Adaptive Deployment of Application-level Sensing and Data Processing Pipelines in a Wireless Network of Embedded Devices

Giorgos Polychronis[1], Manos Koutsoubelias[1], Foivos Pournaropoulos[1], Spyros Lalis[1], Lefteris Georgiadis[2], Thomas Pazios[2], Stratos Tsatsaronis[2], and Isaias Vrakidis[2]

[1] Electrical and Computer Engineering Department
University of Thessaly
Volos, Greece
{gpolychronis, emkouts, spournar, lalis}@uth.gr
[2] Department of Research & Development
METIS Cybertechnology
Athens Greece
{lefteris.georgiadis, thomas.pazios,
stratos.tsatsaronis, isaias.vrakidis}@metis.tech

Abstract. Most IoT devices are nowadays equipped with computing resources so that – besides acting as plain sensor nodes – they can also perform local data processing, aggregation and filtering, before data is forwarded upstream to more powerful servers. In this paper, we present a framework for the flexible and adaptive deployment of application-level sensing and data processing pipelines in a network of such wireless sensor embedded nodes. The system administrator merely provides a high-level, declarative description of the services to be deployed. Based on this input, a helper facility performs a mapping of the specified application services to nodes so as to reduce the wireless network traffic. Furthermore, the service-to-node mapping can be adapted at runtime to handle changes in system configuration. We evaluate our approach for an indicative node topology and different application processing pipeline configurations. Our results show that such an optimized deployment can reduce wireless traffic by up to 51% vs a centralized placement, while the ability to adapt the placement of the data processing pipeline to system configuration changes can achieve up to $3.6x$ savings vs a static deployment that was optimal for a previous system configuration. Also, such adaptations can be performed fast, within a few tens of seconds.

Keywords: IoT · Wireless Sensor Networks · In-Network Processing · Edge Computing · Application Service Deployment · Adaptation.

1 Introduction

The continued advances in embedded computing platforms and short-range wireless communication technologies have enabled the development of wireless devices, which can be flexibly deployed in different kinds of infrastructures and can

support a wide range of sensing applications, giving rise to the so-called Internet of Things (IoT). While the main purpose of such devices is typically to collect measurements through a variety of sensors, most of them also have sufficient computing resources to process data locally, in the spirit of edge computing, before forwarding it to more powerful servers or the cloud.

The ability to process data directly on the sensor nodes, becomes even more important when such data is propagated over wireless links. In this case, support for in-network processing to perform aggregation and filtering is crucial to minimize the wireless traffic and increase the robustness of communication over low-bandwidth links. To achieve this, however, one must have a suitable placement of the data processing logic, depending on the location of the nodes that produce the raw data. Moreover, any static placement may turn out to be sub-optimal if the system configuration changes. Therefore, it is also important to be able to adapt the current deployment at runtime.

In this paper, we present support for such a flexible and adaptive deployment for an industrial-strength IoT system targeting the shipping domain. Our work has the following distinctive characteristics. First, it supports the deployment of entire pipelines of application-level data processing tasks on the wireless IoT nodes. Secondly, the placement of the individual data processing tasks can be adapted at runtime. Thirdly, deployment and adaptation can be performed automatically, without manual commands from the system administrator, based on high-level declarative descriptions.

The main contributions of our work are: (i) We present a complete framework for the flexible and adaptive deployment of application-level data processing pipelines in IoT systems. (ii) The desired deployment is captured in a declarative way, through suitable descriptions based on which the necessary actions are performed in an automated way. (iii) We evaluate the benefits of such flexible and adaptive deployment for various system configurations, showing that it can significantly reduce wireless traffic vs fully centralized or static deployments, with only a short downtime for the application due to the adaptation phase.

The rest of the paper is structured as follows. Section 2 provides an overview of the IoT system we target in our work, based on a concrete use-case from the shipping domain, and the available system support for the deployment of application-level sensing and data processing tasks on the wireless embedded nodes. Section 3 presents our approach for the flexible and adaptive deployment of application services in the IoT system, with minimal input from the administrator. Section 4 presents an evaluation of the proposed approach. Section 5 gives an overview of related work. Finally, Section 6 concludes the paper.

2 System Overview

This section provides an overview of the system we target in our work. We start by describing the physical system infrastructure. Then, we explain how sensing and data processing functionality is achieved by combining smaller application services into more complex data processing pipelines. Finally, we discuss

the core underlying system-level support for deploying and interconnecting such application-level services in the system.

2.1 Embedded nodes & wireless network

We target IoT systems comprising embedded devices used to collect information from a potentially wide variety of sensors. We assume that these nodes are interconnected via short-range wireless links in a multi-hop network that is deployed on the site of interest. As a concrete case, we consider a wireless sensor network installed on a vessel for monitoring and analyzing its status, including, e.g., engine operation, speed, fuel consumption, sea state, etc. We note that METIS already operates such systems in numerous vessels worldwide.

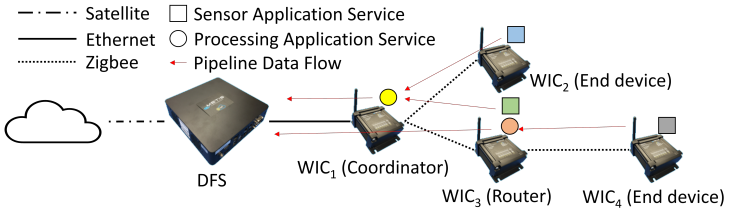


Fig. 1: Flexible deployment and execution of application services on the embedded nodes of the IoT system. Rectangles and ovals denote distinct sensor and processing services, respectively. The arrows between the application services indicate the data flow of the respective pipelines.

Figure 1 illustrates an indicative system setup. The basic building block of the system is the 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.

In terms of networking, WICs feature an Ethernet and Zigbee interface. Zigbee is used to deploy and interconnect WICs at various locations on the vessel in a fast and flexible way, without the need for any wiring (apart from being costly, this also raises the issue of safety if cables need to cross isolated compartments). The ZigBee network is managed by a coordinator WIC, while the rest act as router nodes or end nodes that do not perform any data forwarding. The coordinator WIC uses its Ethernet interface to connect to the so-called data fusion server (DFS) on the ship via the ship's local area network.

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). Note that stable Internet connectivity over satellite may not be available all times, thus the DFS may need to store data for longer periods of time until

this can be uploaded to the cloud. It is therefore important for the local IoT system to operate autonomously and as efficiently as possible even when being disconnected from the cloud.

2.2 Application services and data processing pipelines

To exploit the processing capacity of the embedded nodes, we depart from the paradigm of monolithic applications and adopt a distributed application approach. More specifically, the application logic is split into smaller components, referred to as services, which can be deployed on the available nodes (WICs), as shown in Figure 1. We differentiate between two types of application services: sensor and processing services (denoted in the figure as rectangles and ovals), respectively. Sensor services access specific sensors connected to the embedded nodes to collect measurements and forward them upstream, possibly after some filtering. Processing services collect data produced by one or more sensor services or other processing services to produce derivative, more complex information.

This makes it possible to build data processing pipelines that run in a distributed way on top of the IoT infrastructure. Two indicative examples are shown in Figure 1. One pipeline consists of the green and blue sensor services running on WIC_2 and WIC_3, and the yellow processing service running on WIC_1, which takes as input the values produced by those services to produce a higher-level metric. The other pipeline includes the grey sensor service on WIC_4 and the orange processing service running on WIC_3. Note that sensor services must run directly on the nodes featuring the respective sensors. In contrast, a processing service can run on any node that has sufficient computing resources to host it, including the DFS. For instance, the orange data processing service could be placed on WIC_4 to reduce the amount of data sent over the wireless network.

The formation of data processing pipelines is done indirectly, based on so-called quantity identifiers (QIDs) which uniquely associate data with specific sensor types or metrics produced by the processing services, and additional information that is needed to properly interpret and process the data. More specifically, instead of hard-wiring the components of a pipeline via explicit references or addresses, application services are loosely coupled through producer-consumer relationships captured via the respective QIDs.

For each application service, the information about the QIDs consumed and produced, the logic for parsing and processing data, and any special resources or parameter files required to run the service, is included in a corresponding description. In turn, such descriptions are used to drive the deployment and execution of application services on the embedded nodes.

2.3 Basic system-level support

The deployment of application services and the data exchange between them based on QIDs is supported via system-level software running on the embedded nodes (WICs) and the DFS. Figure 2 gives a high-level overview of the software architecture, which is discussed in some more detail below.

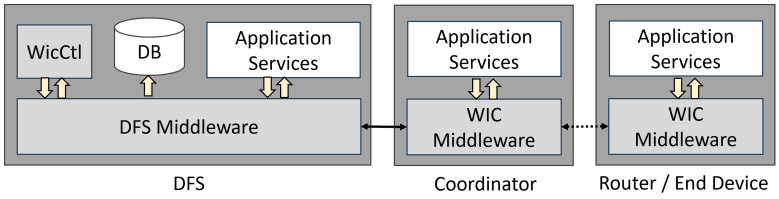


Fig. 2: High-level view of the system software architecture.

All interactions that occur in the system are supported via the middleware running on the DFS and the WIC nodes. The middleware implements a pub/sub transport layer, using a combination of ZeroMQ [2] and MQTT/MQTT-sn [1,15] to support the local interaction between entities running on the same node and the remote interaction between entities running on different nodes over Ethernet/ZigBee, respectively. If a message that is produced by a local entity has remote subscribers, the pub/sub layer will transparently forward it to the respective node(s) and will deliver it to the proper subscriber. Note that the data that is generated by application services flow from the WICs toward the DFS, while requests for system-level operations travel in the reverse direction.

The configuration and control of remote nodes from the DFS is supported by the WIC control tool (WicCtl). This works in a client-server fashion, in the spirit of a simple remote shell, allowing the system administrator to send files and run commands on one or more nodes over ZigBee. Among other things, WicCtl is used to deploy application services on the nodes by sending the service descriptions. The middleware on the WICs is responsible for handling such control/configuration commands, in particular regarding the execution of application services. More concretely, when a command is issued via WicCtl to start a given application service, the middleware on the target node launches a generic service execution process that runs the application logic according to the respective description. Note that the application description must already be available on the node (it can be pre-installed on the node, or sent to it over the wireless network via a WicCtl file transfer command). Conversely, when the runtime layer receives a request to stop an application service, it terminates the corresponding execution process.

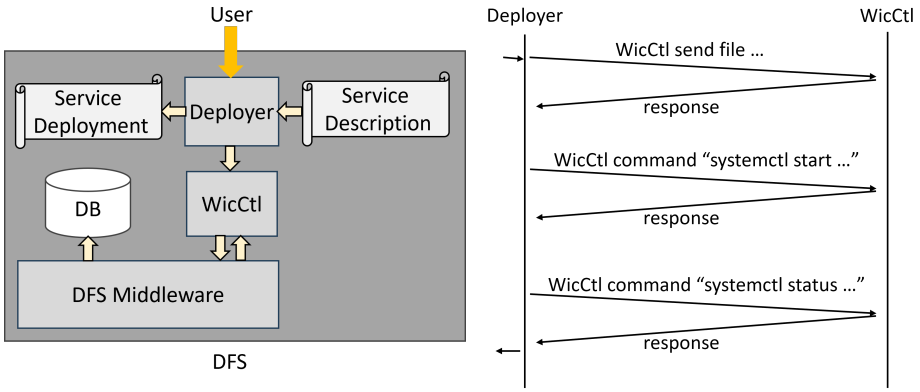
Furthermore, the middleware running on the DFS is configured to store the data that is generated from the application services in a local database. From there, it is uploaded to the cloud in an asynchronous way subject to satellite connectivity (a separate system service is used for this, not shown in the figure).

3 Flexible & Adaptive Application Service Deployment

3.1 Targeted service deployment

Using WicCtl, the system administrator can deploy/start or stop/remove arbitrary application services on specific nodes at any point in time. However, this

procedure can become quite awkward, especially if one wishes to deploy/remove several services or some commands fail (e.g., due to network instability) and have to be repeated.



(a) Addition of the Deployer facility on top of the basic system software stack.

(b) Command sequence (via WicCtl) for application service startup.

Fig. 3: Service deployment via the Deployer.

To simplify system management, we introduce the Deployer facility, shown in Figure 3a, which performs the desired application service deployment based on high-level directives/intents from the system administrator. The input is provided in the form of a json file, which contains the description of one or more deployment operations. An indicative example is given in Listing 1, for the deployment of a simple data processing pipeline consisting of a service that calculates the specific fuel oil consumption (SFOC) of the vessel based on engine power and fuel mass flow data produced by respective sensor services; note that the QIDs capturing the producer-consumer relationships are part of the respective service descriptions (not shown for brevity). Each operation consists of the desired action (deploy/start or stop/remove an application service), the service name, the names of the respective description file, and the target node. In some cases, besides the service description, additional files may need to be sent to the node, e.g., describing special input configuration or sensor data parsing parameters for the service in question. The Deployer automatically extracts such information by inspecting the service description and sends the respective files to the node together with the main service description file.

The Deployer executes the specified operations sequentially, by issuing the respective commands via WicCtl, as shown in Figure 3b. In case the node does not respond, the Deployer retries the command a number of times (the upper bound can be set by the administrator). If the failure persists, the operation is aborted and the Deployer returns a corresponding negative result without

Listing 1 Input file for deploying an indicative data processing pipeline. SFOC (Specific Fuel Oil Consumption) service using data produced by the EnginePower and FuelMassFlow services.

```
operations: [
    {
        serviceName: "EnginePower_svc"
        serviceDescription: "EnginePower_svc_desc"
        command: "start"
        destination: "WIC-2"
    },
    {
        serviceName: "FuelMassFlow_svc"
        serviceDescription: "FuelMassFlow_svc_desc"
        command: "start"
        destination: "WIC-3"
    },
    {
        serviceName: "SFOC_svc"
        serviceDescription: "SFOC_svc_desc"
        command: "start"
        destination: "WIC-1"
    }
]
```

proceeding to the next operation in the deployment description. Furthermore, the Deployer explicitly confirms the success of the requested operation by retrieving relevant status information from the node, again via WicCtl. Based on the outcome of the operations, the Deployer updates the current service deployment state for the entire system, i.e., the services hosted/running on each of the nodes. In our implementation, this information is kept/updated in a special file using a human-readable format so that it can be easily inspected by the system administrator (as well as by other programs, as will be discussed in the sequel).

3.2 Flexible service mapping & deployment

Sensor services must be deployed on the nodes that feature the respective sensors. These nodes are typically installed at very specific locations on the vessel hence are well-known to the system administrator. Therefore, it is straightforward to specify them in the description passed to the Deployer, as done for the EnginePower and FuelMassFlow services in Listing 1.

However, such restrictions do not apply to data processing services, such as the SFOC service in Listing 1, which could be deployed on any node of the system, provided it has sufficient resources. In this case, rather than having the administrator specify a concrete node (as done in Listing 1), it can be desirable to leave the target host open so that it can be selected automatically. Apart from reducing the burden of the system administrator, this makes it possible to optimize service placement in an automated way. Note that this can be hard/awkward to do manually in case multiple application services are arranged in multi-stage data processing pipelines that must be deployed and run concurrently in system configurations with a large number of nodes.

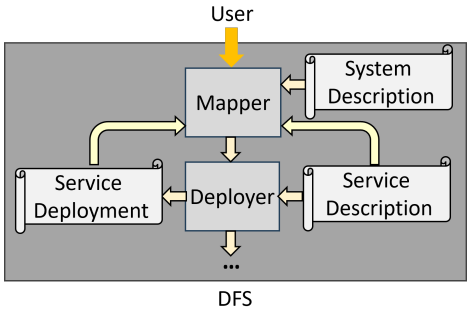


Fig. 4: Supporting flexible and adaptive service deployment via the Mapper.

This functionality is implemented through another facility, the Mapper, which operates on top of the Deployer as shown in Figure 4. Like the Deployer, the Mapper takes as input from the administrator a file describing the application services to be deployed. The difference is that, in this case, the target hosts of certain services can be left unspecified. The Mapper maps such non-anchored services to the available system infrastructure by finding a suitable host for them. Depending on the configuration, it can automatically implement the corresponding deployment plan by invoking the Deployer, or simply return it as a proposal to the system administrator.

3.3 Service mapping algorithm

The Mapper inspects the system description, service descriptions and current service deployment. It uses this information to populate its internal data structures, which are subsequently used to compute the service-to-node mapping. Table 1 and Table 2 summarize the information that is kept for each node and service, respectively.

The $s.svcProd$ and $s.isProd$ fields are computed by matching the input QIDs of s with the output QIDs of other services to find the respective producer-consumer relationships. Namely, $s.svcProd = \{s' : s.inQIDs \cap s'.outQIDs \neq \emptyset\}$, while $s.isProd = \exists s' : s.outQIDs \cap s'.inQIDS \neq \emptyset$. The $s.inRate$ field is computed by summing-up the output rates of all services that produce data consumed by s, taking into account the number of relevant QIDs. More formally, $s.inRate = \sum_{s' \in s.svcProd} s'.outRate \times |s.inQIDs \cap s'.outQIDs|$. Note that for sensor services $s.svcProd = \emptyset$ and $s.inRate = 0$. The $s.host$ field encodes the host where service s is mapped. Note that some services may already have preassigned hosts (recall that the sensor services must be installed on specific nodes). Such anchored services are ignored in the mapping operation, which only tries to map services with $s.host = NULL$. At the end of the mapping operation, the $s.host$ field of each non-anchored service will be assigned a node value. Finally, for each node n of the system, $n.resAvlMap$ is used in the mapping operation to capture the resources that can be used for the hosting of additional services.

Table 1: Node information based on the system and service descriptions.

Notation	Description
n	Node object.
$n.id$	Node identifier.
$n.resTot$	Total resource capacity for hosting services.
$n.resAlloc$	Resource capacity allocated to hosted services.
$n.parent$	Parent in the wireless routing structure.
$n.hops$	Number of wireless hops to DFS.
$n.resAvlMap$	Available resource capacity for service mapping.

Table 2: Service information based on the system and service descriptions.

Notation	Description
s	Service object.
$s.id$	Service identifier.
$s.resReq$	Resource capacity required to host s.
$s.outQIDs$	QIDs for which s produces data.
$s.inQIDs$	QIDs for which s consumes data.
$s.outRate$	Rate at which s outputs data (msgs/sec).
$s.svcProd$	Services that are data producers for s.
$s.isProd$	Whether s is a data producer for other services.
$s.inRate$	Aggregate input data rate from all producers.
$s.host$	Node where s is deployed or mapped.

When the Mapper receives a set of services as input, let *svcInput*, it forms the union of all services that are already deployed, let *svcDeployed*, and the set of non-anchored services in *svcInput* which have to be mapped to a node ($s \in svcInput \wedge s.host = NULL$), let *svcToMap*. It then verifies that the requested mapping has no unresolved references. In other words, for each service $s \in svcToMap$ it is checked that all services acting as data producers for it are already deployed or are part of the requested mapping: $s.svcProd \subseteq svcDeployed \cup svcInput$. Then, the Mapper proceeds with the actual mapping operation to find a suitable service-to-node mapping for the non-anchored services in *svcInput*. The optimization objective, subject to resource constraints/requirements, is to minimize the data traffic over the wireless network.

The heuristic for the mapping operation is given in Algorithm 1 in the form of pseudocode. The top-level function is Map(). As a first step, the mapping information is initialized and the services to be mapped are sorted according to their input data rates (in descending order) via function SortByInputRate(). The rationale is to give priority to the services consuming/ingesting the largest amount of data, hoping to map them on a host as close as possible to the respective data sources. Then, the algorithm incrementally maps each service to a node (or the DFS). In each iteration, it invokes function FindFirstSvcCovered() which scans the sorted list of unmapped services and returns the first service (with the highest aggregate input rate) whose data producers are already deployed or have been successfully mapped. Then, the best hosting option is found for this service, via function FindMinTrafficCostHost(), and the resource

Algorithm 1 Mapping non-anchored data processing services on nodes.

```
1: function Map(nodes, svcDeployed, svcInput)
2:     svcMapped ← {s ∈ svcInput : s.host ≠ NULL}                ▷ anchored services
3:     svcToMap ← svcInput − svcMapped  ▷ non-anchored services, to be mapped
4:     for each n ∈ nodes do                                      ▷ init service mapping info
5:         n.resAvlMap ← n.resTot − n.resAlloc
6:         for each s ∈ svcMapped where s.host = n do
7:             n.resAvlMap ← n.resAvlMap − s.resReq
8:         end for
9:     end for
10:    dfs.resAvlMap ← ∞                                          ▷ assume DFS has ample resources
11:    svcToMapSorted ← SortByInputRate(svcToMap)
12:    while svcToMapSorted ≠ ∅ do
13:        s ← FirstSvcCovered(svcToMapSorted, svcDeployed ∪ svcMapped)
14:        n ← FindMinTrafficCostHost(nodes + dfs, s)
15:        s.host ← n
16:        n.resAvlMap ← n.resAvlMap − s.resReq
17:        svcMapped ← svcMapped + s
18:        svcToMapSorted ← svcToMapSorted − s
19:    end while
20: end function

21: function FindFirstSvcCovered(svcToMapSorted, svcAvl)
22:    for each s ∈ svcToMapSorted do
23:        if s.svcProd ⊆ s.svcAvl then
24:            return s
25:        end if
26:    end for
27: end function

28: function FindMinTrafficCostHost(nodes, s)
29:    minCost, maxResAvl ← ∞, 0
30:    for each n ∈ nodes where n.resAvlMap ≥ s.resReq do
31:        cost ← 0
32:        for each s′ ∈ s.svcProd do
33:            cost ← cost + OutputTrafficCost(s, n, s′)
34:        end for
35:        if ¬s.isProd then                                      ▷ add output cost to DFS
36:            cost ← cost + OutputTrafficCost(s, n, NULL)
37:        end if
38:        resAvl ← n.resAvlMap − s.resReq
39:        if cost < minCost ∨ (cost = minCost ∧ resAvl > maxResAvl) then
40:            host, minCost, maxResAvl ← n, cost, resAvl
41:        end if
42:    end for
43:    return host
44: end function

45: function OutputTrafficCost(s, n, s′)
46:    if s′ ≠ NULL then          ▷ cost for data produced by s′ for s hosted on n
47:        return s′.outRate × |s.inQIDs ∩ s′.outQIDs| × hops(n, s′.host)
48:    else                ▷ cost for data produced by s hosted on n to reach the DFS
49:        return s.outRate × |s.outQIDs| × hops(n, dfs)
50:    end if
51: end function
```

availability of that node is updated accordingly. Note that the DFS is modeled as a node with infinite resources, $n.resAvl = \infty$, so there will always be at least one available node to host a service (in other words, it is guaranteed that the mapping problem is not infeasible).

Function FindMinTrafficCostHost() finds the node n that has sufficient available resource capacity to host the service in question while incurring the lowest traffic cost over the wireless network. The traffic cost is calculated by summing-up the data traffic caused by each service s' that is a data producer for s. In turn, this cost is calculated via OutputTrafficCost() by multiplying the respective output rate $s'.outRate$ by the number of QIDs consumed by s and the number of hops between node n and the host of the data producer $s'.host$. Moreover, if s does not serve as a data producer for another service, its own output data traffic toward the DFS is added to the total cost. This is because this data is not consumed by another service (running on a node in the wireless sensor network) but simply ends-up in the main database on the DFS.

Notably, Map() returns a *proposed* deployment. To implement the suggested deployment, the Mapper subsequently invokes the Deployer. This can be done incrementally by invoking the Deployer separately for each service, or via a single invocation passing as input the complete deployment plan.

3.4 Adapting the current service deployment

In addition to the initial flexible mapping and deployment of application services on the nodes of the system, it can be desirable to adapt the current deployment at runtime. For instance, new nodes may be added in the system that can be used to host application services, the topology / routing structure of the wireless network may change as a side effect of changing the placement of some nodes, or certain services may change their data rates. Such changes may render the current deployment sub-optimal.

The Mapper can be configured to support such adaptation. In this case, after performing the initial deployment, it periodically checks the system description to detect changes or it can be manually invoked by the administrator after such changes occur. If any changes are detected, the Mapper re-computes a new mapping of all processing services using the Map() function. Then, the total data traffic over the wireless network for the current and new mapping is calculated in the spirit of function OutputTrafficCost() in Algorithm 1, for every producer-consumer service pair of the data processing pipelines.

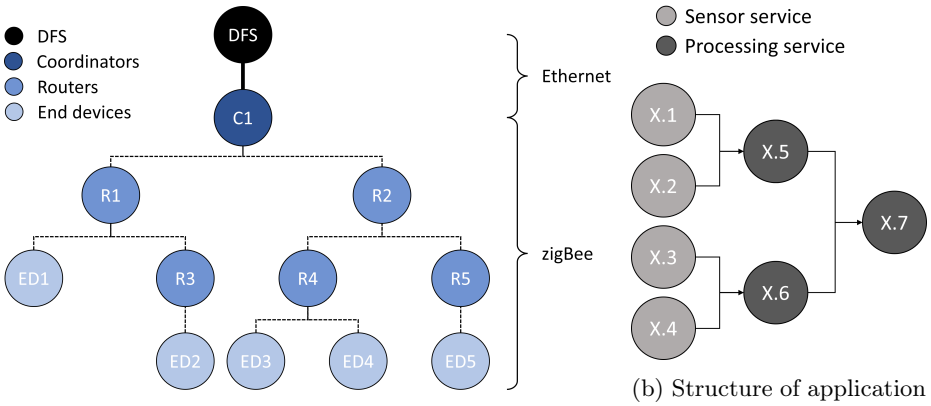
Finally, the decision whether to actually implement the new mapping is taken based on an improvement threshold that is set as a configuration parameter by the system administrator. If the relative improvement of the new (planned) vs the current service deployment exceeds this threshold, the Mapper proceeds to implement the new mapping by invoking the Deployer. Alternatively, the Mapper can be configured to suggest the adapted deployment to the system administrator, who can examine the proposed mapping to decide whether to adopt the proposal. In this case, the administrator can implement the respective deployment, if desired, by invoking the Deployer manually.

4 Evaluation

We evaluate the quality of service-to-node mapping produced by our algorithm for different system configurations with several nodes and indicative data processing pipelines. First, we describe the test configurations and some alternative service mapping approaches which we use as benchmarks for our algorithm. Then, we present and discuss the obtained results.

4.1 System configuration

We perform experiments for a system configuration with eleven embedded IoT nodes, arranged in the topology shown in Figure 5a. We investigate the case where the system administrator wishes to deploy two data processing pipelines. Both have the same structure, shown in Figure 5b, but rely on different sensors and corresponding sensor services.



(a) System topology.

(b) Structure of application data processing pipelines.

Fig. 5: Experimental setup.

We explore three scenarios regarding the embedded nodes that feature the sensors that are required by the sensor services, given in Table 3. Recall that sensor services must be deployed on the nodes that have the corresponding sensors, thus the location of sensors determines the placement/mapping of the respective sensor services. Also, we investigate three scenarios for the data production rates of the application data processing pipelines, given in Table 4.

Finally, we experiment with three different service hosting capacity scenarios, assuming that each embedded node can host 2, 3 or 4 application services (including the sensor services). As an exception, the DFS can host an unlimited number of application services; for all practical purposes, it is considered to have infinite service hosting capacity.

Table 3: Sensor locations.

Sensors	Nodes 1	Nodes 2	Nodes 3
1.1	R5	R4	ED3
1.2	ED1	R3	ED5
1.3	ED2	ED4	ED4
1.4	ED3	ED5	ED2
2.1	R3	ED2	R4
2.2	R4	R5	R3
2.3	ED4	ED1	R4
2.4	ED5	ED3	R5

Table 4: Data rates (msg/sec).

Services	Rates A	Rates B	Rates C
X.1	1	0.25	1
X.2	0.25	0.25	1
X.3	1	1	1
X.4	0.25	1	1
X.5	1	0.25	0.25
X.6	0.25	1	0.25
X.7	0.25	0.25	0.25

4.2 Benchmarks

We compare our heuristic with the following benchmarks:

(1) **Proximity:** A data processing service is placed as close as possible to its sources, by selecting hosts based on their average distance with the nodes that run the respective producer services. Ties are broken by picking the host closest to DFS.
(2) **Optimal:** Data processing services are mapped optimally on the nodes by running an exhaustive search algorithm that checks all possible placement options. This always produces the best possible solution.
(3) **DFSonly:** All data processing services are simply placed on the DFS. No data processing takes place on the embedded nodes / in the wireless network. This is used as a baseline for the results obtained via the proposed approach and the rest of the benchmarks.

The metric of comparison is the aggregate data traffic generated over the wireless network when running both processing pipelines concurrently. This is calculated as explained in Section 3.4, on the service-to-node mapping produced by each approach. In the following, all results are presented by reporting the relative reduction of wireless message traffic vs DFSonly.

4.3 Benchmark comparison

In a first set of experiments, we compare our algorithm with the above benchmarks for the deployment of the two application processing pipelines on the system, for each of the 27 combinations of the sensor placement, production rate and resource capacity scenarios. The results are shown in Figure 6. Each row corresponds to a different sensor placement scenario in Table 3, while each column corresponds to a different data rate scenario in Table 4. The code at the top left corner of each plot indicates the specific sensor placement / production rate combination. In each case, the different resource capacity scenarios are shown along the x-axis, while the y-axis shows the reduction of wireless traffic achieved vs the DFSonly approach (higher is better). As it can be seen, the proposed

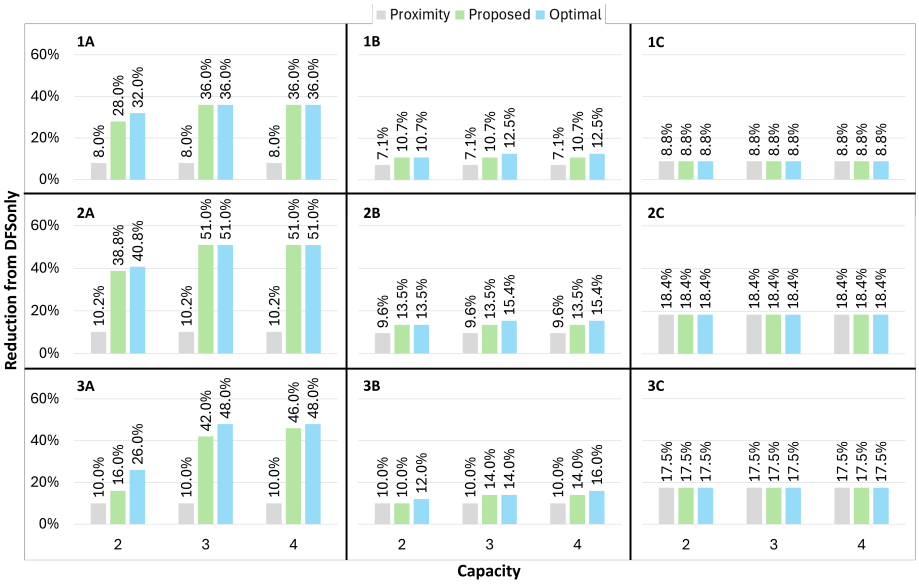


Fig. 6: Reduction of wireless message traffic vs DFSonly. Each row (1, 2, 3) corresponds to a different service placement scenario as per Table 3. Each column (A, B, C) corresponds to a different data production rate scenario as per Table 4.

heuristic outperforms the proximity heuristic in most of the scenarios and has the same performance in all the rest; it is also optimal or close to optimal for sufficient service hosting capacity of the nodes.

The difference between the two heuristics is significant in column A of Figure 6, where there is greater asymmetry in the production rates of the services that send input data to the same processing service. This is because the proposed heuristic tends to map each processing service on nodes that are closer to the source with the higher rate, while the proximity heuristic picks hosts that are located between the sources ignoring their production rates. An important observation is that the latter does not take advantage of increased resource availability of the nodes. The reason is that if the sources of a processing service are located in different subtrees of the wireless network, the service will be placed on the root node of these subtrees, or (if this does not have sufficient resources) on a node that is even closer to the DFS or on the DFS itself. Apart from achieving only a small wireless traffic reduction vs DFSonly, this fails to exploit the hosting capacity of the subtrees. In contrast, the proposed heuristic achieves consistently better deployments as the node hosting capacity increases. More specifically, the proximity heuristic reduces the wireless traffic merely by 8% - 10.2% vs DFSonly, whereas the proposed heuristic achieves a substantial reduction of 36% - 51%, up to $5x$ more improved deployment vs the simple proximity heuristic. In fact, in scenarios 1A and 2A, the proposed heuristic manages to find the optimal

solution already when the nodes have hosting capacity 3, while in scenario 3A it produces a service-to-node mapping that is very close to the optimal, with a difference of just 6% and 2% for capacity 3 and 4, respectively.

In the column B of Figure 6 showing the results for the Rates B scenarios, data rate asymmetry exists only between services X.5 and X.6 that produce data used as input for X.7. As a consequence, even the optimal heuristic achieves a rather small reduction of wireless traffic vs DFSonly. This indicates that there is only a small improvement opportunity. Therefore, both the proposed heuristic and the proximity heuristic get stuck in local optima, failing to improve the service-to-node mapping when the capacity of the nodes increases. As an exception, in scenario 3B, the proposed heuristic produces an improved deployment plan when capacity increases from 2 to 3. Notably, the proposed heuristic again outperforms the proximity heuristic in all cases (except one where they produce equal service-to-node mappings). In cases where the service hosting capacity is 4, it achieves a traffic reduction of up to 10.7% - 14% vs DFSonly, while the proximity heuristic achieves a reduction of just 7.1% - 10%. These small improvements are justified given the small room for improvement. Nevertheless, the proposed heuristic manages to perform close to optimal in all cases where the capacity is larger than 2, producing deployment plans that are on average about $1.4x$ better than those of the Proximity heuristic.

In column C of Figure 6, the proposed heuristic but also the simpler proximity heuristic both produce optimal deployment plans in all cases. This is because there is full symmetry in the data rates between the services that produce data for the same processing service, hence there is no benefit in placing a processing service closer to one of its sources. As a result, for all processing services whose sources are hosted on nodes in different subtrees of the routing hierarchy, the best hosting option is to host them on the node that is the root of the these subtrees. In several cases, the best hosting option for such services is the DFS itself, which has infinite hosting capacity. In turn, this relaxes the capacity pressure on all other nodes, which can host the remaining services optimally.

4.4 Adaptation

The system configuration may vary during the lifetime of an application processing pipeline, which may be required to run for a very long time. For example, the system administrator may wish to increase the rate at which sensor services sample the respective sensors or the rate at which processing services generate data upstream towards the DFS or other processing services. Such changes may render the current service deployment non-optimal.

In the next set of experiments, we explore the inefficiency of static service placement in case of changes in the system configuration. To this end, we use sensor placement A from Table 3 and a range of different system configurations, given in Table 5, regarding the data rates of each service in the application processing pipelines running in the system.

Figure 7 shows the results for the scenario where the system starts from configuration 1 and successively goes through configurations 2 to 8. The initial

Table 5: Different data rate configurations for the two application pipelines.

Services	Data Rate Configurations (msg/s)							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
X.1	1	1	1	1	0.25	0.25	0.25	0.25
X.2	0.25	0.25	0.25	0.25	1	1	1	1
X.3	1	1	0.25	0.25	1	1	0.25	0.25
X.4	0.25	0.25	1	1	0.25	0.25	1	1
X.5	1	0.25	1	0.25	1	0.25	1	0.25
X.6	0.25	1	0.25	1	0.25	1	0.25	1
X.7	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25

service deployment plan as produced by the Mapper (using Algorithm 1) is optimal for the first configuration. The red line plots the wireless traffic assuming this deployment remains static throughout all configuration changes. As a reference, the black line shows the wireless traffic for DFSonly; this remains constant because the sum of the service data rates is the same in all configurations and all processing services are placed on the DFS. One can clearly observe that the initial deployment can become significantly inefficient depending on the system configuration changes that occur in the sequel, up to $2.3x$ worse than DFSonly.

As mentioned in Section 3, the Mapper can be configured to adapt service placement by invoking the Deployer if the estimated improvement exceeds a given threshold. Figure 7 shows the resulting wireless traffic for such an adaptive deployment using three different threshold settings 0, 0.3 and 0.5 (green, blue and purple dashed line, respectively). Lower thresholds lead to more frequent adaptations of service deployment. The most aggressive threshold of 0 triggers an adaptation at every configuration change and manages to reduce wireless traffic up to $3.6x$ vs a static service deployment. Larger thresholds naturally lead to fewer transitions, resulting in higher wireless traffic for certain configurations. Note that the 0.5 threshold is marginally beneficial, as in some cases it does not manage to keep the load of the wireless network traffic below that of DFSonly. Overall, these results illustrate the importance of adapting the placement of application services to the current system configuration.

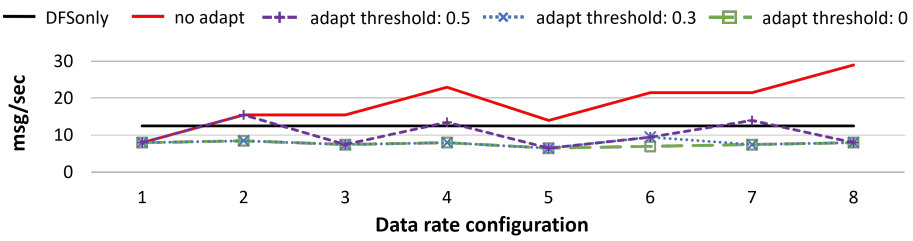


Fig. 7: Wireless traffic for the different data rate configurations in Table 5.

4.5 Adaptation delay

Adapting the service placement can reduce the wireless network traffic, but may also lead to application downtime. Namely, the data processing pipeline will remain suspended as long one of its services is being moved to another host. In the worst case, this may extend to the full duration of the adaptation phase. Next, we discuss how to compute a rough estimate of the adaptation delay.

Let $startT(k)$ and $stopT(k)$ be the time needed for the Deployer to perform a service start and stop operation on a node that is k hops away from the DFS (recall that these operations translate to a sequence of file transfer, service start/stop, and check commands to the target node). We have experimentally measured these overheads in our lab setup. For $startT$ we assume a service description file of 400 bytes, which is typical for the services running in practice.

Further, let $P.s.host$ denote the node that hosts service s under placement P, function $hops(P, s) = hops(DFS, P.s.host)$ return the number of hops between the DFS and $P.s.host$, and $P1 \bigotimes P2 = \{s : P1.s.host \neq P2.s.host\}$ denote the set of services that have a different host in two placements $P1$ and $P2$. Then, the total time that is required to perform an adaptation, i.e., to make the transition from service placement $P1$ to a new placement $P2$, can be calculated as

$$adaptT(P1, P2) = \sum_{s \in P1 \bigotimes P2} stopT(hops(P1, s)) + startT(hops(P2, s))$$

Table 6 gives the estimated adaptation delay for the sequence of configuration changes in Figure 7 for each of the adaptation thresholds. Recall that thresholds 0.3 and 0.5 do not lead to an adaptation in every configuration change hence in some cases the delay is 0. As it can be seen, the overhead is quite small, in the order of a few tens of seconds. This is perfectly acceptable for systems where major configuration changes are expected to be rather infrequent.

Table 6: Adaptation delay for the series of configuration changes in Figure 7.

Threshold	**Adaptation Delay (sec)**						
	1 → 2	**2 → 3**	**3 → 4**	**4 → 5**	**5 → 6**	**6 → 7**	**7 → 8**
0.0	7.5	14.8	7.5	21.6	7.5	14.8	7.5
0.3	7.5	14.8	7.5	21.6	0	7.8	7.5
0.5	0	7.6	0	21	0	0	15

We have verified through experiments in our testbed that the estimated delay produced with the above formula is very close to the time it takes to perform the corresponding adaptation of service placement in reality, with a small inaccuracy of about 5%. Consequently, the formula is a useful tool which can be used by the system administrator to anticipate the delay of a proposed adaptation and the potential downtime of the respective application pipelines.

5 Related Work

There is a significant amount of work focusing in the placement of operators (processing tasks) in the fog/edge also combined with or without the cloud. Such applications have the form of directed acyclic graphs, where the nodes are the operators and edges denote the producer-consumer relationships between operators. The authors of [14] study the problem of operator placement for the edge-cloud with the objective to optimize the end-to-end latency and deployment costs. They propose a MILP model with a technique to reduce the search space. Similarly, [13] has the objective to minimize the latency and proposes a heuristic to tackle the problem with two variants for reducing the search space to take faster placement decisions. In [10], different placement heuristics, such as a greedy approach and a local search starting from an initial greedily-built solution, are presented and evaluated for different single or multi objective goals including application response time, application availability and network usage. [7] deals with a similar problem, which is placing operators in the fog but in addition optimizes the placement periodically. There are also works addressing the placement problem with the objective to reduce the amount of data sent over the network. In [11], authors focus on the problem of service placement in the fog while trying to minimize the application delay, network usage and cloud placements. They propose a genetic algorithm to tackle the problem. Another placement problem is studied in [8] where multiple sensors generate data and the goal is to find fog nodes close to the sources to store the data sources, so that the network usage is minimized. The proposed solution considers the generation rate of each data type and picks for each placement the best node given various centrality indexes. Our work also focuses in a placement problem where application-level data processing pipelines are deployed inside a WSN so that the wireless traffic is minimized. Similarly to [13] and [8] we propose a solution that greedily picks the most suitable node for the placement of each processing task, but this is tailored for our particular system/application model and objective.

Another topic well studied in the literature is data aggregation in sensor networks [3]. Aggregators are placed inside the network with most common objectives to reduce the traffic and extend the network's lifetime. A data aggregation approach is studied in [12] for networks arranged in clusters. Authors in [18] consider networks where sensors generate different type of data, while data of the same type can be aggregated. They propose a protocol to route the data which also considers the aggregation of the data. These works aim to extend the lifetime of the network. In [17], the objective is to improve the service latency by reducing the amount of data transferred by applying in-network filtering. [6] proposes an in-network outlier detection. Another in-network approach is investigated in [9] with objective to increase transmission opportunities. Our work also supports processing data inside the network to reduce the messages sent over the wireless network, via application-level services that can be dynamically deployed on and moved between nodes.

There is a wide range of work on the dynamic deployment of code in WSNs. In [4] IoT devices execute scripts on a lightweight container that runs on top

of a lightweight OS. Sensorware [5] provides a runtime environment for flexibly deploying and executing application-level scripts. In the middleware presented in [19] application tasks extend the basic system functionality, while in [16] the application has the form of agents that are dynamically instantiated in a WSN to support in-network processing. In these works, the application logic is given in the form of scripts, while in our case the application services are captured in the form of purely declarative descriptions that are parsed on the node in order to configure a generic service execution engine so that it performs the required sensing and processing operations. Notably, the so-called generic agents in [16] are similar to the data processing services in our work, allowing such application logic to change its location in the WSN so as to reduce wireless network traffic.

6 Conclusion

We have presented work on how to automatically deploy and adapt the placement of application-level data processing pipelines in a wireless network of embedded sensor devices so as to minimize the data traffic over the wireless network. Our evaluation shows that optimized deployment can reduce radically wireless traffic up to 51% vs a simple centralized placement at the root of the wireless network, and is close to optimal in most cases. Also, adapting the deployment to system configuration changes can reduce the wireless traffic up to $3.6x$ vs a previously optimal placement that remains static during execution, while such adaptations can be performed fast, within just a few tens of seconds.

Our software architecture is modular, allowing extensions to be introduced in a flexible way. For instance, the service-to-node mapping logic could be changed to balance the services running on the nodes and the amount of data transmitted by each node to achieve an even resource utilization and energy consumption. Another possible optimization objective is to minimize contention on the wireless channel or to reduce the end-to-end latency of data production.

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. Mqtt. `https://mqtt.org/`
2. zeromq. `https://zeromq.org/`
3. Abbasian Dehkordi, S., Farajzadeh, K., Rezazadeh, J., Farahbakhsh, R., Sandrasegaran, K., Abbasian Dehkordi, M.: A survey on data aggregation techniques in iot sensor networks. Wireless Networks **26**, 1243–1263 (2020)

4. Baccelli, E., Doerr, J., Kikuchi, S., Padilla, F.A., Schleiser, K., Thomas, I.: Scripting over-the-air: Towards containers on low-end devices in the internet of things. In: International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops). pp. 504–507. IEEE (2018)
5. Boulis, A., Han, C.C., Shea, R., Srivastava, M.B.: Sensorware: Programming sensor networks beyond code update and querying. Pervasive and Mobile Computing **3**(4), 386–412 (2007)
6. Branch, J.W., Giannella, C., Szymanski, B., Wolff, R., Kargupta, H.: In-network outlier detection in wireless sensor networks. Knowledge and information systems **34**, 23–54 (2013)
7. Hiessl, T., Karagiannis, V., Hochreiner, C., Schulte, S., Nardelli, M.: Optimal placement of stream processing operators in the fog. In: 3rd International Conference on Fog and Edge Computing (ICFEC). pp. 1–10. IEEE (2019)
8. Lera, I., Guerrero, C., Juiz, C.: Comparing centrality indices for network usage optimization of data placement policies in fog devices. In: 3rd International Conference on Fog and Mobile Edge Computing (FMEC). pp. 115–122. IEEE (2018)
9. Lin, S.C., Chen, K.C.: Improving spectrum efficiency via in-network computations in cognitive radio sensor networks. IEEE Transactions on wireless communications **13**(3), 1222–1234 (2014)
10. Nardelli, M., Cardellini, V., Grassi, V., Presti, F.L.: Efficient operator placement for distributed data stream processing applications. IEEE Transactions on Parallel and Distributed Systems **30**(8), 1753–1767 (2019)
11. Sarrafzade, N., Entezari-Maleki, R., Sousa, L.: A genetic-based approach for service placement in fog computing. The Journal of Supercomputing **78**(8), 10854–10875 (2022)
12. Shobana, M., Sabitha, R., Karthik, S.: Cluster-based systematic data aggregation model (csdam) for real-time data processing in large-scale wsn. Wireless Personal Communications **117**, 2865–2883 (2021)
13. da Silva Veith, A., de Assuncao, M.D., Lefevre, L.: Latency-aware strategies for deploying data stream processing applications on large cloud-edge infrastructure. IEEE transactions on cloud computing (2021)
14. de Souza, F.R., da Silva Veith, A., Dias de Assunção, M., Caron, E.: Scalable joint optimization of placement and parallelism of data stream processing applications on cloud-edge infrastructure. In: Service-Oriented Computing: 18th International Conference. pp. 149–164. Springer (2020)
15. Stanford-Clark, A., Truong, H.L.: Mqtt for sensor networks (mqtt-sn) protocol specification. International business machines (IBM) Corporation version **1**(2), 1–28 (2013)
16. Tziritas, N., Georgakoudis, G., Lalis, S., Paczesny, T., Domaszewicz, J., Lampsas, P., Loukopoulos, T.: Middleware mechanisms for agent mobility in wireless sensor and actuator networks. In: 3rd ICST Conference on Sensor Systems and Software (S-Cube). pp. 30–44. Springer (2012)
17. Wu, H., He, J., Tömösközi, M., Xiang, Z., Fitzek, F.H.: In-network processing for low-latency industrial anomaly detection in softwarized networks. In: Global Communications Conference (GLOBECOM). pp. 01–07. IEEE (2021)
18. Yun, W.K., Yoo, S.J.: Q-learning-based data-aggregation-aware energy-efficient routing protocol for wireless sensor networks. IEEE Access **9**, 10737–10750 (2021)
19. Zhang, J., Ma, M., He, W., Wang, P.: On-demand deployment for iot applications. Journal of Systems Architecture **111**, 101794 (2020)