Flexible Distributed Computing Across End-Devices, the Edge and the Cloud

Alexandros Patras, Spyros Lalis, and Christos D. Antonopoulos

Electrical and Computer Engineering Department, University of Thessaly

e-mail:{patras,lalis,cda}@uth.gr

Abstract

The emergence of the edge/fog computing paradigm has increased the programming complexity of applications so that they can work seamlessly in the new distributed and heterogeneous system landscape. In this paper, we investigate a structured dataflow approach which simplifies application development and offers great flexibility regarding the deployment of the application across end-devices, edge computing infrastructure and remote cloud systems. Our prototype is built on top of the Node-RED framework, with extensions in order to support the transparent deployment and distributed execution of application flows. We use a real-world application example to illustrate our approach as well as to explore the performance trade-offs for different deployment scenarios on a real distributed computing setup.

1 Introduction

A large number of IoT and pervasive computing application scenarios revolve around a rather simple architectural approach, whereby low-end and/or mobile devices send data to and receive actuation/control requests from powerful server machines in the cloud. This approach, though straightforward to implement, has several drawbacks. Firstly, a large amount of low-level data are routed over the Internet to remote machines, leading to scalability issues. Secondly, due to the latency of the Internet, it may not be possible to support control/feedback loops with tight real-time constraints. Last but not least, privacy-sensitive data ends-up in the cloud from where it may leak to third parties, either intentionally for business purposes or unintentionally as a result of attacks.

An alternative approach is to adopt a more complex system architecture, which allows part of the data processing and decision making to be performed at the edge, on machines close to the end-devices, or in part even directly on the end-devices themselves. However, writing applications that span across end-devices, edge computing infrastructure and remote cloud systems is a non-trivial task: the developer has to structure the application in different parts; each part must be written/prepped so that it can run on the target host; the interface between the different parts of the application has to be cleanly defined and implemented so that it can be performed over a network; finally, each part must be installed on its host, and be properly instantiated and linked together with other parts of the application. And this process has to be repeated, in the worst case from scratch, if one wishes to apply / experiment with a different deployment.

In this paper we present work done to support the programmer in the above task, by adopting a combination of component-based and dataflow-oriented programming. More specifically, we let an application be expressed as a graph, where the nodes represent components and the edges between them represent unidirectional links used for data exchange. The developer also provides hints on how components should be placed on the hosts of the system. At deployment time, the sub-graphs are instantiated on the target hosts, along with automatically generated connector logic that takes care of inter-component binding and communication over the network. To accelerate prototyping, we have built our support on top of Node-RED [1], which provides several features that are in line with our vision. Our contribution is in the extensions that support transparent component deployment and inter-component binding and communication over the network, without touching the Node-RED core. Moreover, we perform experiments on a real distributed setup and provide performance results that show the benefits and trade-offs of a more flexible application deployment that exploits computing resources at the edge.

The rest of the paper is structured as follows. Section 2 describes an indicative application example, where the edge computing paradigm is particularly appropriate. Section 3 gives an overview of the Node-RED framework. Section 4 presents the extensions made to Node-RED to enable a more flexible application deployment and execution, while Section 5 discusses the results of our performance experiments. Section 6 outlines related work. Finally, Section 7 concludes the paper.

2 Application Example: Camera-based Security

Several applications have been identified in the literature as typical examples where edge/fog computing can be advantageous compared to conventional cloud computing. Here we focus on one such application, which is based on the Airport Visual Security System case study that is described in the OpenFog Reference Architecture [2]. For the purpose of our work we have simplified and adapted the application while keeping its most salient features and performance-critical aspects.

In a nutshell, the purpose of the application is to monitor a security-sensitive area by processing camera feeds in order (a) to recognize movement and (b) unwanted persons entering the area, and issue respective notifications to nearby personnel or even take an automated action, such as raising an alarm or locking a door. As shown in Figure 1, the application can be partitioned in several high-level functions: (F1) take video of the area of interest; (F2) detect motion in the video frames; (F3) detect individual faces and crop the respective frame areas; (F4) match the cropped images against the images of different persons stored in a database; (F5) raise soft/hard alarms in case movement or an unwanted person is detected. Function F1 is achieved using camera sensors, F2, F3 and F4 are implemented using suitable image processing and face recognition

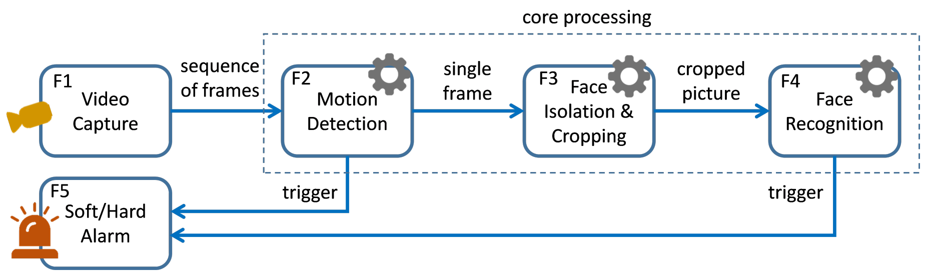

Figure 1: High-level functional diagram for the camera-based security application.

algorithms, while F5 will typically trigger an actuator or user interface on some end-device.

This application can be implemented using a low-cost local infrastructure, deploying only the video cameras and alarms in the area of interest. The core image processing and face recognition functions can be performed by employing ready-to-use services offered by cloud providers like [3, 4, 5], or by running own/custom computer vision software such as [6, 7] on powerful server machines in the cloud. While this approach has the well-known advantages of cloud computing, it also comes with important scalability and responsiveness issues. Firstly, the available network bandwidth may not suffice to push a large number of video streams all the way to the cloud. Secondly, the latency of the Internet and the relaxed quality of service guarantees of cloud systems may introduce large end-to-end actuation and user notification delays.

As an alternative approach, the core processing functions of the application can be implemented in a modular fashion, as independent components that can run, if desired, on different machines —some of them located at the edge of the Internet— leading to better performance and scalability. In this particular case, while F2 takes as input a continuous stream of frames which can arrive at a high rate, it only outputs selected frames when motion is detected. Also note that F3 and F4 are activated only when F2 actually produces some output. Therefore placing F2 close to F1, for instance on a machine with a fast wired/wireless connection to the camera devices, which may also feature special hardware acceleration, might be a better choice in terms of both scalability and responsiveness of the application. To further reduce bandwidth requirements, F3 could be at the edge as well. On the other hand, if the network latency is sufficiently small, F4 could be hosted in the cloud on machines with lots of storage and powerful database support.

3 Node-RED Overview

Node-RED [1] is built on top of NodeJS [8], with the purpose of enabling rapid and simple development of IoT applications. It consists of (i) a collection of runnable software components called *nodes*, (ii) a graphical user interface (GUI) for picking nodes and linking them together into a flow graph, and (iii) a runtime environment for deploying and executing such flows. The collection of nodes can be enriched in an open-ended manner, and developers are free to write their own nodes that implement missing functions or custom glue-logic needed for the target applications. A large number of nodes embodying a wide range of different functions have already been contributed by the community, making Node-RED a very powerful component ecosystem.

The GUI allows the developer to interactively browse the list of available nodes, also referred to as *node pallete*, choose the ones to be used in the application, drag and drop them on the so-called *canvas*, and link them together into a flow that achieves the desired functionality. The linking between nodes is done by drawing a line that connects an output port of the source node with the input port of the destination node. When the application flow graph is finalized, it can be deployed by pressing a button. To give a concrete example, Figure 2 shows a Node-RED flow for the camera-based security application, following the component structure of Figure 1.

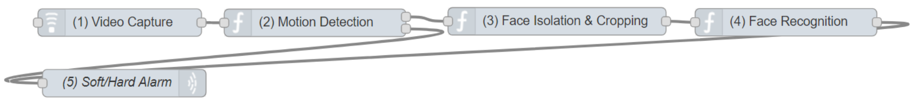

Figure 2: Node-RED flow graph for the camera-based security application.

Node components are written in JavaScript. According to the conventions of the Node-RED framework, nodes can have at most one input port and zero or more output ports; they may also communicate through shared objects stored in a global or flow-specific context. The actual node logic resides in a handler function, which is invoked by the Node-RED runtime environment when a message arrives at the input port in order to processes it and produce messages for the output ports. Apart from simple functions, nodes can implement more complex data processing directly in JavaScript or by proxying other programs that are invoked through suitable middleware. One may also exploit the node-gyp [9] tool of NodeJS to create bindings between external libraries and nodes, and the NodeJS Package Manager to install contributed modules. Meta-information on the functionality, input/output ports and configuration options of a node is provided in a separate HTML file, which is accessed by the GUI in order to guide the user during the interactive node selection, linking and configuration process.

Finally, the Node-RED runtime environment takes as input a flow graph and deploys it on the local machine. More specifically, it creates an instance for every node in the graph, and routes messages between the nodes as dictated via the respective links.

4 Extending Node-RED for Flexible Distributed Computing

Node-RED is intended primarily for running application logic on the same machine, and the runtime environment takes care of component instantiation and message forwarding at a local scope only. While it is possible for an application to exploit functions running on remote machines, the respective software management and communication becomes the responsibility of the developer, and has to be done in a manual way, separately from the application flow that is built/run using Node-RED.

We have extended Node-RED so that it can be used to build applications that can span across machine boundaries in a transparent way. The main difference compared to regular Node-RED flows is that the developer has to specify the placement of nodes on hosts according to the desired deployment. Doing this is trivial, and one can explore different deployments scenarios with practically zero effort. For our prototyping purposes, we leave the core of the Node-RED framework untouched, and introduce our extensions on top of it while exploiting the out-of-the-box functionality as much as possible.

For the orchestration of the system we adopt a centralized approach whereby a distinguished machine, called the *master*, acts as the application manager and coordinator for the distributed execution environment. The machines that can be used to host/run one or more application nodes, called *slaves*, run the conventional Node-RED environment. Slaves register with the master using a simple directory protocol, and accept requests for configuring and executing flows locally. This master-slave interaction is performed using the HTTP management API of the Node-RED framework. Figure 3 shows the high-level system architecture.

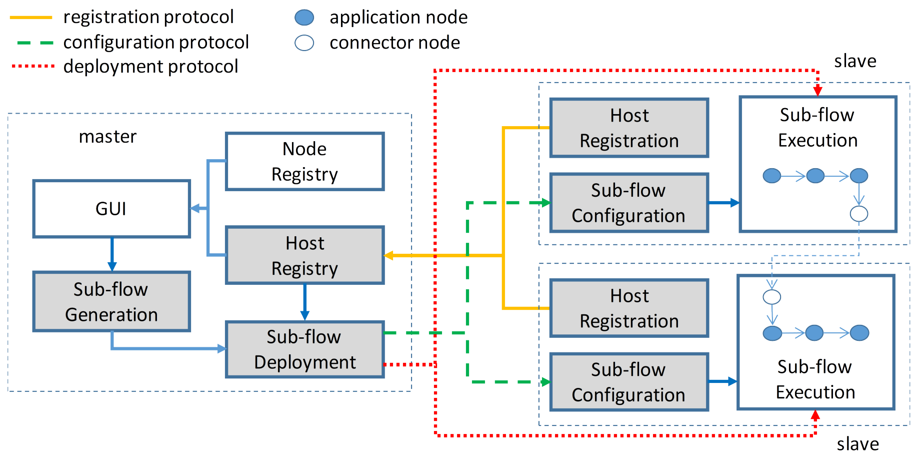

Figure 3: High-level view of the system architecture. Our extensions to the Node-RED framework are marked in grey. The protocol for the deployment of node flows is already supported by Node-RED.

The developer builds the application flow on the master environment, by dragging-dropping nodes on the canvas and linking them together as usual. The user can find the machines that are available for hosting nodes and inspect their properties by interactively browsing the master directory. The placement of the nodes on hosts is specified via a syntactical convention, namely by appending the host identifier as a suffix in the name of the node. For example, if node *VideoCapture* should run on host *CamDev*, the full name of the node component should be *VideoCapture_CamDev*. It is thus easy to define and change node placement, by setting these suffixes accordingly. Once the application flow is defined, the user can deploy and run it by pressing a button of the GUI.

Based on the node placement information, the master splits the global application flow into sub-flows each comprising only nodes that should be co-located on the same host. For each link that spans across hosts, additional connector logic is generated, which comprises a communication endpoint and suitable data serialization for each side. This is implemented via proper nodes, re-using the networking and serialization support of Node-RED (plus application-specific serialization add-ons for custom data types, if any), which are then linked to the corresponding application-level nodes in the respective local sub-flows. Finally, the master describes each sub-flow independently, in the form of a JSON file that follows the standard Node-RED flow specification format, and sends it for execution to the respective host. The allocation of the UDP/TCP ports for the communication endpoints on the hosts is done by the master via a suitable configuration protocol, before activating the individual sub-flows. The entire process, including the generation of the *connector* logic and required nodes installation, is automated and does not require any involvement of the application developer.

5 Performance Experiments

We have used our prototype to investigate the performance of different deployments for the camera-based security application described in Section 2. This is structured as a Node-RED flow as already shown in Figure 2. Function F1 is implemented as a node that reads a sequence of video frames that are stored in JPEG format, simulating a video stream. The individual frames are transmitted "as is" without employing any compression technique. The node for F2 is based on existing software that compares subsequent frames [10]. If the difference between them is above a threshold, the node outputs the last frame and a corresponding notification towards F5. F3 runs a face detection algorithm that uses Haar Feature-based Cascade Classifiers on the input frames and produces cropped frames that contains a face in grayscale color. F4 is implemented using the Local Binary Patterns Histograms face recognition algorithm on the incoming cropped frames, generating a notification in case a positive match is found. These algorithms are already implemented as part of the OpenCV library [6], and are accessed via respective NodeJS bindings [11]. Finally, F5 is a simple node that consumes the notifications coming from F2/F4 and prints a warning.

We use a small part of the ChokePoint [12] video dataset as input, with a duration of 22 seconds: the first 5.5 seconds show an empty corridor without any movement; during the next 5.5 seconds a person passes through the corridor facing the camera, but his face is not in the face database; the following 5.5 seconds are identical to the first phase; the last 5.5 seconds show another person whose face is included in the face database. The original video dataset has an image resolution of 800x600 pixels at 30 frames per second. To let each frame fit into a single UDP/IP packet and avoid frame fragmentation and reassembly at the application level, the video was cropped to 525x500 pixels. Also, due to limitations in the upload bandwidth of our edge setup, the frame rate was artificially reduced to 1.25 frames per second. The cropped frames produced by F3 vary in size. We train the face recognition model using the Labeled Faces in the Wild face database that contains faces of 5,749 people in 13,233 images [13].

In a first run that is performed on a single machine we record the amount of data that would travel over the network if the application components were deployed on different hosts. This is done by adding serialization and de-serialization nodes for each link between two components, along with the corresponding UDP-out and UDP-in nodes. We measure the size of application-level messages as well as the traffic at the level of UDP/IP traffic using the *iptraf* Linux tool [14]. Table 1 presents the results. We observe that the placement of application components significantly affects the amount of data that travels over the Internet. More specifically, placing F2 at the edge rather than on a remote cloud reduces the number of frames and total amount of data sent over the Internet by 60%. If F3 is placed at the edge as well, the outbound traffic drops by more than 95% (fewer frames, which are also much smaller in size due to face cropping).

Table 1: Data traffic between the components of the camera-based security application. The bandwidth is estimated for the frame rate of the original video footage without employing compression.

	Application	**UDP/IP**	**Bandwidth @ 30 fps**
F1 -> F2	27.5 MB	55 MB	20 Mbps
F2 -> F3	10.3 MB	20.6 MB	3.7 Mbps
F3 -> F4	0.5 MB	1.1 MB	0.2 Mbps

In a real-world setting there could be numerous video streams that need to be processed concurrently, leading to very high bandwidth requirements even if compression is employed. For instance, a H.264 or H.265 IP camera generates 12 Mbps at 30 fps [2]. To support even just 10 cameras with a pure cloud-based approach one would already require more than 100 Mbps of bandwidth. According to a recently published Akamai report on the state of the Internet, the global average peak speed of Internet connectivity stands at 44.6 Mbps at an average speed of just 7.2 Mbps [15]. Clearly, to support such an application at scale, one would have to run F2 and probably also F3 at the edge.

In a second series of experiments we run the application for four different deployments. The platforms used to host the application components and the deployment scenarios are summarized in Table 2. In all cases, the nodes for F1 and F5 are co-located on a laptop computer that represents an end-device. The placement of each of the nodes for F2, F3 and F4 depends on the deployment scenario. They run either on a desktop that stands for the edge computing infrastructure (in the same LAN as the laptop) or on a remote cloud system. Our main goal is to investigate the delay for the notifications issued by F2 and F4 purely as a function of deployment. To avoid any side-effects due to heterogeneity in terms of processing capacity, we use hosts of similar processing characteristics at both the edge and the cloud. Also, application nodes run within a single thread in Node-RED, and do not take advantage of any multi-core system capability.

We measure the end-to-end delay for the motion detection notification issued by F2 and the face recognition notification issued by F4. Both delays are recorded at

Table 2: Host platform characteristics and deployment scenarios.

	End-Device	**Edge Computer**	**Remote Cloud**
Platform	Linux Atom CPU, 2C@1.66 Ghz 512 KB Cache, 1 GB RAM	Linux Image in VM I7 CPU, 4C@4.2 Ghz 8 MB Cache, 8 GB RAM	Linux via Docker I7 CPU, 8C@3.4 Ghz 8 MB Cache, 32 GB RAM
Cloud	F1, F5	none	F2, F3, F4
Edge/Cloud 1	F1, F5	F2	F3, F4
Edge/Cloud 2	F1, F5	F2, F3	F4
Edge	F1, F5	F2, F3, F4	none

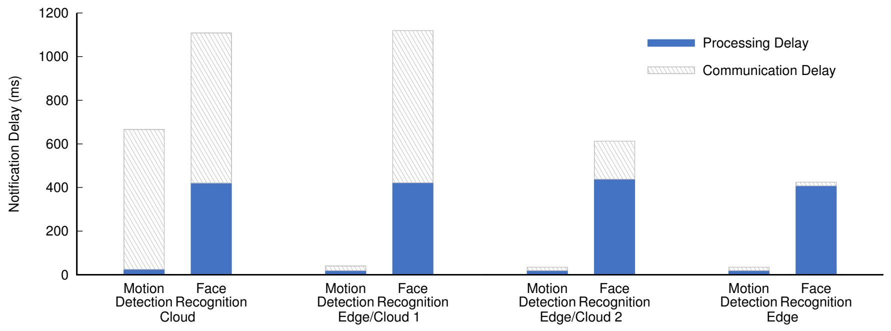

Figure 4: Notification delay for motion detection and face recognition.

F5, when these notification messages arrive, using as a reference the time when F1 outputs frames with motion and frames with a person whose picture is in the face database, respectively (we know the sequence numbers of those frames through manual inspection). To estimate how much of the delay is due to processing, we record the time it takes for each component (including the serialization/deserialization components) to produce its output, and sum-up the processing times of all components along the pipeline that is relevant in each case (F2 for the motion detection delay; F2 + F3 + F4 for the face recognition delay).

Figure 4 shows the results (median over ten runs for each deployment). The whole bars show the notification delays. The solid part of each bar denotes the aggregated processing delay, while the striped part shows the rest of the delay, due to the latency of the communication that takes places between the application components.

The notification for motion detection is much faster in all cases where F2 is hosted at the edge (Edge and both Edge/Cloud variants) vs. on the remote cloud system (Cloud). This is expected, because in all deployments F2 is hosted on equally fast machines, but in the Cloud deployment it is further away from the end-device where F5 resides.

However, the results for the face recognition notification delay are not that intuitive. The Edge deployment has the lowest delay, as expected. But the face recognition notification delay for Edge/Cloud 1 is slightly larger than that of the Cloud deployment, even though F2 runs at the edge on a host machine that is as powerful as the cloud host. This is due to the extra overhead of passing the data flow through the host at the edge before going to the cloud. While F2 reduces the number of frames that travel upstream towards F3, the amount of processing done by F2 (just 5 milliseconds per frame, in all configurations) is too small to outweigh the time it takes to "relay" a frame via the application layer (about 7 ms). In contrast, this overhead is not visible in the Edge/Cloud 2 deployment because of the heavier aggregated processing of F2 and F3 on the edge host (a total of 55 ms per frame), but also due to the very significant reduction by F3 in the number and size of frames that flow out of the edge host to the cloud, leading to a lower face notification delay compared to the Cloud deployment. This shows that performance only improves by placing the *right part* of the application at the edge.

6 Related Work

Another effort to support distributed computing based on Node-RED is Distributed Node-RED (DNR) [16]. The ideas behind DNR are described in [17], but the system was released just recently, at the same time when we developed our own support. An important difference is that in order to support more flexible linking patterns the communication between nodes running on different hosts revolves around a pub/sub scheme via MQTT [18], with the master node acting as the broker. Given the results of our experiments, we believe that this indirection is likely to be too costly in terms of latency and perhaps even sheer throughput for streaming applications like the one discussed here. DNR also changes the core of the Node-RED framework, whereas our functionality is introduced via extensions that leave the core untouched. We could not find any information about experimental deployments and performance measurements for DNR.

Flogo [19] is a recent framework that is similar to Node-RED, allowing the developer to build an application flow through a graphical interface. The main difference is that the application components are written in the Go programming language. Also, the programmer has to take care of the deployment of the components on the respective machines, but it is easy to do the wiring via the GUI. The creators of Flogo claim that it is more lightweight and faster compared to other technologies like Node-RED.

Several frameworks support programming with data streams and dataflow graphs, like Apache Storm [20], Apache NiFi [21] and Cask [22]. These are mainly visualization and management tools intended for data analysis and aggregation from many different data sources. Kura [23] and Fiware [24] are centralized IoT middleware frameworks. They offer a high-level API which hides the communication between the application running on the server and the end-devices that run suitable adapters/gateways. However, there is no support for distributing the application logic across different hosts.

TensorFlow [25] adopts a distributed dataflow computing model, but it is primarily geared towards supporting the time-consuming training of neural networks. The developer writes the application using a custom programming notation, while a graphical representation is used to visualize (rather than specify) the program structure in a user-friendly way. However, the application can have a large number of primitive function/operator components, as opposed to the much coarser software components we target in our work. TensorFlow also enables the transparent exploitation of heterogeneous computing resources (GPUs) via abstract functions and corresponding runtime support.

Yet another way for developing applications that can be distributed in a flexible way is to compose them out of microservices [26]. Individual microservices could then be grouped into larger clusters and be deployed on remote hosts through a suitable container system like Docker [27] and deployment system like Kubernetes [28]. An issue that would have to be addressed in this case is how to link together microservices without going through some centralized glue logic or broker as in [29, 30], for instance by using a decentralized message bus [31]. In our work inter-component links are implemented without any intermediate broker, through direct UDP/IP or TCP/IP transport channels. There are also proposals for new programming languages that ease the development of microservices and their communication via a custom runtime environment [32]. We use the well known JavaScript language on a fully supported runtime.

7 Conclusion and Future Work

We have extended the Node-RED framework in order to enable the seamless distributed placement and execution of application flows on different hosts, making it possible to exploit computing resources at the edge and remote cloud systems in a flexible way. We have also presented experimental results using a realistic application, illustrating the performance trade-offs for different deployment scenarios.

In the future we wish to investigate the introduction of a logical layer via abstract components, in order to support even more flexible deployment scenarios, such as the exploitation of pre-installed components, the sharing of components among different application flows, and the transparent scaling-out of components at different edges of the Internet. Another direction is to investigate deployment trade-offs for the case where the hosts at the edge are less powerful than the ones in the cloud.

Acknowledgements

This work was supported in part by the H2020 Framework Program of the European Union through the UniServer Project (Grant Agreement 688540) http://www.uniserver2020.eu.

References

[1] Node-RED, http://nodered.org.

[2] OpenFog Reference Architecture for Fog Computing, *OpenFog Consortium Architecture Working Group, OpenFog Consortium*, 2017.

[3] Microsoft Azure Cognitive Services, https://azure.microsoft.com/en-us/services/cognitive-services.

[4] Google Cloud Vision API, https://cloud.google.com/vision.

[5] IBM Bluemix Visual Recogntion API, https://www.ibm.com/watson/developercloud/visual-recognition.html.

[6] I. Culjak, D. Abram, T. Pribanic, H. Dzapo and M. Cifrek. A brief introduction to OpenCV, *Proceedings of the 35th International Convention MIPRO*, pages 1725-1730, 2012.

[7] D. E. King. Dlib-ml: A Machine Learning Toolkit, *Journal of Machine Learning Research 10*, pages 1755-1758, 2012.

[8] NodeJS runtime, https://nodejs.org.

[9] Node-gyp tool, https://github.com/nodejs/node-gyp.

[10] Motion Detection for NodeJS using OpenCV, https://github.com/Daan-Grashoff/Motion-detection-node-opencv.

[11] NodeJS OpenCV Bindings, https://github.com/peterbraden/node-opencv.

[12] Y. Wong, S. Chen, S. Mau, C. Sanderson, B.C. Lovell. Patch-based Probabilistic Image Quality Assessment for Face Selection and Improved Video-based Face Recognition IEEE Biometrics Workshop, *Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 81 - 88. IEEE, 2011.

[13] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments, *Technical Report, University of Massachusetts, Amherst*, pages 07 - 49, 2007.

[14] IPTraf, http://iptraf.seul.org.

[15] Akamai State of the Internet Report, https://www.akamai.com/us/en/about/our-thinking/state-of-the-internet-report.

[16] Distributed Node-RED, https://github.com/namgk/dnr-editor.

[17] N. Giang, M. Blackstock, R. Lea, V. C. M. Leung. Developing IoT Applications in the Fog: a Distributed Dataflow Approach, *5th International Conference on the Internet of Things*, 2015.

[18] Message Queue Telemetry Transport (MQTT), http://mqtt.org.

[19] Project Flogo, http://www.flogo.io.

[20] Apache Storm, http://storm.apache.org.

[21] Apache NiFi, https://nifi.apache.org.

[22] Cask, http://cask.co.

[23] Kura, http://www.eclipse.org/kura.

[24] Fiware Connection to the Internet of Things, https://www.fiware.org.

[25] Google Brain Research Team. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265-283, 2016.

[26] K. Bakshi. Microservices-based software architecture and approaches, *IEEE Aerospace Conference, Big Sky, MT*, pages 1-8. IEEE, 2017.

[27] Docker, https://www.docker.com.

[28] David K. Rensin. Kubernetes - Scheduling the Future at Cloud Scale, *OSCON*, 2015.

[29] J. Innerbichler, S. Gonul, V. Damjanovic-Behrendt, B. Mandler and F. Strohmeier. NIMBLE collaborative platform: Microservice architectural approach to federated IoT, *Global Internet of Things Summit (GIoTS), Geneva*, pages 1-6, 2017.

[30] L. Sun, Y. Li and R. A. Memon. An open IoT framework based on microservices architecture, *China Communications, vol. 14, no. 2*, pages 154 - 162, 2017.

[31] P. Kookarinrat and Y. Temtanapat. Design and implementation of a decentralized message bus for microservices,*13th International Joint Conference on Computer Science and Software Engineering (JCSSE), Khon Kaen*, pages 1-6, 2016.

[32] C. Xu, H. Zhu, I. Bayley, D. Lightfoot, M. Green and P. Marshall. CAOPLE: A Programming Language for Microservices SaaS, *IEEE Symposium on Service-Oriented System Engineering (SOSE), Oxford*, pages 34 - 43. IEEE, 2016.