unified Federated Management layer. Based on this layered design, the IoTEF architecture offers: (i) a common software stack for both the edge and the cloud; (ii) replication-based local fault tolerance on the edge devices to overcome nodes failure, network connectivity problems, or other harsh environments; (iii) data processing closer to the data sources for minimizing latency and consuming a smaller amount of network bandwidth; (iv) exactly-once data delivery for environments in which data may be processed more than once or not processed at all; and (v) a unified federated management for managing several clusters from a single management interface. We evaluate the capabilities of IoTEF by applying them to the smart buildings use case of the ongoing Otaniemi3D project at the Aalto University campus in Otaniemi. A demonstrator system has been implemented for both the edge and the cloud in which the data fault tolerance is tackled by employing the Apache Kafka publish/subscribe platform. We also deploy Kubernetes framework, combined with its federated scheme, offering a single management interface and enabling unified fault-tolerant management. Experimental results have validated the use case implementation in which our solution is capable of minimizing latency, reducing network bandwidth, and handling both node and network failures.

Furthermore, our proposed solution can be configured on the edge- and cloud-side clusters in which each computing node is able to execute Linux OS and other required softwares. However, many edge devices do not support these technologies. In such cases, we can integrate these edge nodes with another small cost-efficient processing module which supports Linux clone. Thus, making them compatible for our proposed infrastructure. The IoTEF solution can then become technologically-viable for such systems.

6.2 Limitations and Future Work

We implement and evaluate our proposed solution on the edge devices that are connected through LAN network. In the case of mobile edge devices which move between different networks, the IP addresses will change based on the new network. As a consequence, the entire system must reconfigure itself to resume data communication. The other edge components can still be able to communicate with the cluster for data publication and consumption. To achieve such mobile systems, our framework should implement a service enabling to update the whitelist of our nodes. However, there could be many limitations to such systems. As the devices move to a new network, the firewall settings need to be updated to open specific ports for communication. Moreover, the data processing latency, throughput, and delay values are also changed based on the location, network, or other parameters. In addition to the previous limitations, security at the edge-level is one of the main concerns in which the system should identify whether it is the same device that has been moved from one network to another. Our future work will address the implementation of IoTEF architecture on these mobile use cases along with the edge-level security. This security mechanism is particularly related to the device identification and data encryption on resource-constraint devices, ensuring secure data communication. Further, the architecture will be enhanced by enabling

Kafka transactional messaging (i.e., exactly-once data delivery) between two or more clusters.

Acknowledgements This research is supported by the EUs Horizon 2020 research and innovation program (grant 688203) and Academy of Finland (Open Messaging Interface; grant 296096, APTV; grant 277522, and SINGPRO; grant 313469). The authors would like to thank Jaakko Kotimäki and Markus Murhu from the CS IT department of Aalto University for their help in the experiments setup.

Funding Information Open access funding provided by Aalto University.

References

1. Choi, M., Park, J., Jeong, Y.-S.: Mobile cloud computing framework for a pervasive and ubiquitous environment. J. Supercomput. **64**, 331–356 (2013)
2. Atzori, L., Iera, A., Morabito, G.: The Internet of Things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
3. Främling, K., Holmström, J., Ala-Risku, T., Kärkkäinen, M.: Product agents for handling information about physical objects, WorkingPaper 153/03. Helsinki University of Technology, Laboratory of Information Processing Science (2003)
4. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: vision and challenges. IEEE IoT J. **3**(5), 637–646 (2016)
5. Schmid, S., Bröring, A., Kramer, D., Käbisch, S., Zappa, A., Lorenz, M., Wang, Y., Rausch, A., Gioppo, L.: An architecture for interoperable IoT ecosystems. In: Interoperability and Open-Source Solutions for the Internet of Things - Second International Workshop, InterOSS@IoT 2016, Held in Conjunction with IoT 2016, Stuttgart, Germany, November 7, 2016, Invited Papers, pp. 39–55 (2016)
6. Huang, Y., Garcia-Molina, H.: Exactly-once semantics in a replicated messaging system. In: Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany, pp. 3–12 (2001)
7. Javed, A., Heljanko, K., Buda, A., Främling, K.: CEFIoT: a fault-tolerant IoT architecture for edge and cloud. In: 4th IEEE World Forum on Internet of Things, WF-Iot 2018, Singapore, February 5-8, 2018, pp. 813–818 (2018)
8. Buda, A., Kinnunen, T., Dave, B., Främling, K.: Developing a campus wide building information system based on open standards. In: Lean and Computing in Construction Congress (LC3): Volume I - Proceedings of the Joint Conference on Computing in Construction (JC3), July 4-7, Heraklion, Greece, pp. 733–740 (2017)
9. Dave, B., Buda, A., Nurminen, A., Främling, K.: A framework for integrating BIM and IoT through open standards. Autom. Constr. **95**, 35–45 (2018)
10. Avizienis, A.: Toward systematic design of fault-tolerant systems. IEEE Computer **30**(4), 51–58 (1997)
11. Mei, J., Li, K., Zhou, X., Li, K.: Fault-tolerant dynamic rescheduling for heterogeneous computing systems. J. Grid Comput. **13**(4), 507–525 (2015)
12. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**(2), 74–80 (2013)
13. Pradhan, D.K., Reddy, S.M.: A Fault-Tolerant communication architecture for distributed systems, IEEE trans, journal=Computers, **31**(9), 863–870 (1982)
14. Laprie, J., Arlat, J., Béounes, C., Kanoun, K.: Definition and analysis of hardware- and software-fault-tolerant architectures. IEEE Computer **23**(7), 39–51 (1990)
15. Hwang, S., Kesselman, C.: A flexible framework for fault tolerance in the grid. J. Grid Comput. **1**(3), 251–272 (2003)
16. Jhawar, R., Piuri, V., Santambrogio, M.D.: Fault tolerance management in cloud computing: a system-level perspective. IEEE Syst. J. **7**(2), 288–297 (2013)
17. Su, P.H., Shih, C., Hsu, J.Y., Lin, K., Wang, Y.: Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware. In: IEEE World Forum on Internet of Things, WF-Iot 2014, Seoul, South Korea, March 6-8, 2014, pp. 45–50 (2014)
18. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A.C., Peterson, L.L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007, pp. 275–287 (2007)
19. Peinl, R., Holzschuher, F., Pfitzer, F.: Docker cluster management for the cloud - survey results and own solution. Journal of Grid Computing **14**(2), 265–282 (2016)
20. Newman, S.: Building microservices - designing fine-grained systems, 1st edn. O'Reilly (2015)
21. Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015, pp. 18:1–18:17 (2015)
22. Bernstein, D.: Containers and cloud: from LXC to Docker to Kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
23. Turnbull, J.: The Docker book: containerization is the new virtualization, James Turnbull (2014)
24. Kakadia, D.: Apache Mesos Essentials. Packt Publishing Ltd (2015)
25. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)
26. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: 2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010 (2010)
27. Weyrich, M., Ebert, C.: Reference architectures for the internet of things. IEEE Softw. **33**(1), 112–116 (2016)

28. Ganchev, I., Ji, Z., O'Droma, M.: A generic IoT architecture for smart cities. In: 25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014), pp. 196–199 (2014)
29. Tracey, D., Sreenan, C.J.: A holistic architecture for the Internet of Things, sensing services and big data. In: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013, pp. 546–553 (2013)
30. Premsankar, G., Francesco, M.D., Taleb, T.: Edge computing for the internet of things: a case study. IEEE IoT J. **5**(2), 1275–1284 (2018)
31. Rodrigues, J., Marques, E.R.B., Lopes, L.M.B., Silva, F.M.A.: Towards a middleware for mobile edge-cloud applications. In: Proceedings of the 2nd Workshop on Middleware for Edge Clouds & Cloudlets, MECC@Middleware 2017, Las Vegas, NV, USA, December 11 - 15, 2017, pp. 1:1–1:6, ACM (2017)
32. Kertesz, A., Pflanzner, T., Gyimothy, T.: A mobile IoT device simulator for IoT-Fog-Cloud systems. J. Grid. Comput. **17**, 529–551 (2018)
33. Krco, S., Pokric, B., Carrez, F.: Designing IoT architecture(S): a european perspective. In: IEEE World Forum on Internet of Things, WF-Iot 2014, Seoul, South Korea, March 6-8, 2014, pp. 79–84 (2014)
34. Kelaidonis, D., Rouskas, A., Stavroulaki, V., Demestichas, P., Vlacheas, P.: A federated edge cloud-IoT architecture. In: 2016 European Conference on Networks and Communications (EuCNC), pp. 230–234. IEEE (2016)
35. Alam, M., Rufino, J., Ferreira, J., Ahmed, S.H., Shah, N., Chen, Y.: Orchestration of microservices for IoT using docker and edge computing. IEEE Commun. Mag. **56**(9), 118–123 (2018)
36. Munir, A., Kansakar, P., Khan, S.U.: IFCIOT: integrated fog cloud IoT: a novel architectural paradigm for the future Internet of Things. IEEE Consum. Electron. Mag. **6**(3), 74–82 (2017)
37. Sarkar, C., Nambi, S.N.A.U., Prasad, R.V., Biswas, A.R., Neisse, R., Baldini, G.: DIAT: A scalable distributed architecture for IoT. IEEE IoT J. **2**(3), 230–239 (2015)
38. Cheng, B., Papageorgiou, A., Cirillo, F., Kovacs, E.: Geelytics: Geo-distributed edge analytics for large scale IoT systems based on dynamic topology. In: 2nd IEEE World Forum on Internet of Things, WF-Iot 2015, Milan, Italy, December 14-16, 2015, pp. 565–570 (2015)
39. Chang, H., Hari, A., Mukherjee, S., Lakshman, T.V.: Bringing the cloud to the edge. In: 2014 Proceedings IEEE INFOCOM Workshops, Toronto, ON, Canada, April 27 - May 2, 2014, pp. 346–351 (2014)
40. Elias, A.R., Golubovic, N., Krintz, C., Wolski, R.: Where's the bear?: automating wildlife image processing using IoT and edge cloud systems. In: Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, IoTDI 2017, Pittsburgh, PA, USA, April 18-21, 2017, pp. 247–258 (2017)
41. Ramprasad, B., McArthur, J., Fokaefs, M., Barna, C., Damm, M., Litoiu, M.: Leveraging existing sensor networks as IoT devices for smart buildings. In: 4th IEEE World Forum on Internet of Things, WF-Iot 2018, Singapore, February 5-8, 2018, pp. 452–457 (2018)
42. Tai, S., Rouvellou, I.: Strategies for integrating messaging and distributed object transactions. In: Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms, New York, NY, USA, April 4-7, 2000, Proceedings, vol. 1795 of Lecture Notes in Computer Science, pp. 308–330. Springer (2000)
43. Rimal, B.P., Jukan, A., Katsaros, D., Goeleven, Y.: Architectural requirements for cloud computing systems: an enterprise cloud approach. J. Grid Comput. **9**(1), 3–26 (2011)
44. Kubler, S., Främling, K., Derigent, W.: P2P Data synchronization for product lifecycle management. Comput. Ind. **66**, 82–98 (2015)
45. Lukša, M.: Kubernetes in action. Manning Publications Company (2018)