Journal of Internet Services and Applications, 2023, 14:1, doi: 10.5753/jisa.2023.3084

STEER: An Architecture to Support Self-adaptive IoT Networks for Indoor Monitoring Applications

Bruna M. O. S. Cordeiro [**Federal University of Goiás** | ***brunamos.bm@gmail.com***]
Roberto Rodrigues Filho [**University of Campinas** | ***robertor@ic.unicamp.br***]
Iwens G. S. Júnior [**Federal University of Goiás** | ***iwens@inf.ufg.br***]
Fábio M. Costa [**Federal University of Goiás** | ***fmc@inf.ufg.br***]

Instituto de Informática, Universidade Federal de Goiás, Alameda Palmeiras, Quadra D, Campus Samambaia, Goiânia, GO, 74690-900, Brazil.

Received: 29 December 2022 • **Accepted:** 16 May 2023 • **Published:** 27 July 2023

Abstract IoT infrastructures are becoming increasingly more difficult to manage. One of the main issues is the high volatility present in the infrastruture, which increasingly demands self-adaptive solutions. As a proposal to handle this challenge, this paper presents STEER (Sdn-based inTEnt drivEn iot netwoRks), a new approach for the dynamic adaptation of IoT networks for indoor monitoring applications, based on the unification of Intent-Driven Networks (IDN) and Software-Defined Networks (SDN). Particularly, we explore the ability of IDNs to dynamically interpret an application's intent, using an IDN-based mediator attached to an SDN-controller to autonomously adapt the IoT network behavior at runtime, thus realizing the intent according to the current operating context of the network. We demonstrate the approach using a representative application scenario related to IoT indoor environment monitoring in the domain of indoor air quality monitoring. The experiments allowed us to validate the applicability of the approach and show the system-wide effect of dynamic adaptation to the current operating environment on improving performance according to the metric under consideration, in this case, the number of application-level messages exchanged in the network.

Keywords: Intent-Driven Networks, Software-Defined Networks, Internet of Things, Self-Adaptive Systems

1 Introduction

IoT infrastructures are becoming more and more complex and challenging to manage. One of the main reasons for this complexity is the high volatility of contemporary distributed systems (Blair (2018)), notably caused by frequent changes in the network environment. In order to preserve functionality and satisfy application demands, IoT networks, therefore, need to adapt at runtime. Such adaptation must consider the application's goals as it interacts with the network and the application constraints as the operational environment changes. Thus, autonomic approaches must be in place to enable the network to determine the need to change and rapidly adapt to better configurations without human intervention.

In this context, approaches such as *self-driving networks*, as in Mai *et al*. (2021), have gained popularity. They mainly focus on mechanisms to enable runtime adaptation of the network without disrupting its services, such as classification of the network execution context and the use of machine learning to predict change and to plan adaptations accordingly. However, an important, albeit less explored aspect of the decision-making process refers to the goals of the applications that interact with the network.

By combining the concepts of Software-Defined Networks (SDN) and Intent-Driven Networks (IDN), this paper proposes a novel approach for programmable self-adaptive IoT networks. The approach, which we call STEER (*Sdn-based inTEnt drivEn iot netwoRks*), enables self-adaptation of the network at runtime, without requiring predefined rules and taking into account only the network operational context and high-level goals, referred to as *intents*, set by the applications using a declarative syntax. We exploit the IDN concept to enable interpretation of the application intents, along with SDN as the mechanism to adapt network behavior accordingly. Our implementation of IDN is centered around a *Mediator*, which interprets application intents to decide, at runtime, which behavior should be installed on the network. Behavior installation, in turn, is carried out by the Mediator using a third-party SDN controller.

The *Mediator* implements a control loop for selecting the network behavior that yields maximum performance, considering a given performance metric, the network environment, and an application intent. The network behavior selection algorithm selects the optimal behavior without human interference or predefined domain-specific information. The algorithm has two main phases, *exploration* and *exploitation*. The exploration phase is responsible for locating the optimal behavior by executing the available behaviors and monitoring their performance. The exploitation phase, in turn, executes the optimal behavior and monitors it to detect performance degradation and changes in the network operating environment, which in turn may trigger a new exploration.

We evaluate our approach in the context of indoor monitoring applications. Particularly, we use a real dataset from an indoor air quality application to create representative scenarios to explore our approach. The proposed solution addresses self-adaptation in such scenarios and is motivated by the wide adoption of networked sensors to monitor indoor environments, coupled with their demands for dynamic adaptation. In this context, we evaluate our selection algorithm in terms of convergence time (*i.e.*, how long it takes to converge to the optimal behavior) and accuracy (*i.e.*, if the *Mediator* selects the correct behavior). We also evaluate the overhead in the network performance incurred by network behavior adaptation. The findings suggest that our end-to-end approach manages to accurately select the optimal behavior and detect environment changes for the chosen scenarios. We also discuss the different parameter settings for the proposed algorithm and their effect to lower the overhead as measured by the chosen performance metric.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes our approach for SDN-based intent-driven IoT networks, followed by the proposed architecture to realize it. Section 4 describes an application scenario and use case in the domain of air quality monitoring, which we use to both illustrate the approach and evaluate its performance. Section 5 presents the results obtained through a set of experiments that use the air quality monitoring application with real data on a simulated network environment. Section 7 concludes the paper and discusses future work.

2 Background and Related Work

This section introduces important terms used throughout the text and contrasts the proposed approach with other relevant and related work in the literature.

This work focuses on providing an approach to support self-adaptive IoT (Internet of Things) Networks. According to Gubbi *et al.*Gubbi *et al.* (2013), IoT refers to a collection of objects with networking capabilities able to exchange information with one another. In this context, an IoT network is a local network formed out of the connection of such devices. Moreover, the self-adaptation of such networks (e.g., Min *et al.* (2021)) entails the ability to change the behavior of its devices as (often unexpected) events happen. Enabling self-adaption is becoming increasingly important as the scale at which these devices are deployed is becoming increasingly large (*e.g.*, at city scale, as described by Zanella *et al.* (2014)), making human-led decision-making approaches infeasible for managing and adapting the network.

As we describe later in this section, many techniques have been applied to allow the adaptation of behavior in IoT networks. In our approach, detailed in Sec. 3, we apply an increasingly popular approach for adapting IoT network behavior (see Bera *et al.* (2017)), namely Software Defined-Networks (SDN). In general, the concept of SDN, as discussed by (Kreutz *et al.* (2015)) separates the network data plane (that forwards data packets) from the control plane (a logically centralized module that defines how to handle network traffic), allowing more flexibility in adapting the network to handle dynamic traffic. Furthermore, we explore the concept of Intent-Driven Network, as defined by Elkhatib *et al.* (2017), which consists in the application defining how it intends to exchange data in the network and making such intents available to the network so that it can adapt to better suit the generated traffic. This enables a promising alternative to the current opaque traffic generated by applications, about which the network has no information and thus cannot adapt itself to better manage traffic. In Section 3, we describe our vision to unify both approaches in order to support self-adaptive IoT networks.

The remaining of this section discusses and compares relevant work in the literature that falls within three key areas for realizing our approach: adaptation mechanisms for IoT networks (Sec. 2.1) ; autonomic management of IoT networks (Sec. 2.2); and intent-driven networks (Sec. 2.3).

2.1 Adaptation Mechanisms

Adaptation in IoT networks may happen at different levels on network devices: at the operating system level, as in Rodriguez-Zurrunero *et al.* (2018); at the level of communication protocols, as in Junior *et al.* (2020); and at the application level, as in Shafi *et al.* (2012) and Aschenbruck *et al.* (2012). Rodriguez-Zurrunero *et al.* (2018) describe an adaptive operating system for wireless sensor networks that enables the adaptation of modules running on the OS to reduce the use of computing resources. By disabling modules, the running system reduces CPU and memory utilization. Junior *et al.* (2020), on the other hand, explore changes in the RPL routing algorithm. The proposed solution is able to switch between existing RPL instances to accommodate sudden demands of temporary critical applications. When a critical application is started, the scheduler changes the RPL instance for regular applications and assigns a new instance to the critical application, meeting the requirements of both kinds of applications. Finally, Shafi *et al.* (2012) and Aschenbruck *et al.* (2012) describe an over-the-air code-shipping approach to reprogram nodes in the network.

There are also approaches that enable (re)programming of the network behavior as a whole, as in Noor *et al.* (2019) and Azzara *et al.* (2014). Both introduce macroprogramming language models that abstract the programming of the network by abstracting the interaction with the nodes. These abstractions allow developers to focus on the network behavior as a whole, instead of focusing on fine-grain coordination of node tasks to achieve the intended network behavior, thus facilitating the creation of dynamic and large-scale IoT systems.

Moreover, as an alternative mechanism for changing network behavior, the use of SDN for IoT networks is becoming increasingly popular. Centralized controllers calculate and specify flow tables in IoT devices to determine how incoming packets should be routed. Controllers are also able to recalculate and change network data routes according to new data flows and events in the network (Galluccio *et al.* (2015); Bera *et al.* (2017)). In our work, we leverage the logically centralized controllers to collect metrics from the IoT network and install behavior to change IoT devices at the application level. Although the above approaches deal with OS-level adaptation, they may be seen as complementary to ours as our mechanism to adapt application-level behavior could also be used to change OS-level components. On the other hand, we believe the use of SDN makes our approach more generic as compared to applying adaptation to a specific routing algorithm (e.g., adaptive RPL).

2.2 Autonomic Management of IoT Networks

Autonomic management of IoT networks has become a topic of interest in the literature. This is due to the volatility (*i.e.*, constant and often unexpected changes) of IoT infrastructures, which makes rapid and assertive network adaptation of paramount importance. In such changing large-scale infrastructures, human-centric approaches to support reconfiguration are insufficient to timely react to changes.

In this context, *Self-driving Networks* (Júnior *et al.* (2021), Mai *et al.* (2021), Jacobs *et al.* (2018))is an approach that has gained traction. Important elements of the approach include the use of decision making techniques (*e.g.*, machine learning), high level network goals defined by the network manager and an adaptation mechanism to change network configuration and behavior at runtime.

In particular, Júnior *et al.* (2021) propose an interesting self-managing solution mixing context-awareness and SDN to adapt IoT networks and compare it with human-led management. Mai *et al.* (2021), on the hand, levarage in-network machine learning to optimize network functions such as load balancing, congestion control and DDoS attack detection. Finally, Jacobs *et al.* (2018) describe the use of high-level network intents, as an alternative to the low-level definition of network policies and to determine network behavior. They also explore an approach based on machine-learning to rightly identify network configurations from intents expressed in natural language according to operators feedback.

Similarly, our work employs an approach based on user intents, which are effected on the network at runtime and autonomously, using a decision-making algorithm and network metrics collected at runtime. Differently, however, our approach is inspired on emergent software systems (Rodrigues-Filho and Porter (2017)), which allows the dynamic exploration of the search space of possible solutions.

2.3 Intent-Driven Networks

Specifically considering the use of intents to drive runtime adaptation of the network, different approaches have been proposed (Cerroni *et al.* (2017); Pang *et al.* (2020)). An important design decision refers to who defines the intents, which may be the network manager or the application developers. Also, the scope of intents needs to be considered.

Pang *et al.* (2020) survey important work that explores Intent-driven Network for realizing self-managing networks. The majority of reviewed approaches use intents to express network requirements that are translated into policies for network execution. Cerroni *et al.* (2017), on the other hand, focus on defining an interoperable and vendor-agnostic intent-based northbound interface (NBI) for service orchestration in different domains. These works show the importance of abstracting network requirements and goals through intents to either be used as goals to guide network self-management or abstract details of the NBI to support interoperability across domains.

In our work, we follow the definition of intent-driven networks (IDN) proposed by Elkhatib *et al.* (2017), in which intents are defined by application developers with the goal of informing the network about how a given application intends to use it, allowing the network to adapt and accommodate the specific needs of each application. This approach contrasts to similar proposals in which intents are defined by the network manager and specify high-level policies on how the network should behave with respect to all applications currently executing on it (Jacobs *et al.* (2018)).

3 SDN-based Intent-Driven IoT Networks

Software-defined intent-driven IoT networks are characterized by the ability of self-adaptation guided by high-level goals (*i.e.*, intents) defined by the applications that interact with the network. Network functions are defined in the form of a predefined set of *behaviors*, and there can be a number of alternative behaviors for a given function. The requirements and goals of an application regarding a given network function are expressed in the form of an *intent*, which is transparently translated at runtime into the most appropriate behavior, considering the current operating context of the network. This allows dynamic adaptation of the network behavior as the operating context changes. Network managers are responsible for defining and implementing the set of available behaviors, while the specification of intents is carried out by application developers. As a result, application developers do not need to know the configuration of the network in order to program its dynamic behavior.

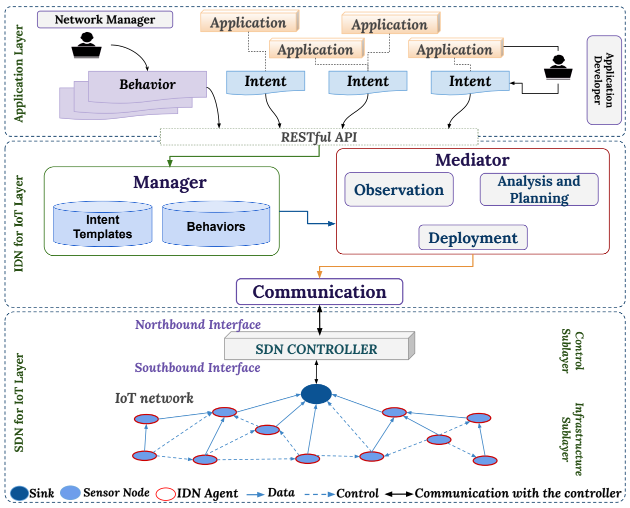


Figure 1. STEER architecture.

Fig 1 presents the STEER architecture, composed by three layers:

- the *SDN for IoT Layer*, which consists of an existing SDN solution for IoT and is further subdivided into two sub-layers: infrastructure and control;
- the *IDN for IoT Layer*, composed by a *Manager*, which maintains a set of templates for intent definition, along with the set of available behaviors, a *Mediator*, which is responsible for interpreting intents and deploying the most appropriate behavior on the network, and a *Communication* module, responsible for the interaction with the SDN layer; and

- the *Application Layer*, which is composed by the user applications and tools to create intents and behaviors.

In the *SDN for IoT Layer*, we use SDN-WISE (Galluccio *et al.* (2015)), whose control plane enables the deployment and adaptation of application functions in the data plane of IoT network nodes. Moreover, the fact that SDN-WISE is open source enabled its extension for the purposes of this work, as described below. The infrastructure sub-layer comprises the IoT network nodes, which are of two kinds: sensor nodes, which capture data from the physical environment, and sink nodes, responsible for sending data to and receiving requests from the *SDN Controller*. In this work, we extended SDN-WISE nodes by adding an *agent* that carries out the dynamic change of network behavior at the application level on each network node. At the control sub-layer, in turn, the *SDN Controller* carries out network configuration by means of control packets sent to the nodes via the southbound interface. The control packets are created from commands (produced from the interpretation of intents) received from the *IDN for IoT Layer* via the northbound interface.

In the *IDN for IoT Layer*, which represents the main contribution of this work, the *Manager* module is responsible for receiving, via a RESTful API, the intents and behaviors, which are sent to the *Mediator* for interpretation and to the *Manager* for storage in the repository, respectively. The *Mediator* runs the *Observation*, *Analysis & Planning*, and *Deployment* loop, which is inspired by the control loop of emergent software systems proposed by Rodrigues-Filho and Porter (2017) and described next.

The control loop executed by the *Mediator* aims to locate the most suitable network behavior given the operating environment to which the network is subjected at the time. In case the operating environment changes, the loop is responsible for identifying such changes and locating the optimal behavior for the new environment without human interference or domain-specific information. Hereafter, we refer to the control loop executed by the *Mediator* as the *behavior selection algorithm*. Note that due to the modular nature of STEER, we can easily replace the selection algorithm presented in this paper with any other suitable decision-making algorithm.

The behavior selection algorithm is a loop that alternates between the execution of two main phases: the exploration and exploitation phases. The algorithms that drive these phases are shown in Fig 2 and Fig 3, respectively. Because the *Mediator* has no predefined information about the network behaviors or the operating environments, whenever the network is exposed to a new environment, it has to experiment with the available behaviors to learn how well they perform. The exploration phase consists of the sequential execution of each available network behavior for a brief period of time (namely, the observation window). After executing each behavior, the *Mediator* collects performance measurements from the network. At the end of the exploration phase, the algorithm selects the behavior with the best performance, which is then deployed and executed in the network.

Fig 2 presents the algorithm for the exploration phase. This algorithm receives, as input parameters, the intent, the set of network behaviors that realizes the intent, the

```
int[] exploration(Intent intent,
                  Behavior behaviors[],
                  Communication com,
                  Observation obs) {
    int metrics[] = new int[behaviors.length()];
    for (int i = 0; i < behaviors.length(); i++) {
        obs.metricMonitor().start();
        behaviors[i].execute(com,intent,obs);
        thread_sleep(OBS_WINDOW); // 30000
        obs.metricMonitor().stop();
        behaviors[i].stop();
        metrics[i] = obs.metricMonitor().getMetric();
    }
    int result[] = obs.metricCalcResult(metrics);
    // metricCalcResult() always returns a 2-position
    // array, where result[0] points to the best
    // behavior in behaviors and result[1] holds the
    // metric value of the best behavior
    return result;
} // after this, the exploitation algorithm starts
```

Figure 2. The exploration phase of the behavior selection algorithm.

Communication object, which allows interaction with the network, and the *Observation* object, which allows the algorithm to collect metrics from the network and implements three functions that assist the behavior selection algorithm in deciding which behavior has the best performance: $metricCalcResult()$, which determines the best value from an array of metric values (*e.g.*, best performance may mean the lowest or the highest value, depending on the semantics of the metric); $metricThreshold()$, which sets the metric threshold that indicates performance degradation (*e.g.*, twice as higher than the known best metric value means performance degradation); and $newBest()$, which, given a new metric value and the previously known best, determines whether or not the new value is better. The *Observation* object thus encapsulates the logic that handles metric processing and analysis, freeing the selection algorithm from the need to have code to handle specific metrics and making it reusable for any network-wide metric.

The exploration algorithm consists of a loop that iterates through all behaviors, executes each one in the network for the observation window time frame (*e.g.*, 30 seconds), and collects the performance measurements. As a result, the algorithm returns an array with two elements. The first element points to the best-performing behavior in the behaviors array, and the second element holds the performance measurement corresponding to the best behavior. After the exploration phase locates the best-performing network behavior, the exploitation phase starts. This phase is when the network executes the best behavior for the environment. If the environment would never change, this phase would consist of only executing the best behavior. However, since IoT network environments are constantly changing, this phase is responsible for identifying environment changes and restarting the exploration phase.

Fig 3 presents the exploitation phase algorithm, which receives the same input as the exploration phase plus a pointer to the best behavior in the behaviors array and the best behavior's performance measurement, which are both results of the exploration phase. The exploitation algorithm executes the best-performing behavior for the observation window time frame and extracts the performance metric from the network. After this observation window, the exploitation algorithm decides whether it should finish execution and return to the exploration phase (because the network has been subjected to a new environment) or continues to observe the best-performing behavior for another observation window. The exploitation algorithm uses two mechanisms to detect a new environment, the detection of degraded performance and periodic exploration.

The code starting in line 21 of Fig 3 detects sequential degraded performance metrics. Suppose the best behavior presents three consecutive degraded metrics (*i.e.*, a measurement twice as large as the optimal behavior's performance). In that case, the exploitation algorithm determines that there has been a change in the network's operating environment. We defined 3 as the anomaly threshold in the presented algorithm, but note that this is a parameter for the exploitation algorithm. For unstable environments that suddenly generate high measurements for a defined short period of time, a higher threshold may be more suitable. On the other hand, if the environment is extremely stable and never generates suddenly degraded measurements, the anomaly threshold should be set to a lower value.

Periodic exploration, in turn, is presented in line 30 of Fig. 3. Sometimes a change in the environment does not negatively impact the performance metric. In this case, the only mechanism that the exploitation algorithm has to detect a change in the environment is to periodically experiment with a known sub-optimal behavior to verify whether it continues to perform sub-optimally. In the case where a known sub-optimal behavior performs better than the previously optimal behavior, the exploitation algorithm understands that the environment has changed and that a new exploration phase should start.

The main parameter for the periodic exploration algorithm defines how frequently the exploitation phase has to test a different known sub-optimal behavior. This parameter is expressed in terms of the number of metric collection cycles, *e.g.*, after 15 metric collections, test a known sub-optimal behavior. If the frequency of environment changes is high, this parameter should be set to a low value so that it can detect a change sooner. As the frequency decreases, the parameter should be set to a higher value so that the network stays executing the optimal behavior longer. Also, note that a strategy for selecting sub-optimal behaviors during periodic exploration is required. In this work, we opted for a round-robin-based selection of the next sub-optimal behavior, but other strategies could also be used (e.g., random selection).

The observation window time frame, the periodic exploration threshold, and the anomaly threshold are parameters of the exploitation phase of the selection algorithm and should be set to specific values according to the behavior, intents, and operating environment. In this work, we define these parameters manually when executing the experiments and after analyzing their impact. We discuss the impact of defining these parameters in Sec. 5 when evaluating the approach. A methodology for defining such parameters is outside the scope of this work.

The *Application Layer*, besides hosting the actual user applications, provides the tools and APIs for the definition and

```
 1  void exploitation(Intent intent,
 2                    Behavior behaviors[],
 3                    Communication com,
 4                    Observation obs,
 5                    int bestBehavior[]) {
 6      bool exploiting = true;
 7      int anomaly_count = 0;
 8      int periodic_exp = 0;
 9      int metric = 0;
10      bool behaviorExecuting = false;
11      while (exploiting) {
12          obs.metricMonitor().start();
13          if (!behaviorExecuting) {
14              behaviors[bestBehavior[0]].execute(com,
15                  intent,obs);
16              behaviorExecuting = true;
17          }
18          thread_sleep(OBS_WINDOW); // 30000
19          obs.metricMonitor().stop();
20          metric = obs.metricMonitor().getMetric();
21          // degraded performance
22          bool degraded = obs.metricThreshold(metric,
23                                   bestBehavior[1]);
24          if (degraded) {
25              anomaly_count++;
26              if (anomaly_count == ANOMALY_THRESHOLD) {
27                  behaviors[bestBehavior[0]].stop();
28                  behaviorExecuting = false;
29                  exploiting = false;
30              }
31          } else { anomaly_count = 0; }
32          // periodic exploration
33          if (periodic_exp == PE_THRESHOLD) {
34              behaviors[bestBehavior[0]].stop();
35              behaviorExecuting = false;
36              int newMetric = executeNextBehavior(
37                                  behaviors,
38                                  bestBehavior,
39                                  intent, com, obs);
40              if (obs.newBest(newMetric,
41                          bestBehavior[1])) {
42                  // a known better sub-optimal behavior
43                  exploiting = false;
44              }
45              periodic_exp = -1;
46          }
47          periodic_exp++;
48      }
49  } // after this, the exploration algorithm starts
```

Figure 3. The exploitation phase of the behavior selection algorithm.

installation of intents and behaviors. Behaviors provide alternative implementations for each network function and implement the interface shown in Fig 4. They are defined by the network manager, which sends them to the *IDN for IoT Layer*, via its RESTful interface, for registration in the behaviors repository. Intents, in turn, are defined by application developers and sent to the *IDN for IoT Layer*, via its RESTful interface, for interpretation by the *Mediator*. When defining a new intent, the application developer queries the *IDN for IoT Layer* to get an *intent template*, as well as a list of the capabilities (e.g., sensor types) available on the network. Intents are defined in JSON (*JavaScript Object Notation*) in the form of attribute-value pairs. An example intent definition is presented in Section 4.

Furthermore, the behavior is implemented using network communication functions to interact with the agent software running on the sensor nodes. The communication functions

```
interface Behavior {
    void execute(Communication com,
                 Intent intent,
                 Observation obs);
    void stop();
}
```

Figure 4. The network behavior interface.

interface is shown in Fig. 5, and they are part of the *Communication* module in the *IDN for IoT Layer*. There are three functions: $poll()$, $periodic\_collection()$ and $define\_alert()$.

The $poll()$ function collects sensed data from a specific sensor node. The execution of this function generates a message to a specific node asking for sensed data. Once it gets to the node, the agent measures the requested data and sends it back to the behavior. Since all communication between the behavior and the sensor nodes is asynchronous, all communication functions, including $poll()$, have a callback object that has the logic that gets executed when the sensed data returns to the behavior.

On the other hand, the $periodic\_collection()$ is used to register a periodic function that executes in a specific sensor node. The execution of this function generates a message to the agent of a specific sensor node to register a collection function. This collection function executes periodically, given the time specified in the function. Once the function is registered on the sensor node, sensed data is periodically sent from the sensor node to the behavior until the behavior sends a message to delete the periodic function.

Finally, the $define\_alert()$ function is used to register an alert function in a specific sensor node. The execution of this function generates a message to the agent in the sensor node and registers an alert function. This alert function is executed every second in the sensor node. The alert function collects sensed data and verifies whether the new data falls outside the defined limit. If that occurs, the sensor sends the newly sensed data back to the behavior, alerting it that the newly sensed data has surpassed the alert threshold.

Note that the selection algorithm can locate multiple best behaviors for multiple intents by executing the selection algorithm multiple times in different threads – each thread for a different intent. Also note that, for each new intent, an instance of the selection algorithm explores a set of behaviors to verify which one has the best performance and, while doing so, it may disturb the performance of the already converged instances of the selection algorithm, thus triggering the exploration phase in them, until, eventually, they all converge. However, this aspect of running multiple intents and studying the effects of multiple simultaneous instances of the selection algorithm is outside the scope of this work. Furthermore, there are other ways to cope with the realization of multiple intents in the network. For instance, with a single instance of the selection algorithm for a set of intents to be realized. The algorithm then considers a combination of behaviors, one for each of the provided intents. In this case, the resulting search space of possible behaviors to realize a set of intents increases exponentially. We further discuss this issue in Sec. 6. In the next section, we describe a use case that illustrates the approach.

```
interface Communication {
    void poll(String sensor_type, String node_id,
        MessageCallback msgCallback);
    void periodic_collection(String sensor_type,
        String node_id, int frequency,
        MessageCallback msgCallback);
    void define_alert(String sensor_type,
        String node_id, float threshold,
        MessageCallback msgCallback);
}
```

Figure 5. The communication module interface.

4 Use Case: Air Quality Application

This section describes the IoT application and scenario that we leverage to evaluate our approach. We have chosen the air quality application domain, in which context we define the intent template and an intent, and implement a set of alternative network behaviors. We also use this application to determine the parameters of the behavior selection algorithm and to design experiments to evaluate the algorithm's convergence accuracy and time.

The air quality application domain was chosen due to its importance in maintaining safe indoor environments and its strict requirements for IoT networks in terms of the quality of data sensing and the frequency of data collection. This often requires the IoT network to adapt its behavior depending on the network operating conditions.

An air quality application mainly consists of a set of physical sensors collecting the physical environment's temperature and detecting compounds that are found suspended in the air. These applications are often used to monitor the air quality in locations such as schools (Madureira *et al.* (2015)), supermarkets (Almutairi *et al.* (2019)), buildings (Floris *et al.* (2021)), and other indoor environments (Fernández-Agüera *et al.* (2019)).

To evaluate our approach, we designed an air quality application that periodically performs measurements of Total Volatile Organic Compounds (TVOC) in the air. Note that keeping track of TVOC measurements is important because long exposure to excessive amounts of TVOC has a negative impact on individuals' health (Madureira *et al.* (2015)). We built the application using STEER, defining an intent template, a specific intent, and a set of network behaviors for the intent template. We also used the COOJA (Eriksson *et al.* (2009)) network simulator to emulate the network sensor nodes and evaluate the behavior selection algorithm.

For our experiment, we use a real dataset[1] with TVOC values collected from Goiânia's city hall corridors and offices using our application.[2] We also configure the simulator to have one sink node, which serves as a gateway for applications executing outside the network, and five sensor nodes, running the STEER IDN agent software. All nodes are organized in the topology shown in Fig. 6. For the Software Defined Network (SDN), we use SDN-WISE (Galluccio *et al.* (2015)), which is an SDN controller and an implementation of a forwarding plane communication protocol specifically designed to run on low-energy devices.

[1]Dataset: `https://github.com/brunacordeiro/steer`.
[2]Goiânia is the capital city of the state of Goiás, located in the Central-West region of Brazil.

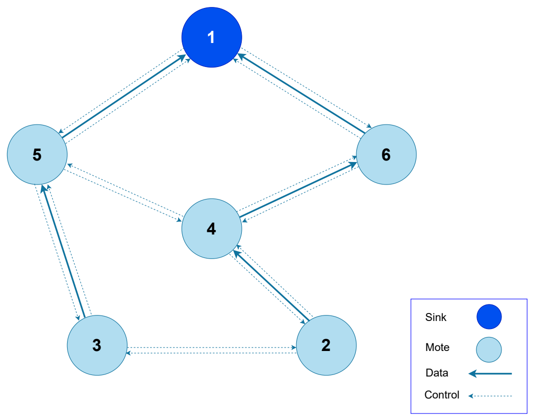


Figure 6. Network topology for the Air Quality application evaluated in the COOJA simulator.

An intent template was designed considering the needs of air quality applications, in particular, the need for periodically sampling sensor data from the network. The template has a list of attributes and *final* or *suggested* values for them: a template name (final: "periodic_sampling"), a list of sensor data types (suggestions: temperature, TVOC, humidity, CO2), a list of nodes that are part of the intent (suggestion: all, or a list of node IDs), a metric name (final: "message_count"), the goal for the metric (final: "minimize"), a numerical interval for sampling sensor data (suggestion: "5", or integer value), and the unit used for the sampling interval (suggestion: "seconds", "minutes", "hours"). The template serves as a recipe for defining concrete intents. Once an application developer obtains a template, they generate a JSON object with all the template attributes and accordingly provides values for them, thus defining a concrete intent.

The attributes in the intent template become the names in the JSON "name-value" pairs, and the suggested or final values in the template become the values. Final values are unchangeable, meaning that the application must keep the same in the intent as they appear in the template. Suggested values, on the other hand, are chosen by the application developers.

The intent we use in the evaluation is shown in Fig. 7. The depicted intent is a JSON object consisting of a series of attribute-value pairs defined for one or more applications executing outside the network. Network behavior implementations use the application-provided values to realize the intent in the network. In this use case, the behavior collects TVOC measurements every 5 seconds from all sensor nodes and makes them accessible to the applications that specified or used the intent.

Based on the intent template, a set of network behaviors is implemented. The network manager defines intent templates and implements network behaviors that realize the overall goal expressed in the intent templates. In our case study, all network behaviors implement the necessary end-to-end mechanisms to periodically collect sensor data and make the data available to the applications.

Network behaviors are implemented following the *Behavior* interface, which defines the $execute()$ and $stop()$ methods (see Fig. 4). For our use case, we have implemented three unique behaviors that leverage the available network interaction methods defined in the communication component (located at the *IDN for IoT layer* – see Fig. 1). Given the three network communication functions ($poll()$, $periodic\_collection()$ and $define\_alert()$), we created three unique behaviors that realize the air quality applications' intent (shown in Fig. 7).

```
"Intent": {
    "template_name": "periodic_sampling",
    "data_type": "TVOC",
    "nodes": "all",
    "metric": "message_count",
    "goal":"minimize",
    "periodicity": {
        "value": "5",
        "unit": "seconds"
    }
}
```

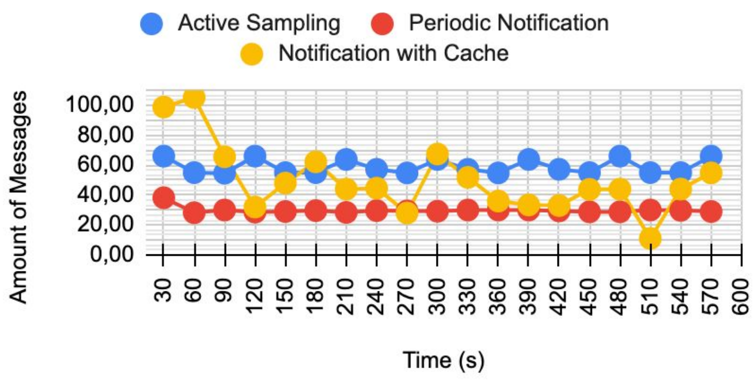
Figure 7. The air quality use case intent is expressed as a JSON object. This intent commands the network to retrieve TVOC values every 5s from all sensor nodes in the network.

The first implemented behavior is called "*Active Sampling*". This behavior periodically calls the $poll()$ function to collect data from all the network nodes. To realize the air quality use case intent, the "*Active Sampling*" behavior executes the $poll()$ function for each of the 5 sensors every 5s to collect TVOC values.

The second implemented behavior is called "*Periodic Notification*". It uses the $periodic\_collection()$ function to be notified of sensor data periodically, without having to actively sample data. To realize the air quality intent, the "*Periodic Notification*" behavior calls the $periodic\_collection()$ function on the agent running in each of the 5 nodes, setting the period of notification to 5s and the sensor data to TVOC. After invoking $periodic\_collection()$ on all network nodes, no other application-level message is generated by the behavior; all subsequent messages are generated from the nodes that send TVOC values to the behavior every 5s.

Finally, the third and last behavior is called "*Notification with Cache*". The general idea behind its implementation is to maintain newly sensed data in cache. The behavior can then provide cached values to the application without having to query the IoT network constantly. To maintain the cached data always fresh, the "*Notification with Cache*" uses the $define\_alert()$ function to register an alert function on all nodes, which collects TVOC values every second and verifies if the newly collected value is different from the cached value (i.e., the last value sent to the behavior). If the newly collected value is different, then the sensor notifies the behavior, sending it the newly collected value. Once the cached value is updated, the behavior deletes the old registered alert function in the sensors and creates a new one with a new threshold based on the new cached value. Otherwise, the sensors do not perform any action and wait another second to sense a new TVOC value. Note that this behavior only generates messages in the network when TVOC values change. When the TVOC value remains unchanged, no application-level messages are exchanged in the network.

Based on the implementation of the network behaviors to realize the air quality intent, and considering the amount of application-level messages generated on the network as the primary performance metric, we can expect the "*Active*



a)

	Active Sampling	Periodic Notification	Notification with Cache
Total Amount of Messages	1121,20	**566,00**	947,60
Average Messages per Collection	59,01	**29,79**	49,87
Standard Deviation Messages per Collection	4,86	2,11	22,87

b)

Figure 8. (a) Graph showing the three available behaviors being exposed to environment A (TVOC measurements collected during the day), showing "*Periodic Notification*" as the optimal behavior; (b) number of messages exchanged for each behavior, showing that "*Periodic Notification*" generates the minimum amount of messages both in total and on average per collection.

Sampling" and "*Periodic Notification*" behaviors to generate a relatively constant amount of messages, as they generate messages every 5s regardless of the network operating environment. On the other hand, the "*Notification with Cache*" behavior generates messages as the cached data become stale, so the number of generated messages is highly dependent on the variation of the TVOC value sensed by the network. During periods when the TVOC value changes frequently, a larger amount of messages is generated in the network to update the cached value, whereas, during periods when the TVOC value remains constant, no messages are exchanged in the network.

5 Evaluation

This section evaluates the *Mediator's* behavior selection algorithm. We evaluate the algorithm in terms of convergence time, accuracy of convergence (*i.e.*, if the Mediator selects the optimal behavior for the current environment), and overhead on the performance metric as a result of inadequate parameter settings.

We divide the evaluation into four parts. The first one refers to the ground truth, where we explore statically defined behaviors in two different operating environments, showing that there are different optimal behaviors for different operating environments. The two subsequent parts analyze the selection algorithm in static and dynamic environments, showing convergence time (*i.e.*, time to select the optimal behavior) and the algorithm's accuracy (*i.e.*, if the algorithm selects the correct behavior). Finally, we conclude the section by providing an analysis of the main parameters of the algorithm and their impact on adding an overhead to the behavior performance metric.

All experiments were conducted on a computer with a Core i7-4700MQ 2.40GHz and 16GB of RAM running Windows 10 64 bits. The IoT network was simulated on the COOJA Wireless Sensor Network simulator (Eriksson *et al.* (2009)) in a VM running Ubuntu 14.04 with 4GB of RAM. On the host operating system, we executed the SDN platform SDN-WISE Galluccio *et al.* (2015), specially tailored for controlling wireless sensor networks. The *Mediator* was built for SDN-WISE and ran as a Java application on top of the controller. All experiments were executed in 5 rounds, and the presented results are an average across all executions. The code and dataset used to perform the experiments are open-source and available for download.[3]

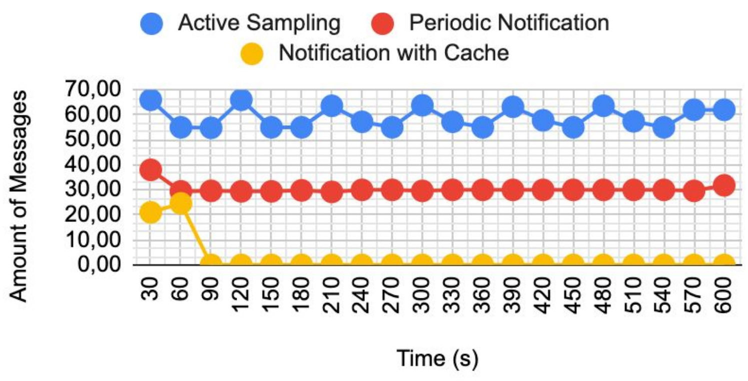
5.1 Ground Truth

This section presents the ground truth that establishes which network behavior performs optimally when exposed to two different operating environments. The operating environments are based on the selected dataset containing measurements of TVOC collected during day and night shifts. The TVOC measurements vary with different frequencies according to the time of day and place where the measurements are taken, which directly influences the performance of the network behaviors.

The available network behaviors are evaluated according to the number of messages they generate. The fewer generated messages being exchanged in the network, the better. The rationale is that the number of exchanged messages in the network directly impacts energy consumption and congestion. Thus, minimizing message exchange is generally desirable and was chosen as the main performance metric.

The "*Active Sampling*" and "*Periodic Notification*" behaviors maintain the number of generated messages in the network constant as they generate messages periodically at predefined intervals specified in the intent. On the other hand, the "*Notification with Cache*" behavior only generates messages when the cached value becomes stale. The cached value, in turn, only becomes stale when the sensed data change, and therefore this behavior is impacted by the frequency of change; otherwise, when sensed data remain unchanged, there is no reason to update the cached value, and no message is generated on the network. We, therefore, expect that the "*Notification with Cache*" behavior performs well when the TVOC values remain mostly constant; conversely, we expect poor performance when TVOC values

[3] `https://github.com/brunacordeiro/steer`



a)

	Active Sampling	Periodic Notification	Notification with Cache
Total Amount of Messages	1179,60	605,80	**45,60**
Average Messages per Collection	58,98	30,29	**2,28**
Standard Deviation Messages per Collection	4,22	1,88	7,04

b)

Figure 9. (a) Graph showing the three available behaviors being exposed to environment B (TVOC measurements collected during the night), showing "*Notification with Cache*" as the optimal behavior; (b) number of messages exchanged for each behavior, showing that "*Notification with Cache*" generates the minimum amount of messages both in total and on average per collection.

change more frequently.

We performed two experiments, presented in Fig. 8 and Fig. 9. The experiments were conducted by running each behavior on the network statically, meaning that each behavior was deployed on the network nodes as a regular IoT application with no adaptive capabilities. Each behavior was executed for 9.5 minutes (570s) and exposed to two different datasets, a dataset of TVOC readings collected during daytime (Fig. 8) and TVOC readings collected during the night (Fig. 9). Since the TVOC readings were collected from a government building that is only open during normal working hours (from 8AM to 6PM), we expect a high frequency of changes in the TVOC readings during daytime. On the other hand, TVOC readings collected during the night are expected to be mostly constant.

The y-axis in both graphs shows the amount of messages generated at each point of metric collection, while the x-axis shows the time (in seconds) the metric collection takes place. Every 30s, we collect the amount of messages for the currently executing behavior. The amount of messages as displayed in the graph is the result of the sum of messages perceived in the 30s that precede the moment when the collection was made. All the remaining graphs shown in this paper follow this description, including the ones presented in the next sections.

Fig. 8(a) shows the number of application messages generated by the three available behaviors as they are exposed to the operating environment generated by TVOC values collected during the day, which change frequently. These changes in TVOC readings make the "*Notification with Cache*" behavior generate more messages than the "Periodic Notification" behavior because the cache value becomes stale more often, forcing the network to generate more messages to update the cache.

The "*Active Sampling*" and "*Periodic Notification*" behaviors generate a relatively constant number of messages throughout the experiment because the number of messages they generate is not dependent on the dataset but rather on the intent that defines the periodic collection rate (*i.e.*, how often network nodes send sensed data to the behavior), which, in this case, is set to 5s. Furthermore, Fig. 8(b) presents a table with the total number of application-level messages generated by the network when exposed to the TVOC daytime dataset environment, showing that "*Periodic Notification*" generates the minimum amount of messages, both in total and on average per collection, and, therefore, the "*Periodic Notification*" behavior is the optimal one for that environment.

Likewise, Fig. 9 shows a similar graph (a) and table (b) for the same set of behaviors being exposed to a different environment. This time, the operating environment is defined by the readings of TVOC values collected during the night, which remain constant primarily due to the nature of TVOC. The graph on Fig. 9 (a) shows the "*Notification with Cache*" behavior generating the minimum number of messages in the network throughout the entirety of the experiment. The graph also shows the remaining two behaviors performing very similarly to how they performed in the previous environment (depicted in Fig. 8). The table (Fig. 9(b)), in turn, shows the total number of messages exchanged in the network, the average number of messages exchanged per collection, and the standard deviation of the amount of messages during the entire experiment. The table points to the "*Notification with Cache*" behavior as the optimal one as it generated the least amount of messages.

The two analyzed operating environments were designed based on real measurements of TVOC. Considering an air quality application, our ground truth experiments show that depending on the time of the day, a different implementation of the network behavior (*i.e.*, how the network nodes collect TVOC measurements) has a direct impact on the network performance. Also, our experiments suggest that no single behavior has optimal performance in all cases, which leads to the necessity to adapt the network behavior whenever the operating environment changes. The following section shows how our approach selects the optimal behavior for the two operating environments without human interference.

5.2 Convergence in Static Environments

This section explores our behavior selection algorithm. We particularly focus on behavior selection in static environ-

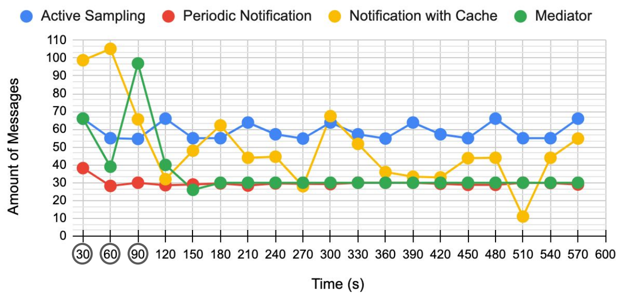


Figure 10. The three static network behaviors (lines yellow, blue, and red) executing under operating environment A (TVOC measurements collected during the day). The graph also shows the *Mediator* (green line) exploring the different behaviors and converging to the optimal behavior from instant 120s onward.

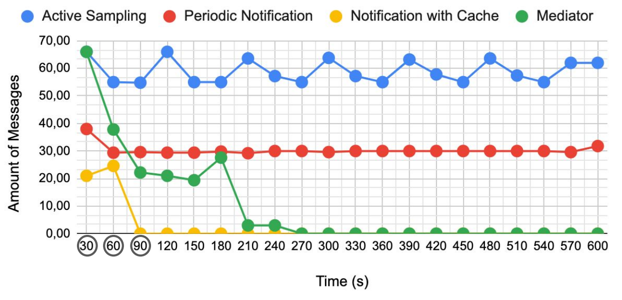


Figure 11. The three static network behaviors (yellow, blue, and red lines) executing under operating environment B (TVOC measurements collected during the night). The graph also shows the *Mediator* (green line) as it explores the different behaviors and converges to the optimal behavior from instant 120s onward.

ments (*i.e.*, network operating environments that remain unchanged). This experiment aims to show that our algorithm is accurate when converging to optimal behavior for the two operating environments evaluated in the ground truth experiments (Sec. 5.1).

The experiments were conducted in two parts for each static operating environment. First, we execute all static behaviors for each operating environment, resulting in the ground truth graphs depicted in the previous section. Second, we execute the *Mediator*, which runs the behavior selection algorithm, deciding which behavior should be deployed. The goal of the experiments is to show convergence accuracy, pointing out the overhead the exploration phase adds to the performance metric.

As discussed in Section 3, the selection algorithm has two main phases: exploration and exploitation. In static environments, the exploration phase executes once, exploring the different available network behaviors and selecting the one that yields the maximum reward (in our case, the minimum message count). After selecting the optimal behavior, the algorithm switches to the exploitation phase. The exploitation phase must decide when to return to exploration, either by identifying degradation of the performance metric or performing periodic exploration. For this pair of experiments, however, we disabled the periodic exploration part of the exploitation phase. Since the operating environment remains unchanged, we verify whether the behavior installed by the *Mediator* performs identically to the optimal static behavior after convergence.

Fig. 10 shows the *Mediator* converging towards the "*Periodic Notification*" behavior after the exploration phase, which occurs during the first three message count collections (at instants 30s, 60s, and 90s). During exploration, the *Mediator* executes the available network behaviors to verify how well each behavior performs in the current environment. The exploration phase has a fixed duration that allows the *Mediator* to test each available network behavior. In our experiment, the exploration phase takes 3 consecutive collection cycles. In the first cycle, the *Mediator* executes "*Active Sampling*". It then executes "*Periodic Notification*" in the second cycle. And, finally, at 90s it executes the "*Notification with Cache*" behavior. After exploration, the *Mediator* installs the "*Periodic Notification*" behavior in the network, which, according to the ground truth (Sec. 5.1), is the optimal behavior for the operating environment.

Similarly, Fig. 11 shows the *Mediator* converging to the "*Notification with Cache*" behavior when exposed to environment B (when TVOC readings were collected during the night). The exploration phase also occurs in the first three message count collection cycles (at 30s, 60s, and 90s). From 120s onward, the *Mediator* executes the "*Notification with Cache*" behavior, which converges to the same number of messages generated by the static behavior.

The results depicted in Fig. 10 and Fig. 11 demonstrate that the *Mediator* is able to accurately converge to the appropriate network behavior with no predefined domain-specific information. The exploration phase with fixed duration experimenting with each behavior only once works well in the experiments we tested, mainly because the performance of each behavior in both operating environments is very distinctive. This exploration algorithm, however, may not be adequate for behaviors that start with high metrics and later stabilize in a low value. For these types of environments and behavior, algorithms that dynamically change the duration of the exploration phase may be more suitable. Multi-armed bandit algorithms such as UCB1 (Auer *et al.* (2002)) and ϵ-greedy (Sutton and Barto (2018)) are examples of algorithms that could work for such environments and behavior. Overall, however, our experiments show that our behavior selection algorithm converges quickly and accurately to the appropriate behavior, with an overhead on the number of generated messages produced only during the exploration phase.

Note that the proposed algorithm works well for the tested scenarios, and although it is a simple solution, it serves as a baseline for more complex scenarios. Also, note that for other, more complex, network behaviors and environments, performing a single test of each behavior during the exploration phase may not be sufficient for the algorithm to converge toward the optimal solution.

5.3 Convergence in Dynamic Environments

This section presents our experiments with the behavior selection algorithm in dynamic environments. For these experiments, we start in a specific environment; mid-execution, we change to a different operating environment. These experiments simulate operating environment changes and how our behavior selection algorithm performs under such dynamic environments.

We run two distinct experiments illustrated in Fig. 12 and

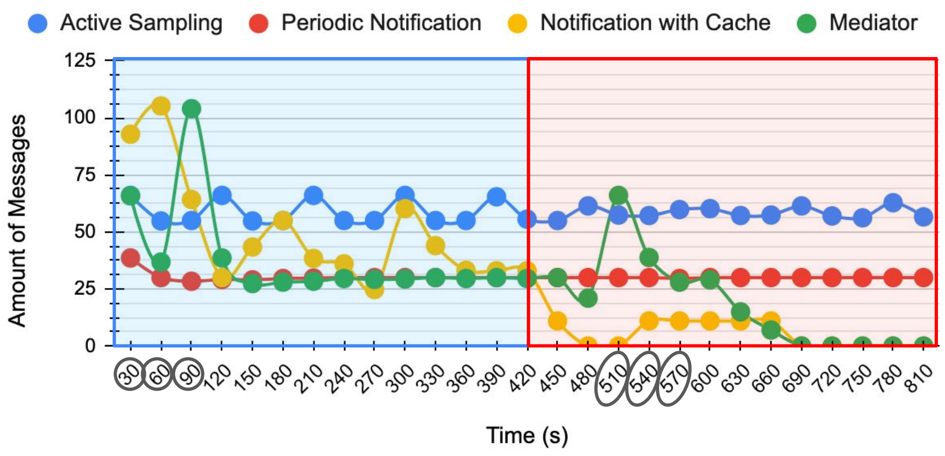


Figure 12. The three network behaviors executing statically, along with the Mediator, which in turn executes in a dynamic operating environment. The experiment starts in environment A – daytime measurements of TVOC (determined by the blue area) and changes to environment B – nighttime measurements of TVOC (determined by the red area). The *Mediator* explores and converges to the optimal behavior for each environment.

Fig. 13. We start the first experiment (Fig. 12) with environment A (TVOC collected during the day) and change to environment B (TVOC collected during the night). In the first part of the experiment, as expected, the *Mediator* behaves in the same way as it does when we subject it to environment A (as depicted in Fig. 10). The *Mediator* explores the three available behaviors (at instants 30s, 60s, and 90s) and selects the "*Periodic Notification*" behavior.

After converging to the optimal behavior, the behavior selection algorithm has to decide when to return to the exploration phase again. Ideally, the best moment to return to the exploration phase is when the operating environment changes. For the experiment illustrated in Fig. 12, the periodic exploration happens at a fixed time interval defined to occur every 6 minutes (360s) after convergence. This means that every 6 minutes, the *Mediator* tests a different known sub-optimal behavior to ensure that these behaviors remain sub-optimal and that no changes occurred in the environment. This periodic exploration is essential for the *Mediator* to make sure that the known optimal behavior remains optimal and that the operating environment continues the same. This is crucial for cases where a change in the environment does not have a direct negative impact on the performance metric for the currently executing behavior.

In the graph (presented in Fig. 12), periodic exploration occurs at 480s. During periodic exploration, the *Mediator* tests a different behavior. In our first experiment, it tests the "*Notification with Cache*" behavior and gets a resulting message count of 21, which is better than the best-known behavior message count (≈30 messages per collection) produced by the "*Periodic Notification*" behavior. Because the recently tested behavior yields a better message count, the *Mediator* subsequently triggers exploration at 510s, testing all behaviors again in order to determine a new best-performing behavior. In our experiment, at 600s, the *Mediator* converges to the "*Notification with Cache*" behavior, which is the optimal behavior for the new environment (TVOC collected during the night).

Note that the periodic exploration time span set to 6 minutes is ideal for the experiment because it triggers a periodic exploration right after the operating environment changes. This was made by design and showed the perfect parameter definition for the selection algorithm. This may not always be the case, and if the periodic exploration time span is set to a higher value, the *Mediator* will operate sub-optimally for longer. On the other hand, if the time span for the periodic exploration is set to a lower value, the *Mediator* would test sub-optimal behaviors more often, and if the environment remains unchanged for longer, this would result in a higher overhead in terms of message count.

The second experiment is shown in Fig. 13. It starts in environment B (TVOC collected during the night) and changes, mid-execution, to environment A. In the first part of the experiment, the *Mediator* behaves just like it does during the experiment with environment B (Fig. 11). After the operating environment changes, the performance of the "*Notification with Cache*" behavior degrades drastically (at 450s). As part of the *Mediator* exploitation phase algorithm, the *Mediator* maintains the execution of the behavior for 3 consecutive metric collection cycles after the metric degrades (450s, 480s, and 510s) and before triggering the exploration phase (which starts at 540s). This is because the anomaly threshold parameter makes the Mediator more tolerant to degraded metrics before triggering exploration, making the system more stable when subjected to sporadic oscillations in the performance metric.

Setting the anomaly threshold to 3 delays the exploration phase's start for 90s, which, in this case, can be seen as too long since the environment and the network are very stable. The higher the threshold value, the more tolerant to metric value outliers the network becomes. However, for stable environments, setting the threshold to a high value only delays the triggering of the exploration phase. Finally, after the exploration phase, the *Mediator* converges to the optimal behavior for the new environment, which, in our experiment, is"*Notification with Cache*".

This pair of experiments demonstrates that our behavior selection algorithm correctly selects optimal behavior even after the operating environment changes. During the exploitation phase, which starts when the optimal network behavior

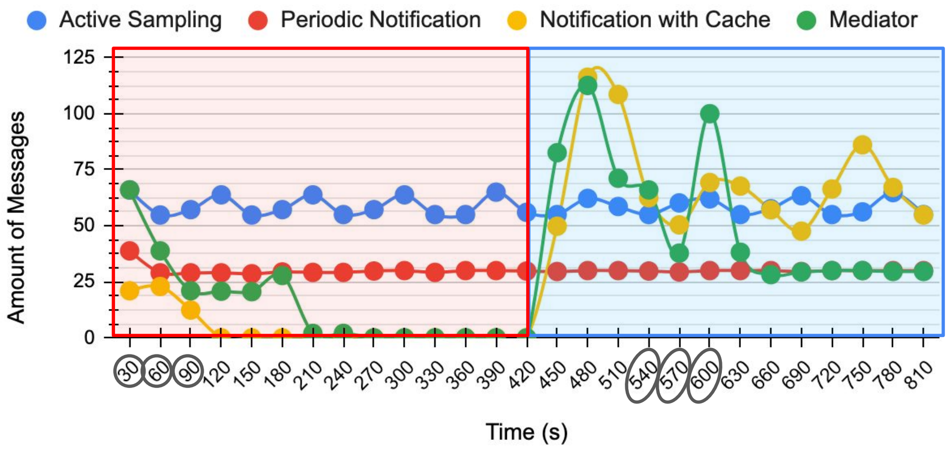


Figure 13. The graph shows the three network behavior executing statically in a dynamic operating environment and the Meditator. The experiment starts on environment B – night time measurements of TVOC (determined by the red area) and changes to environment A – daytime measures of TVOC (determined by the blue area). The mediator explores and converges to the optimal behavior for each different environment.

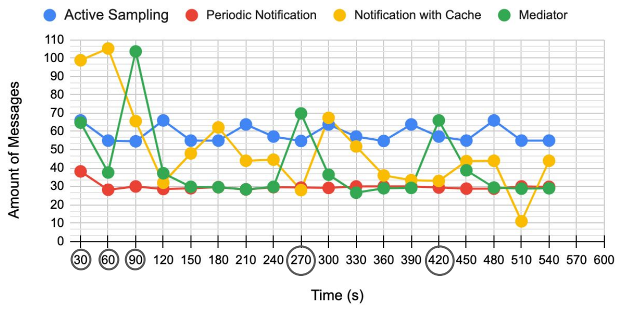


Figure 14. Convergence of the *Mediator* to *"Periodic Notification"* (the optimal behavior) for environment A (day time collection of TVOC). The *Mediator's* periodic exploration time span is set to 150s, with explorations at 270s and 420s.

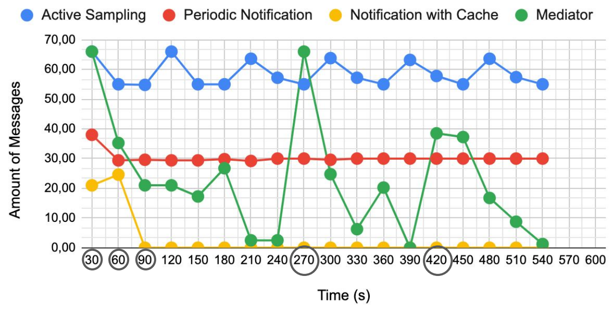


Figure 15. Convergence of the *Mediator* to *"Notification with Cache"* (the optimal behavior) for environment B (night time collection of TVOC). The *Mediator's* periodic exploration time span is set to 150s, with explorations at 270s and 420s.

is selected in the exploration phase, the *Mediator* implements two different mechanisms to detect operating environment changes. The first mechanism, illustrated in Fig. 12, is periodic exploration, where the *Mediator* periodically executes a different known sub-optimal behavior to verify that it remains sub-optimal. The second, shown in Fig. 13, is through performance metric degradation. Both mechanisms are controlled by parameters, and depending on the parameter settings, the network may suffer from performance overhead and unnecessarily long convergence time.

5.4 Algorithm's Parameters Analysis

This section analyses the parameters of the behavior selection algorithm. We first illustrate and discuss the effects of the periodic exploration time span for the algorithm, as shown figures 14, 15, and 16. We conclude the section with a discussion of the anomaly threshold parameter values and their consequences for different hypothetical operating environments.

The results depicted in Fig. 14 and Fig. 15 show the overhead in message count due to an overly short value for the periodic exploration time span, whereas Fig. 16 shows the delay in triggering exploration after an environment changes due to a long time span setting for the periodic exploration parameter. The periodic exploration time span defines how frequently a periodic exploration is performed by the *Mediator*. Periodic exploration is a mechanism used by the *Mediator* to verify whether or not the operating environment changed. This is necessary due to the fact that some operating environment changes do not affect the performance metric of some network behaviors, making it difficult to detect changes.

Fig. 14 shows the three static behaviors (blue, red, and yellow lines) executing under operating environment A, where the optimal behavior is "*Periodic Notification*". The graph also shows the execution of the *Mediator* (green line), which runs the exploration phase during the first three collection cycles (at 30s, 60s, and 90s), converging to the optimal behavior at 120s. After collection, due to the setting of the periodic exploration time to 120s, every 120s the *Mediator* tests a known sub-optimal behavior to check whether the given behavior continues to perform sub-optimally. The first time span count starts at 150s and triggers the periodic exploration at 270s (120s later), then it starts counting again at 300s and triggers periodic exploration at 420s. During the first periodic exploration, the Mediator tests the "*Notification with Cache*" behavior (at 270s), and during the second periodic exploration, the Mediator tests the "Active Sampling" behavior (at 420s). Because the environment does not change, testing a sub-optimal behavior every 120s makes the network behavior become temporarily sub-optimal twice during the experiment's lifetime. Ideally, the longer the environment

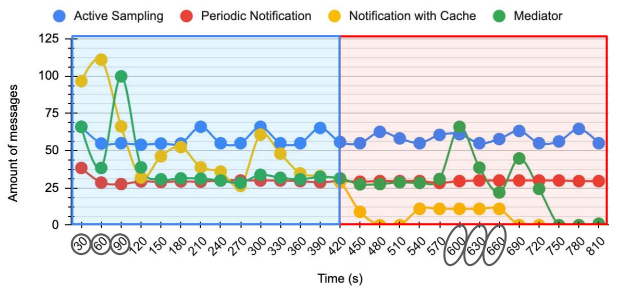


Figure 16. The *Mediator* and the three network behaviors executing in a dynamic environment. The experiment starts with operating environment A and changes to environment B. Periodic exploration is set to 480s, delaying the triggering of the exploration phase after the environment changes at 600s.

remains unchanged, the less frequently the *Mediator* should test sub-optimal behaviors to reduce the negative impact of periodic exploration on the network performance.

Similarly, Fig. 15 also shows the periodic exploration time span set to a low value, forcing the *Mediator* to perform periodic exploration too frequently for the experiment lifetime. Differently, though, in this experiment, the network behaviors were executed under environment B, for which "*Notification with Cache*" is the optimal behavior. After the *Mediator* converges to the optimal behavior (at 120s), the periodic exploration occurs every 120s, executing at 270s and 420s. The *Mediator* tests the "*Active Sampling*" behavior during the first periodic exploration (at 270s) and the "*Periodic Notification*" behavior during the second one (at 420s). Besides having the expected peak in performance after testing known sub-optimal behaviors, frequent periodic exploration prevents the optimal behavior selected by the *Mediator* to stabilize and increase the network message count. Therefore, at least in the scenario exploited in this experiment, less frequent triggering of periodic exploration allows the selected optimal behavior to properly converge to a very low message count.

Fig. 16 shows the opposite effect of mistuning the periodic exploration time span. Instead of having the effect of generating a higher message count due to frequent exploration, the graph shows the *Mediator* having a slower reaction to environment change. Fig. 16 presents the three fixed behaviors and the *Mediator* being subjected to a dynamic environment. In this experiment, the network is first subjected to environment A (TVOC collected during the day), and at 420s, the environment changes to B (TVOC nighttime collection). In this experiment, periodic exploration is set to occur every 480s, starting from the 120s and occurring only at 600s for the first time. As a consequence, the network remains on a sub-optimal behavior for over 150s before triggering the exploration phase and converging to the optimal network behavior, which happens at 690s.

The other crucial parameter for the behavior selection algorithm is the anomaly threshold. This parameter determines how tolerant to outlier performance metric values the *Mediator* is. A high threshold value is crucial for environments that randomly generate a series of peaks in the performance metric, but remain in the same operating environment. Peaks in performance metrics (*i.e.,* performance degradation) often are an indication of environment changes, but sometimes it is just a period of instability generated by unpredictable factors. Therefore, always reacting to performance degradation (*i.e.*, always triggering exploration) may not be the best course of action, especially if the performance degradation is just a momentary instability of the operating environment. Executing a complete exploration phase generates high overhead in the network performance metric. Thus, being tolerant to degraded performance for moments of instability is necessary, but being too tolerant makes reaction time to environment change very slow.

An example of the effects of this parameter is shown in Fig. 13. The anomaly threshold was set to 3, meaning that the *Mediator* only triggers a full exploration phase after detecting 3 consecutive measurements of unusually high metric values (*e.g.*, 3 consecutive measurements of the metric at values two times higher than the previously collected values). In that case, the *Mediator* only triggers exploration 120s after the environment changes. Considering that the environment is very stable, the *Mediator* could trigger exploration sooner and converge to the new optimal behavior sooner.

This set of experiments with the periodic exploration parameter shows that this parameter is highly dependant on the environment and available network behavior characteristics, and for the different environments or sets of behavior a different parameter value can be more adequate. Better behavior selection algorithms should use meta-learning approaches to tune such parameters. Besides online parametric tuning/learning, a promising line of research is to investigate coordination between the behavior selection algorithm and environment classification algorithms to eliminate the need for periodic exploration and identifying periods of environment instability and to only trigger exploration when a previously unknown environment is detected.

6 Discussion

This section discusses the limitations of our approach and presents opportunities to be explored in future work. We divide the discussion into three sections: limitations of the behavior selection algorithm; limitation of the current integration of the IDN and SDN layers; and a discussion on scalability.

6.1 Behavior Selection Algorithm

The last part of the evaluation of the behavior selection algorithm shows the consequences of not adequately choosing the algorithm's parameters. For instance, we have shown situations in which the algorithm takes too long to react to changes in the environment or explores too often when it could simply remain executing the best-known behavior longer. Thus, an important addition to the selection algorithm is an extension that explores parameter auto-tuning to adjust their values as the network encounters new environments. Besides the need for online parameter tuning, the current approach would also benefit from classifying operating environments. Environment classification would enable the selection algorithm to identify previously seen conditions and remember the decisions made. This allows the algorithm to create a mechanism to avoid exploring behaviors for previously seen conditions and spend more time exploiting best-known behaviors rather than searching for them.

The execution of multiple intents in the network was not investigated in this work. As noted in the approach description section (Sec. 3), multiple instances of the selection algorithm could execute concurrently to locate the best behavior for each intent provided to the network. Furthermore, to realize multiple intents, a unique network-wide metric (*e.g.*, message count, or energy consumption) should be used. This unique network metric would allow the selection algorithm to converge towards a set of behaviors that would yield optimal performance for the given network operating condition. Clearly, however, deploying new behaviors to realize new intents in the network would impact the network-wide observed metric, disturbing the execution of the multiple instances of the selection algorithm. The multiple executing instances would eventually converge, as they would make decisions based on the newly observed metric in the new operating environment created by adding new intents to be realized in the network. Nevertheless, the impact of multiple simultaneous instances of the selection algorithms needs to be further investigated.

Other approaches to explore multiple intents in the network can be investigated in future work. Another example is to update the selection algorithm to receive as input multiple intents and update the algorithm to execute one behavior for each intent received. This approach would generate a combination of behaviors to realize a set of intents. For instance, if there are two intents: $I1$ and $I2$, each with two behaviors $I1-B1$, $I1-B2$, $I2-B1$, $I2-B2$, the exploration phase would execute all combination of behaviors (*e.g.*, $I1-B1$ and $I2-B1$, then $I1-B1$ and $I2-B2$, and so on) and select the combined behaviors with the best performance according to the network-wide metric. This approach would certainly lead to a large search space to explore as the number of intents to be realized in the network increases.

The multiple intent discussion leads to two concerns: the exploration of large search spaces and the consideration of conflicting goals. As the number of intents increases in the network, the number of combined behaviors to evaluate increases exponentially. Thus, the need to iterate through all possible combinations of behaviors at least once before making decisions becomes impractical in extreme cases. These extreme situations require a new strategy to cope with larger search spaces. Reinforcement learning approaches, in turn, such as the one discussed in Ontanón (2017), could potentially be applied to help navigate through large search spaces ($\approx$millions of actions). In that paper, the author describes a multi-armed bandit solution for real-time gaming that is able to navigate through a search space with millions of actions.

Conflicting goals represent another important challenge. The realization of multiple intents would lead to conflicting goal resolution when the metric that determines the criteria to select behaviors for different intents is different. In this case, the multi-object optimization (MOO) approach should be considered instead of our proposed behavior selection algorithm, which is facilitated by the modularity of our approach, which in turn allows replacing the selection behavior algorithm with other algorithms. This scenario, however, is outside the scope of this paper, although related research in the literature could be explored. For instance, Fei *et al.* (2017) surveys MOO algorithms for Wireless Sensor Networks. Some of these algorithms could potentially be used to further explore and consolidate STEER when considering conflicting metrics.

6.2 IDN and SDN integration

For the integration of the IDN and SDN layers, our approach presents limitations on the extensibility of both metric collection and the execution of the agent on sensor nodes. Although the IDN module is equipped with the $Observation$ interface that can be implemented to deal with any metrics, our current implementation only uses the metrics that SDN-WISE constantly collects from the network: nodes energy consumption, nodes RSSI, network topology and other network-related metrics. To make STEER able to deal with other types of metric, it would be necessary to create a mechanism to extend SDN-WISE‘ metric collection to allow the collection of new metrics from the sensor nodes and send them to the controllers as Report Packets. In the current implementation of STEER, arriving Report Packets are made available to any class implementing the $Observation$ interface. This mechanism would allow the creation of new implementations of $Observation$ for new metrics, without the need to alter the proposed behavior selection algorithm as previously described.

The agent running on sensor node, isn turn, is static and implements network communication functions that are used by behaviors to interact with the network. To add more flexibility in the way behaviors interact with the network, the IDN layer could exploit the network function feature of SDN-WISE and install new agents, as demanded by behaviors, equipped with new ways to interact with the network. This would allow executing behaviors to change code in target sensor nodes at runtime; for instance, to add code that performs a specific aggregation function or a data classifier after data collection in the node.

Finally, we adapted the sensor node at the application level only. However, we could further interact with the SDN controller from the behavior to also make changes at the network level. For instance, we could change the routing algorithm or install new network functions to perform tasks such as load balancing among sensor nodes. We could also deploy a classifier on the sensor node responsible for replying with processed data instead of sending raw sensing data for the behavior to process.

These extensions to the integrationo of the IDN and SDN layers would make our approach more adaptable and generic, possibly also makin git more useful for a broader range of application domains.

6.3 Scalability

As a final set of limitations of the approach, we now discuss some aspects of scalability. The first noticeable aspect of our experiment is the small number of nodes in the evaluated scenarios. In this paper, we evaluated STEER with only

5 executing sensor nodes and a sink node, resulting in a deployment setting with a maximum of 6 nodes. Although 6 nodes are realistic for monitoring the air quality of a single room in a building, we built our solution to scale past a 6-node network infrastructure. STEER leverages an SDN for an IoT-based approach, and we rely on the SDN solution to scale. In the literature, the controller placement problem has been widely tackled to scale SDN and demonstrated to be useful in the real world (Huang *et al.* (2017); Lange *et al.* (2015); Min *et al.* (2021).

We rely on SDN to bootstrap the network and define and adapt the flow tables of the sensor nodes, especially to extract metrics from the network and send application-level commands to agents running at the application level on the nodes. Considering that the nodes are up and running and a network is perfectly executing with hundreds of nodes and that we can reach any node and send application-level commands to them and collect metrics, we can then focus on the aspects that directly affect the scalability of STEER. These aspects are the number of concurrent intents (and the resulting combinatorial number of behaviors) and the nature of selected metrics.

The number of behaviors is determined by the intents, and the intents may only affect a set of nodes. The selection algorithm could be made to only consider the affected nodes defined by the intents rather than run all combinations of all behavior in all nodes. This could be used as a strategy to help manage scalability. Furthermore, as previously discussed, strategies to navigate large search spaces could be adopted to allow the realization of multiple intents in the network.

Finally, the collection of metrics may have a significant impact on scalability (Gardikis *et al.* (2016)), and further research should be conducted to explore this in the context of STEER. The messages exchanged between the network nodes and the application running outside the network go through a centralized controller and, thus, do not require active metric extraction. However, many other metrics may impact performance. For instance, calculating energy consumption involves querying the energy levels of a set of nodes multiple times to calculate the total consumption. This impacts the number of messages in the network, and thus a strategy to query a subset of nodes may help scale the system.

7 Conclusion

This paper presented an end-to-end implementation of a novel approach to support autonomous adaptation of IoT networks for indoor monitoring applications. The approach is the result of unifying two concepts that have gained importance in the literature: Intent-Driven Networks (IDN) and Software Defined Networks (SDN). The combination of the two concepts was realized through a layered architecture, where the SDN layer was built using an existing solution (SDN-WISE), while the IDN layer is a contribution of this work. The IDN layer receives and interprets application intents, realizing them through the runtime selection and execution of the best-performing network behavior (among a set of available behaviors), considering the current network operating environment.

We evaluated the approach using a simulated network and real IoT indoor monitoring application data. The experiments were conducted in the domain of air quality monitoring. We defined an intent template and a representative intent instance for an air quality monitoring application, together with three distinct network behaviors to realize the intent. We subjected our IoT network to two different operating environments and evaluated the behavior selection algorithm, implemented as part of the intent *Mediator*, in terms of convergence time and accuracy.

Our results show that the proposed selection algorithm correctly selects the optimal behavior for both environments. We also show that the convergence time is determined by the duration of the selection algorithm's exploration phase and that our exploration phase is appropriate for the behavior and environment that we used in the experiments. Finally, we showed that the overhead on the performance metric is a consequence of adapting the network behavior and that by properly setting the parameters for the exploitation phase, we may reduce such overhead.

As we presented a complete end-to-end implementation of a novel approach, some of its aspects were not fully addressed. In future work, we envision investigating the process of generating network behavior implementations directly from the application's intent. This work also provides the opportunity to investigate conflicting intents and how the network should handle them. Finally, the investigation of reinforcement learning algorithms to learn the best performing network behavior in situations where i) it is not easy to identify an optimal behavior after executing each behavior once in the exploration phase, and ii) when the number of available behaviors is large, which increases the search space and requires strategies to handle scalability. Finally, although the intent and behaviors used in the experiments are representative of an important class of IoT applications, further work is necessary to validate and evaluate the approach in other scenarios, especially those involving other patterns of interaction between application and network besides the collection of sensing data.

Acknowledgment

This research is also part of the INCT of the Future Internet for Smart Cities funded by the National Council for Scientific and Technological Development (CNPq) proc. 465446/2014-0, the Coordination for the Improvement of Higher Education Personnel (CAPES) proc. 88887.136422/2017-00, and the São Paulo Research Foundation (FAPESP) procs. 14/50937-1 and 15/24485-9. Bruna Michelly de O. S. Cordeiro thanks CAPES for her masters scholarship. Roberto Rodrigues Filho thanks FAPESP for funding his postdoctoral work under the process 2020/07193-2.

Declarations

Authors' Contributions

All authors contributed to the writing of this article, read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Data can be made available upon request.

References

Almutairi, A., Alsanad, A., and Alhelailah, H. (2019). Evaluation of the indoor air quality in governmental oversight supermarkets (co-ops) in kuwait. *Applied Sciences*, 9(22). DOI: 10.3390/app9224950.

Aschenbruck, N., Bauer, J., Bieling, J., Bothe, A., and Schwamborn, M. (2012). Selective and secure over-the-air programming for wireless sensor networks. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE. DOI: 10.1109/ICCCN.2012.6289278.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256. DOI: 10.1023/A:1013689704352.

Azzara, A., Alessandrelli, D., Bocchino, S., Petracca, M., and Pagano, P. (2014). Pyot, a macroprogramming framework for the internet of things. In *Proceedings of the 9th IEEE international symposium on industrial embedded systems (SIES 2014)*, pages 96–103. IEEE. DOI: 10.1109/SIES.2014.6871193.

Bera, S., Misra, S., and Vasilakos, A. V. (2017). Software-defined networking for internet of things: A survey. *IEEE Internet of Things Journal*, 4(6):1994–2008. DOI: 10.1109/JIOT.2017.2746186.

Blair, G. (2018). Complex distributed systems: The need for fresh perspectives. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1410–1421. DOI: 10.1109/ICDCS.2018.00142.

Cerroni, W., Buratti, C., Cerboni, S., Davoli, G., Contoli, C., Foresta, F., Callegati, F., and Verdone, R. (2017). Intent-based management and orchestration of heterogeneous openflow/iot sdn domains. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–9. DOI: 10.1109/NETSOFT.2017.8004109.

Elkhatib, Y., Coulson, G., and Tyson, G. (2017). Charting an intent driven network. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–5. IEEE. DOI: 10.23919/CNSM.2017.8255981.

Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., and Marrón, P. J. (2009). Cooja/mspsim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–7. DOI: 10.4108/ICST.SIMUTOOLS2009.5637.

Fei, Z., Li, B., Yang, S., Xing, C., Chen, H., and Hanzo, L. (2017). A survey of multi-objective optimization in wireless sensor networks: Metrics, algorithms, and open problems. *IEEE Communications Surveys & Tutorials*, 19(1):550–586. DOI: 10.1109/COMST.2016.2610578.

Fernández-Agüera, J., Dominguez-Amarillo, S., Fornaciari, M., and Orlandi, F. (2019). Tvocs and pm 2.5 in naturally ventilated homes: Three case studies in a mild climate. *Sustainability*, 11(22). DOI: 10.3390/su11226225.

Floris, A., Porcu, S., Girau, R., and Atzori, L. (2021). An iot-based smart building solution for indoor environment management and occupants prediction. *Energies*, 14(10). DOI: 10.3390/en14102959.

Galluccio, L., Milardo, S., Morabito, G., and Palazzo, S. (2015). Sdn-wise: Design, prototyping and experimentation of a stateful sdn solution for wireless sensor networks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 513–521. IEEE. DOI: 10.1109/INFOCOM.2015.7218418.

Gardikis, G., Koutras, I., Mavroudis, G., Costicoglou, S., Xilouris, G., Sakkas, C., and Kourtis, A. (2016). An integrating framework for efficient nfv monitoring. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 1–5. DOI: 10.1109/NETSOFT.2016.7502431.

Gubbi, J., Buyya, R., Marusic, S., and Palaniswami, M. (2013). Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660. DOI: 10.1016/j.future.2013.01.010.

Huang, T., Yu, F. R., Zhang, C., Liu, J., Zhang, J., and Liu, Y. (2017). A survey on large-scale software defined networking (sdn) testbeds: Approaches and challenges. *IEEE Communications Surveys & Tutorials*, 19(2):891–917. DOI: 10.1109/COMST.2016.2630047.

Jacobs, A. S., Pfitscher, R. J., Ferreira, R. A., and Granville, L. Z. (2018). Refining network intents for self-driving networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, pages 15–21. DOI: 10.1145/3229584.3229590.

Júnior, J. C., da Cunha, D. C., and Ferraz, C. A. (2021). Integrating context awareness and sdn for a lightweight approach to adaptive networking. In *Anais do XIII Simpósio Brasileiro de Computação Ubíqua e Pervasiva*, pages 91–101. SBC. DOI: 10.5753/sbcup.2021.16007.

Junior, S., Riker, A., Silvestre, B., Moreira, W., Oliveira-Jr, A., and Borges, V. (2020). Dynasti—dynamic multiple rpl instances for multiple iot applications in smart city. *Sensors*, 20(11):3130. DOI: 10.3390/s20113130.

Kreutz, D., Ramos, F. M. V., Veríssimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2015). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76. DOI: 10.1109/JPROC.2014.2371999.

Lange, S., Gebert, S., Zinner, T., Tran-Gia, P., Hock, D., Jarschel, M., and Hoffmann, M. (2015). Heuristic approaches to the controller placement problem in large scale sdn networks. *IEEE Transactions on*

Network and Service Management, 12(1):4–17. DOI: 10.1109/TNSM.2015.2402432.

Madureira, J., Paciência, I., Rufo, J., Ramos, E., Barros, H., Teixeira, J. P., and de Oliveira Fernandes, E. (2015). Indoor air quality in schools and its relationship with children's respiratory symptoms. *Atmospheric Environment*, 118:145–156. DOI: 10.1016/j.atmosenv.2015.07.028.

Mai, T., Garg, S., Yao, H., Nie, J., Kaddoum, G., and Xiong, Z. (2021). In-network intelligence control: Toward a self-driving networking architecture. *IEEE Network*, 35(2):53–59. DOI: 10.1109/MNET.011.2000412.

Min, Z., Sun, H., Bao, S., Gokhale, A. S., and Gokhale, S. S. (2021). A self-adaptive load balancing approach for software-defined networks in iot. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 11–20. DOI: 10.1109/ACSOS52086.2021.00034.

Noor, J., Tseng, H.-Y., Garcia, L., and Srivastava, M. (2019). Ddflow: visualized declarative programming for heterogeneous iot networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 172–177. DOI: 10.1145/3302505.3310079.

Ontanón, S. (2017). Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, 58:665–702. DOI: 10.1613/jair.5398.

Pang, L., Yang, C., Chen, D., Song, Y., and Guizani, M. (2020). A survey on intent-driven networks. *IEEE Access*, 8:22862–22873. DOI: 10.1109/ACCESS.2020.2969208.

Rodrigues-Filho, R. and Porter, B. (2017). Defining emergent software using continuous self-assembly, perception, and learning. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(3):1–25. DOI: 10.1145/3092691.

Rodriguez-Zurrunero, R., Tirado-Andrés, F., and Araujo, A. (2018). Yetios: An adaptive operating system for wireless sensor networks. In *2018 IEEE 43rd Conference on Local Computer Networks Workshops (LCN Workshops)*, pages 16–22. IEEE. DOI: 10.1109/LCNW.2018.8628500.

Shafi, N. B., Ali, K., and Hassanein, H. S. (2012). No-reboot and zero-flash over-the-air programming for wireless sensor networks. In *2012 9th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 371–379. IEEE. DOI: 10.1109/SECON.2012.6275799.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. (2014). Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32. DOI: 10.1109/JIOT.2014.2306328.