This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

Stream Processing on Clustered Edge Devices

Rustem Dautov*, Salvatore Distefano†‡
* SINTEF Digital, Oslo, Norway
rustem.dautov@sintef.no
† Università di Messina, Messina, Italy
‡ Kazan Federal University, Kazan, Russian Federation
sdistefano@unime.it, s_distefano@it.kfu.ru

***Abstract*—The Internet of Things continuously generates avalanches of raw sensor data to be transferred to the Cloud for processing and storage. Due to network latency and limited bandwidth, this vertical offloading model, however, fails to meet requirements of time-critical data-intensive applications which must act upon generated data with minimum time delays. To address such a limitation, this paper proposes a novel distributed architecture enabling stream data processing at the edge of the IoT network, broadening the principle of enabling processing closer to data sources adopted by Fog and Edge Computing. Specifically, this architecture extends the Apache NiFi stream processing middleware with support for run-time clustering of heterogeneous edge devices, such that computational tasks can be horizontally offloaded to peer devices and executed in parallel. As opposed to vertical offloading on the Cloud, the proposed solution does not suffer from increased network latency and is thus able to offer 5-25 times faster response time, as demonstrated by the experiments on a run-time license plate recognition system.**

***Index Terms*—Internet of Things; Edge Computing; Big Data; Stream Processing; Horizontal and Vertical Offloading; Apache NiFi; License Plate Recognition.**

I. Introduction, Motivation, and Contribution

THE traditional Cloud-centric data processing model adopted by the Internet of Things (IoT) is only suitable for scenarios with rather relaxed time constraints, as it fails to meet pressing requirements in terms of reaction time and network latency, especially in the presence of considerably big data streams. As time-critical IoT applications and services demand for near real-time data processing and reaction, they cannot rely on (potentially outdated) results obtained by sending data over the network to a remote processing location. This becomes particularly challenging in the context of bandwidth-constrained wireless connections ubiquitously present at the edge of IoT networks, thus limiting the disruptive potential of the IoT. Aiming to address this challenge, the Fog computing paradigm still remains limited in its capacity to support processing of extreme amounts of continuously flowing data in a ubiquitous manner similar to the Cloud, whereas network latency (albeit much lower) is still present. Supported by the ever-growing processing capabilities of devices at the network edge, the Edge Computing paradigm aims at pushing intelligence to devices that not only provide sensing and actuation resources, but also act as computational nodes in their own right. This way, execution of processing tasks immediately after data are generated and reduction of network traffic and latency, can be enabled by exploiting own processing capabilities of edge devices.

In this light, this paper further extends the scope of Edge Computing towards a wider range of data-intensive IoT application scenarios, where computational tasks can be offloaded to collocated edge devices. Specifically, it proposes a novel distributed Stream Processing architecture to enable *horizontal offloading* at the edge, by clustering devices and utilizing a shared pool of their contributed resources to process computational tasks offloaded by peers, i.e. *Clustered Edge Computing*, as apposed to the traditional vertical offloading to Fog/Cloud nodes. By pushing intelligence to the very edge of the network as close to the data source as possible, the proposed architecture aims to minimize the amount of data sent to a remote server, reduce network latency, and thus achieve faster processing results. Furthermore, considering storage limitations of edge devices, the proposed architecture also benefits from in-memory (stream) processing to minimize the number of disk I/O operations.

This proposed approach is implemented as the Edge Cluster Stream Processing (*ECStream Processing*) middleware enabling time-constrained data-intensive applications to be entirely deployed and executed at the edge. Accordingly, the contribution of this paper is three-fold: *i)* a *dynamic* horizontal offloading pattern for distributed data processing on clustered edge devices able to deal with the node churn at *run-time*; *ii)* a *decentralized* Stream Processing architecture, extending the Apache NiFi middleware with new clusterization services for in-memory processing of data streams on clustered edge devices, distributing modules among clustered nodes. The proposed approach goes beyond the traditional data parallelism model (e.g. MapReduce) towards a *task parallelism (pipeline) model*, wherein atomic tasks are offloaded to *peer* edge devices, rather than the full workflow, as in the traditional data parallelism; *iii)* a *comparison* of stand-alone local processing against Cloud (vertical offloading) and Clustered Edge Computing (horizontal offloading) through a case study, to demonstrate the viability of the approach and its potential exploitation in cluster capacity planning.

The remainder of the paper describes the ECStream

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

Processing approach, by first outlining the research context resorting to a motivating video processing application domain and then reviewing the literature to highlight main limitations of existing solutions (Section II). As a potential way of addressing such limitations, Section III presents the main aspects of ECStream Processing, including the stack architecture and the corresponding clusterization workflow. Section IV describes preliminary clusterization stages, while core activities are detailed in Section V (discovery and selection) and Section VI (deployment and operation). A preliminary implementation of the ECStream Processing Cluster middleware is proposed in Section VII, while Section VIII describes a case study on a run-time license plate recognition system, deployed on clustered edge devices available on-board a vehicle. This way, ECStream Processing is compared to Cloud-based horizontal offloading models through experiments with promising results and feedback. Section IX concludes the paper with final remarks and an outlook for future work.

II. Research Context and Related Works

Delayed data analysis and feedback generation often cannot be tolerated by critical systems, which rely on timely (i.e. quasi real-time) operation. These limitations primarily affect application domains involving, e.g., live video analysis where multiple image sensors, independently or combined, continuously capture video streams for online processing at the intersection of IoT and Fog/Edge computing [1]. Examples include intelligent surveillance systems (e.g. object/face detection and recognition), smart mobility (e.g. dashcams and infotainment systems), Industry 4.0 (e.g. machine vision for product/equipment surface inspection and staff tracking), emergency management, robotics, etc.

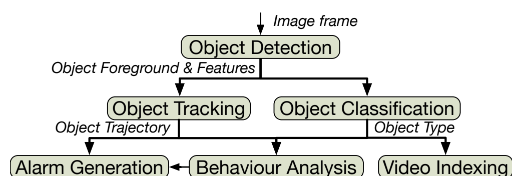


Fig. 1: A generic video processing workflow [2].

A typical video processing workflow can be conceptually split into several steps, as depicted in Fig. 1. Various objects, present in input frames, are first detected and then classified (i.e. recognized) according to a background knowledge base. At the same time, detected objects might be tracked through a sequence of frames to analyze object behavior and actions. Another element of such systems is some kind of notification/alarming, as well as storing of intermediate processing results and video indexing. Depending on the system architecture, different processing steps can take place at various physical and logical locations. For example, the initial object detection can take place immediately at the source, whereas more complex operations are undertaken on a remote server.

Raw video streams, continuously recorded by relatively modern capture devices, constitute a rather *big* data set from a perspective of an edge device and, consequently, can be treated as a Big Data challenge, hardly manageable by the device on its own (unless specifically conceived for that), thus becoming a perfect motivating scenario for the present work. A common technological trend to address such a limitation is to resort to vertical offloading, also looking for a right balance between low network latency of the Fog [3], [4], [5] and high computing capabilities of the Cloud [6], [7], [8] by hierarchical resource allocation and orchestration architectures that transparently provision containerized resources. These are often coupled with some optimization techniques and algorithms as in [9], [10], where the resource allocation problem for optimal placement of video analytics queries in such a hierarchy is formulated. However, the performance of such client-server architectures is affected by the network connection, worsening with the number of hops – a limitation hardly addressable by vertical offloading approaches due to the inevitable requirement to send data remotely. In these circumstances, minimization of the amount of data transferred over the network comes as a natural fit and is acknowledged as one of the main concerns for the IoT research community [11].

As a next step towards keeping computation locally, Edge Computing enabled data filtering and aggregation, as well as relatively simple processing to be executed on edge devices themselves as part of a more complex data processing workflow, the rest of which is expected to be accomplished by Fog and Cloud nodes [12]. Until recently, a shortcoming of the existing approaches focusing on data processing at the edge [13], [14] was the lack of support for pooling computing resources of multiple collocated edge devices, which only became possible with the recent advances in hardware and networking technologies. As a result, there are existing works [15], [16], demonstrating how edge devices can be clustered and managed through middleware at run-time, thereby achieving even lower latency. Similar to the Cloud- and Fog-level coordination, these approaches rely on equipping edge nodes with agent-like virtual containers to enable orchestration and management. This way, edge devices are able to communicate with each other to split, delegate, and share processing tasks.

Existing initial attempts to enable collaborative processing among edge devices by means of horizontal offloading typically rely on a central (Fog) coordinator [8], [17], which is in charge of cluster establishment and management. Clustering and orchestration of edge devices using hybrid, hierarchical Cloud-Fog-Edge offloading techniques are also proposed in [8], [18], [19], still via centralized Fog/Cloud-based coordination. This limitation is partially addressed by recent works focusing on *Mobile Cloud Computing* [20], *Mobile Ad-hoc Clouds* [2], [17], *Mobile Edge Computing* [21] and *Cyber Foraging* [22], which are able to pool resources of collocated mobile phones and IoT devices into *cloudlets* [17] or *fog colonies* [8], to support distributed processing. Similar to their centralized predecessor *Mobile Grid Computing*, such approaches do not address the heterogeneity of edge IoT devices restricting their scope only to mobile devices, providing basic infrastructure clustering mechanisms (e.g. pooling), not able to dynamically connect, discover, select,

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

and orchestrate devices. In more generic IoT contexts, *Multi-access Edge Computing* [21] addresses networking issues by extending Mobile Edge Computing with support for wireless (radio) connectivity at the edge.

As opposed to these existing approaches, the research effort presented in this paper aims at enabling a decentralized architecture, where participating clustered edge devices can act as both cluster initiators/coordinators and worker nodes. This architecture takes into account the mobile and heterogeneous nature of edge devices and enables dynamic discovery, selection and management of suitable nodes at run-time. Similar ideas are proposed and discussed, albeit at a more conceptual level, in [23], [24], where the authors motivate for horizontal offloading at the edge and outline a high-level architecture of a future system. As explained below, the proposed approach builds upon an existing Stream Processing middleware for in-memory data analytics, currently designed for static cluster configurations, and extends it with mechanisms for dynamic clustering and task offloading at system run-time. This approach goes beyond the traditional data-parallel processing model (i.e. MapReduce) and is able to 'unpack' Stream Processing workflows into finer-grained atomic tasks, thereby adopting a task-parallel processing model on clustered edge devices.

III. ECStream Processing

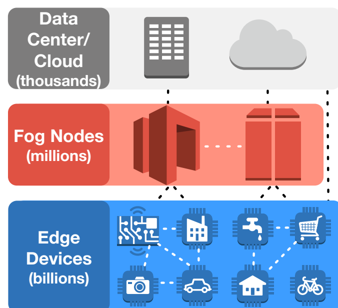


Fig. 2: IoT data offloading and processing patterns.

The challenges raised by data-intensive and time-critical IoT scenarios, such as online video processing, call for a solution bridging infrastructure and software layers. Such a solution is expected to foster the convergence of multiple paradigms, spanning across Edge, Fog, and Cloud Computing in a *computing continuum* (see Fig. 2) coupled with Big Data (batch/stream) processing techniques. To involve edge devices in this continuum, Edge Computing has to be enhanced with clustering techniques extending its application domain towards *Clustered Edge Computing (CEC)*. Combined with in-memory Stream Processing, CEC paves the way for the proposed *Edge Cluster Stream (ECStream) Processing* approach, resulting in a flexible solution for time-constrained IoT data processing on a cluster of collocated edge devices. This way, data processing is no longer Fog/Cloud-centric, but is rather Edge-centric – i.e. the workload is distributed among clustered edge nodes to avoid network latency and improve performance, while the Fog/Cloud servers remain as secondary processing/storage

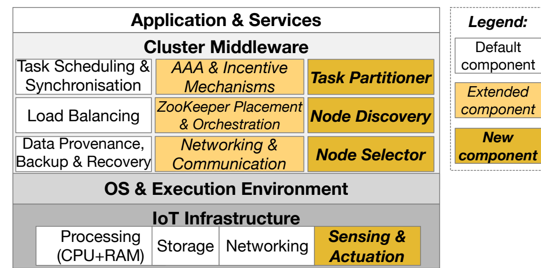


Fig. 3: ECStream Processing stack.

locations. This is highlighted in Fig. 2, where traditional *vertical offloading* to Fog and Cloud nodes (black dashed lines) are extended with *horizontal offloading* (white dashed lines) enabled by CEC.

A. Stack Architecture

The envisioned ECStream Processing has to deal with run-time clusterization, task decomposition, distribution, scheduling and orchestration, resource and data management, serialization and synchronization. Furthermore, since edge devices are usually resource-constrained, specifically in terms of storage facilities, a light-weight solution based on in-memory, online data processing of continuously streaming raw data has to be adopted. To this purpose, among multiple available open-source options,[1] we opt for Apache NiFi[2] – a light-weight, customizable, fault-tolerant Stream Processing framework. As opposed to more widely used implementations, such as Apache Storm, Spark, or Flink, the main advantage of NiFi is its low footprint – i.e. the smallest NiFi agent, written in C++ and specifically tailored to IoT devices, consumes as little as 5MB of memory. Based on the concept of *Flow-Based Programming*, NiFi allows defining control logic as a workflow composed of multiple interconnected processing steps (i.e. *processors*). The built-in set of NiFi processors ranges from simple mathematical operations, data translation or format conversion, to more complex analytical operations, and can be further extended with user-customized processors. NiFi features also include support for cluster management (i.e. ZooKeeper), scheduling algorithms, data serialization, backup and replication, network communication, monitoring, accounting, authentication and authorization (AAA), security and privacy using TLS encryption, and improved usability (e.g. IDE for visual workflow design).

To implement the ECStream Processing stack, we aimed to build upon existing functionality, making use of available NiFi features wherever possible, extending built-in or developing new ones, specifically conceived for clustering and management of edge devices. Fig. 3 shows the resulting four-layer ECStream Processing stack architecture, differentiating between *i)* completely novel (darker boxes with bold italic labels), *ii)* existing and extended (yellow boxes with italic labels), and *iii)* existing and taken *as-is* (white boxes) components (bottom-up):

1) *IoT Infrastructure* is no longer restricted to traditional servers, but also includes edge devices with their sensing

[1] https://github.com/manuzhang/awesome-streaming
[2] https://nifi.apache.org/

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

and actuation facilities. In the video processing context taken as a reference, cameras and other smart devices (e.g. smartphones) with processing, networking and sensing capabilities can be part of the infrastructure.

2) *OS & Execution Environment* serve as a unified platform for deploying and running middleware and software on top of the heterogeneous infrastructure. IoT heterogeneity is still an open issue, since edge devices can differ in both hardware and software capabilities. A well-established solution is *containerization* [25] – i.e. a light-weight form of virtualization, allowing to run multiple independent applications in 'sandboxed', isolated environments, thus achieving interoperability, while ensuring security and privacy crucial for the IoT.

3) *Cluster Middleware* is the core of the ECStream Processing stack. It is deployed on the resulting (homogeneous) execution environment and provides clusterization functionality. It extends the original NiFi architecture with six components, three of which are brand new (*Task Partitioner*, *Node Discovery* and *Node Selector*, while the rest are existing modules enhanced with CEC-oriented features. Specifically, *Networking & Communication* is extended with support for overlay networking, *ZooKeeper Placement & Orchestration* is enhanced with functionality for deploying and orchestrating distributed tasks in edge clusters, and *AAA & Incentive Mechanisms* include new primitives for subscription management (e.g. user contribution profiles) and incentive mechanisms (e.g. reputation systems, gamification, social incentives, financial rewards).

4) *Applications & Services* benefit from ECStream Processing results according to their business logic, even issuing downstream feedback actuation commands, thus promptly closing the loop to meet application time constraints.

B. ECStream Processing Workflow

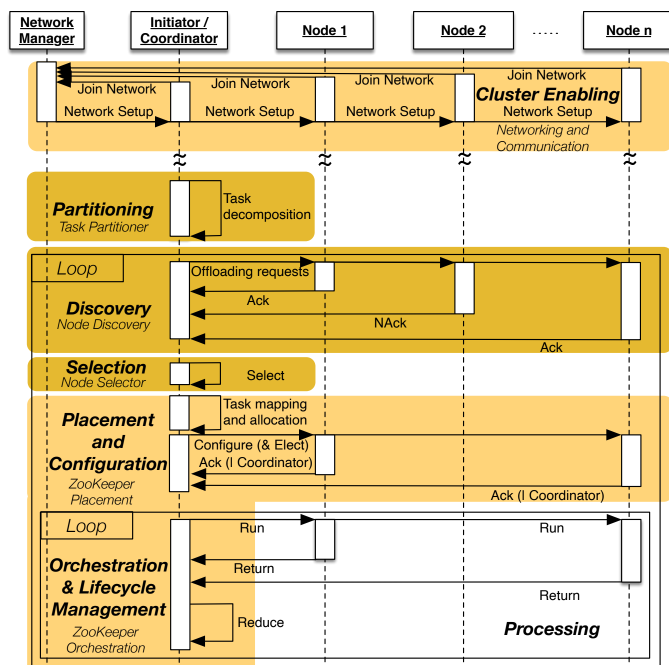


Fig. 4: Sequence diagram of the clusterization process.

The clusterization workflow is depicted in Fig. 4 and is executed by the ECStream Processing stack shown in Fig. 3. It continuously loops to dynamically adapt the edge cluster configuration to potential issues (due to the node churn) at run-time, thus requiring to perform all the activity steps online, demanding for a light-weight, low-latency implementation.

Clusterization is initiated by an edge node, the *Initiator*, willing to offload computational task to peers. To this purpose, it may be required to first establish a connection between nodes through a *Cluster Enabling* operation performed by a bootstrap node, the *Network Manager*. Once the connection is established, clusterization is triggered by the *Initiator*, broadcasting offloading requests to edge nodes as part of the ECStream Processing *Discovery* service. In a video processing application, the camera usually acts as the *Initiator*, delivering offloading requests to nearby network nodes. Upon receiving a request, eligible devices in the *Initiator* range can decide whether to support it. If so, they will then go through the *Selection* stage driven by various factors, such as mobility patterns, potential security issues, physical and network distances, etc. Then, it assigns tasks to selected nodes by sending them configuration parameters at the *Placement and Configuration* step. If there are at least 3 nodes in the cluster (i.e. a minimum sufficient number to run a leader election protocol), a *Coordinator* is elected among the nodes contextually. In case there is already an elected *Coordinator*, only an *Ack* is replied to the configuration message. For the sake of simplicity, Fig. 4 assumes that the roles of *Initiator* and *Coordinator* are undertaken by the same node. This way, the cluster is established and configured, ready to run offloaded tasks in parallel on its nodes (*Processing*), supervised by the customized *ZooKeeper Placement & Orchestration* module that performs *Orchestration & Lifecycle Management*. All these steps are explained below in more details.

IV. Preliminary Stages

A. Cluster Enabling

To support clusterization, networking issues have to be addressed first. Constituted by multiple mobile and portable smart devices that can move across different geophysical and network locations, the IoT ecosystem is very dynamic in its nature. Since the dynamic nature of such topologies is underpinned by wireless connectivity coupled with mobility patterns, possibly inducing the traversal of different network domains, it is important to take into account issues such as (sudden) introduction of address/port translators or security-oriented appliances (e.g. firewalls) between nodes, which may outright block or significantly modify inter-node communications, hindering the process of node discovery and clusterization. To this end, the built-in *Networking & Communication* module (see Fig. 3) is extended with support for *ad-hoc* topologies and overlay networking facilities. This further improves the *cluster discovery range*, as well as *node reliability*, allowing to enroll and keep edge nodes traversing different

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

subnets, even in the presence of network barriers that might otherwise impede their direct interaction.

To implement this, we based on existing work [25], [26] that enables (transparent) network communications between nodes in different subnets via *overlay networks*. As depicted at the top of Fig. 4, an (overlay) *Network Manager* (NM) gets contacted by other nodes aiming to establish an always-on command-and-control stream of messages, compliant with WebSocket-based Web Application Messaging Protocol. Network barriers are overcome through WebSocket-based (reverse) tunnelling by piercing 'middle boxes' for implementing overlay networks among edge nodes and transporting node-initiated network tunnels. Specifically, transparent Layer-3 (L3) networking is enabled by the *NM* that instantiates, manages, and routes tunnels to each node during clusterization, as well as during actual data processing afterwards.

Referring to the IoT video processing workflow, an example of cluster enabling is shown in Fig. 5(a), where nodes with network restrictions, although Internet-connected, send a join request to the *NM*, which replies with a setup configuration to establish WebSocket reverse tunnelling and, as a result, enable communication with other nodes.

B. Partitioning

Our target scenario assumes that a running edge node cannot meet processing requirements of a data-intensive application due to some resource constraints, and opts for sharing computational tasks, thereby triggering offloading at run-time, not at design/deployment time. It is also assumed that the initial setup and the requirements for the tasks to be offloaded are known and will further drive the partitioning activities by the Initiator. From the horizontal offloading perspective, such tasks can be allocated to nearby devices composing a cluster. To do so, the original (complex) application logic has to be decomposed into simpler tasks tailored to edge device capabilities, thus treating the original application as a sequence of atomic data processing operations. Furthermore, it is also mandatory to identify the requirements of partitioned tasks driving the discovery and selection steps to cluster matching devices.

Admittedly, partitioning is challenging to be implemented in an automated manner, as it cannot be generalized to any class of problems, since each sequential algorithm usually requires a specific partitioning model. Even restricting the scope to our target domain (i.e. data-intensive video processing applications) could not be enough, since it is required to go beyond pure data parallelism to let resource-constrained edge devices be able to run simpler stream processing tasks, rather than the full workflow. It is therefore necessary to identify and split concurrent blocks of the target sequential algorithm, and then apply a partitioning strategy that can possibly combine different task decomposition models. Admittedly, this requires deep knowledge of the target application, and descriptions of decomposed tasks have to include both semantic (e.g. information to be exchanged or processed, functionality to be performed) and syntactic (e.g. data structure and format) aspects. Taking these self-describing building blocks, the system can then chain complex workflows, validate information flows, input data and output results automatically. Software composability and task decomposition are partially explored by literature [27], [28] and deserve further investigation, since partitioning is still an open problem, especially in the context of the IoT and Edge Computing scenarios.

On this premise, a convenient solution for partitioning a (NiFi) stream processing workflow proposed in this paper is to first identify atomic tasks in a workflow and then parallelize them. This is quite a trivial approach, since a stream processing workflow mainly consists of task sequences, conditional branches and parallel fork-join constructs without loops [29]. The resulting workflow decomposed into a sequence of atomic tasks, will be then executed by exploiting a pipeline parallel model [30] on clustered edge devices, thus maximizing the throughput.

In the reference video processing scenario, a complex workflow can be partitioned as shown in Fig. 1, and a task to be offloaded could request for devices equipped with GPUs, optimized for such kind of processing. Further details on the underlying partitioning algorithm can also be found in [31].

V. Discovery and Selection

A. Discovery

To integrate edge devices into a common cluster, it is required to discover them on the network first. The discovery process should happen dynamically at run-time, since many edge devices are expected to be mobile (e.g. smartphones, tablets, and other hand-held portable devices), i.e. joining and leaving the wireless network unpredictably. This becomes particularly challenging as far as edge devices with sensing/actuating capabilities are concerned – i.e. as opposed to more traditional nodes, these need to be (semantically) described to be discoverable.

The discovery process can be conceptually split into two sequential steps: network discovery and functional discovery. The *network discovery* mainly focuses on finding connected nodes reachable by the *Initiator*. Apart from the TCP/IP-based networks (i.e. WiFi and Ethernet), it is also possible to discover peer devices through other wireless channels, such as Bluetooth, possibly implementing discovery in parallel and even hierarchical ways [29].

Once an edge node is discovered, the *functional discovery* checks whether the node is eligible for running a task among those identified by partitioning. The presence of a wide range of heterogeneous edge devices does not guarantee that all of available-discovered nodes will necessarily be capable of processing the current workload for a number of reasons (e.g. missing hardware/software components, low computational capabilities, high network latency, etc.). In these circumstances, it is important to check first whether a particular node is indeed suitable for processing a given task – that is, to check their functional suitability for a

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

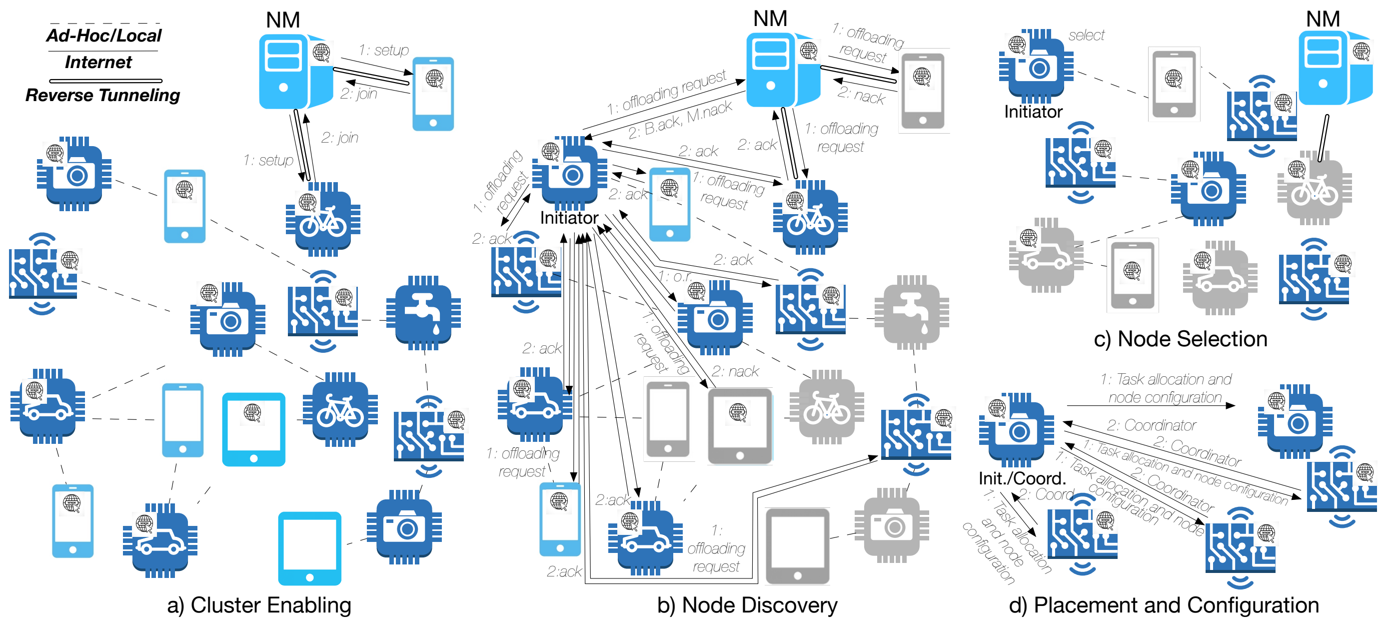


Fig. 5: An example of the ECStream Processing algorithm applied to the IoT video processing workflow.

task. To enable such kind of match-making, it is expected that previously discovered network nodes reply with a self-description specifying the provided resources and services (see Listing 1 for an example). Resource properties have to match with task requirements for proper allocation to ensure the node can provide expected hardware resources and software components, e.g. specific sensor/actuator, type of power supply (power line vs battery), network connection (wired vs wireless), mobility pattern (static vs mobile), security and privacy mechanisms, etc. For example, they may not be equipped with a relevant image recognition software, a camera, or have sufficient battery charge, and, therefore, will not acknowledge their suitability to participate in the given scenario.

The example shown in Fig. 5(b) depicts the discovered devices in the IoT image processing example. The network discovery, initiated by a node requiring task offloading (i.e. the *Initiator* smart camera), only discovers Internet-connected devices, exploiting the *NM* mediation for edge nodes with network restrictions. Grey nodes include both not Internet-connected devices and not available ones, rejecting the discovery request through a *Nack*.

B. Selection

The functional compliance check performed by potential cluster nodes during discovery is not yet enough to establish a cluster. Network nodes have a limited view on the arrangement of a cluster – that is, they are only able to evaluate their individual functional capabilities to address task requirements, but not their suitability to be engaged in a cluster. For example, a device might be equipped with sufficient hardware resources, as well as image processing software (i.e. thus meeting task requirements). However, it might turn out that, due to its network location and configuration, network latency between the cluster *Initiator* and this node is unacceptably high, which might become a bottleneck in the future. Admittedly, the node itself is not expected to be aware of this context-related information, which becomes known only to the *Initiator*, once it collects node acknowledgments.

Accordingly, the selection of edge nodes becomes an important duty of the *Initiator* that collects replies from all the nodes, and, therefore, has a global view on the system, including context-related information. While discovery takes into account single task requirements when identifying a node, selection considers *global policies* to further filter the previously discovered nodes, aiming to achieve a balanced and robust topology (see Listing 2 below for an example). The *Initiator* has to evaluate available nodes, which acknowledged its offloading requests, with respect to their suitability to global selection policies. Upon receiving acknowledgments, the *Initiator* may follow a policy to select e.g. only those devices that exhibit sufficient computing capabilities to process a task, whereas less powerful ones are to be excluded. Such a selection procedure also serves to 'homogenize' and balance the future cluster, so that it is composed of nodes relatively equal in their computing capabilities and network latency (i.e to avoid delayed processing by weaker nodes and further dis-synchronization). Selection policies might also include costs, which are strictly related to the incentive mechanisms. In this case, the *Node Selection* component of the ECStream Processing framework in Fig. 4 also has to interact with *AAA & Incentive Mechanisms* to enforce a selection policy taking into account credits, rewards and related technologies.

Noteworthy, during the selection process, more than one node could meet the requirements of a task and, vice-versa, the same node could meet requirements of multiple tasks. However, since we need to implement a pipeline parallelism, at most one task can be assigned to a node to maximize the pipeline speedup. Allocating

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

tasks to nodes is quite challenging – a problem known as *mapping* in parallel computing, which, even in the presence of constraints, falls into the class of NP-hard *generalized assignment problems*. To solve the ECStream Processing mapping of tasks to edge devices subject to the pipeline constraint, analytical solutions cannot thus be a valid option, even with a low number of nodes (tens as in CEC). To this purpose, some heuristics, usually based on greedy approximation algorithms (e.g. first-/best-/worst-fit allocation) with polynomial time complexity, already applied in Edge Computing contexts [32], can be adopted. Further investigation of this problem can be found in [33].

Fig. 5(c) depicts how the *Initiator* smart camera applies selection policies in the reference image processing scenario. One policy restricts the scope of the cluster to non-battery powered devices, thus excluding all mobile devices (e.g. smartphones, tablets and smart vehicles) from the cluster. Furthermore, since other two cameras are available, another policy selects the one directly connected to cluster nodes, while the camera reached by tunneling, due to network limitations, is discarded to reduce image processing delays.

VI. Deployment and Operation

A. Deployment

The previously identified tasks have to be deployed on the selected nodes and configured accordingly as part of *Placement and Configuration*. This is performed by the *Coordinator*, now identified by election among the selected peers orchestrated by ZooKeeper (usually the *Initiator* acting as the driver of the clusterization process). To inject the application logic into clustered edge nodes (i.e. the workflow tasks and corresponding configurations), the following mechanisms were modified or added to the original NiFi to implement the ECStream Processing middleware in Fig. 3:

a) Custom prioritizers: a mechanism for specifying the order of delivering jobs to processors, extending NiFi default prioritizers (e.g. 'First In – First Out', 'Last In – First Out', etc.) with parametric custom prioritizers is developed. Based on flowfile attributes, a custom processor can prioritize queueing flowfiles and thus define the processing order. Such prioritizers only act on the task processing scheduling and do not modify the cluster configuration or the workflow topology.

b) Parametric flowfiles: in Stream Processing, individual processors composing a workflow have no direct communications and the workflow deployment is performed by forwarding a flowfile from one processor to another through a queue. Furthermore, this implies that cluster nodes do not know about downstream processors and nodes, thus preventing any dynamic run-time flowfile routing based on characteristics of upcoming processors. This does not allow to deploy a flowfile with specific requirements, e.g. containing a video frame, on a node with matching resources, e.g. a GPU-based node processor. A flowfile attribute-based compliance check overcomes this issue by comparing its attributes to the node resource properties. If there is a match, the node keeps and processes the flowfile, otherwise it rolls back, placing the flowfile back on the input queue to be forwarded to a different node.[3] Such a mechanism allows to assign tasks to nodes according to the mapping schema defined in the previous steps.

c) RESTful interface: NiFi can be accessed and managed through a RESTful interface (which is also exploited by its workflow design interface) that allows to programmatically query and manage cluster nodes, as well as selectively connect or disconnect nodes according to task requirements and available node resources. This can be defined as a script or a custom processor triggered before deploying and executing the workflow topology so as to avoid inconsistent and unstable behavior of the cluster at run-time.

Fig. 5(d) depicts how the *Initiator*, supported by the described mechanisms, places and configures video processing tasks on selected cluster nodes, and is then elected as the *Coordinator* by the nodes through *ZooKeepers*, thus also acknowledging the assignments before starting execution.

B. Operation

Once the edge cluster is established and configured, the nodes start running allocated tasks concurrently, sending results to the *Coordinator* for reduction and aggregation, as requested by the original workflow. At the same time, the *Coordinator* manages and orchestrates the overall processing, periodically scanning the network to discover and select new nodes. The selected nodes will then be configured as new cluster nodes in order to run corresponding workflow tasks. This process is iterated till completion.

Node churn management mechanisms can be implemented by exploiting NiFi built-in ZooKeeper – a commonly present facility in Apache software projects. It allows keeping track of disconnected/failed nodes and update the cluster topology with respect to available nodes. This module (*ZooKeeper & Orchestration* in Fig. 4) has been further extended with the described run-time orchestration functionality tailored to edge clusters, as described below. In the case some incentives have been negotiated by the involved parties, this module has to also interact with the *AAA & Incentive Mechanisms* to enforce corresponding policies and finalize pending transactions.

VII. Preliminary Implementation

In the preliminary implementation of the ECStream Processing middleware,[4] the NiFi code baseline has been extended to implement the described enhanced functionality according to the architecture in Fig. 3. Following the NiFi distributed philosophy and using embedded *ZooKeeper* instances, it is deployed on top of edge devices using a decentralized *zero-master* coordination approach. This means that participating devices are equipped with the

[3]To prevent a flowfile from being infinitely queued due to the absence of relevant processing nodes, it is possible to implement a custom processor that will remove the flowfile from the queue for later processing, if there are no suitable processing nodes available.

[4]https://github.com/rdautov/ekstream

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

same middleware and equally suitable to act as both the *Initiator/Coordinator* of the cluster and usual worker nodes. Once deployed and configured, each NiFi instance is responsible for a range of background routine operations, including the default ones for networking and cluster communication, security, job scheduling, synchronization, backup and recovery, distributed coordination, data provenance, as well as the novel features discussed above. The higher *Application & Services* level comes with an intuitive thin client used to define workflows and transformations, and monitor the run-time cluster operation. This level deals with the actual flow-based programming, where users are able to design data flow topologies, made of data sources, processors, and connections between them.

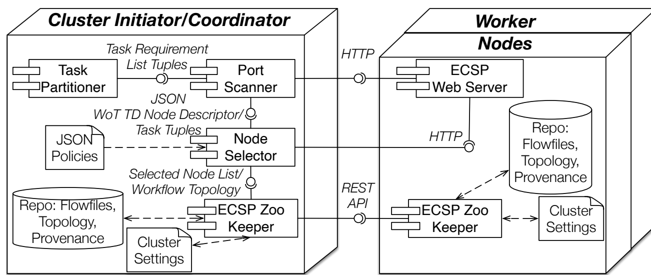


Fig. 6: ECStream Processing middleware implementation on top of Apache NiFi.

Fig. 6 schematically represents the design of the ECStream Processing (ECSP) middleware on Apache NiFi, describing the interactions between the *Initiator/Coordinator* and worker nodes. As detailed below, in this preliminary implementation, we primarily focus on the basic features to implement a minimal, yet viable middleware able to establish and manage a cluster of edge nodes.

A. Task Partitioner

The preliminary implementation of *Task Partitioner* relies on a basic decomposition algorithm, which splits the original workflow into connected sub-workflows, i.e. tasks, as discussed in Section IV-B. This way, each task of the workflow is considered as a NiFi processor represented by its (functional and non-functional) requirements, and connected to others according to the original workflow. These tasks/sub-workflows are described by a list of requirements and expressed as tuples

$$(Task_ID, Req_1, ..., Req_n)$$

where $Task_ID$ is the task identifier, and $Req_i = (Name_i, Op_i, Value_i)$ with $i = 1, .., n$ is a requirement triple representing a constraint applied to a property ($Name_i$) with a threshold value ($Value_i$) through a relational operator (Op_i). For example, a simplified computational task expressed by the tuple

$$(T_1, (CPU, >=, 1), (RAM, >=, 1), (Storage, >=, 10), (OS, =, Linux))$$

requires at least a 1GHz CPU, 1GB of RAM and 10GB of storage on a Linux-running device.

B. Port Scanner and Web Server

Node Discovery can be implemented in several different ways, ranging in their complexity based on the network configuration and constraints. As far as the network discovery of online nodes is concerned, this was implemented by means of the TCP port scanning facilities and integrated into the NiFi Web Server initialization code as the *Port Scanner*. As a result, the *Initiator* is able to scan network hosts on a specific port to detect other nodes running the ECStream Processing middleware and, therefore, potentially ready to join the cluster. To avoid situations when some other software occupies the given port, nodes discovered via the *Port Scanner* are also expected to report their unique ID, as part of the JSON heartbeat payload. If no node ID is reported, the network device is assumed not to be running the ECStream Processing middleware, and therefore is no longer considered for clustered processing.

It is important to remark that TCP scanning, a simple, effective and standardized solution for network topology discovery, requires that network nodes remain routable and are not subject to address/port translation (or any other kind of filtering). As discussed in Section IV-A, this is achieved via the *NM* that provides overlay networking capabilities to establish a virtual network for unhindered communication among nodes.

Listing 1: Device description in JSON-TD.

```
{ "id": "SBC_1",
    "coordinates": [38.26, 15.60],
    "ip": "172.30.127.77",
    "properties": {
        "CPU": { "type": "number",
            "description": "CPU clock in GHz",
            "href": "/node/properties/CPU" },
        "RAM": { "type": "number",
            "description": "RAM capacity in GB",
            "href": "/node/properties/RAM" },
        "Storage": { "type": "number",
            "description": "Storage capacity in GB",
            "href": "/node/properties/storage" },
        "Power": { "type": "number",
            "description": "Power type: -1 Power Line;
                0-100 Battery level",
            "href": "/node/properties/power" },
        ... }
}
```

Listing 1 reports a JSON description of a single-board computer (SBC), adopting the Web of Things (WoT) Thing Description (TD) model,[5] provided by the worker node Web Server in response to a request from the *Initiator Port Scanner*. A node is characterized by a unique ID, geographical and network location, available hardware resources, available software functionality, etc. It is also able to exchange *heartbeats* – light-weight JSON messages carrying all these relevant fields as payload.

C. Node Selector

The next step in the cluster configuration process is the node selection, wherein the *Initiator*, based on node selection policies and task requirements – on the one hand, and available nodes and resources – on the other, is able to configure the cluster as required. Node selection policies are evaluated against collected node self-descriptions and may

[5]https://www.w3.org/TR/wot-thing-description/

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

range from simple rules specifying rather static threshold values (e.g. a cluster node should have at least 1GB RAM and 1GHz CPU) to more sophisticated constraints that take into account how well individual nodes can co-operate within a cluster. That is, a selection policy might, for example, restrict nodes with an excessive response time – a potential shortcoming that might eventually affect the timely operation of the cluster in the long run. It is assumed that monitoring of such metrics at run-time is implemented using standard Linux resource and network utilization facilities. With system performance as a priority, the implemented prototype relies on JSON for representing policies – a simple, yet efficient way of capturing the node selection logic.

Listing 2: Task selection policy in JSON.

```
{ "policyId": "policy-1",
   "rule": {
      "ruleId": "noBatteryDev",
      "node:/properties/power": {
         "op": "="
         "type": [-1] }
   },
   ...
}
```

A simple selection policy expressed in JSON is reported in Listing 2. If a node description (similar to Listing 1) satisfies the task requirements of this policy, the node is selected to run the considered task. To exploit the pipeline parallelism, only one task can be allocated to a node. This means that allocation can potentially be unfeasible, if there are not enough nodes meeting task requirements.

Algorithm 1: Discovery and Selection algorithm involving Port Scanner (PS) and Node Selector (NS).

```
Input: Task list tasks and selection policies selPols
Output: Selected node with allocated task list selNodesTasks
discoveredNodes ← PS.scanNetworkPort(8080);
foreach task ∈ tasks do
    if discoveredNodes ≠ null then
        selected ← false;
        foreach node ∈ discoveredNodes do
            devDescription ← PS.getDescription(node);
            if NS.taskFiltering(devDescription,task.reqs)
             then
                nodeRes ← PS.query(devDescription.URI);
                if NS.reqResMatching(task.reqs, nodeRes)
                 then
                    if selected ← NS.select(selPols, nodeRes)
                     then
                        selNodesTasks.add(node,task);
                        discoveredNodes.remove(node);
                        tasks.remove(task);
                        break;            ▷ First fit algorithm.
                    end
                end
            end
        end
    else
        exit(-1);               ▷ Unfeasible: not enough devices!
    end
    if selected=false then
        exit(-2);               ▷ Unfeasible: unallocated tasks!
    end
end
return selNodesTasks;
```

Algorithm 1 illustrates the overall discovery and selection process triggered by the *Initiator*, which first scans the network for online nodes running the NiFi Web Server on port 8080 (scanNetworkPort(8080)), and then starts checking if task requirements, expressed as tuples, are met by the available nodes ($discoveredNodes$), which are then then contacted for their JSON descriptions (getDescription()). If the resources in $devDescription$ exposed by the node match the requirements of the current task (taskFiltering()), the algorithm queries the node for further details on the exact resource property values (query()). If the requirements do not match, the algorithm proceeds to the next available node in $discoveredNodes$. Otherwise, the algorithm compares values of task requirements and node resources (reqResMatching()) and, if the selection policies are also satisfied by the considered node (select()), the node is selected ($selNodesTasks$.add()) for the current task. The respective node and tasks will not be further considered by the algorithm ($discoveredNodes$.remove() and $tasks$.remove()). Finally, if there are not enough nodes or they are not able to satisfy all task requirements, the algorithm exits with errors. Otherwise, the assignment is deemed accomplished, and the list of paired tasks and nodes is returned. The *Port Scanner* provides scanNetworkPort(), getDescription(), and query(), while the *Node Selector* exposes taskFiltering(), reqResMattching(), and select().

As stated above, the selection process can be considered as a mapping problem, which has been demonstrated to be NP-hard. Algorithm 1 implements a first-fit greedy approximation heuristic, which allocates tasks one by one to a first-fitting (discovered) node. This way, the above algorithm, which is at the core of the overall clusterization process in Fig. 4, can be solved in polynomial time ($O(n^2)$). For the sake of feasibility, it is assumed that the number of discovered nodes m is of the order of magnitude of the number of tasks to be allocated n ($m \sim n, m \geq n$), since reqResMatching() and select(), required to process a single task-node pair, have constant time complexity ($O(1)$).

D. ZooKeeper

ZooKeeper[6] is a service for maintaining configuration information, naming, providing distributed synchronization and group services. It also provides basic facilities to implement consensus, group management, leader election, and presence protocols. At its core, *ZooKeeper* is a distributed file system with so-called *ZNodes* that store snapshots of the current system state. *ZooKeeper* can be configured to run either as a centralized stand-alone service, or as multiple *embedded* instances in a distributed manner. In the latter case, the available built-in synchronization and state management facilities allow to synchronize all instances with minimum delay in a reliable and fault-tolerant manner. This is especially useful for distributed cluster setups, where individual nodes may become unavailable due to failures or network barriers. With multiple embedded *ZooKeeper* instances, each node is continuously updated with the current state of peer nodes and jobs in the queue, thus ensuring their eventual execution even in the case of node failures. Distributed *ZooKeepers* are also useful during the election of the *Coordinator* of the cluster (initially assigned to the *Initiator* by default), whenever the

[6]https://zookeeper.apache.org/

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

current *Coordinator* node goes offline. The leader election is implemented using a reliable and efficient protocol [34], ensuring that the cluster has its *Coordinator* at all times. All nodes in the cluster will then continuously send heartbeat/status information to the *Coordinator*, which may also disconnect non-responsive nodes. Additionally, when a new node joins the cluster, it must first connect to the currently-elected *Coordinator* to obtain the most up-to-date flow. These activities, executed through *ZooKeeper*, have relatively small impact on the *Coordinator*, and are comparable to the overheads of the rest cluster nodes.

Once the *Initiator* knows all nodes and their tasks within the cluster, it is time to deploy the workflow topology. This functionality is implemented by NiFi RESTful API and *ZooKeeper*, adapted to ECStream Processing purposes. Among other things, the API provides entry points for querying and updating the current cluster configuration by, e.g. connecting/disconnecting nodes or specifying stand-alone processes (i.e. executed on a single node). Accordingly, the *Initiator* first updates its own settings, which are then synchronized across the cluster by embedded *ZooKeepers*.

ZooKeeper has also been extended to implement *Orchestration and Lifecycle Management* by running a continuous looping routine, during which the configured computational topology is executed on the cluster nodes in parallel. The *Coordinator Port Scanner* keeps on scanning the network for new potential worker nodes. Whenever a new node appears on the network, it needs to go through the same initial steps and, if successful, will be added to the cluster in a seamless and transparent way – i.e. there is no need to stop and restart the already running cluster in order for a new node to be integrated. This is also facilitated by *ZooKeeper* that handles the node churn and synchronizes topology changes across all cluster nodes.

VIII. Proof of Concept

To fully demonstrate viability of the proposed approach, a pilot scenario had to meet the following requirements:

a) The amount of data generated by the pilot and the application logic are large and complex enough to require Stream Processing techniques for their management.

b) Data processing, involved in the target scenario, is computationally intensive and goes beyond the capabilities of a single edge device, thereby requiring offloading.

c) The pilot application logic can be decomposed into simpler, 'parallelizable' tasks.

d) The pilot scenario has strict time constraints.

e) The surrounding urban IoT environment, composed of various edge devices, is dynamic – that is, different connected devices may randomly appear in close proximity to the source of data at unpredictable rates. On the other hand, the environment should not be too dynamic either, as it happens, for example, in vehicular *ad-hoc* networks characterized by very short-lasting connections.

f) Collocated edge devices, albeit resource-constrained and/or mobile, are powerful enough to run Linux OS and have an executable environment, such as JRE. This means that target devices are equipped with a microprocessor and a wireless networking interface.

Based on these requirements, a target scenario can be identified in the domain of relatively complex image/video processing in an urban environment, where mobile/portable nodes can share their idle resources. In this context, among a number of public surveillance and monitoring applications, a particularly novel and challenging topic is run-time license plate recognition. At present, a network of traffic monitoring cameras typically covers only road junctions and intersections to detect speed limit violations, whereas most of the roads are not monitored at all. Moreover, such cameras are usually only involved in off-line image recognition – i.e. they transfer captured images (together with the violating speed values) to a server, responsible for the actual recognition of license plates.

This limitation could be potentially addressed by a pervasive network of personal image capturing devices, such as vehicle dashcams and personal smartphones. Indeed, there are millions of drivers worldwide who use on-board cameras to continuously record the surrounding environment. The current use, however, is limited to offline manual analysis of the recorded video in case of various incidents (accidents, car break-ins, 'hit-and-go', etc.), since run-time automated image analysis is currently beyond the capabilities of a single device. The situation might change with the ubiquitous presence of increasingly powerful edge devices, either personal hand-held gadgets or smart road-side infrastructure. These vast processing capabilities open up opportunities for using on-board dashcams to perform run-time situation assessment by pooling computing resources of edge devices and distributing the workload in an ECStream Processing fashion.

Limiting the scope of the generic video processing workflow depicted in Fig. 1, the envisaged scenario, therefore, is the following. The dashcam installed in a vehicle acts as the source of the video stream (and possibly as the WiFi access point in the case there is no built-in access point in the vehicle), whereas other WiFi-enabled smart devices available within the car, including personal smartphones, tablets and an on-board infotainment system, can connect to the network and communicate with each other. The dashcam is then able to sample the video stream into individual frames and distribute them among participating nodes for parallel license plate recognition.

A. Testbed Setup

To compare and evaluate the proposed approach to the existing technological baseline, below we present and explain how the challenge of automated run-time license plate recognition can be potentially faced. We thus identify three possible setups, in addition to a fourth setup running on clustered edge devices. In all setups, OpenALPR[7] is used as the underlying license plate recognition software.

1) Stand-alone OpenALPR on a single device (EC) - This setup represents a situation when an onboard dashcam has

[7]http://www.openalpr.com/

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

to perform license plate recognition against the captured video stream on its own, in a typical Edge Computing (EC) fashion. Admittedly, the dashcam market is saturated with different types of devices, varying in their hardware specs and architectures. A common practice is to use a smartphone with a special app to act as a dashcam. As an average representation of this plethora of dashcam models, this setup employs a Raspberry Pi board running Raspbian OS with OpenALPR, thus sampling the video stream and immediately feeding the resulting images to OpenALPR.

2) Stand-alone OpenALPR on Google Compute Cloud (IaaS) - Cloud-based functionality can be implemented by deploying the free OpenALPR API on a public IaaS Cloud platform and making it available as a RESTful Web service. This will receive and process incoming images and return the results back to the user. In our experiments we used a Google Compute Engine[8] virtual machine deployed in the EU to implement this setup.

3) Stand-alone OpenALPR Cloud API (SaaS) - Apart from a freely available software library to be installed on-premises, OpenALPR also offers a commercial Cloud SaaS service[9] (deployed in the US) providing a RESTful API for license plate recognition. Client applications can either stream video or transfer a single image and receive notifications on the recognized license plate.

4) OpenALPR on an Edge Cluster (CEC) - This setup implements the proposed ECStream Processing architecture, in which video frames from a dashcam are distributed among a cluster of edge devices (e.g. passenger smartphones) for Stream Processing in a CEC fashion.

TABLE I: Testbed hardware specs and network speed test.

Setup	Hardware	Uplink, Mbit/s	Downlink, Mbit/s	Round Trip Time, ms
EC	Raspberry Pi 3 (1.2GHz ARM Cortex-A53, 1GB RAM)	n/a	n/a	n/a
IaaS	n1-standard-1 (located in EU, 2.52 GHz vCPU, 3.75GB RAM)	1.24	1.63	482
SaaS	Amazon EC2 (located in US)	0.77	0.93	524
CEC	Raspberry Pi 3 + 1-6 Samsung Galaxy J5 (1.2GHz Cortex-A53, 1.5GB RAM)	16.58	26.9	144

The configuration of all four testbeds is summarized in Table I, which covers both hardware and network specification. At its current stage, the prototype implementation relies on a pre-recorded dashcam stream as the input video source,[10] captured during a ride through London downtown in HD quality of 1,920 × 1,080 pixels, resulting in 1,350 KB aggregate payload transferred on average, at the frequency of 30 frames per second. Each device is assumed to be running Debian-based Linux OS (Linux Deploy[11] was used to emulate the Linux environment on top of Android OS on smartphones) and a pre-deployed instance of the ECStream Processing Cluster middleware with customized NiFi processors. Target license plates can be dynamically pushed to vehicles involved in the run-time license plate recognition, which then re-configure their internal cluster nodes to the new plates. Upon detection, a notification with a corresponding screenshot, GPS location, and a timestamp is issued to interested parties (e.g. police).

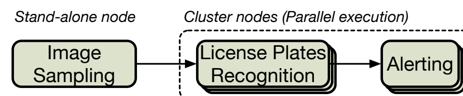


Fig. 7: The streaming workflow in the context of the license plate recognition scenario.

To be processed in parallel on the edge cluster, the license plates recognition workflow is partitioned into three custom NiFi processors, as shown in Fig. 7. `Image Sampling` takes an input video stream, samples it into separate frames, and transfers the resulting images to an output port for recognition. `License Plates Recognition` is responsible for detecting and recognizing license plates in incoming images by invoking the OpenALPR library. As an output, it provides a list of license plates with a matching confidence value. `Alerting` notifies interested parties whenever target license plates are recognized. It can be configured for using various channels, such as API, e-mail, or SMS.

To trigger the ECStream Processing clusterization process shown in Fig. 4, on-board devices have to be located on the same WLAN. This way, the dashcam (i.e. Raspberry Pi), acting as the Initiator, is able to scan the network to discover worker nodes through its Port Scanner. Recipient nodes reply to the incoming request with their JSON-TD self-descriptions, as explained in Section VII. Then, the Initiator, now becoming the Coordinator, is able to select suitable nodes. In the considered scenario, all nodes are suitable for task offloading (both license plates recognition and alerting) and there are no selection policies to be enforced by the Coordinator. ZooKeeper thus allocates the tasks to the available nodes, replicating them to run in parallel. As a result, the dashcam is tasked with a stand-alone operation of sampling the video stream and broadcasting frames, whereas the worker nodes perform license plate detection/recognition and alerting in parallel.

B. Experiments and Benchmarking

The main benchmarking metric for the license plate recognition experiments is the *response time* – i.e. the time difference between the instant when an image is first sampled by the dashcam and the instant when the system accomplishes the license plate recognition task. This metric is two-fold, and includes *i)* time delays associated with network latency and data (de-)serialization when transferring images (*overhead*), and *ii)* time spent on actual data processing (*processing time*). Further metrics of interests for our case study are the *throughput* – i.e. the number of frames each setup is able to process in a second, and the *speedup* – i.e. the ratio between the response times obtained by the sequential processing of the license plate recognition workflow on the different setups and the ones obtained by parallel execution on the edge cluster varying

[8]https://cloud.google.com/compute/
[9]https://www.openalpr.com/cloud-api.html
[10]https://youtu.be/MM3W3FS-W8Q
[11]https://github.com/meefik/linuxdeploy

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

the number of nodes. To achieve statistically significant results, the experiments were conducted over several days with more than 1,000 iterations per setup.

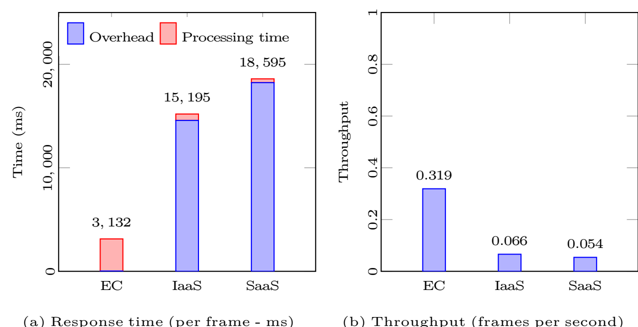


Fig. 8: Benchmarking results on traditional setups.

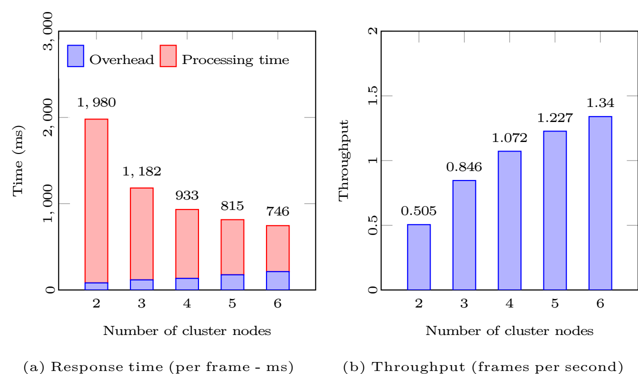


Fig. 9: Benchmarking results on a CEC edge cluster.

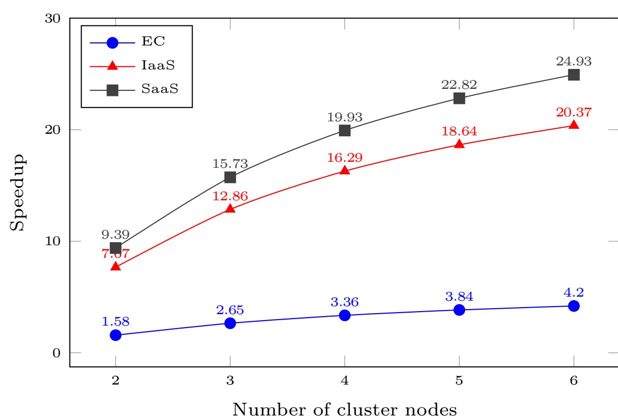


Fig. 10: Speedup of the CEC setup parallel processing vs EC, IaaS and SaaS stand-alone processing.

The experimental results for all four setups are summarized in Fig. 8 (traditional setups) and Fig. 9 (edge cluster). The 95% confidence interval is negligible due to the high number of experiments (>1,000) and thus is not shown in the figures. Fig. 8a depicts average time delay for the traditional setups. As it follows from the chart, running license plate recognition in a stand-alone mode on a single node (i.e. dashcam) takes 3,132 ms with almost no overheads. On contrary, in the vertical Cloud-enabled setups, the main delay is caused by the image transfers, whereas the actual image processing is relatively fast due to the excessive hardware resources of the Cloud. Fig. 8b presents the same results in terms of throughput. Admittedly, the Cloud-enabled setups fail to provide continuous support for run-time license plate recognition, whereas the stand-alone OpenALPR node can process 0.319 frames per second.

Fig. 9a refers to CEC and illustrates how the performance improves as more nodes join the cluster. That is, starting from two nodes in the cluster (the dashcam and one smartphone), where the average response time for a single frame is 1,980 ms, the cluster grows up to 6 nodes (e.g. more passengers get in the car and contribute to the edge cluster with their devices) that are able to process incoming images in parallel with a response time of 746 ms (533 ms for processing and 213 ms of overhead) per frame. Please note the slight increase in the overhead (due to more intensive data serialization, scheduling, and network transferring requirements), as the number of cluster nodes grows. Fig. 9b refers to throughput and suggests that within a fully-loaded car (4 passengers and the driver), the pooled resources of 6 edge devices are enough to process 1.34 frames per second – a sufficiently high rate for run-time license plate recognition. A lower number of devices could be still acceptable (∼1-1.227 frames per second for 4-5 nodes), while Cloud-based solutions, with delays higher than 15-18 seconds, cannot be considered for run-time license plate recognition.

By looking at the histogram charts, it becomes clear that local processing (either in a stand-alone local mode or in a cluster) is already able to outperform the Cloud-enabled architectures by avoiding the congested network communication and the related overheads. Admittedly, the results refer to this specific license plate recognition task and the corresponding experimental setup (i.e. image size, sampling frequency, available cluster nodes, network bandwidth, etc.). Nevertheless, it is expected that for similar data-intensive tasks (suitable for the proposed ECStream Processing) there will be a threshold number of nodes to share the workload horizontally, sufficient to substitute the remote vertical offloading. The increase in performance is best depicted by the speedup graph in Fig. 10, which compares the three stand-alone setups against the edge cluster composed of 2-6 nodes.

C. Threats to Validity and Discussion

In the conducted experiments, smartphones fully contribute their available computing resources to the cluster, whereas in reality they are expected to be running some user applications and related background jobs. Potentially, the minimum share of contributed resources can also be defined as a non-functional requirement, such that, for example, devices not able to guarantee at least 50% of their hardware capacities are not selected. This can also apply to battery charge – e.g. devices with insufficient charge levels are not allowed (although in the presented in-vehicle scenario, there is a possibility to charge a device). Given the finite bandwidth of the wireless network, the increased number of cluster nodes and associated inter-node data exchange may potentially lead to saturation of

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

the network, as well as to quickly drain the device battery. Albeit beyond the scope of this paper, these issues will need to be explored in the future, potentially applying intelligent estimation techniques, as proposed in [35].

It is also worth benchmarking the clusterization process as well, to provide a fair overview of the viability of the presented solution. As it was explained, the current implementation of node discovery and selection is based on broadcast network scanning, which makes this process relatively fast (i.e. up to 3 seconds to scan up to 256 LAN addresses, collect acknowledgments, and reconfigure device settings accordingly). The performance drops, however, with restarting the devices – that is, after each node has overwritten its cluster settings, it is required to reboot in order for the new configuration to take place. This process might take up to 1 min (depending on the number of cluster nodes and deployed NiFi processors). Same applies to a situation, when a node joins an already running cluster – i.e. having received new cluster settings, it needs to update its configuration taking up to 1 minute. This lack of support for 'hot deployment' is seen as a limitation of the current version of Apache NiFi, albeit this feature is already proposed to be included in one of the future releases. The clusterization process is anyway a one-off process that is not expected to affect the system performance in the long run. Furthermore, the clusterization overheads are comparable to or even lower than the ones of the Cloud setups, where only the time required to launch a virtual machine, depending on multiple criteria, typically takes at least 50 seconds [36].

At last, speaking of the scalability of the proposed solution, the experiments do not yet demonstrate significant results, and this aspect needs to be further investigated. However, the main goal of the proposed edge clustering is to improve the performance by reducing network latency. As shown in Fig 9a, the overhead increases proportionally to the number of cluster nodes. By interpolating these values, it is assumed that an edge cluster of about 100 nodes should have an overhead similar to the Cloud ($\sim$15 sec). It is also necessary to consider the overhead to manage the cluster for a resource-constrained edge device acting as the *Coordinator*, as well as security issues, all increasing linearly with the number of nodes. For these reasons, the proposed approach is suitable for clustering and managing few nodes in the neighborhood, in the order of tens, thus minimizing overhead and security issues. In the case of more complex computational task to offload, it is recommended to resort to Fog and Cloud computing. In this light, the scale of the above experiments can be considered appropriate to show feasibility and effectiveness of ECStream Processing.

IX. Conclusion and Future Work

This paper presented a novel approach to perform data processing at the very edge of IoT. As opposed to the established practice to offload computation to a Cloud in a vertical manner, the proposed approach relies on enabling local clusters of edge devices on top of the NiFi Stream Processing middleware. This way, edge devices, belonging to the cluster, are able to spread workload among themselves – that is, implement a horizontal offloading pattern – and minimize the amount of data sent over potentially congested network. As demonstrated by the proof-of-concept implementation and benchmarking experiments, the proposed approach outperforms Cloud-centric setups. This way, traditional on-board video recording systems can be turned into online video analytics platforms to support a wide range of situation assessment scenarios in urban environments. These might range from simple object detection of vehicles and people to more sophisticated tracking of subjects and reaction to critical situations.

Along with the generally positive results demonstrated by the prototype implementation, there are some potential enhancements to be taken into account and addressed as part of future work. The histogram charts in Fig.s 8 and 9 suggest an interesting and challenging problem of generalizing the experimental observations across a wider scope of processing tasks to identify an optimal configuration for a specific task at hand. That is, there are expected to be a *minimum* and a *maximum* number of cluster nodes that will underpin a balanced clustered architecture. The minimum number of nodes justifies establishing a cluster that will outperform the remote offloading, whereas the maximum number ensures that no unnecessary/redundant nodes are added to the cluster. This allows to plan and adjust the edge cluster to the problem at hand – e.g. as shown in Fig. 9b, it is possible to tune the cluster throughput and achieve the required video frame rate by adding new nodes accordingly (e.g. 5 nodes for processing 1.2 frames per second).

M2M operation via Bluetooth, LoRa, Zigbee, or some other Personal Area Network (PAN)/Ad-Hoc (MANET) related protocols also deserve to be investigated. Such protocols are mainly based on Master-Slave role profiles (i.e. one-to-one links), thus not immediately ready to support the zero-master, many-to-many cluster topologies (including NiFi). Indeed, any such solution should aim to establish a (full) mesh to enable the cluster middleware to work as intended, albeit with unavoidable configuration overheads. Moreover, the constrained bandwidth of existing wireless technologies also limits their potential utilization for data-intensive scenarios. As discussed above, network discovery is also worth to be investigated in this respect.

Another relevant aspect is security and privacy. Edge clustering could be a way of securing computational task offloading by enforcing security properties during node selection by, for example, filtering remote or not trusted nodes. Such policies should be properly designed and evaluated, thus calling for specific techniques and tools.

References

[1] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, A. Puliafito, and R. Buyya, "Metropolitan intelligent surveillance systems for urban areas by harnessing iot and edge computing paradigms," *Software: Practice and Experience*, vol. 48, no. 8, pp. 1475–1492, 2018.

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2983402, IEEE Transactions on Cloud Computing

[2] M. Jang, M.-S. Park, and S. C. Shah, "A mobile ad hoc cloud for automated video surveillance system," in *2017 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2017, pp. 1001–1005.

[3] S. Yang, "IoT Stream Processing and Analytics in the Fog," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 21–27, 2017.

[4] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, 2017.

[5] F. Haider, D. Zhang, M. St-Hilaire, and C. Makaya, "On the planning and design problem of fog computing networks," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2018.

[6] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Comp.*, vol. 14, no. 2, pp. 24–31, 2015.

[7] R. Vilalta, A. Mayoral, D. Pubill, R. Casellas, R. Martínez, J. Serra, C. Verikoukis, and R. Muñoz, "End-to-End SDN orchestration of IoT services using an SDN/NFV-enabled edge node," in *Optical Fiber Comm. Conf.* IEEE, 2016, pp. 1–3.

[8] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource provisioning for IoT services in the fog," in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2016, pp. 32–39.

[9] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose, "VideoEdge: Processing camera streams using hierarchical clusters," in *2018 IEEE/ACM Symposium on Edge Computing*. IEEE, 2018, pp. 115–131.

[10] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *ACM Int. Conf. on Distributed and Event-based Systems*, 2016, pp. 69–80.

[11] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things: A vision, architectural elements, and future directions," *Future Gen. Comp. Sys.*, vol. 29, no. 7, pp. 1645–1660, 2013.

[12] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Pushing Intelligence to the Edge with a Stream Processing Architecture," in *The 2017 IEEE International Conference on Internet of Things (iThings 2017)*. IEEE, 2017.

[13] R. Roman, J. Lopez, and M. Mambo, "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges," *Future Generation Computer Systems*, 2016.

[14] M. R. Rahimi, J. Ren, C. H. Liu, A. V. Vasilakos, and N. Venkatasubramanian, "Mobile cloud computing: A survey, state of art and future directions," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 133–143, 2014.

[15] A. Manzalini and N. Crespi, "An edge operating system enabling anything-as-a-service," *IEEE Comm. Mag.*, vol. 54, no. 3, pp. 62–67, 2016.

[16] N. Fernando, S. W. Loke, and W. Rahayu, "Computing with nearby mobile devices: A work sharing algorithm for mobile edge-clouds," *IEEE Transactions on Cloud Computing*, vol. 7, no. 2, pp. 329–343, 2019.

[17] M. Chen, Y. Hao, Y. Li, C.-F. Lai, and D. Wu, "On the computation offloading at ad hoc cloudlet: architecture and service modes," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 18–24, 2015.

[18] H. Guo and J. Liu, "Collaborative Computation Offloading for Multiaccess Edge Computing Over Fiber–Wireless Networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 5, pp. 4514–4526, 2018.

[19] R. Dautov, S. Distefano, and R. Buyya, "Hierarchical data fusion for smart healthcare," *Journal of Big Data*, vol. 6, no. 19, 2019.

[20] C. Zhu, H. Wang, X. Liu, L. Shu, L. T. Yang, and V. C. M. Leung, "A novel sensory data processing framework to integrate sensor networks with mobile cloud," *IEEE Systems Journal*, vol. 10, no. 3, pp. 1125–1136, 2016.

[21] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[22] M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Comm.*, vol. 8, no. 4, pp. 10–17, 2001.

[23] M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Dustdar, O. Scekic, T. Rausch, S. Nastic, S. Ristov, and T. Fahringer, "A deviceless edge computing approach for streaming IoT applications," *IEEE Int. Comp.*, vol. 23, no. 1, pp. 37–45, 2019.

[24] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A serverless real-time data analytics platform for edge computing," *IEEE Int. Comp.*, vol. 21, no. 4, pp. 64–71, 2017.

[25] G. Merlino, D. Bruneo, F. Longo, S. Distefano, and A. Puliafito, "Cloud-Based Network Virtualization: An IoT Use Case," in *Int. Conf. on Ad Hoc Net.* Springer, 2015, pp. 199–210.

[26] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4Things: a sensing-and-actuation-as-a-service framework for IoT and cloud integration," *Ann. Telecomm.*, vol. 72, pp. 53–70, 2017.

[27] D. J. Lilja, "Experiments with a Task Partitioning Model for Heterogeneous Computing," in *Proceedings of the Workshop on Heterogeneous Processing*. IEEE, 1993, pp. 29–35.

[28] U. Catalyurek and C. Aykanat, "A hypergraph-partitioning approach for coarse-grain decomposition," in *The 2001 ACM/IEEE Conference on Supercomputing*. ACM, 2001, pp. 28–28.

[29] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Data processing in cyber-physical-social systems through edge computing," *IEEE Access*, vol. 6, pp. 29 822–29 835, 2018.

[30] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[31] R. Dautov and S. Distefano, "Automating IoT Data-Intensive Application Allocation in Clustered Edge Computing," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2019.

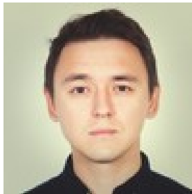
[32] F. Lin, Y. Zhou, X. An, I. You, and K. R. Choo, "Fair resource allocation in an intrusion-detection system for edge computing: Ensuring the security of internet of things devices," *IEEE Consumer Electronics Magazine*, vol. 7, no. 6, pp. 45–50, 2018.

[33] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2009.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[35] D. Trihinas, G. Pallis, and M. D. Dikaiakos, "ADMin: Adaptive monitoring dissemination for the internet of things," in *2017 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2017, pp. 1–9.

[36] M. Mao and M. Humphrey, "A performance study on the VM startup time in the cloud," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 423–430.



Rustem Dautov holds a PhD in Computer Science from the University of Sheffield, UK. He is a Research Scientist at SINTEF, Norway, where he is involved in several R&D projects at the European and national levels. He has previously been a Postdoctoral Researcher and a Lecturer in IoT at Kazan Federal University, Russia, and a Marie Curie Fellow at SEERC, Greece. His research focuses on software engineering for IoT, Edge and Cloud Computing.

Salvatore Distefano is an Associate Professor at the University of Messina, Italy and a Fellow Professor at Kazan Federal University, Russia. His research interests include Cloud, Fog, Edge computing, IoT, crowd-sourcing, Big Data, software and service engineering, performance and reliability evaluation and QoS. He is involved in several national and international projects. He is a member of international conference committees and journal editorial boards such as IEEE Trans. on Dependable and Secure Computing. He has also co-founded the SmartMe.io startup.

2168-7161 (c) 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF BIRMINGHAM. Downloaded on April 17,2020 at 04:01:34 UTC from IEEE Xplore. Restrictions apply.