Flexible and Efficient Deployment of Data Processing Pipelines on Wireless IoT Systems

Giorgos Polychronis, Manos Koutsoubelias,
Foivos Pournaropoulos and Spyros Lalis
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
{gpolychronis, emkouts, spournar, lalis}@uth.gr

Lefteris Georgiadis, Thomas Pazios
Stratos Tsatsaronis and Isaias Vrakidis
Department of Research & Development
METIS Cybertechnology
Athens, Greece
{lefteris.georgiadis, thomas.pazios,
stratos.tsatsaronis, isaias.vrakidis}@metis.tech

***Abstract*—There is an increasing number of IoT devices that do not merely act as sensor nodes but are also sufficiently powerful to perform some local data processing before data is sent upstream to more powerful servers or the cloud. In this paper, we present a framework for the flexible deployment and execution of sensing and data processing application components on embedded, industrial-strength nodes that can be connected to a wide variety of sensors. This is done using textual, declarative descriptions instead of actual code, in conjunction with a pub/sub approach for data exchange between such components. As a result, it becomes possible to build and deploy in-network data processing pipelines in efficient way, even on top of low-bandwidth wireless links. Our evaluation shows that the proposed approach can reduce the size of application logic by 95.3% and the deployment time by up to 91.1% compared to a more conventional deployment of binaries. Also, the ability to process data near or even directly on the nodes that are connected to the sensors proving the raw measurements, can reduce the number of messages by 97.7% and improve message latency by up to 44.3% vs sending all data and processing it on a server.**

***Index Terms*—IoT, wireless sensor networks, service deployment, in-network processing, publish/subscribe communication**

I. Introduction

Thanks to the continued advances in embedded computing platforms and last mile wireless technologies, there now is a plethora of devices, equipped with all kinds of sensors and Internet connectivity, which can be installed in homes, streets, fields and different infrastructure to support a wide range of sensing applications, giving rise to the so-called Internet of Things (IoT). Besides their sensors, IoT devices may have non-negligible computing resources. This, in turn, allows such nodes to be used for processing data near the sources, in the spirit of edge computing, to reduce the amount of data sent upstream to servers and/or datacenters over the Internet. While this approach increases flexibility, it also raises the problem of managing the deployment of application software on such nodes. This problem becomes even more important when nodes are accessible only over low-bandwidth wireless links and have limited or no direct Internet connectivity.

In this paper, we present a framework that supports the deployment of application-level sensing and data processing tasks on IoT devices in a flexible and efficient way. The key idea is to express application logic through descriptions rather than actual code, which significantly reduces the amount of data that needs to be transmitted over the wireless links in order to deploy an application service, and avoids having to install and run suitably prepared binaries on nodes. Furthermore, the framework makes it possible to organize application services in distributed data processing pipelines with indirect communication between the different components, which is supported under the hood through a publish-subscribe mechanism that performs the required data forwarding.

The main contributions of our work are: (i) We present a complete framework for the flexible and efficient deployment of sensing and data processing pipelines in IoT systems. (ii) The application services are captured via descriptions rather than actual code, and the data exchange between them is achieved via indirect pub/sub communication. (iii) We evaluate the framework on a testbed with industrial-strength IoT nodes, showing that the proposed approach has significant advantages over conventional code shipment and data processing data outside the IoT network.

The rest of the paper is structured as follows. Section II gives an overview of related work. Section III provides an overview of the IoT systems we target in our work, based on a concrete use-case from the shipping domain. In Section IV, we describe how our framework supports the development of application-level sensing and data processing tasks, while Section V discusses the underlying system-level mechanisms for the deployment, execution and interconnection of such services in the IoT system. Section VI presents an evaluation of our framework. Finally, Section VII concludes the paper.

II. Related Work

Data aggregation is the main approach to reduce network traffic and save energy in wireless sensor networks (WSNs). A survey on data aggregation is given in [1]. The authors of [2] investigate clustered networks where data aggregation is performed by the cluster heads. [3] discusses WSNs with different types of sensed data, where aggregation can be achieved between data of the same type. To this end, they

propose a routing protocol which considers, besides other metrics, the potential for data aggregation. Both of these works try to extend the network lifetime. In [4], the authors focus on in-network data filtering to reduce the data volume transferred. Work in [5] proposes an approach for outlier detection in a WSN where the detection happens inside the network. [6] considers a cognitive sensor network where in-network processing allows to increase transmission opportunities. Our work also allows in-network processing and data aggregation to decrease the volume of data sent over the wireless network. However, this is done by application-level logic that can be deployed on the nodes in a dynamic, flexible and efficient way.

There is also a wide range of work on the dynamic deployment of code in WSNs. In [7], the authors propose an architecture for over-the-air firmware updates, consisting of a web server, an android application for the operator to upload new firmware images on the server, and a wifi module on the device that gets the new firmware from the server and then flashes the firmware through the serial to the microcontroller. A similar approach is followed in MENDER [8], where new software can be uploaded on the server also through REST APIs. [9] focuses on low-end IoT devices that run a lightweight OS and, on top of that, a lightweight container capable to execute scripts. Sensorware [10] is a framework that runs on top of a node's OS and provides a runtime environment for the flexible deployment and execution of control scripts that implement application-level logic. In [11], the authors present a middleware which allows to extend the basic system functionality via application-level tasks. [12] describes system-level mechanisms for the dynamic instantiation and strong migration of application-level agents in a WSN, to support flexible in-network processing and minimize the number of messages sent over wireless links. Like the above, our work allows to deploy and run application-level logic on IoT nodes, without changing the basic firmware or system software. The main difference is that we do not send code or scripts over-the-air, but merely declarative descriptions that are parsed and executed on the node.

III. System Overview

Our work focuses on wireless IoT systems comprising embedded devices that are used to collect information from a potentially large variety of sensors. As a concrete case, we consider an IoT system installed on a vessel with the purpose of monitoring and analyzing its status, including, e.g., engine operation, speed, fuel consumption, sea state, etc. METIS already operates such systems in numerous vessels worldwide.

Figure 1 provides a high-level view of an indicative system infrastructure. The basic building block is the so-called wireless intelligent collector (WIC), an industrial-strength device approved for operation on commercial vessels. It has an embedded computing board with different interfaces (such as RS-232, RS-485, CAN, etc) through which it can be connected to and get data from a wide range of sensors. WICs also feature an Ethernet and Zigbee interface.

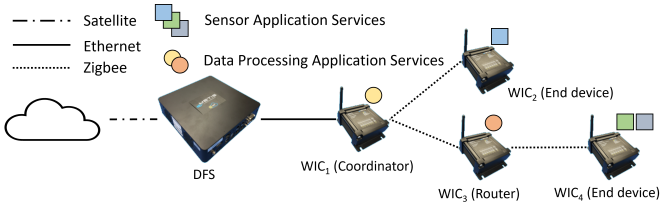

Fig. 1: Flexible deployment and execution of application services on the embedded nodes of the IoT system.

Zigbee is used to deploy and interconnect WICs on the vessel without any wiring. An installation may have multiple Zigbee networks, each managed by its own coordinator WIC. The rest of the WICs in a network act as router nodes or end nodes (that do not forward data to/from other nodes).

The Ethernet interface is used to connect the coordinator WIC(s) to the so-called data fusion server (DFS) on the ship. The DFS is where collected data is stored and possibly pre-processed, before forwarding it to the cloud via satellite for further analysis, long-term storage and integration with the ship owner's ERP systems (not shown in the figure).

This work concerns the IoT system on the vessel. More specifically, we wish to: (1) Support the flexible deployment of application-level logic that reads and processes sensor data. (2) Allow processing to be performed "in" the network of the IoT devices, thereby reducing the amount of data transmitted to the server and lowering the pressure on the low-bandwidth wireless network. (3) Relieve the administrator from having to connect and issue low-level commands to the individual embedded devices. The next sections describe in more detail the framework we have developed to achieve these objectives.

IV. Application-level Abstractions

A. Application services and data processing pipelines

To exploit the sensing and processing capacity of such an IoT infrastructure, we adopt a distributed application approach. More specifically, the application logic is split into smaller components, referred to as *services*, which can be deployed on different nodes, as shown in Figure 1. We differentiate between *sensing* and *processing* services. Sensing services access specific sensors to collect measurements and forward them upstream, possibly after some filtering. Processing services collect data produced by sensing services and/or other processing services to produce more complex metrics.

Based on these two types of application services, one can build data processing chains or pipelines that run in a distributed way on top of the IoT infrastructure. For instance, a high-level fuel consumption metric for the ship can be produced by a data processing service (e.g., the orange service running on WIC_3) that combines readings from the engine's power and fuel flow sensors (e.g., produced by the green and grey sensor services running on WIC_4). In fact, depending on the node's resources, the data processing service, could even be placed on the same node as the sensor services to further reduce the data sent over the wireless network.

B. Formation of data processing pipelines

The type of data that is generated in the IoT system is uniquely referred to through so-called *quantity identifiers* or QIDs. In other words, sensor services generate data for certain QIDs, while data processing services consume data for one or more QIDs and, in turn, generate data for other QIDs. As a consequence, the producer-consumer relationship between application services in a pipeline is captured via the produced/consumed QIDs, rather than being specified in a direct way by the identifiers of the respective application services.

This significantly increases flexibility in different ways. Firstly, the developer of a service that produces or consumes data, does not have to refer to specific endpoints for the respective data consuming and/or data producing services. Secondly, one can naturally capture the case where data for the same quantity needs to be consumed by multiple application services. Thirdly, the data producer for a given quantity may change in time without braking any references. Last but not least, a data processing pipeline may be deployed in an incremental way, without running into all kinds of race conditions regarding the instantiation of application services and initialization of the respective communication endpoints. Namely, a processing service simply remains idle as long as one of the required inputs is not available. This is particularly important when deployment occurs over low-bandwidth links, where the delay can be non-negligible.

C. Application service descriptions instead of code

To keep application services small in size and to support a flexible deployment without having to perform code updates on the IoT nodes, we capture the respective logic via declarative text-based descriptions, rather than actual code. More specifically, each service is "encoded" in a json file, which includes all the information that is required to instantiate and run the service on an IoT device (WIC).

Listing 1 shows an excerpt of the description for a sensor service that reads a ship's engine power values from a corresponding sensor via CAN, and forwards these measurements upstream under a given QID. The service description specifies the interface and configuration setting for accessing the sensor as well as the method used to receive the information with the associated trigger and period. The rules for parsing the information received from the sensor in order to extract the desired quantity and for producing the value that will be forwarded upstream, are given in a separate file shown in Listing 2. Note that, in the general case, sensors can generate more than one values combined into a bundle (string) from where one may extract several quantities. Each extracted value is stored in a temporary variable, which can then be combined/filtered through suitable transformation and acceptance expressions (in the given example, a single value is extracted and forwarded upstream without any transformation or filtering).

Data processing services have similar descriptions. The main difference is that one must specify the data inputs that are required to produce the data that will be forwarded upstream.

Listing 1 Description for a sensing service.

```
serviceName: "EnginePower_svc" #name of application service
sensorConnectionMode: "CAN" #means of sensor connection
sensorConnectionPort: "/dev/ttymxc0" #serial port
baudrate: "4800" #baudrate for serial communication
databits: "8" #data bits for serial communication
parity: "N" #parity for serial communication
stopbits: "1" #stop bits for serial communication
protocol: "ASAP" #protocol for data encoding
parsing: ".../protocolParse.json" #file for parameter parsing
execMethod: "Listen" #listen for incoming data
execTrigger: "TimeBased" #publish data based on a timer
execPeriod: "15" #publication frequency (sec)
```

Listing 2 Parsing parameters.

```
outputQName: "EnginePower" #name of output quantity
outputQID: "EP0123456789" #id of output quantity
talkerID: "" #id of data sender/source
sentenceID: "41" #id of sentence with several data items
                 #(a sensor may produce different sentences)
fields: [
    {
        position: "1" #position of data item in sentence
        type: "Number" #type of data to be parsed
        varname: "var1" #variable to save the data
    }
]
transformExpr: "var1" #data processing (can be complex)
acceptanceRange: "" #acceptable range of values to publish
```

Listing 3 Description for a processing service.

```
serviceName: "SFOC_dev_svc"
outputQName: "SFOC_dev"
outputQID: "SFOCDEV00112233"
execTrigger: "TimeBased"
execPeriod: "15"
variables: [
    {
        inputQID: "EP0123456789" #id of input quantity id
        method: "Latest" #use the most recent value received
        validityPeriod: "20" #freshness of values (sec)
        varname: "EP"
        acceptanceExpr: ""
    },
    {
      inputQID: "FMF578899103"
      method: "Latest"
      validityPeriod: "20"
      varname: "FMF"
      acceptanceExpr: ""
    }
]
transformExpr: "(FMF*940/EP)/
 (0.000005809*EP^2-0.0079556*EP+193.25)"
acceptanceRange: "(0.1, inf)"
action: "Drop" #drop data that do not meet above condition
```

An indicative description for such service is given in Listing 3. Temporary variables are introduced for each input quantity, along with basic rules for assigning values to them based on the incoming data stream and their validity. In this case, the first variable captures the most recent output of the service that produces measurements for the engine power (Listings 1 and 2), while the second variable captures the most recent output of a another service that produces measurements for the ship's fuel mass flow (the description for that service is very similar to the engine power service and is not shown for brevity). Similar to the parsing rules for sensor services, these variables can be transformed/combined and filtered via corresponding expressions, provided as part of the service

description, to produce the final value that will be forwarded upstream. In the specific example, the data processing service calculates the specific fuel oil consumption of a ship (SFOC) based on ship's engine power and fuel mass flow, and its deviation vs a SFOC reference model/function (obtained from different sea trials). The range-action expression is used to drop all values that are below a certain threshold; such values are considered to be of no interest/use to the higher-level monitoring services or engineering team, and thus are not forwarded upstream to the DFS. Of course, the ability to drop unwanted data as early as possible in the pipeline further reduces the traffic on top of the wireless network.

V. System-level Support

The flexible deployment and data exchange between application services is supported via a set of resident system-level services running on the IoT nodes. The overall software architecture is given in Figure 2. The next subsections describe in more detail the main system services and the underlying communication support.

A. System services

The main system-level services are:

SrlInt: The serial interface service is a general purpose service that executes the logic of an application sensor service based on the respective description, by reading the measurements of the specified sensor, possibly performing some pre-processing and publishing the result under a specific QID.

BasicProc: The basic processing service is similar to the srlInt, for the execution of an application data-processing service. Instead of reading data from sensors, it consumes data published under the specified QIDs, performs the specified processing and finally produces its own values.

ZeroBroker: This service handles the communication of the system-level services on the same IoT device, using the ZeroMQ technology [13]. Also the ZeroBroker acts as a forwarder for data that needs to be propagated over the network and for data coming from the network, via the GTP.

GTP: The so-called general transmission platform undertakes the actual data transfers to and from the network. Since the coordinator is both connected to the Zigbee network and directly to the DFS over Ethernet, the coordinator's GTP acts as a bridge between the wired and wireless network.

WicCtl: The WIC control service is a tool for the user used to send commands and files to the WIC devices, but also to receive information from them. Notably, among other things, WicCtl is used to deploy application services on the WICs by sending the service descriptions and launching corresponding instances of SrlInt or BasicProc services.

B. Pub-sub communication

Generally all the communication between system services, happens in a publish-subscribe manner. The topics used have the form $MsgType|Device|Service|ID$. The first field describes the type of the message, the second is the device name from which the message originates, the third field is the service name that the message originates, and the last field is the unique identifier of the message. For example, in the case of a message produced by the SrlInt or BasicProc services, the ID field is the QID of the respective value/measurement.

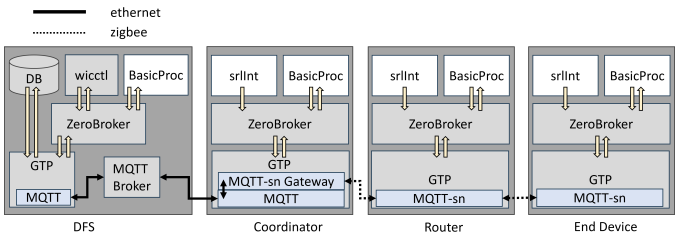

Fig. 2: Software architecture.

As already mentioned, the communication inside the WIC device is handled by the ZeroBroker. This is where all the other system services subscribe to their required topics, while advertising and publishing their own topics. More precisely, SrlInt and BasicProc services publish data in topics with a message type "measurement", the name of the local device, the name of the application service they execute, and finally the respective QID given in the respective description files. Conversely, when a service needs to receive data, e.g. a BasicProc service consuming a certain input QID, it subscribes for the respective topic on the ZeroBroker. The GTP service subscribes to all the topics advertised within the device so that it can forward them to the network as needed.

The GTP service handles the communication between WIC devices. This is done over MQTT-sn [14] if the WIC is connected to the Zigbee network. The coordinator, which is connected to the DFS over Ethernet, receives messages from the MQTT-sn and forwards them over MQTT [15] to MQTT broker running on the DFS. Similarly, the messages issued by the DFS are received over MQTT and are forwarded to the rest of the WICs over MQTT-sn. Note that measurements flow from the end devices to the DFS, while file transfers and control commands flow in the opposite direction (the "file" and "command" message types are reserved for the topics that are used to perform the respective interactions).

When a new device joins the network, the local GTP service announces the name of the device (using the "advertise" message type) so that the coordinator and the DFS discover the new arrival. Then it subscribes to all the all the topics that are required to receive commands, files and other advertisements. The GTP service on the DFS talks with MQTT directly to the MQTT broker. Except the subscriptions for required for receiving commands, files and advertisements it also subscribes to all the "measurement" topics, regardless the values of the other fields of the topic, so it can receive all the produced measurements published in the IoT network.

VI. Evaluation

We evaluate our implementation using a system in the spirit of Figure 1. For practical reasons, we use a test installation in the lab rather than in a vessel. Our testbed has only 3 WICs and

in each experiment we use a different subset of the topology shown in the figure.

The evaluation has two aspects. On the one hand, we measure the efficiency/speed of description-based service deployment vs deploying actual code/binaries. On the other hand, we show the benefits of in-network data processing vs having all sensor measurements being processed on the DFS.

A. Description-based vs code-based service deployment

In this set of experiments we show the benefits of the description-based service deployment. We use the DFS, WIC_1, WIC_2 and WIC_3, arranged as shown in Figure 1. We test two alternative service deployment scenarios.

In the first scenario, we assume that each application service needs to be implemented with code, in the spirit of a SrlInt service that is hardwired for a specialized sensing task. In this case, to deploy and run the application service on a WIC, we need to send the binary of 25.5 KB over the network. Note that this is the size for a reduced version, customized to retrieve and send measurements from a single sensor, stripped from all the configurability of the SrlInt system service. In the second scenario, the deployment of the application service via the configurable SrlInt service. In this case, we must send two description files (see sensor service example discussed in Section IV) with a total size of 1.2 KB.

The deployment is started from the DFS and the target node for service deployment is WIC_3. In both scenarios, we measure (i) the transfer delay for the respective required files from the DFS to node WIC_3, and (ii) the end-to-end service activation delay. The latter includes the time needed to send the command to remotely start the application service, the time needed for the WIC to start the respective service, and finally the time needed for the WIC to reply with an acknowledgement to the DFS.

We run the same experiments varying the number of application services that are running on WIC_2 and WIC_3 during the deployment of the new application service, generating competing traffic that must be sent over the wireless network towards the DFS (via the coordinator WIC_1). More specifically, we measure the application service deployment delay when 0, 1, 2, 3 and 4 services run on these WICs with each service publishing a 8-byte-measurement every 1 sec. We repeat this test 50 times and report the averages.

The results are given in Figure 3. Each stacked bar shows at the bottom the file transfer delay and on top the service activation delay. In both cases, the dominant part is the file transfer delay, whereas the service activation delay is comparably negligible. In total, when no other services are running in the system, the service deployment delay is 29.62 sec when sending the binary vs merely 2.72 sec when using descriptions, reduced by more than 90%. The presence of concurrently running services increases the difference even more. With 4 services running on each WIC, binary-based deployment requires 33.46 sec while description-based deployment merely takes 2.98 sec. This is expected as the transfer of larger files is more sensitive to contention on the wireless links.

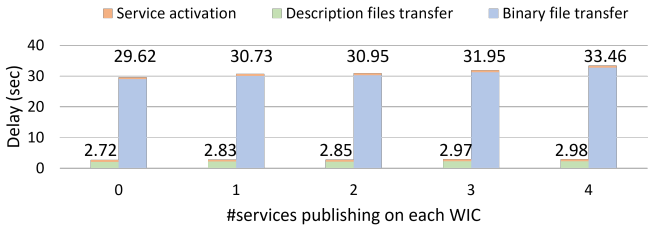

Fig. 3: Application service startup delay for description-based vs code-based deployment as a function of additional application services running concurrently on the WICs and generating traffic for the wireless network.

B. Reduce message traffic

In the next set of experiments, we consider the data generation and processing pipeline for the SFOC service introduced in Section IV. In this case, we use the DFS, WIC_1, WIC_3 and WIC_4 as shown in Figure 1. We assume that the two sensor services for the ship's engine power and fuel mass flow run on WIC_4. We investigate two scenarios: when the SFOC service runs on the DFS vs when the service runs on the same node as the sensor services (WIC_4).

In both scenarios, we use real measurements from a ship stored in files and configure the application sensor services to read these values from these files instead of accessing a real sensor via a physical/serial interface. We let the data processing pipeline run for a period of 1 hour and measure the total number of measurement messages sent over the Zigbee network (all wireless links). Each of the sensor services produces 240 measurement messages, so a total of 480 messages are sent over each wireless link. In contrast, by deploying the SFOC service directly on WIC_4, only 11 messages SFOC deviation messages are generated and sent towards the DFS, yielding a reduction of 97.7%.

Note that this difference becomes more significant as the hop distance h to the DFS increases. For instance, in our experiments where the sensor services were 2 hops away from the DFS, the total number of messages sent over the Zigbee network is 960 vs 22. For the general case of h hops, the flexible deployment of the SFOC service on the node where the sensor services are running, is $h \times (480 - 11)$ messages.

C. Reduced and more stable message reception delay

Another benefit of placing a data processing service as close as possible to the sensor services producing the required data values (QIDs), is the reduction of the contention on the links of the wireless network leading to more stable behavior. We investigate this aspect through another set of experiments, using the DFS, WIC_1, WIC_2 and WIC_3 as shown in Figure 1. In this case, we let a varying number of sensor services run concurrently on WIC_2 and WIC_3, each producing a double precision floating point value (8 Bytes) every 1 second. We run experiments with each WIC hosting 1, 2, 3 and 4 such sensor services, and record the delay until the messages containing the respective sensor measurements reach the DFS.

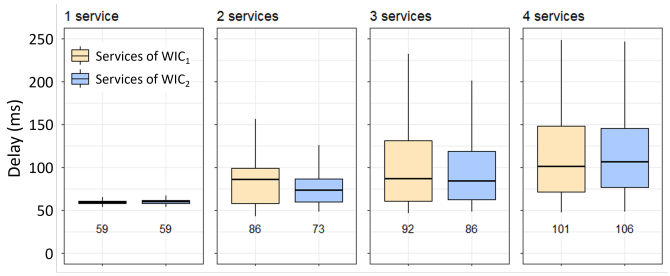

Fig. 4: Message reception delay at the DFS for the values produced by the sensor services on the WICs.

Figure 4 shows the recorded delay in the form of boxplots, visualizing the key statistical properties of the delay for the values generated by the sensor services running on each WIC. The number bellow the boxes give the median values in milliseconds (also marked by the horizontal line inside each box). As can be seen, the performance decreases quite significantly as the number of sensor services that run on the WICs increases, with the median delay growing on average from 59 to 79.5 to 89 up to 103.5 milliseconds when 1, 2, 3 and 4 sensor services run on each WIC. The variance and worst-case values also grow significantly.

Now, assume we wish to have a data processing service that consumes the values produced by the (orange) sensor services of WIC_2, and another data processing service for the (blue) sensor services of WIC_3. If these two services were deployed on the DFS, the delays with which they would produce their own values would be very close to those observed in Figure 4. In contrast, if each service were deployed directly on the WIC where the respective sensor services reside, in the spirit of the experiment discussed in Section VI-B, the data production delays (time needed for the produced values to reach the DFS) would be practically equivalent to the ones in the left-most plot of Figure 4, independently of the number of sensor services running locally on each node. Thus, the flexible deployment of application services near the data sources not only reduces the number of messages sent over the wireless network but also leads to lower latency and stable performance.

VII. Conclusion

We have presented a framework that enables the user to easily deploy sensing or data processing application services on wireless embedded devices featuring or connected to different sensors. Each application service is described by json files and can be started by sending the respective description to the desired node. In our evaluation, we have shown that the usage of service descriptions leads to significant reduction of the service's start-up time delay compared to deploying the service's code. Additionally, the capability of our system to place data processing services directly on the embedded nodes inside the wireless network can be beneficial in terms of network traffic.

As a continuation to this work, we plan to extend our framework to transparently place and migrate such application services without the involvement of the user (so that the user does not have to specify the host of each service). The service placement will be chosen and adapted by the system in an automated way to reduce the traffic in the wireless network or balance the load on the embedded nodes.

Acknowledgments

This work has been co-financed by the European Union-NextGenerationEU and Greek national funds through the Greece 2.0 National Recovery and Resilience Plan, under the call RESEARCH-CREATE-INNOVATE, project VEPIT – Vessel Energy Profiling based on IoT (code: TAEDK-06165).

References

[1] S. Abbasian Dehkordi, K. Farajzadeh, J. Rezazadeh, R. Farahbakhsh, K. Sandrasegaran, and M. Abbasian Dehkordi, "A survey on data aggregation techniques in iot sensor networks," *Wireless Networks*, vol. 26, pp. 1243–1263, 2020.

[2] M. Shobana, R. Sabitha, and S. Karthik, "Cluster-based systematic data aggregation model (csdam) for real-time data processing in large-scale wsn," *Wireless Personal Communications*, vol. 117, pp. 2865–2883, 2021.

[3] W.-K. Yun and S.-J. Yoo, "Q-learning-based data-aggregation-aware energy-efficient routing protocol for wireless sensor networks," *IEEE Access*, vol. 9, pp. 10 737–10 750, 2021.

[4] H. Wu, J. He, M. Tömösközi, Z. Xiang, and F. H. Fitzek, "In-network processing for low-latency industrial anomaly detection in softwarized networks," in *IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2021, pp. 01–07.

[5] J. W. Branch, C. Giannella, B. Szymanski, R. Wolff, and H. Kargupta, "In-network outlier detection in wireless sensor networks," *Knowledge and information systems*, vol. 34, pp. 23–54, 2013.

[6] S.-C. Lin and K.-C. Chen, "Improving spectrum efficiency via in-network computations in cognitive radio sensor networks," *IEEE Transactions on wireless communications*, vol. 13, no. 3, pp. 1222–1234, 2014.

[7] A. I. Ahmed, E. I. Shahin, L. A. Said, and A. H. Madian, "A scalable firmware-over-the-air architecture suitable for industrial iot applications," in *3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*. IEEE, 2021, pp. 228–231.

[8] "Mender," https://mender.io/how-it-works.

[9] E. Baccelli, J. Doerr, S. Kikuchi, F. A. Padilla, K. Schleiser, and I. Thomas, "Scripting over-the-air: Towards containers on low-end devices in the internet of things," in *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2018, pp. 504–507.

[10] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava, "Sensorware: Programming sensor networks beyond code update and querying," *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 386–412, 2007.

[11] J. Zhang, M. Ma, W. He, and P. Wang, "On-demand deployment for iot applications," *Journal of Systems Architecture*, vol. 111, p. 101794, 2020.

[12] N. Tziritas, G. Georgakoudis, S. Lalis, T. Paczesny, J. Domaszewicz, P. Lampsas, and T. Loukopoulos, "Middleware mechanisms for agent mobility in wireless sensor and actuator networks," in *3rd ICST Conference on Sensor Systems and Software (S-Cube)*. Springer, 2012, pp. 30–44.

[13] "zeromq," https://zeromq.org/.

[14] A. Stanford-Clark and H. L. Truong, "Mqtt for sensor networks (mqtt-sn) protocol specification," *International business machines (IBM) Corporation version*, vol. 1, no. 2, pp. 1–28, 2013.

[15] "Mqtt," https://mqtt.org/.