2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)
Banff Center, Banff, Canada, October 5-8, 2017

Data and Control Points: A Programming Model for Resource-constrained IoT Cloud Edge Devices

Stefan Nastic*, Hong-Linh Truong*, and Schahram Dustdar*

***Abstract*— Recent emergence of IoT Cloud systems has fostered proliferation of various applications mainly driven by urgent need to respond to volume, velocity and variety of data generated by IoT Cloud, but also to enable timely propagation of actuation decisions, crucial for business operation, to the Edge of the infrastructure. In such systems, utilizing currently untapped Edge resources such as sensory gateways, and enabling the IoT devices as first-class execution environments plays a crucial role. However, enabling virtually exclusive access to the underlying devices, e.g., field bus sensors and supporting flexible, application-specific customizations for such devices still remain a challenge. In this paper, we introduce *Data- and Control Points* - a novel programming model and framework for developing applications specifically tailored for resource-constrained Edge devices. Our framework offers programming constructs that enable applications to define custom configurations for and their own view of the underlying devices. By providing an illusion of an exclusive access to the underlying sensors and actuators, our framework supports execution of multiple applications within a single Edge device.**

I. Introduction & Related Work

Recent advances in IoT and Edge computing have resulted in numerous approaches in terms of programming frameworks and middleware for developing application business logic suitable for resource-constrained IoT devices such as sensory gateways. However, such approaches mainly focus on hiding the heterogeneity of data sources (e.g., sensors) [1]–[6], defining data processing schemes [1], [6], and dealing with mobility [1], [3], [4], privacy [5] and scalability [4], [6]. In spite these and other approaches that address similar issues, e.g., on communication protocols level [7] or by utilizing SOA principles [8], [9], enabling virtually exclusive access to the underlying devices, e.g., field bus sensors and supporting flexible application-specific customizations for such devices are still not fully addressed in the literature. This inherently prevents utilizing the Edge devices as generic execution environments that can be potentially shared by multiple IoT Cloud applications. Still in large-scale IoT Cloud systems, leveraging the computational resources of the Edge devices is especially important, as their currently untapped processing capabilities can be used to optimize IoT Cloud applications by making edge devices first-class execution environments, i.e., by moving parts of application logic away from cloud platforms towards the edge of IoT Cloud infrastructure.

Continuing our previous line of research [10], [11], in this paper we introduce *Data- and Control Points* - a novel programming model and framework that provides a set of programming abstractions for developing common monitor and control tasks. The Data- and Control Points represent low-level channels to the sensors/actuators in an abstract manner and mediate the communication with the connected devices, e.g., digital, serial or IP-based. The supporting framework provides mechanisms which act as multiplexers of the data and control channels, thus enabling the device services to have their own view of and define custom configurations for such channels, e.g., sensor poll rates, data units or data stream filters. By providing an illusion of an exclusive access to the underlying devices, our framework supports execution of multiple applications within a single Edge device.

The remainder of the paper is structured as follows: In Section II we outline the framework architecture; Section III introduces the Data and Control Points and presents the main concepts of the programming model; In Section IV, we describe the main runtime mechanisms of the edge-devices framework. Finally, Section VI concludes this paper.

II. Overview of DRACO Framework

The main aim of DRACO (*Data And Control pOints*) framework is to facilitate the development of common monitor and control tasks in IoT Cloud systems. As discussed in our previous work, these task are the main building blocks of edge-device applications/services and the main constituents of reusable domain libraries [10]. Generally, such libraries form the cornerstone for building higher-level cloud-centric IoT Cloud applications.

Figure 1 shows a high-level overview of the DRACO framework. In general, our framework follows a layered architecture and runs inside resource constrained Edge device, enabling local execution of device-level applications. In a broader sense it acts as an interlayer between low-level devices such as sensors and actuators and the high-level services which are executed on cloud platforms. Starting from bottom up, the *Edge device middleware layer* is generally responsible to mediate communication with the underlying physical devices, maintain configuration models and provide and execution runtime for the monitor and control tasks. To enable communication with the physical devices this layer provides *Drivers and Com. Protocols* component. Its main responsibility is to provide the supporting driver implementations, which enable direct communication with the devices, e.g., via general purpose IO (GPIO) pins, field bus communications over protocols such as I^2C or ModBus, or communication over IP-based networks. Th Edge device middleware layer also provides *Configuration Models* repositories such as light-weight NoSQL database. The configuration models are stored locally in the device and among other things they specify how the underlying devices are connected. For example, in case of direct pin connection such models contain meta-data such as pin class (e.g., analog in), name and hardware-related data, e.g., multiplexer addresses or value correction constants. Finally, the *Runtime Services* constitute the tasks execution runtime

*This author is with Distributed Systems Group, TU Wien, Austria. Contact him via: {lastname}@dsg.tuwien.ac.at

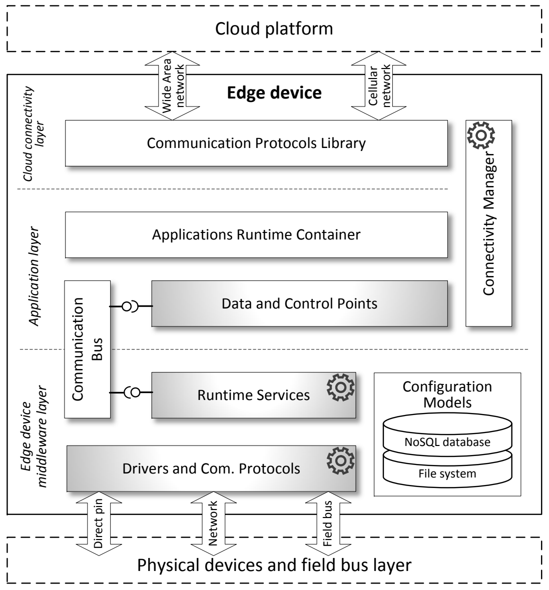

Fig. 1. DRACO high-level architecture overview.

and provide the sporting runtime mechanisms for Data and Control Points. This component is discussed in more detail in Section IV.

The next layer is the *Application layer* and its main purpose is to provide application development and execution support. The crucial part of this layer is the *Data and Control Points* component. It provides concrete implementation our programming model's abstractions and APIs, which are used by domain expert developers, as we describe in more detail subsequently. Further, the *Applications Runtime Container*, provides an execution runtime for the Edge-device applications. It is important to mention that the DRACO framework does not impose many limitations regarding the application model. For example, such applications can be based on OSGi container or even stand alone applications. In the current prototype, we rely on a striped-down JVM (based on Oracle Compact Profiles) to run the programming model, thus the framework only requires the application runtime to be JVM compatible. The *Connectivity Manager* is a cross cutting component (between the *Application layer* and *Cloud connectivity layer*), which provides a flexible mechanisms for the Edge-device applications and services to communicate with the cloud. The Connectivity Manager relies on the *Communication Protocols Library* to offer a set of higher-level communication protocols such as CoAP or MQTT to the applications. It supports the applications to dynamically configure and utilize the available protocols, without having to deal with the low-level implementation details. In this paper we do not discuss the *Cloud connectivity layer* to more detail, since it is out of scope of DRACO framework.

III. Data and Control Points: A Programming Model for Edge Devices

In our framework we envision two distinct usage patterns for the Data and Control Points and the aforementioned tasks. First, they can be used to develop "stand alone" edge-device applications, which do not necessarily depend on the cloud. An example of such application would be a logging application that reads sensory data and stores it locally at device side, e.g., in a light-weight database. Second, they can be used to develop domain libraries. In this context a domain library contains a set of reusable tasks that are responsible to encapsulate domain-specific knowledge, most notably domain model and common behaviors, in a reusable manner. For example, a building automation expert developer could develop a domain library to facilitate development of higher-level functionality for building management systems. Generally, a control task is any permissible sequence of actuating steps which can be used to control physical devices, via the actuators they expose. Further, a monitor task includes processing, correlation and analysis of sensory data streams to provide meaningful information about the state changes of the underlying devices or the surrounding environment.

A. Main programming abstractions and application model

Data and Control Points represent and enable management of data and control channels (e.g., device drivers) to the low-level sensors/actuators in an abstract manner. Generally, they mediate the communication with the connected devices (e.g., digital, serial or IP-based), enable application-specific customizations of the channels and also implement communication protocols for the connected devices, e.g., Modbus, CAN or I^2C.

Figure 2 shows a simplified UML diagram of the main components of our programming model. From the figure we notice that the EdgeApplication contains multiple Tasks. Further such tasks can have multiple DCPoints associated with them. The DCPoint is an abstract class which provides main operators and lifecycle management hooks for the Data and Control Points. Both DataPoints and ControlPoints inherit from this component and encapsulate the specialized behavior for reading sensory data (DataPoints) and preforming the actuations (ControlPoints). In general, DCPoints allow the developers to perform concurrent reads and writs, regardless of whether the low-level drivers support sequential or concurrent reads and writes. In this way the applications have an impression of exclusive usage of the available devices. Another important feature of DCPoints is that they enable developers to configure custom behavior of underlying devices. For this purpose each DCPoint can have a ConfigurationModel associated with it. For example, an application can configure sensor poll rates, activate a low-pass filter for an analog sensory input or configure unit and type of data instances in the stream. However, there are physical limitation, which need to be considered, such as a sensor might sample data at a different rate then specified in the ConfigurationModel for the DataPoint abstracting the sensor. The most important case is when the poll interval specified by an application is shorter than sensor's minimum interval. In this case the corresponding DataPoint issues a warning to the application (e.g., not supported configuration), but it resends the last available reading, given the configuration, until a new fresh reading is available. This enables developers to handle such situation dynamically, while allowing the applications to run without runtime interrupts.

The most important concept supporting the DCPoints are the VirtualBuffers, which are provided and managed by our framework. In general, such buffers enable virtualized access to and custom configurations of underlying sensors and actuators. They act as multiplexers of the data and

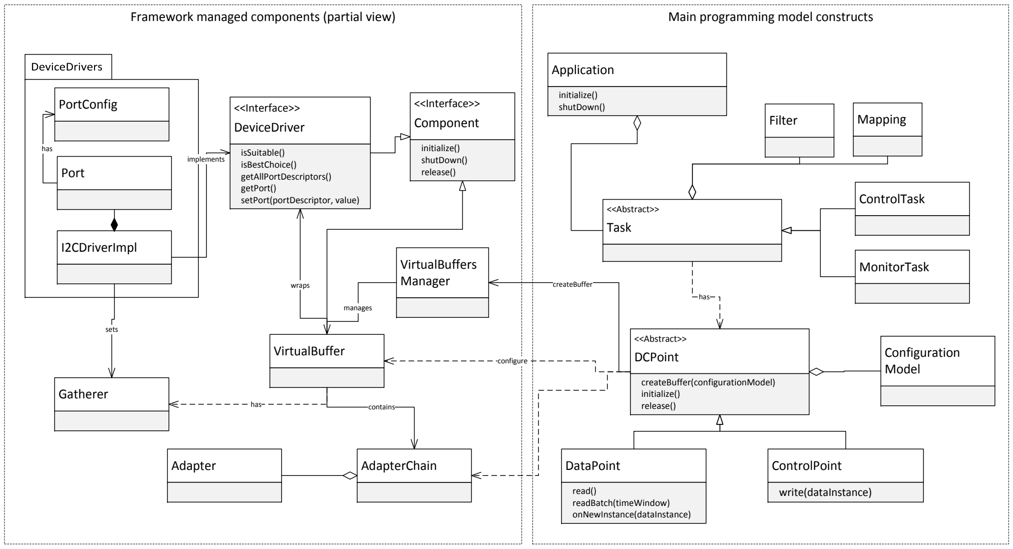

Fig. 2. Simplified UML diagram of Data and Control Points.

control channels, thus enabling the device applications to have their own view of and define custom configurations for such channels. To this end, the VitualBuffers wrap the DeviceDrivers and share a common behavior with them, inherited through the Component Interface. For example, they can be initialized, shutdown and released. Both buffers and drivers lifecycle are managed by the VirtualBuffersManager. Moreover, a virtual buffer references a set of Gatherers and can contain an optional AdapterChain. Generally, a gatherer is a higher level representation of a port. For example, in case of a sensing device (DataPoint) the gatherer represents the most resent value of the hardware interface. The principle for ControlPoints is similar to the one for the sensing devices. The only difference is that in case of actuation request the gatherers act as serializes instead of observers. To support application-specific configurations such as sensor poll rate, filters or scalers, each virtual buffer can have an AdapterChain. Adapter chains reference different Adapters, which are specified and parametrized via DCPoint's ConfigurationModel. Any raw sensing value is passed through such adapter chain before being delivered to a DataPoint.

The DeviceDrivers Package contains a set of driver implementations. For readability purposes, in the figure we only show the component for I^2C protocol, but each implementation follows similar principle. It contains a set of Ports, which is a framework internal representation of devices attached to the bus. Such Ports are dynamically instantiated by the VirtualBuffersManager at device bootup during driver initialization phase, based on the provided PortConfig. At the moment, PortConfig is specified as a JSON file that contains the meta-data such as port class (e.g., analog in), name and hardware-related data, e.g., multiplexer address or value correction constants.

B. Application data model

Besides supporting development of monitor and control tasks the Data and Control Point enable the domain expert developers to define a custom application data model. Figure 3 depicts a simplified UML data model of the DCPoints. It can be seen as a meta model that enables applications to define a custom data (domain) model. This is especially important for defining groups of DCPoints that represent some logical entity in the physical environment. For example, this model can be used to describe a complex device, which contains multiple sensors or an application-specific domain model entity, e.g., room. To this end, the DataInstance acts as a wrapper of a sensory reading (value) and enriches it with additional information such as timestamp. Moreover, the DataType enables defining custom data instances types. It extends the built-in Java types and provides additional information about the data instance such as its unit (e.g., Kelvin). In this context, the most important feature provided by our framework is the support for complex data types and complex data instances. A complex type is represented as a record, hence it consists of named fields that have again have a type associated with them. Similarly, a complex data instance is a combination of simple data instances and it additionally provides a processing hook, which allows the developers to specify additional filters or aggregations of the data instances. For example, it could contain a functionality to compute an average of the requested sensory readings. However, in such cases, it is developer's responsibility to deal with the instances compatibility, e.g., their units. More importantly our framework provides support for synchronizing the individual readings within a complex data instance. Therefore, developers only need to declare a complex instance and the framework takes care of collecting the relevant readings from

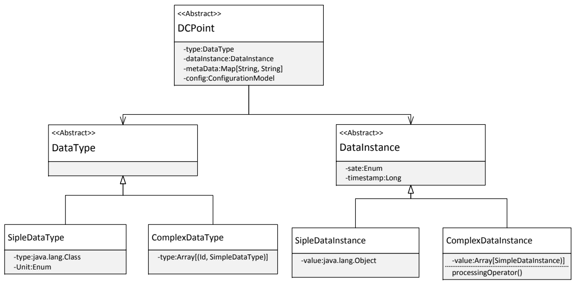

Fig. 3. Simplified UML data model.

multiple streams and delivering the complex instance to the application when it is fully initialized or updated.

C. Application-level programming constructs

Listing 1, gives a general example of how developers define a DataPoint. It shows a data point with one stream of simple data instances that represent, e.g., vehicle's tire speed, based on the required sensor properties. By default the data points are configured to asynchronously push the data to the applications at a specific rate, which can be configured as shown in the example. The application defines a call-back handler, which contains some data processing logic, e.g., based of complex event processing techniques. Additionally, the data and control points offer a `read` operator that can be used to sequentially (or in batch) read a set of instances from a stream, e.g., in order to perform more complex stream processing operations.

```java
DataPoint dataPoint = new DataPoint();
// Query available buffers
Collection<BufferDescription> availableBuffers
     = dataPoint.queryBuffers(new SensorProps(...))
// Assign the buffers to the data point
dataPoint.assign(availableBuffers.get(0));
dataPoint.setPollRate(300);
dataPoint.addCallback(this);
// Event handler
void onNewInstance(DataInstance di){  ... }
```

Listing 1. A DataPoint with callback handler.

```java
1  //Configure a custom data channel
2  BufferConfig bc = new BufferConfig("voltage_in");
3  bc.setClass(BufferClass.SENSOR);
4  bc.getAdapterChain().add(
5    new ScalingAdapter(0.0,100.0,10.0));
6  bc.getAdapterChain().add(new LowpassFilter(0.30));
7  BufferManager.create("lowpass-scaled", bc);
8  //Define diagnostics model
9  DataPoint diagnostics =
10   DataPoint.newComplexInst("lowpass-scaled","voltage_in");
11 DataInstance di = diagnostics.read();
12 //Log the diagnostics data locally
```

Listing 2. Custom configuration of DataPoints.

Listing 2 shows a more complex example of a custom data point, together with a simple diagnostics data model. The diagnostic data contains raw engine voltage readings and scaled voltage readings with low-pass filter, e.g., possibly indicating that something is taking the power away from the motor. The listing shows how to define a custom (partial) configuration for the data point. In this case, we define a scaling adapter and a filter, which are added to data point's adapter chain, as shown in lines 2-6. After creating a custom data point (virtual sensor) (line 8) application can treat this sensor as any other sensor. Finally, the example shows how to synchronously read a data instance from the newly created virtual sensor. Storing the data is omitted for readability purposes.

IV. Main Runtime Mechanisms of the DRACO Framework

A. Lifecycle of Data and Control Points

In the DRACO framework both the DCPoints and the VirtualBuffers have clearly defined lifecycles, which are entangled and mainly share a common behavior. For example, when a DCPoint is created it is associated with one or more VirtualBuffers, as shown in Figure 2 and if this DCPoint is released the corresponding buffers are also automatically released by the framework (given that no other DCPoint is referencing them). The main difference in lifecycle management of these components is that the DCPoints are mainly managed by applications (cf. Figure 2 right-hand side), while the DRACO framework manages the VirtualsBuffers and their dependencies (cf. Figure 2 left-hand side).

Figure 4 illustrates the main phases of the VirtualBuffers lifecycle. Depending on the configuration a VirtualBufer is initialized either when a device is connected or when explicitly requested by a DCPoint. The initialization phase includes allocating a buffer instance, creating a corresponding Gatherer and performing configuration directives, specified in the DCPoint configuration model. The latter usually includes creating an adapter chain with corresponding scalers, filters and adapters. This part is automatically handled by the RuntimeServices and it happens transparently to the applications. When in the Ready state VirtualBuffers are discoverable and can be queried by the applications via the DCPoint APIs. Also at this point the corresponding Gatherer is automatically started by the framework and it starts gathering the data from the underlying device or in case of a ControlPoint it is ready to receive serialization requests. After a DCPoint obtains a reference to the buffer it can decide to start it (e.g., to open a data stream) or the buffer is automatically started by the framework if the callback object is provided. After a successful start both the VirtualBuffer and the DCPoint are in the Running state. In this state the DCPoint receives periodic updates from the underlying buffer or it explicitly reads the buffer state via the read operator, i.e., sets a new state via the write operator. Finally, there are two ways to release a DCPoint/VirtualBuffer. An application can manually stop and release them after it has finished using the DCPoint or in case an error occurs, e.g, device disconnected, the VirtualBuffer is moved to a Fault state. When it the Fault state the buffer notifies the DCPoint about the error after which it is automatically released by the framework.

B. Main Information flow

Figure 5 shows the main steps of the information flow within the DRACO framework. For readability purposes, the figure only illustrates the information flow of a sensory reading (DataPoint), but the framework behaves in a similar fashion for the ControlPoins, with a main difference that the direction of the information flow is reversed.

When an application requests a sensory reading, initially the *raw physical value*, e.g. temperature is measured by a sensor that is connected to the Edge device, e.g., via a field-bus. The raw value enters the framework through a driver. The driver handles the protocol on the field-bus and

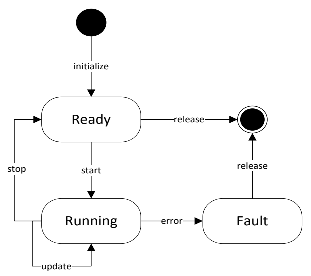

Fig. 4. VirtualBuffers lifecycle overview.

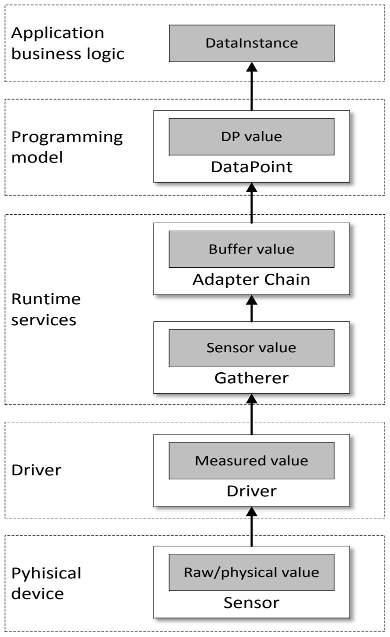

Fig. 5. Information flow of a sensory reading.

acquires the *measured value* from the sensor. In most cases the measured value is a linear function that is applied on the raw value, i.e., on a stream of bits and it scales the raw value between 0% and 100% of the measurement range (provided in the PortConfiguration, cf. Figure 2). Next, the measured value has to be formatted to a common framework internal format, which is independent of the used driver. This is performed by the Gatherer and by default it formats the sensor value as a double. At this stage the sensor reading is formatted as a *sensor value*, which is universal understood by the framework. In the next steps the sensor value is propagated and processed through the AdapterChain, i.e., the framework applies application-specific adapters and filters on it. This transforms the sensor value resulting in a *buffer value* that is delivered to a DCPoint. Finally, the DCPoints creates a DataInstance with the required format, unit and type (cf. Figure 3) and delivers it to the calling application.

V. Evaluation & Prototype Implementation

A. Prototype Implementation

The current prototype is implemented in Java programming language (based on Java SE Embedded). The framework is designed to run on a stripped-down JVM and we have created a lightweight compact profile JVM runtime specifically tailored for constrained devices. The complete source code and supplement materials providing more details about current framework implementation are publicly available[1].

B. Experiments

1) Test bed Gateway and Experiments Setup: In order to evaluate our DRACO framework, we built a test physical gateway (cf. Figure 6). The getaway is based on Raspberry Pi 2, with ARMv7 CUP and 1Gb of RAM. They run Raspbian Linux 8 (based on Debian Jessi) on Linux Kernel 4.1. Further

[1]http://github.com/tuwiendsg/SoftwareDefinedGateways

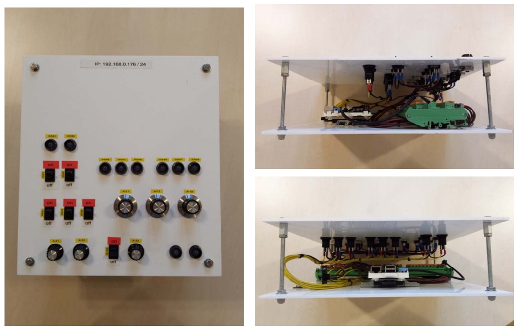

Fig. 6. Testbed gateway for Data and Control Points.

it contains a serial I2C bus system, which is used to attach the test sensors (5 digital inputs and 6 analog inputs, which are used to simulate changes in sensor readings) and actuators (8 light-emitting diodes (LED)).

C. Experiments Results

For the evaluation purposes we have developed two example applications available in the aforementioned Git Hub repository. First application (LogApp) runs inside the test bed gateway, collecting all the sensory inputs (both analog and digital), logging them locally and displaying the changes in sensors readings on stdout. It defines several tenths of the Data Points, which have different configurations such as scaling adapters and filters for the connected sensors. It also logs the raw sensory readings. Second application (ActApp) is also running in the gateway and its main purpose is to demonstrate different actuations, based on the changes in sensory readings. For this purpose it creates several Data Points (actuation triggers) and also several Control Points, which are responsible to perform actuations, i.e., in this case turning on/off the LEDs.

Figure 7 and Figure 8 show memory and CPU usage of the LogApp. Initially (Figure 7) we notice that the DRACO framework consumes below 5% of the CPU when no applications are running. The first spike in CPU consumption happens when the LogApp application is started. The reason for this is that at this point the application instantiates its Data Points and requests the framework to allocate the corresponding VirtualBuffers, AdapterChains, etc. This is also reflected in Figure 8, where we observe an increase in RAM of around $1Mb$. After this point in time the application is running (processing and logging the changes in the sensory readings). These changes are simulated by manually adjusting the analog inputs, i.e., by alternating the digital switches. In general, the application is mostly consuming less then 10% of CPU and its memory usage is fairly stable (with only minimal increase mainly due to created data instances). The smaller spikes in CPU usage represent noticeable changes in sensory readings (e.g., several knobs are rotated). However, even when all the knobs are affected, effectively forcing all the buffers to perform their individual data processing actions, the CPU usage remains below 20%. Moreover, the increased CPU usage is temporary and both the application and the framework quickly return to normal resource usage. Similar things can be observed in Figure 8, as the memory usage during the observation time remains below $13Mb$.

Similar results can be observed in Figure 9 and Figure 10,

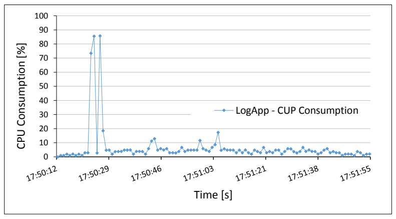

Fig. 7. CUP consumption of the example logging application (LogApp).

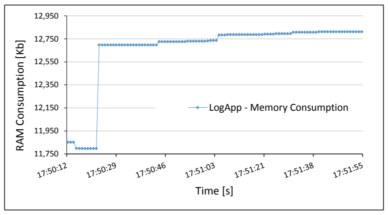

Fig. 8. Memory usage of the example logging application (LogApp).

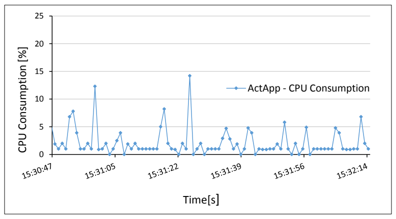

Fig. 9. CUP consumption of the example actuation application (ActApp).

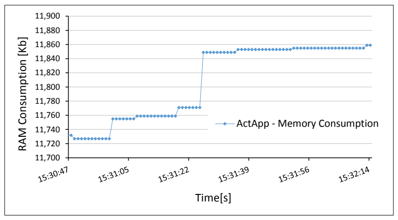

Fig. 10. Memory usage of the example actuation application (ActApp).

where we show the performance of the ActApp, which besides the Data Points also utilizes the Control Points. The main differences are reflected in the overall smaller memory consumption (below $12Mb$) and slightly higher CPU spikes. The main reason for the former is that ActApp instantiate smaller amount of Data Points. The latter is mainly due to the fact that the changes in sensory readings trigger actuations, which also require some processing to be done by the framework, such as serializing the Control Points instances. Also here the changes in sensor inputs were manually simulated by the test sensor knobs and switches. Finally, it is worth noticing that for the both experiments the memory and CPU usage was measured on the process level (i.e., entire JVM). Also, albeit small, in both cases we notice a constant increase in memory usage. This is generally not a desired behavior (e.g., since it can indicate a memory leak). In this case, however, the reason for such behavior is that the figures do not show the garbage collection of old data instances. Additionally, when an application exits it releases all its resources.

VI. Conclusion

In this paper, we introduced Data and Control Points, a programming model for resource-constrained Edge devices. The main aim of the presented framework is to facilitate development of light-weight, edge-centric applications as well as domain-specific libraries that contain reusable, generic monitor and control tasks and domain models. We presented the main features of the supporting DRACO framework: providing a virtually exclusive access to the connected sensors and actuator; enabling application-specific view on such devices; and supporting flexible customizations of the low-level sensing and actuating channels. We demonstrated feasibility of prototype and its suitability for resource-constrained Edge devices, in terms of optimized resource consumption. In the future, we plan to integrate the DRACO framework with our existing provisioning [12] and governance [13], [14] approaches, in order to provide foundations for the novel Deviceless paradigm [15].

Acknowledgment

This work is sponsored by Joint Programming Initiative Urban Europe, ERA-NET, under project No. 5631209. The authors would like to thank Dr. Manfred Siegl, Dr. Alessio Gambi, Sanjin Sehic and Michael Voegler for their help designing and developing the framework.

References

[1] H. Chen, T. Finin, and A. Joshi, "Semantic web in the context broker architecture," tech. rep., DTIC Document, 2005.

[2] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *Handheld and ubiquitous computing*, pp. 304–307, Springer, 1999.

[3] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications," *Human-computer interaction*, vol. 16, 2001.

[4] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam, "Middleware for distributed context-aware systems," in *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, pp. 846–863, Springer, 2005.

[5] J. I. Hong and J. A. Landay, "An architecture for privacy-sensitive ubiquitous computing," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, 2004.

[6] S. Sehic, F. Li, S. Nastic, and S. Dustdar, "A programming model for context-aware applications in large-scale pervasive systems,"

[7] B. Frank, Z. Shelby, K. Hartke, and C. Bormann, "Constrained application protocol (coap)," *IETF draft, Jul*, 2011.

[8] L. M. S. De Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio, "Socrades: A web service based shop floor integration infrastructure," in *The internet of things*, pp. 50–67, 2008.

[9] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *Services Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 223–235, 2010.

[10] S. Nastic, S. Sehic, M. Voegler, H.-L. Truong, and S. Dustdar, "PatRICIA - A novel programing model for IoT applications on cloud platforms," in *SOCA*, 2013.

[11] S. Nastic, H.-L. Truong, and S. Dustdar, "A middleware infrastructure for utility-based provisioning of iot cloud systems," in *SEC*, pp. 28–40, IEEE, 2016.

[12] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar, "Provisioning Software-defined IoT Cloud Systems," in *FiCloud'14*.

[13] S. Nastic, C. Inziger, H.-L. Truong, and S. Dustdar, "GovOps: The Missing Link for Governance in Software-defined IoT Cloud Systems," in *WESOA14*, 2014.

[14] S. Nastic, M. Voegler, C. Inziger, H.-L. Truong, and S. Dustdar, "rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems," in *Mobile Cloud 2015*, 2015.

[15] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: extending serverless computing to the edge of the network," in *Systor*, p. 28, ACM, 2017.