Configuring Edge Devices That Are Not Accessible Via The Internet

Sebastien Andreo
SI CTO EAAS
Siemens AG
Erlangen, Germany
Email: Sebastien.Andreo@siemens.com

Uwe Hohenstein
FT RPD SSP
Siemens AG
Munich, Germany
Email: Uwe.Hohenstein@siemens.com

***Abstract*—In the context of Industrial Internet of Things and Edge Computing, there are often computing devices that run software on the edge device. In real industrial settings, these computing devices are not accessible from the Internet. This raises a problem if software components on the device require configuration changes, such as adjusting some parameters to customize software, restarting one or several applications that behave badly on the device, or rotating passwords to name a few. This paper presents a novel approach to configure software on such computing devices. The details of the cloud-based approach are presented and how the approach has successfully been applied in an industrial project.**

Keywords-internet of things; IoT; cloud computing; industrial use case; configuration.

I. INTRODUCTION)

The Internet of Things (IoT) [9] is a rapidly emerging Internet-based information service architecture where many devices are interconnected [3][36]. Devices are equipped with communication, sensing, computing and actuating capabilities to collect and exchange data with their surroundings to enable analysis, optimization and control. Devices also perform a task in the physical world like opening or closing a valve. Recent technological achievements support the vision of a connected world [22]. In fact, different IoT use cases have successfully been implemented [31].

Many traditional IoT approaches use the Cloud for analyzing devices' data, i.e., devices forward data to the Cloud where intelligent analysis is performed because of the Cloud's high reliability, availability, unlimited scalability and resources. However, a lot of incarnations of IoT have increasingly challenged this approach [1] because transmitting large amounts of data to the Cloud burdens the network traffic. Recent approaches, such as edge [27], fog [6][38] and osmotic computing [33], thus target at processing and putting more data intelligence and decision making processes [16][18][37] at the edge of the network, i.e., near the devices [28][30]. Filtering and pre-processing data occurs before submission to the Cloud, thus decreasing the volume of data and reducing the network traffic [25].

In these cases, computing devices run some installed software on the hardware. However, in real industrial settings, devices are deployed in remote and hard-to-reach environments. As a consequence, devices do not allow any access from the Internet due to strong security requirements.

This raises a problem. Usually, software requires some configuration, which might change from time to time. For example, software is mostly designed in a generic manner to satisfy several customers or deployments. Thus, running the generic software needs some configuration during startup to customize software accordingly. Only then a customer-specific implementation can be avoided.

Changes of the configuration will also occur during operation. For example, transferring data to the Cloud in the context of edge devices requires credentials to access Cloud services, e.g., a Cloud storage, such as AWS S3 storage. This does not seem to be a problem at a first glance. But security policies in industrial contexts require an expiration and periodic password rotation. That is, renewed passwords must then be passed to the device at runtime in order to avoid downtime.

In general, further parameters for the software have often to be adjusted, e.g., thresholds, according to changing system behavior or environment. Similarly, if a software component behaves badly, restarting it might become necessary from the outside, maybe after changing some parameters to remedy a component. And finally, there are often scheduled jobs that require a modifiable Cron specification for periodic tasks.

Hence, a configuration of devices and their components is indispensable. But any kind of such configuration is a problem if the device and components running on that device are not accessible via the internet – as in the case of industrial settings.

One possible solution is to securely log into the computing device and to configure locally and directly. This requires advanced access from outside, e.g., by the Common Remote Service Platform [8] – if possible at all. Without a remote login, configuration must be done directly at the device's location, i.e., usually at the plant's site.

The main contribution of this paper is to tackle these configuration issues. Hence, Section II presents a novel approach to control and configure software on edge devices that are not accessible by the Internet. Details about the organization of software are presented in Section III, before Section IV applies and evaluates the approach in a real industrial context with corresponding use cases. In Section V, we compare the presented approach with other work done in the literature. Finally, Section VI summarizes the works and gives an outlook about future work.

II. APPROACH

In the context of Industrial Internet of Things and Edge Computing, there are computing devices that run some installed software on the hardware. This paper targets at allowing for externally provided configurations. In the following, we use the following terms:

- *Computing Device (CD)*: Can be any device, such as a computer, an industrial PC, industrial boxes, such as Siemens X300 box, or a RaspberryPi. A CD might also be part of embedded hardware.
- *Device Component (DC)*: A piece of software that runs in a computing device to fulfill a certain task within the computing device. In modern IoT architectures, these DCs typically run in virtualization, especially Docker containers.
- *Configuration*: Some data required by a device component DC to adjust its behavior, for instance, to set some thresholds, credentials, and time intervals, or to trigger actions, such as enforcing a restart.

The proposed solution to allow for an external configuration is as follows.

There is a new component *ConfigurationManager-Backend* (*CMB*) running on a separate computer (maybe hosted in the Cloud). The CMB keeps configurations and possesses a publicly available but secured service. Dedicated users can send configuration data to the CMB service for a Device Component on a CD by means of an API; the CMB persists the configuration data internally. Furthermore, CMB can also be asked for configurations. Thus, CMB is a REST service that offers a GET operation to retrieve configuration data and a POST and PUT to store and update configurations. The CMB is responsible for several CDs.

Another component *ConfigurationHandler* (*CH*) is installed on the DC to periodically pull configuration data from the CMB by using a GET request. The GET request works in such a way that each request from CH to CMB returns NOT MODIFIED (e.g., code 304 for HTTP/REST) whenever a configuration has not been changed at the CMB (this can be determined by using the ETAG mechanism and the lastModified timestamp); otherwise, a package with all configurations for the CD is returned. There is a CH for each CD. Each CH obtains credentials to access the CMB REST API to get only its configurations. The credentials are changed periodically and passed to CH using the mechanism explained below.

The configuration itself is stored as a zipped file `package.zip` and organized as follows:

- There is a directory for each device component named like the DC. It is assumed that each DC possesses a unique name or identifier.
- Several files can be put in such a component's directory; the format can be json or XML.

The following is a sample `package.zip`:

```
|- bulk-transfer
   |- x.json
   |- y.json
|- db-inserter
   |- z.json
```

The first level of the hierarchy determines the components, here `bulk-transfer` and `db-inserter`. And files `x.json` and `y.json` contain configurations for the DC bulk-transfer. Further configuration files can be added at any time.

As already explained, CH calls the CMB service periodically, particularly initially after a restart of CH. Whenever a successful response is received, the packaged data is analyzed by CH and internally distributed to all device components DC running on the computing device CD. The communication between CH and components is done by means of a message queue, e.g., supporting MQTT. That is, having received a configuration change, CH splits the configuration `package.zip` into parts according to the returned package structure so that individual configuration files for each DC are extracted. The contents of all the files belonging to the same DC are merged to one file.

CH keeps the latest configuration state for each DC in an internal storage. If something has changed for a component compared to the last state (determined by using a hash key or similar), CH puts a message into the message queue with a topic `/configuration/<DC>` that identifies the DC being supposed to receive it. Otherwise, the configuration will not be pushed since nothing has changed for the respective component.

Each device component listens to incoming configuration changes in the message queue concerning itself (identified by its topic `/configuration/<DC>`). It takes the message payload, i.e., its configuration part. Afterwards, the DC can react according to the new configuration, for instance, adjusting some parameters or performing a certain action. Figure 1 illustrates the approach for such a periodic configuration update.

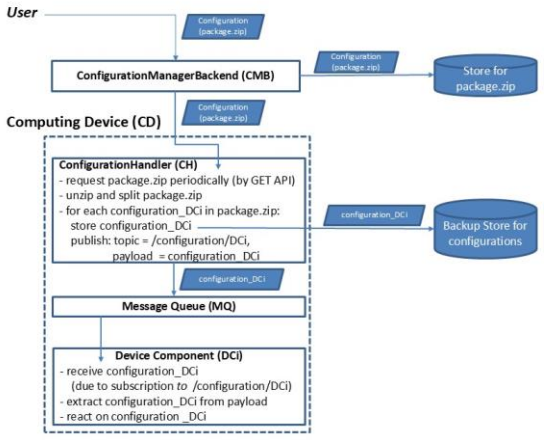

Figure 1. Periodic configuration update.

This is the typical scenario of notifying DCs about configuration changes. Another scenario is important for restarting components. After a restart of a device component DC, the DC might have missed some configuration changes that occurred during downtime. Hence, DC is enabled to ask for its latest configuration explicitly. Therefore, the DC can publish a corresponding message to the message queue with

a topic /request/<DC>. CH listens to topic /request/#, i.e., all those topics (due to "#") starting with /request, and reacts by issuing a request to the CMB service and behaving as described before. Figure 2 illustrates the procedure.

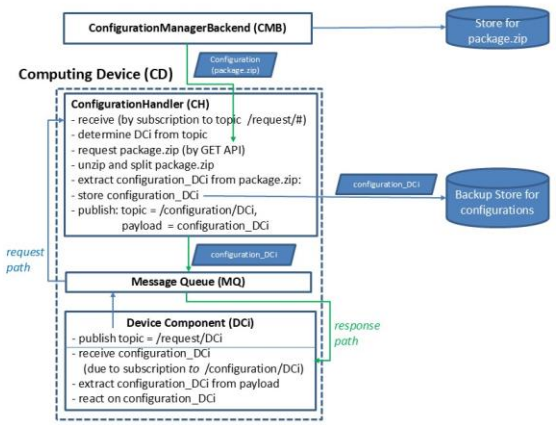

Figure 2. Explicit configuration update.

In case the CMB service is unreachable by CH (e.g., due to network problems), the latest version of a configuration as stored in CH is issued if explicitly asked for a configuration by a DC. Once CMB is reachable again, the usual mechanism works as described before.

Indeed, there is some monitoring of the overall system to detect any issues as early as possible and sending alerts to responsible persons.

Using a message queue has the advantage that all the components do not need to be known in advance or have been registered somewhere. In other terms, new components can be added by using the mechanism immediately. If a component mentioned in the package does not exist, a message is published to the message queue but nothing else happens due to the lack of a consuming DC.

The Cloud is beneficial for CMB due to global accessibility and high reliability, but not mandatory for this approach.

Compared to other approaches mentioned in Section V, advantages are:

- A computing device can be secured by not being accessible from the Internet while still obtaining configuration changes.
- Moreover, the configuration can be done at any time and outside of the computing device CD, independent of its location.

III. ORGANIZATION OF SOFTWARE

The common parts for letting a DC request a configuration at startup and listening to request changes can be placed in a common piece of code so that all the DCs can share the logic (e.g., by inheritance).

The following are some code snippets in python, however, omitting some details, such as proper exception handling.

There is a superclass (indicated in python by ABC) with the common code to be shared with every component:

```
class Component(ABC):
  def __init__(self, broker_url:str, broker_port:int):
     initialize message queue mqtt;
     set component_name and topic;
     on_message = lambda msg: self.on_msg(msg)
     subscribe to message queue with topic and on_message
      as callback;
  def on_msg(self, msg):
      payload = json.loads(msg.payload.decode('UTF-8'))
  def start_listening(self):
     self.mqtt_client.broker_client.loop_start()
  def stop_listening(self):
     self.mqtt_client.broker_client.loop_stop()
  @abstractmethod
  def update_configuration(self):
     pass #  to be implemented by every derived class
  def request_configuration(self):
     data = { "component": self.component_name }
     self.mqtt_client.publish(message=json.dumps(data),
                              topic='/request' + self.component_name)
```

on_message is a callback function that is used to subscribe to the message queue with a particular topic. Functions start_listening and stop_listening start and stop listening to a specific topic in the message queue, resp. update_configuration is an abstract function that must be implemented in a component to react on received configuration changes.

Every component has to be derived from this superclass as follows.

```
class SpecificComponent(Component):
   def main():  # will run in a docker container
      super().__init__(broker_url, broker_port)
      self.start_listening()
      self.request_configuration() # get first configuration for
                                   # start-up
   def update_configuration(self,payload):
      if 'config_a' in payload:
         react on payload['config_a']
      if 'config_b' in payload:
         react on payload['config_b']
```

Here, SpecificComponent runs in a docker container, which executes the main function. Invoking start_listening starts listening on configuration changes. During start-up of the component, a configuration is requested by request_configuration. Any configuration change will automatically invoke update_configuration, where component-specific behavior is implemented how to react.

IV. EVALUATION IN AN INDUSTRIAL CONTEXT

This section discusses the evaluation of the approach in an industrial context by using a concrete application.

A. Industrial Context

Indeed, there are many different industrial IoT projects within our company. However, it turned out that many of them have similar requirements and follow the same behavioral scheme. As one important characteristic, there is no internet access to the devices. Moreover, there is a strong need to deploy project-specific applications at the edge.

This leads to one common architecture to be set up several times in industrial projects. The overall generic approach follows the Lambda [23] architecture and is based upon container technology. Indeed, IoT applications are increasingly deployed using containers [24].

The common use case is to gather data from IoT devices. Data is processed and used twofold.

First, there are several calculations of key performance indicators (KPIs) that are resource-consuming and run on a daily schedule producing some kind of daily analysis and summary. For these applications, a component like a batch layer [23] is sufficient. That is, data is regularly pushed into the Cloud, and analysis and calculating KPIs is then performed in the Cloud using the submitted data. Calculated KPIs and any detected anomalies are visualized in dashboards. Further applications in this context are predictive maintenance etc. since they also have higher needs on compute power.

Second, other use cases behave in the sense of a speed layer [23] and require data in real-time to immediately react on events in the data, e.g., to control a device. Those applications typically run at the edge in the sense of edge computing.

The overall common architecture consists of several components running in Docker containers. Each component has a dedicated task to fulfil.

At first, a *Connector* abstracts from various industrial protocols, such as OPCUA, MODBUS, or BACNET to get sensor data from devices. Hence, this is a central component to handle all the various protocols and their heterogeneity for receiving data from devices. The Web-of-Things [34], particularly the concept of thing description, is the basis for this component; it keeps the information about the device and its protocol and handles data access.

The Connector sends data to a *Forwarder* component immediately by means of an efficient protocol like web sockets. The Forwarder then puts the received data into a message queue with a particular topic.

Using a message queue has the reason to let other application-specific components immediately consume events from the message queue, similar to the speed layer in the Lambda architecture [23].

An *Inserter* listens to the message queue by subscribing to the topics used by the Forwarder. It stores the received data in a timeseries database, such as InfluxDB. Again, other application-specific components are enabled to read data from the database.

The *BulkTransfer* component transfers a bulk of sensors' data from the timeseries database to the Cloud regularly in a configurable interval, e.g., every hour. This means the data for, e.g., the last hour is then transported. This scheduled job is reasonable for Cloud-based analysis and algorithms that do not require streamed data [23].

The rationale behind this architecture, especially using a message queue and a timeseries database, is to allow for project-specific components to be plugged in. Depending on a particular project, further application-specific components can be deployed to consume and process data directly and immediately from the message queue or timeseries database. Those applications can also store data there to be processed by others or being transferred to the Cloud. Application-specific components are especially used for controlling devices.

This is quite a generic and flexible approach. Various configurations are possible in this architecture for dedicated scenarios due to keeping components exchangeable.

B. Application

We applied the configuration approach successfully to achieve several configurations being explained below.

As mentioned previously, the *Inserter* listens to the message queue and stores the received data in a timeseries database. From a performance point of view, it is not reasonable to store record by record. Hence, a bulk approach gathers data until a certain number of records have been received or a certain time threshold has been passed; it then stores the bulk of records. The time threshold is reasonable in order to avoid that records are not stored for a longer period of time because of incomplete bulks. Both the bulk size and the time threshold are configurable for the *Inserter* to adjust to specific loads using our approach.

Next, the *BulkTransfer* runs periodically as some kind of Cron job to move bulks of data from the timeseries database to the Cloud. Here, the schedule is configurable. Intervals can be configured according to how often data is processed in the Cloud by means of a Cron schedule. Moreover, the *BulkTransfer* requires S3 credentials to access the Cloud storage. Due to key rotation, the credentials are periodically changed in the Cloud. New secrets can now be updated so that *BulkTransfer* becomes able to submit data to the Cloud storage.

There are also some more general applications of configurations. Having several components and containers in a device, the communication between them, for instance IP addresses and ports, are timeseries database at startup. Similarly, several configurations for the timeseries database and message queue are configurable. Also, the logging level can be changed at runtime. This turned out to be very important since the logging level can be increased for debugging purposes and reset after having detected issues.

In the architecture, a thing description (TD) plays an important role, particularly to describe the sensors. Whenever new sensors are delivered by a device, via OPCUA or MODBUS connectors, the software components become aware of new sensors by receiving the enhanced TD

with new sensors by means of configuration. Hence, data from new sensors can be processed immediately.

It could happen that a container behaves badly. A configuration parameter is used to enforce a restart, maybe with changing parameters.

Further configurations are used for bypassing components. For instance, the *Inserter* can be configured to directly forward data to the Cloud, then skipping the *BulkTransfer* and allowing for data processing in a streamed manner in the Cloud. However, due to our experiences, this is only useful for smaller amounts of data due to higher costs for the IoT solutions of Cloud vendors.

V. RELATED WORK

There are many reference architectures for IoT, edge, and fog computing [2][4][5][11][13][14][17][19][25][35] in the literature. They provide generic taxonomies for the components of IoT platforms and differentiate several functional components, such as device, sensor, actuator, and gateway. Reference architectures then pose components in three [39] or more layers [12]. They all have in common to pay no attention on how to configure components properly.

State-of-the-art reviews, such as [32] – despite discussing so far unsolved challenges in the field of edge applications – also do not mention configuration problems, especially in case of unreachable devices as a challenge.

Several approaches could benefit from such an approach despite not mentioning configuration issues. For example, Stankovski et al. [26] proposed a distributed self-adaptive architecture for container-based technologies to ensure the QoS for time-critical applications. Monitoring data is used to allocate required resources for each container; end-users, application developers and/or administrators can define operational strategies to handle resources in a better manner. Indeed, these strategies are a form of configuration.

CloudScale is a monitoring system proposed by [21]. The system analyses the performance of distributed applications at runtime, thereby adopting user-specified scaling policies for provisioning and de-provisioning of virtual resources. Policies are again another type of configuration.

Olorunnife et al [24] evaluate various approaches for failure recovery for IoT applications. Monitoring the output of IoT applications. Their approach automatically diagnoses faults with IoT devices and gateways; and effectively manages and re-configures container-based IoT software to achieve a minimal downtime upon the detection of software failures. This technique can particularly be applied to scenarios where IoT software is deployed in embedded or hard to reach scenarios, i.e., with difficult or no physical access. But this approach merely focuses on automatic recovery without any interaction from the outside.

VI. CONCLUSIONS

In this paper, we discuss the problem of configuring devices that are unreachable from the Internet in the context of Internet-of-Things (IoT).

We motivate the need for configurations by presenting typical examples such as Cron schedules for running periodical jobs, parameters or thresholds for components, changing credentials because of password rotation to name a few. Especially the latter one is required in industrial settings due to high security requirements where passwords must be renewed regularly. These kinds of configurations are usually indispensable for an effective operation in industrial contexts. However, if devices are unreachable by the Internet, operators have to perform configurations at the device site causing efforts and costs.

The approach that is pursued to solve this issue is discussed in detail. There is mainly a central service running in the Cloud to keep configurations. New configurations can be submitted to that service. Each IoT device is equipped with a component that polls the central service periodically about configuration changes and distribute configurations to the components running in that device

We evaluate the approach in a real industrial application where several types of configurations are required.

Our future work will evaluate even more complex scenarios, such as enabling or disabling components in the architecture at runtime. We also want to investigate the impact of the approach on the overall system performance. Moreover, we want to tackle further industrial issues for IoT devices, such as enhancing fault tolerance by self-healing and monitoring.

REFERENCES

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aled-hari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies Protocols and Applications”, IEEE Communications Surveys Tutorials, Vol. 17 (4), pp. 2347-2376, June 2015, ISSN 1553-877X.

[2] M. Aazam, I. Khan, A. Alsaffar, and E. Huh, “Cloud of Things: Integrating Internet of Things and Cloud Computing and the Issues Involved”, Int. Bhurban Conf on applied sciences and technology.

[3] L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A Survey”, Computer Networks 2010, Vol. 54 (15), pp. 2787–2805.

[4] M. Bauer, M. Boussard, N. Bui, J. C. De Loof, C. Magerkurth, S. Meissner, A. Nettsträter, J. Stefa, M. Thoma, and J. W. Walewski, “IoT Reference Architecture. In: Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model”, Springer Berlin Heidelberg 2013.

[5] S. Biswas and S. Misra, “Designing of a Prototype of e-Health Monitoring System”, IEEE Int. Conf on Research in Computational Intelligence and Communication Networks (ICRCICN) 2015, pp. 267–272.

[6] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, “Fog Computing: A Platform for Internet of Things and Analytics”, Big Data and Internet of Things: A roadmap for smart environments, pp. 169–186. Springer.

[7] B. Costa, J. Bachiega, R. Carvalho, M. Rosa, and A. Araujo, “Monitoring Fog Computing: A Review, Taxonomy, and Open Challenges”, Computer Networks Vol. 215, Elsevier 2022, pp. 1–30.

[8] “Remote Services – For the High-Performance Operation of Your Plant”, https://www.siemens.com/global/en/products/services /digital-enterprise-services/field-maintenance-services/remote-services.html) [retrieved: March, 2025].

[9] B. Dorsemaine, J.-P. Gaulier, J.-P. Wary, N. Kheir, and P. Urien, “Internet of Things: a Definition & Taxonomy”, 9th Int. Conf on Next Generation Mobile Applications, Services and Technologies, ISBN 978-1-4799-8660-6/15, 2015.

[10] K. Fatema, V. Emeakaroha, P. Healy, J. Morrison, and T. Lynn, “A Survey of Cloud Monitoring Tools: Taxonomy, Capabilities and Objectives”, Journal of Parallel and Distributed Computing 2014, Vol. 74 (10), pp. 2918–2933.

[11] J. Guth, U. Breitenbucher, M. Falkenthal, F. Leymann, and L. Reinfurt, "Comparison of IoT Platform Architectures: A Field Study Based on a Reference Architecture", Cloudification of the Internet of Things (CIoT) 2016, pp. 1–6.

[12] J. Guth, U. Breitenbücher, M. Falkenthal , P. Fremantle, O. Kopp, F. Leymann, and L. Reinfurt , "A Detailed Analysis of IoT Platform Architectures: Concepts, Similarities, and Differences", Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives, Springer 2018, pp. 81-101.

[13] J. Gubbi, R. Buyya, and S. M. P. Marusic, "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions", Future Generation Computer Systems 2013, Vol. 29 (7), pp. 1645–1660.

[14] S. A. S. Haller, M. Bauer, and F. Carrez, "A Domain Model for the Internet of Things", Proc. of IEEE Int. Conf on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber Physical and Social Computing. IEEE (2013).

[15] B. Hazarika and T. J. Singh, "Survey paper on Cloud Computing & Cloud Monitoring: Basics", SSRG Int. Journal on Comput. Science Engineering 2015, Vol. 2 (1), pp. 10-15, ISSN:2348–8387.

[16] J. Kua, G. Armitage, P. Branch, and J. But, "Adaptive Chunklets and AQM for Higher-Performance Content Streaming", ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM) 2019, Vol. 15, pp. 1–24

[17] J. Kim, J. Lee, J. Kim, and J. Yun, "M2M Service Platforms: Survey, Issues, and Enabling Technologies", IEEE Communications Surveys & Tutorials 2014, Vol. 16 (1), pp. 61–76.

[18] J. Kua, S. H. Nguyen, G. Armitage, and P. Branch, "Using Active Queue Management to Assist IoT Application Flows in Home Broadband Networks", IEEE Internet of Things Journal 2017, Vol 4 (5), pp. 1399–1407.

[19] S. Krco, B. Pokric, and F. Carrez, "Designing IoT and Architecture(s)", In: Proc. of the IEEE World Forum on Internet of Things (WF-IoT). IEEE (2014).

[20] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Towards Observability Data Management at Scale", ACM SIGMOD Record, Vol. 49, pp. 18–23.

[21] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm", Proc. of 5th IEEE Int. Conf. on Service-Oriented Computing and Applications (SOCA'12). IEEE, Taipei, Taiwan, pp. 1–8.

[22] I. Lee and K. Lee, "The Internet of Things (IoT): Applications, Investments, and Challenges for Enterprises", Business Horizons 2015, Vol. 58 (4), pp. 431–440.

[23] N. Marz: "How to Beat the CAP Theorem", 13 October 2011. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html [retrieved: March, 2025].

[24] K. Olorunnife, K. Lee, and J. Kua, "Automatic Failure Recovery for Container-Based IoT Edge Applications", Electronics 2021, Vol. 10.

[25] V. Prasad, M. Bhavsar, and S. Tanwar, "Influence of Montoring: Fog and Edge Computing", Scalable Computing: Practice and Experience 2019, Vol. 20 (2), pp. 365-376.

[26] V. Stankovski, J. Trnkoczy, S. Taherizadeh, and M. Cigale, "Implementing Time-Critical Functionalities with a Distributed Adaptive Container Architecture", Proc. of 18th Int. Conf. on Information Integration and Web-based Applications and Services (iiWAS2016). ACM, Singapore, pp. 455–459.

[27] W. Shi and S. Dustdar, "The promise of Edge Computing", Computer, Vol. 49 (5), pp. 78–81, May 2016.

[28] K. Saharan and A. Kumar, "Fog in Comparison to Cloud: A Survey", Int. Journal of Computer Applications, Vol. 122 (3), 2015.

[29] E. Solaiman, R. Ranjan, P. P. Jayaraman, and K. Mitra, "Monitoring Internet of Things Application Ecosystems for Failure", IT Professional 2016, Vol. 18 (5), pp. 8–11.

[30] M. Satyanarayanan, P. Simoens, Y. Xiao et al., "Edge Analytics in the Internet of Things", IEEE Pervasive Computing 2015, Vol. 14 (2), pp. 24–31.

[31] A. Srinivasa and D. Siddaraju, "A Comprehensive Study of Architecture, Protocols and Enabling Applications in Internet of Things (IoT)", Int. Journal of Science & Technology Research 2019, Vol. 8, Issue 11.

[32] S. Taherizadeh, A. Jones, I. Taylor, Z. Zhao, and V. Stankowski, "Monitoring Self-Adaptive Applications within Edge Computing Frameworks: A State-of-the-Art Review", Journal of Systems and Software 2018, Vol 136, pp. 19-38.

[33] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic Computing: a New Paradigm for Edge/Cloud Integration", IEEE Cloud 2016.

[34] Web of Things in a Nutshell. https://www.w3.org/WoT/documentation [retrieved: March, 2025]

[35] L. D. Xu, W. He, and S. Li, "Internet of things in industries: A survey", IEEE Transactions on Industrial Informatics Vol. 10 (4).

[36] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of Things", Int. Journal of Communication Systems 2012, Vol. 25 (9), pp. 1101–1102.

[37] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A Survey on the Edge Computing for the Internet of Things", IEEE Access 2018, Vol 6, pp. 6900–6919.

[38] S. Yi, C. Li, and Q. Li, "A Survey of Fog Computing: Concepts, Applications and Issues", Proc. of Workshop on Mobile Big Data. (Mobidata 2015), pp. 37–42.

[39] L. Zheng, H. Zhang, W. Han, X. Zhou, J. He, Z. Zhang, Y. Gu, and J. Wang, "Technologies, Applications, and Governance in the Internet of Things", Internet of Things – Global Technological and Societal Trends. River Publishers (2009).