The convergence of Internet of Things (IoT) and the Cloud has significantly facilitated the provision and management of services in large-scale applications, such as smart cities. With a huge number of IoT services accessible through clouds, it is very important to model and expose cloud-based IoT services in an efficient manner, promising easy and real-time delivery of cloud-based, data-centric IoT services. The existing work in this area has adopted a uniform and flat view to IoT services and their data, making it difficult to achieve the above goal. In this paper, we propose a software framework, called CARIoT for real-time provisioning of cloud-based IoT services and their data, driven by their contextual properties. The main idea behind the proposed framework is to structure the description of data-centric IoT services and their real-time and historical data in a hierarchical form in accordance with the end-user application’s context model. CARIoT features design choices and software services to realize this service provisioning model and the supporting data structures for hierarchical IoT data access. Using this approach, end-user applications can access IoT services and subscribe to their real-time and historical data in an efficient manner at different contextual levels, e.g., from a municipal district to a street in smart city use cases. We leverage a popular cloud-based data storage platform, called Firebase, to implement the CARIoT framework and evaluate its efficiency. The evaluation results show that CARIoT’s hierarchical structure imposes no additional overhead with less data notification delay as compared to existing flat structures.

CCS Concepts:

- Networks → Cloud computing;
- Software and its engineering → Data flow architectures;
- Abstraction, modeling and modularity;

Additional Key Words and Phrases: Internet of things, data-centric services, cloud

1 INTRODUCTION

The Internet of Things (IoT) is rapidly being proposed for scenarios where various smart things—such as Radio-Frequency IDentification (RFID) tags, sensors, mobile phones, etc.—interact with each other through unique addressing schemes in a pervasive fashion. The IoT represents one of the most innovative technologies, enabling ubiquitous and pervasive computing scenarios. On the other hand, clouds have virtually unlimited capabilities in terms of storage and processing power. The need for large-scale IoT applications has triggered the convergence of the IoT and cloud computing paradigms, leveraging the scalability, performance and pay-as-you-go capabilities of the Cloud and compensating for the technological constraints of IoT systems (e.g., storage, processing and energy) [8, 42]. From the cloud perspective, the Cloud can benefit from IoT systems by extending its functionality and delivering new services in a large number of real life scenarios [36].

During recent years, several efforts have been made towards the convergence of IoT systems and the Cloud, both in the research community [21] and industry (e.g., [24, 46, 47]). A common characteristic of these efforts is their ability to stream data to the Cloud in a scalable and high performance manner, while at the same time providing the means for managing applications and data streams, and providing data-centric IoT services at scale. These efforts are highly beneficial and make more sense for applications with a large number of IoT devices, spread possibly across vast geographical areas. In such a setting, the Cloud will provide a unified and integrated platform for fast, scalable, and efficient development of end-user applications [8].
From a service provisioning perspective, IoT devices constantly make their services accessible at the cloud level (i.e., based on a push model and/or a pull model) [27]. On the opposite end of the system, the interested parties, such as end-user applications and other IoT systems and devices, will make use of these services based on protocols for service discovery and access, and their preferences in terms of quality of service requirements. In this scenario, two critical challenges arise that should be carefully addressed. First, with the presence of millions of smart and intermittent devices, a crucial need is an efficient way to model and expose data-centric IoT services in the Cloud, providing easy and fast delivery of services and their data. For example, in smart city use cases, one efficiency challenge is how to structure the increasing amount of real-time city data, process it (e.g., data gathering and aggregation locally for a city district), and provide the processed data to interested parties. Second, the event-driven nature of services in IoT systems requires a mechanism for the provision of services in real-time. For instance, in a cloud-based disaster recovery system, instant changes in environmental conditions (e.g., traffic and weather sensors) should be propagated to interested applications without any delay.

Addressing these challenges is non-trivial because of the scale, dynamicity and physical context dependency of IoT services, as well as their shareability between multiple applications. The existing work, in this area, has mainly focused on the management of IoT resources in the Cloud. Moreover, in existing frameworks, structuring and provisioning data-centric IoT services in the Cloud is either not concretely addressed [26, 30] or limited to semantic processing of data sets [31] and their relations [22]. In addition, IoT services with real-time requirements are often implemented over publish/subscribe middleware frameworks, focusing only on the type or content of IoT data [5, 42]. To conclude, for large-scale cloud-integrated IoT systems, the state-of-the-art is missing a generic design solution for IoT services and data provisioning that scales well with the size of the IoT network, IoT data diversity, and complex real-time requirements.

In this paper, we propose a Context-driven And Real-time IoT services provisioning (CARIoT) framework, which specifically addresses efficient modeling and design of cloud-based, data-centric IoT services. The essence of CARIoT is structuring the description of IoT services in a hierarchical model, the Service Access Tree (SAT), containing references to services and their real-time and historical data. Each node of a SAT represents a service delivery, processing and notification point for its own children, accessible by other nodes and third-party applications. The description of a SAT is aligned with the logical or contextual attributes of the target IoT application, e.g., physical location of services. Using CARIoT, the massive and growing number of IoT services with divers data types are structured in a hierarchical topology at different contextual granularities. This promises context-based flexibility by providing different levels of abstractions in providing IoT services and their real-time data. CARIoT features design choices and software services for creating and maintaining a SAT and linking it to the context model of the target IoT application. We have developed CARIoT on Firebase—a popular cloud-based, real-time data storage platform. The evaluation results show that, on a local Firebase server, CARIoT promises real-time notification for deep SATs (2 milliseconds for a seven-level SAT) and its structured way of organizing IoT services will induce no overhead compared to flat IoT service access models such as the Orion middleware [34].

The rest of this paper is organized as follows. The principles of service design in IoT cloud integration is presented in Section 2. In Section 3, we motivate the contribution of this paper by demonstrating a real smart city application, then, in Section 4, we present the CARIoT framework. The implementation details and the evaluation results are reported in Section 5 and Section 6, respectively. Then, we present the related work in Section 7 and conclude the paper in Section 8.

2 DATA-CENTRIC IOT SERVICES: BACKGROUND AND CHALLENGES

Prior to presenting the detail of our approach, it is important to clarify the design space that is fundamental and relevant for efficient, real-time service processing. It should be noted that the design principles discussed in this section are inspired partly from the efforts made so far by the
IoT research community, both in terms of core IoT cloud service design challenges [30, 43] and associated use-cases [15, 17]. Figure 1 shows a service-based view of an IoT cloud integration model, including the key components required for managing and processing IoT services in a cloud platform. The model is inspired by the fact that today’s IoT services are often RESTful and act as a wrapper for IoT resources, resulting in the following design concerns:

- **IoT Resource Management**: which refers to the mechanism to register and maintain the list of available IoT resources in the Cloud. This should also be enhanced with a discovery solution to discover the IoT resources that are relevant or of interest to the end-user application. A well-designed resource registry model can result in a more efficient discovery process.

- **IoT Service Delivery and Processing**: which refers to wrapping IoT resources and making them accessible and invocable by end-user applications. Beyond this, if any events of interest are detected, the service delivery mechanism should propagate event-based services and route them to other interested services or applications. Integrated with the Cloud, the scalable storage capabilities of clouds can be utilized to analyze both big data produced by IoT devices [44] and maintain a configurable history of service data. Thus, two complementary components are required for cloud-based, data-centric IoT services delivery: the control plane and the data plane. The control plane is responsible for real-time and near-real-time (minutes, hours, < 24 hours) views, as well as control of devices. The data plane is the historical aggregation and business logic queries, usually across many devices.

- **Service Integration**: At one higher level, end-user applications and other IoT systems may be built using composition and/or coordination of individual services offered by distributed IoT devices. An important insight into the cloud IoT service landscape is to have a repository of all connected service instances, which are executed at runtime to build composed services. The creation and management of composite services can be expressed through, e.g., business processes, interacting with external entities through Web Service operations using standard languages such as the Business Process Execution Language (BPEL) [43]. Similar to the service delivery, the integration service should support event-based interactions between services.

In order to put our work in context and motivate the proposed approach, below we highlight the associated challenges in data-centric IoT services design:

**IoT Resources Registration.** Expecting a huge number of IoT services, a crucial need is efficient storage mechanisms for registering IoT resources (cf. IoT Resource Registry). The basic model is to follow a table-like structure, such as a cloud-based SQL database. However, the structure of the registry for IoT resources is particularly important for the following reasons. First, the sheer number of resources in the Cloud requires a highly scalable registry mechanism to ensure swift and real-time discovery of resources. Second, unlike many conventional distributed systems, resources in IoT systems relate to each other semantically and contextually, e.g., a traffic monitoring service that invokes services offered by co-located IoT resources. This calls for a resource registration model that adheres better to the physical and contextual structure of IoT services in the environment.

**Event-Driven Processing.** The observation design pattern in the COAP protocol [27, 41] is a recognized basic solution for observing device resources and notifying registered observers about changes in a resource. However, in a broader context, event-based interactions between IoT cloud services may appear in more complex forms than in the above. For instance, as a case of multiple notifications, multiple monitoring services may be interested to receive the data reported from...
a smoke sensor, with differing criteria, e.g., the notification time period. As another example, in contextual notifications, devices populated under a pre-defined context boundary may be of interest for event reporting, e.g., listening to all sensing events produced in a given region of a city. Note that this dynamic and multi-dimensional model of processing event-driven services cannot be easily implemented over publish/subscribe middleware such as MQTT [23]. This calls for an efficient context-driven model for processing event-based IoT services which allows event filtering at different levels, from a sensor reading to changes in a physical context and shared notifications.

**Historical Data Provisioning.** IoT devices are often characterized by their mobility and being transient due to power limitation. Consequently, the services exposed by such devices are neither reliable nor available all the time. On the other hand, their historical data is a good source of information when they are not connected to the Cloud. With respect to mobility, IoT devices may switch between different networks and operation environments, resulting in different perceptions of the produced data (i.e., can be interpreted as context-based data). Ideally, the applications that communicate with such types of services should be instantly notified about, e.g., presence or absence of a mobile service by the Cloud. Besides that, keeping track of historical service data can be highly beneficial when, e.g., the mean value of data is needed or the target service is out of access. These raise the challenge on how to handle the huge amount of data generated by data-centric services integrated to clouds and make the data easily accessible for interested applications.

### 3 AN APPLICATION SCENARIO

In many IoT cloud application scenarios, the development of data-centric IoT services may involve one or more of the above challenges, in particular the use-cases that require coordination and interaction between multiple IoT systems or device groups. In this section, we study a real application scenario and investigate its design aspects with respect to the aforementioned challenges.

The application scenario is taken from ClouT [15]—a recent research project that focuses on IoT-based smart cities. The main goal of this project is to leverage the integration of the IoT and cloud computing to establish an efficient communication and collaboration platform exploiting all possible information sources to make cities smarter and address associated challenges, such as efficient energy management. Urban context-aware applications and safety and emergency management are two of those application domains studied in the ClouT project. The purpose of the urban context-aware application trial, deployed in the city of Fujisawa, is to detect and leverage sensorised social web and IoT for detecting city events such as festivals, conferences or accidents. In addition, Fujisawa classifies city events in terms of their property such as size and contents. Detected and classified events are useful for further services such as navigation and event recommendation.

A considerably larger deployment, in the context of ClouT project, is Traffic Mobility Management in Santander, enabling citizens and visitors to get access to enhanced urban mobility experience and leveraging city transportation resources efficiently. SmartSantander has around 180000 stationary and mobile sensors of various types in the municipality of around 180000 residents [25]. This smart city project is intended to gather and process information from many different data sources related to transport, mobility, public buses fleet, bikes, taxis and trains. This information is combined with information related to traffic parameters, such as speed, occupancy degree, indoor and outdoor parking lots, as well as with environmental indicators such as CO2, NO2, O3, and noise. SmartSantander API (see Section 4.6 for detailed description) provides an interface to access IoT data. The /measurements root is introduced for storing and retrieving IoT events. An example event, sent to the server, looks like [32]:

```
{"eventData": {"eventName":"2016-04-22 13:59:37", "eventLocation": {"latitude":43.471997, "longitude":-3.800185}, "text":"comment", "imageData": "link/ID", "imageFormat": "JPEG", "expirationTime":1398167978883}, "title": "Leakage", "type": "WATER"}
```

As an indication of the size of produced application data over the SmartSantander framework, within three months about 50 GB data has been collected from 1112 sensor nodes of nine different...
The main sequence of actions, in the above applications, is illustrated in Figure 2. Firstly, IoT data is sent to the Processing service to sensorize the data and transfer it to the database. Event Detector retrieves historical sensorized IoT data from the database, and detects events by analyzing spatial-temporal information of the IoT data. Event Detector also sends information of interest for third-party applications (i.e., City IoT Data Consumer) to Event Classifier for further analysis of events. Event Classifier classifies events in terms of, e.g., their popularity. Classified event information is used for navigation or recommendation. For instance, the Genova pilot application uses the cloud storage functionalities, provided by the ClouT platform, to store historical data of sensors. This is achieved by using a specific software agent that performs data polling from the Genova infrastructure services. The cloud storage offers an API to get access to the stored data by different applications on-demand:

GET http://host:port/clout/devices/{deviceID}/services/{serviceID}/resources/ {resourceName}/GET

For example:

GET http://<endpoint>:8080/clout/devices/SmartPlug_0/services/ PowerService_SmartPlug_0/resources/status/GET

In addition, it provides an API to subscribe to data change notifications of a given resource:

POST http://host:port/clout/devices/{deviceID}/services/{serviceID}/ resources/{resourceID}/SUBSCRIBE

For example:

POST http://<endpoint>:8080/clout/devices/WSPT_XBEE_4565/services/ for_WSPT_XBEE_4565/resources/for/SUBSCRIBE

The service discovery mechanism, in ClouT, is based on UDDI (Universal Description, Discovery and Integration). Despite its potential, UDDI lacks support for appropriate scalability, autonomy of individual registries, and volatility of IoT devices [20]. Thus, from Figure 2 and the above examples, it can be concluded that discovering one or a set of relevant services is not a trivial task, in particular for smart city scenarios in which applications are often interested in services bounded to a location context. Additionally, in the event-based interaction model of ClouT, the event subscription model is built on a flat device-oriented description of IoT resources (cf. last POST example). However, IoT applications are often interested in events published within a particular context boundary such as a street or a building. Thus, there is a clear need for efficient context-based discovery of data-centric IoT services and real-time access to their data in cloud-based IoT platforms. To the best of our knowledge, there is no approach that specifically addresses these challenges for cloud-based IoT applications with the above requirements.

4 THE CARIOT FRAMEWORK

The design of CARIoT is founded on the idea of context-based structuring of IoT resources and providing the required services and data at different context levels in an efficient manner. Our approach for context-driven and real-time provisioning of data-centric IoT services is guided by two fundamental questions: i) how to design an efficient context-based hierarchy of IoT resources that enable multi (contextual) levels of reading, updating, monitoring and storing IoT service data
in the Cloud; and ii) how to process and publish real-time service data to interested parties at different context levels. Driven by these questions, we highlight four key design requirements:

- **Hierarchical context modeling and abstraction for IoT resources**: Inspired by the hierarchical contextual parameters of most large-scale IoT applications, we propose a hierarchical service provisioning framework adhering to the contextual architecture of IoT systems. For example, the location context for a smart city can be hierarchized from high level city municipal districts down to neighborhoods and streets.

- **Multi-level context-aware access to real-time data generated by IoT resources**: In addition to real-time access to individual IoT data items, we propose to provide multi-level access to real-time data based on the context scope of target IoT data. In particular, we define the provision of real-time data at higher levels of abstraction, in addition to actual device-level access. For example, a third-party application may be interested to receive real-time weather information for a particular district of a city (based on pre-defined criteria).

- **Multi-level access to historical IoT data**: Besides real-time data, the service provisioning framework should support multi-level access to historical data of an actual resource or a set of resources in a context boundary. This can be further generalized to multi-level historical data access for large-scale IoT applications. For example, a vehicle may need to subscribe to average smoke level of a street during the last number of hours.

- **Context-driven processing of queries for real-time and historical data access**: Finally, being used in large-scale applications with a massive number and variety of IoT resources and context abstractions, CARIoT should be enhanced with a mechanism for efficient processing of queries for accessing real-time and historical data. This is an important feature as, without such a mechanism, identifying criteria for such data for each individual device or context boundary would be almost impossible in large-scale deployments.

Prior to describing the design details of our approach, we present the conceptual model of our framework and briefly discuss how we address the above design objectives in a single service provisioning architecture. The overall design approach is a *context-driven tree-based IoT service provisioning model*. Figure 3 depicts the general structure of such a tree, including the core design elements. Each node of the tree is created based on hierarchical contextual parameters of the IoT application. We name these types of nodes *Context Nodes*. At the level of leaf nodes, in a hierarchical path, IoT resources are located, called *Resource Nodes*. Each node of the tree maintains a history of data collected in that particular context boundary of the application, e.g., smoke level of a street during the past \( n \) number of days. Beyond that, each node can publish real-time data of its context as *events*, e.g., a road accident. For both of these data types (i.e., historical data and real-time data), any other nodes of the tree can subscribe to receive the data of interest. For example, the traffic control system of street \( S_1 \) may depend on the load on the nearby street \( S_2 \), thereby the context node associated to \( S_1 \) can subscribe to traffic events published by the context node of \( S_2 \).

Being proposed for large-scale applications, it would not be feasible to identify data subscriptions individually for each context node. We propose Notification Rules Distribution to provide an efficient and generic way of describing notification rules for the end-user application and distributing them throughout the service provisioning tree based on the properties of each context node. In the rest of this section, we explore the aforementioned design choices in detail.
4.1 Service Access Tree

An important design aspect of CARIoT is extracting efficiently the target IoT application’s context and mapping it to the hierarchy of context elements. We believe that the tree structure is a right design choice for context-driven service structuring and provisioning in cloud-based large IoT systems. This is analogous to typical graph-based data structures. However, the hierarchical top-down access to IoT resources and the corresponding contextual boundaries are hardly achievable by a general graph-based modeling of IoT resources. The graph-based model is suitable when a single IoT resource needs to be associated to several contexts that are not hierarchically related, e.g., in the smart home scenario presented in [18]. However, the main issue in this type of modeling is that it does not scale well for large-scale deployments. It should also be noted that a given graph-based context model may be convertible to a tree or a forest with multiple roots.

To design the context-based service provisioning model, the main context elements in IoT systems should be identified. As described in [35], the main context elements for a typical Internet-connected thing include identity, location, time and activity. These are basically derived from the primary context types introduced for computing systems [2]. As an example of identity, the context information may include information about the user of a smartphone. The user’s identity is made available and augmented with other information, such as the user’s profile or activity. Besides such context types which are more relevant for the user of things, location and time are the core context types linked to smart things themselves.

Contextual information, in CARIoT, is categorized into non-temporal and temporal context attributes. Each node of a SAT has one non-temporal context attribute and we consider one or more temporal context attributes for data items associated to a node, as shown in Figure 4. For example, a context node in the smart city scenario may be associated to a city district (i.e., location context) and support real-time data notification and maintenance of historical data (e.g., a date range) related to that district of the city.

Each IoT resource has a resource type, which can be an instance of either of the following types: sensor, actuator, or tag [14]. The Service Access Tree (SAT) is the core part of the system for context-driven service provisioning. As discussed above, the SAT is built to reflect the target IoT application’s context information. It can include the key context type of location, or other types such as user-related context and the IoT domain. We propose to add the IoT domain to the hierarchy of the SAT as IoT resources are often shared between many domains. Although location is the most relevant data type for context hierarchy, in some applications service provisioning may be based on only a domain or sub-domain context. As an example, the URI for an atmospheric resource represents a combination of domain and location hierarchy for service access.

SAT Builder is one of the main components of the CARIoT framework which builds and maintains a SAT. We define a SAT as follows. A SAT $T$ is defined as $T = (r, \{t^1_1, t^2_1, t^3_1, \ldots, t^n_1\})$, where $r$ is the root of the tree and $t^i_1$ is a subtree of $T$. Each subtree $t^i$ contains a set of subtrees (its children) $\{t^{i+1}_1, t^{i+1}_2, t^{i+1}_3, \ldots, t^{i+1}_k\}$. Note that if a subtree does not possess any children, it is obviously a leaf node and, in the context of our work, it is considered as an IoT resource node $s^i$. Thus, a given resource $s^m$ has predecessors $(t^{m-1}_1, t^{m-2}_1, \ldots, t^i_1, r)$. This means that for getting access to a service provided by resource $s^m$, the path $P = r/t^1_1/\ldots/t^{m-2}_1/t^{m-1}_1/s^m$ must be traversed, which actually represents the URI of an IoT resource.

Fig. 4. IoT context modeling dimensions in CARIoT
Each IoT device, added to the system, should come with the description of its resources along with the hierarchy of parent context nodes, e.g., resource $s$ with pre-defined context path $t^{j-2}/t^{j-1}/t^j$. Then, the SAT Builder component will traverse the tree to find the context path $t^{j-2}/t^{j-1}/t^j$ and add the new resource $s$ under $t^j$. If this path does not exist, SAT Builder will first create the path, and then add the resource to subtree $t^j$. It should be noted that the context path is a unique path. Resources of a device can share the same context path or introduce their own paths, e.g., a smoke sensor can be associated to the city district context, while the light sensor belongs to the street context. A SAT is developed by dynamic insertion of new resources and the context hierarchy will be completed accordingly, based on the paths introduced by added resources. Whereas the whole tree data structure is stored in the Cloud, intermediate gateway devices and the leaf IoT devices can take part in processing service requests, discussed in the next subsections.

### 4.2 Real-Time Notification

As shown in Figure 4, a temporal context can be of type Real-time, TimeRange, or DateRange. While the last two aspects are proposed as de facto semantical modeling concepts for IoT resources [14], we add the real-time aspect to particularly meet the real-time requirements. TimeRange and DateRange support the provision of historical service data, in contrast to the Real-time feature which publishes resource events to the interested nodes of the SAT.

A context node is empowered with the ability to generate events. The kind of events that can be generated by a context node include adding a child, removing a child, and disconnection of a resource node, e.g., adding a new data item for a resource, removing an old item, or a dead IoT device due to failure, respectively. Conversely, context nodes can listen to real-time events generated by other nodes in the tree. In general, two types of event subscribers are envisaged: IoT devices and end-user applications. As an example of the former, the information board in the Fujisawa smart city application acts as an IoT device interested in receiving real-time data from surrounding sensors in the environment. Emergency management applications and central traffic monitoring applications are examples of the latter category, to name a few.

In order to achieve context-driven event dissemination among SAT nodes, CARIoT distinguishes between two kinds of event subscriptions: subscribing to events from a context node and from a resource node, respectively. A given node of a SAT, $t^i$, can subscribe to the receipt of events from both context nodes and resource nodes. Suppose $t^i$ subscribes to smoke alert events generated by node $t^j$. If $t^j$ is a resource node, all events generated by this node will be forwarded directly to $t^i$. Otherwise, all events matching the subscription and generated by the nodes in all subtrees of $t^i$ will be forwarded to $t^i$.

As mentioned above, we consider three basic types of events for a given subscriber node $t^i$: child $t^{i}_{n+1}$ has been added under $t^i$, i.e. $t^i.onAddChild(t^{i}_{n+1})$, child $t^i_m$ has been removed from $t^i$, i.e. $t^i.onRemoveChild(t^{i}_{m})$, and a resource node $s^m$ is disconnected, i.e. $s^m.onDisconnect()$. Within resource nodes, the first two event types are linked to new, removed or updated sensed data items, respectively. However, in a context node, they refer to adding or removing a node in its descendant context nodes or resource nodes. Specifically, they are used for maintaining the SAT structure for changes in the applications and the network architecture. For example, when a new device joins the network, it offers new resources which should be added to the relevant content node in the SAT. Likewise, when all resources under a context node are removed (due to, e.g., removing the IoT device as resource owners), the node itself should be removed as well.

$onDisconnect()$ is an important event which allows the subscribers to handle absence of a resource node in the SAT due to devices that fail or leave the network without notification (i.e., without invoking $onRemoveChild()$). In this case, the subscribers that have subscribed to the receipt of $onDisconnect()$ event from a resource node will be notified. As part of the real-time notification mechanism, the CARIoT framework is in charge of detecting such events and publishing them to interested subscribers in the SAT.
We illustrate the use of the above concepts with an example on safety and emergency management in the city of Fujisawa. In the famous Subana street, there are often many tourists and there is a need to provide evacuation information to tourists in case of emergency. Emergencies can be detected using sensors deployed throughout the street such as atmospheric sensors delivering CO, NO2, O3, dust, temperature, humidity, luminance, and air contaminants measurements. Based on our tree-based design model, the real-time information about this street can be accessed through:

GET http://<endpoint>:8080/clout/smartcity/japan/fujisawa/subana_street/atmospheric

The obtained atmospheric information can be used by the government or city in case of emergency, e.g., Tsunami. In addition to the above street-level monitoring, real-time information (such as weather, traffic and day events) about the Enoshima area in this street can be displayed through the designated information board—called Enoshima info surfboard. For example, to update the day event information, the following PUT call must be made:

PUT http://<endpoint>:8080/clout/smartcity/japan/fujisawa/subana_street/enoshima/dayevents{updateJSONData}

Considering the scalability of design, in the SmartSantander scenario presented in Section 3, we showed that the flat model of city events management would not address this issue. Using the real-time notification model of CARIoT, each event is associated to the corresponding location context according to the GPS position where the event takes place, resulting in a scalable design for service access. Therefore, the example data given in Section 3 will be accessed through context nodes districtA/locationB/leakage, where each node publishes events in its own boundary, e.g., to access leakage events at the level of locationB:

GET http://<endpoint>:8080/smartsantander/districtA/locationB/leakage

4.3 Multi-Level Historical Data Access

The temporal context types TimeRange and DateRange support the provision of historical service data. Analyzing historical data is often the first step in understanding the meaning of data or predicting some data patterns for the future. End-user applications may perform some basic statistics on IoT data to find anomalies, and they may also clean the data by removing bad data points or filtering out noise. In urban planning, historical data from IoT devices may be used to plan for the future. For example, by analyzing electricity consumption of an urban area from previous years, we can predict the demand for the next year and take the necessary action to satisfy the demand [1].

In our service provisioning framework, each node of the SAT will maintain the list of data items gathered in all subtrees of that node. This makes more sense for resource nodes, where environmental IoT data is sensed and stored. In context nodes, historical data is essentially the analyzed data at a particular context level. In other words, context node t_i may contain a set of data analysis functions \( A = \{ \delta_1(S_{r_1}, D_1), \ldots, \delta_n(S_{r_n}, D_n) \} \), where \( S_{r_i} \) denotes the set of resources to analyze and \( D_i \) specifies the range for analyzing the historical data. The latter can vary from a time range to a date range.

CARIoT processes the requests at context nodes levels by collecting historical data of each individual resource in \( S_r \) based on the value of \( D \) and then it executes the associated function \( \delta \). The data collection process is recursive such that all subtrees of \( t_i \) will be traversed recursively down to resource nodes, and then the data meeting the function criteria will be returned up towards \( t_i \). Again, it should be noted that the whole SAT and the historical data reside on the cloud platform and CARIoT will execute the historical data retrieval requests in the Cloud and return the results to the requesting party.

4.4 Context-driven Notification Processing and Distribution

The above two features of real-time and historical data processing would need a notification mechanism for defining events of interest, identifying corresponding event publishers and subscribers,
and actuations (if any) as a response to generated event notifications. Being used for large-scale IoT applications, it would not be feasible to subscribe to notifications individually for each node of the SAT. On the other hand, the data access logic in typical huge IoT applications is essentially described at a high-level, e.g., rerouting vehicles to a new road if the smoke level of a given street exceeds a given threshold. Therefore, the CARIoT framework needs a context-driven notification processing and distribution solution that allows describing notification criteria at the SAT’s root level and distributing them to lower level subtrees.

Bridging the vast information gap between context sources and context-aware services has been a major challenge in pervasive systems. As will be discussed in Section 7, a number of context processing frameworks have been proposed to address context data distribution and dissemination, and ontology-based and conceptual context processing [9, 10, 39, 48]. Our main concern is to devise a distributed context query approach that is distributed through the SAT and enables context-aware provision of services to interested context nodes in the SAT, as well as to third-party applications. Among existing solutions, the COPAL context processing modeling and programming framework meets the basic requirements of our framework such as context data modeling and processing [29, 40]. It also comes with the COPAL-ML macro language for the development of context-aware ubiquitous applications. We believe that COPAL serves as a good basis for modeling the relations between context data producers and subscribers in CARIoT. However, it does not meet the main requirement of CARIoT, i.e., efficient context-driven notification processing and distribution. Prior to discussing our solution for this part, notification processing and distribution in CARIoT is presented below.

In CARIoT, notifications are distributed across the SAT using Notification Rules (NRs). A NR is described in the form of Condition:Action. The key concepts for describing a NR are: i) Condition Context: context nodes that publish notifications; ii) Action Context: context nodes that should receive the notifications. For example, a simple NR is:

$$(\text{fujisawa}.\text{subanast}.\text{atmospheric}.\text{smoke} > c): \text{fujisawa}.\text{enoshima}.\text{dayevents} \ \text{DISPLAY} \ e$$

where \(\text{fujisawa}.\text{subanast}.\text{atmospheric}.\text{smoke}\) is the Condition Context generating the smoke event \(e\) and \(\text{fujisawa}.\text{enoshima}.\text{dayevents}\) is the Action Context receiving \(e\) and displaying it if it is beyond the defined threshold of \(c\). To process a NR, CARIoT first finds the relevant Condition Context and Action Context based on the description of the NR. Then, it configures Action Context to receive notifications from Condition Context.

In CARIoT, we envisage two forms of identifying Condition Context and Action Context: context node-based and context type-based. The context node-based form allows application-specific real-time data notification, in which a specific data producer context node in the SAT is linked to another specific data consumer context node of the SAT, as shown in the above simple NR in which the node \(\text{fujisawa}/\text{enoshima}/\text{dayevents}\) subscribes to events generated by \(\text{fujisawa}/\text{subanast}/\text{atmospheric}/\text{smoke}\). The context type-based form is aimed to link data producers and data subscribers based on the type of context nodes. For example, in the smart city system, a NR can be defined as:

$$(\text{type} \cdot \text{city} \cdot \text{street} \cdot \text{atmospheric} \cdot \text{smoke} > c): \text{type} \cdot \text{city} \cdot \text{street} \cdot \text{intersec} \cdot \text{light} \ \text{INCREASE} \ s$$

Based on this NR, for any context node \(t^i\) with context type of \(\text{city} \cdot \text{street} \cdot \text{atmospheric} \cdot \text{smoke}\), if the smoke level is greater than threshold \(c\), the length of a traffic light period on intersections should increase \(s\) seconds. This implies that each node of the SAT should be associated with a context type, e.g., the context type of \(\text{fujisawa}\) is \(\text{city}\). This form of notification is very useful for large SATs in which it would be a complex task to define context node-based NRs (i.e., individually specifying all NRs for each pair of Condition Context and Action Context). In fact, context type-based NRs ensure higher flexibility and enable an efficient way of linking IoT data producers and subscribers.

As mentioned above, we introduce the general form of \(\text{Condition} : \text{Action}\) as a basis for the distribution of NRs. The Backus Normal Form (BNF) syntax for describing NRs is as follows:
In the above grammar, \(<\text{cond-opr}>\) denotes the standard conditional operations such as \(>\) and \(<\). The \(<\text{act-opr}>\) symbol specifies actuation operations, e.g., INCREASE, DECREASE, DISPLAY, etc. \(<\text{val}>\) identifies a value for evaluating conditions or performing actuations. \(<\text{ctx-expr}>\) represents a context expression, which is either a context type expression or a context node expression. To distinguish these two types, context type-based expressions are prefixed with type: ‘\(\text{ctx-type-expr}\)’ or ‘\(\text{ctx-type-rest}\)’.

Below, we present how NR processing is designed using COPAL as the reference context modeling and language solution. Figure 5 illustrates the components for describing and processing NRs. NR is central to the model, interacting with ConditionContext and ActionContext to evaluate the defined condition and action expressions. Both of them contain references to an implementation of interface ContextExpr, which is realized either as ContextType or Context Node. ConditionNode, generating IoTEvent, is dependent on the ConditionContext component. Likewise, ActionNode, receiving IoTEvent, is dependent on ActionContext.

![Fig. 5. Components for describing and processing notification rules](image)

Shared by ContextType and ContextNode, a key design concern is their search() method, which searches the SAT for the tree nodes specified in NRs. The search() method for ContextNode is straightforward, implementing the code for accessing the actual context node’s hierarchical name. However, for implementing this method in ContextType, we introduce the Context Access Tree (CAT) which contains a reference tree-like structure of context types hierarchies of a given IoT domain. As shown in Figure 6, each node of the CAT maintains a list of nodes in the SAT which are associated to the context type of that particular node in the CAT. For example, \(\text{type.city.street}\) in the above example NR is a node of the CAT which contains a list of actual street names (e.g., Subana street) in the SAT. In this way, CARIoT maintains a tree data structure which only represents the context types, their relation and links to actual instances of each context type in the SAT.
This will not only make NRs processing more efficient, but also allow more flexibility in defining context types in the CAT, e.g., using an ontology-based descriptive specification model of the target application domain to create the CAT.

4.5 CARIoT’s Architecture

Having discussed all different design aspects of CARIoT, in this subsection we present an overview of its system architecture built on the above design choices. Figure 7 shows the main components of CARIoT and their dependencies. At the lowest level, our platform is built on the real-time data storage system of the target cloud platform. This component supports basic APIs for storing and retrieving IoT data from the cloud storage system. In the implementation section, we discuss this component further within a popular storage platform.

CAT and SAT, in the center, store IoT applications’ real-time data and the associated context types tree, respectively. As explained in Section 4.1, when a device is added to the system by the developer, the hierarchy of context nodes and context types, and its resources are known. The developer needs to invoke the API, offered by SAT Builder, along with the above information as arguments in order to enable initialization of the SAT and the CAT and their later updates. Therefore, SAT Builder interacts with the SAT and the CAT to first create these trees and update them later when a device is added. Context Discovery is in charge of searching the context types in NRs and retrieving the associated real context data from the SAT. Then, the Notification Rules Processor component will be able to link the target ActionContext to the defined ConditionContext. The Real-Time Notification component processes the real-time event notifications from the storage and publishes them to the subscribing parties. Finally, the Historical Data Access component communicates with the SAT to process the historical data access requests at different context levels.

4.6 Use Case Study

To illustrate the benefits of our approach, we consider one of the field trials studied in the ClouT project and remodel it based on the modeling solution proposed in this paper. The chosen field trial is based on the SmartSantander IoT platform [12]. The SmartSantander IoT API is the main interface to access the platform tier from deployed elements, external sensors and experimenters or applications. Figure 8 shows the different endpoints mapping the functionalities offered from the IoT API interface. The management of the resources is made through the /resources root, while access to the historical values is made through the /observations root, which may contain several measurements for each resource, and the /measurements root with only specific measurements. Specific requests can be made using the /queries root and subscription is performed through /subscriptions. In addition, the /observation root is also used to inject new data to SmartSantander.

Figure 9 illustrates the redesigned Santander IoT API using the CARIoT framework. While the new model still supports all APIs offered in the original version, it allows inclusion of contextual information and their association to IoT resources. Specifically, /measurements and /observations are assigned to and provided by each individual resource, in addition to supporting /queries.
greater advantage is achieved by supporting subscriptions at both the context node level and the resource level. Therefore, the programmer is provided with a context-driven and simple API for IoT service access and IoT data processing. Beyond that, the context-driven SAT enables efficient mapping between IoT services and their context.

In order to quantify the benefit of redesigning the Santander IoT API based on CARIoT, we focus on the key feature of data subscription in SmartSantander and its notification model, and measure the time overhead in IoT data notification (cf. Section 6). The SmartSantander system is built on the FiWare Orion Context Broker platform [34]. All IoT data items in Orion are stored under a root context node, called entities. On the other hand, under the subscriptions node, we can define subscriptions. We evaluate this by adding the following sample IoT data items to Orion:

```json
{
  "id": "$ID",
  "type": "User",
  "city_location": {
    "value": "$Dist",
    "type": "CityDist"
  },
  "temperature": {
    "value": 23.8,
    "type": "Number"
  }
}
```

where ID and Dist will be set in the sample program developed for IoT data management, e.g., for a district in the city of Madrid. On the other hand, we add the following subscription to the list of subscriptions available at Orion Context Broker:

```json
{
  "description": "A subscription to get info about room1",
  "subject": {
    "entities": [{
      "id": "12500",
      "type": "User"
    }],
    "condition": {
      "attrs": [
        "temperature"
      ]
    },
    "notification": {
      "http": {
        "url": "http://<myLocalIPAddress>:port",
        "attrs": [
          "temperature"
        ]
      },
      "expires": "2017-01-14T14:00:00.00Z",
      "throttling": 5
    }
  }
}
```

## 5 IMPLEMENTATION

The realization of the presented design approach largely depends on the target application and its conformity to the hierarchical context boundaries presented in this paper. Beyond that, we need to provide the generic tree structure for populating IoT resources and provisioning real-time multilevel access to service data. The ideal way to realize the latter is to investigate existing cloud computing platforms for efficient data storage and processing and build our framework on a platform that meets the design goal of context-driven and real-time service provisioning.

The proposed data structure for maintaining data-centric services in CARIoT can be implemented either on NoSQL data storage models that ensemble tree structures or other types of NoSQL databases. Although the former adheres to the hierarchical model of service access in CARIoT, in those models the hierarchical path of services and their data are basically stored according to the \((key, value)\) storage model [16], in which the key should contain the full hierarchical path of a service. The latter models apply other techniques such as columnar [4], document-based [33], or plain key-value models [38]. Therefore, the hierarchical data access model of CARIoT will be eventually converted to the native data storage model of the chosen cloud storage model. This means that the hierarchical service access model of CARIoT will not impact the performance of the proposed framework. However, the storage implementation choice matters with respect to the expressiveness of the API for hierarchical service access and supporting the real-time event notification types in CARIoT.

We choose Firebase [16] as the suitable cloud-based platform from the first category discussed above. Data in a Firebase database is stored as JSON and synchronized in real-time to every connected client. The hierarchical structure for describing the data plays a key role in providing a meaningful perception of individual data items stored in Firebase. Moreover, Firebase provides the primitive functions for supporting the three event types supported by CARIoT, including adding a
node, removing a node, and disconnection of a resource node. For example, a mobile application can upload and add its current sensory information and the changes will be saved automatically in real-time so that other users of the mobile application or third-party applications can be updated about such real-time changes.

CARIoT is implemented as a Node.js server application, deployed in the Cloud and integrated with Firebase. We have developed CARIoT’s components as Node.js modules, deployed within a Ubuntu 12.04 instance which is set up in Amazon EC2. Node.js is an open-source, cross-platform JavaScript runtime environment for developing server-side web applications. CARIoT resides between Firebase and the end-user application. It processes the hierarchical resource access tree of the application (i.e., currently as a JSON file) and then creates basic services for service delivery, service notification, and historical service data access, as shown in Figure 7. Although the framework API is generic and independent of the end-user application, its current implementation is based on Firebase’s solution for notification and historical data querying.

On the other side of the system, to enable end-user applications or IoT devices to interact with CARIoT, we can use either native REST API provided by Firebase or native libraries, e.g., the Node.js Firebase module. On IoT platforms such as Raspberry Pi nodes, both of the above approaches can be used, even though native libraries better support event-based notifications. However, more resource-constrained devices should rely only on the REST API for service data provisioning and notifications. Firebase offers streaming REST for streaming push to any device that can support HTTPS. To better understand the data structuring and processing of our framework, as a proof-of-concept experiment, we have developed the sample scenario of Fujisawa smart city over CARIoT using the Firebase platform. Figure 10 illustrates the Firebase data console for the SmartCityIoT-Cloud application that we have developed. As shown in the figure, the IoT data is organized in accordance with the hierarchical location context of IoT devices spread in the city. Under each location context (e.g., Subanastreet and Shanonstreet), both real-time and historical data are made available to interested applications and other IoT devices.

6 EVALUATION

We consider two main aspects of the evaluation for CARIoT. First, we need to investigate the overhead of the hierarchical structure of the SAT and the CAT and assess the impact of the proposed structure on the efficiency of service provisioning. Second, the efficiency of CARIoT’s service access model should be evaluated in terms of timeliness as compared to the state-of-the-art flat service access model (i.e., Orion).

6.1 Evaluation of Service Hierarchies

The first part of evaluation is devoted to measuring the overhead of CARIoT in terms of context tree creation and processing. To evaluate the overhead of introducing the CAT for NR processing, we have built a sample SAT for an exemplary smart city system, in which the SAT basically describes the location-based IoT services and data. Figure 11 shows the exemplary SAT on the right side,
Provisioning of Data-Centric IoT Services in the Cloud

illustrating a part of a location-based processing structure of IoT services in smart cities. On the left side, the CAT associated with the SAT is shown, which is considered as a reference location context description for the given smart city application. Each context node of the CAT maintains the list of nodes in the SAT which are assigned to that particular context. For example, the Lot node contains links to the three physical context nodes in the SAT. In our implementation, a link is the complete URL of a context node in the SAT, e.g., https://smartcityiotcloud.firebaseapp.com/root/city1/dist1/prk1/lot1.

Fig. 11. An exemplary SAT and the associated CAT for smart city applications

To measure the time overhead of CAT building and processing, we developed a sample application which uses the CARIoT framework to build both the CAT and the SAT and add IoT data items to resource nodes and context nodes of the SAT. The application adds an equal number of data entries to all resource nodes and does the same for context nodes, e.g., 100 data entries to all resource nodes and 10 entries to all context nodes. As a worst case scenario, we assume each data update in CARIoT requires to check the CAT and update it as well (if a particular context node in the CAT does not exist). This scenario is performed for three different cases: i) the CAT is created in parallel with data insertion to the SAT, ii) we assume that the CAT is already built and only the SAT is updated (however the CAT needs to be checked for each SAT update), and iii) it is assumed that the CAT is not part of our framework and CARIoT only builds and updates the SAT.

Figure 12 depicts the behavior of CARIoT when performing the above scenarios for different number of data entries, ranging from 100/10 (100 to each resource node and 10 to each context node in the SAT) to 500/50. For example, to add 300 data entries to each of nine resource nodes and 50 to each of seven context nodes, the time overhead will be 1552 milliseconds (ms) for the first case (i.e., building the CAT and the SAT together). It can be concluded from the graph that, considering the number of data entries added in each round, the above three cases are very close in terms of time overhead, meaning that adding the CAT imposes no significant time overhead to the system. The graph also shows that for lower insertion rates (e.g., 100/10), the second case (i.e., the CAT is already built) slightly outperforms the other two cases. In this experiment, we chose to add 10% ratio between the number of data entries added to resource nodes and context nodes, while other ratios such as 50% and 100% produce the same results. This indicates that the balance of tree in terms of number of data entries under each resource or context node does not influence the performance of the system.

The second evaluation goal is to investigate if redesigning the IoT data access model from Orion’s model to CARIoT’s model impacts the efficiency of Orion. To this end, we evaluated the efficiency by measuring the notification delay when the number of data items added to Orion increases. The measurement was carried out once based on Orion’s data access model and the second one when changing to CARIoT’s IoT data access model. This means that the temperature data
entry in Section 4.6 will change to \{"id":ID, "type":"User", "temperature":\{"value":23.8, "type":"Number"\}\}, and thus, the data entries will be added to entities/city/districts/.... Likewise, the NR changes to listen to data updated for the chosen district of the city.

Figure 13 depicts the behavior of these two approaches in terms of notification delay when the number of IoT data entries increases up to 50000 (i.e., a batch of events). As an IoT data subscriber, we have developed a Node.js HTTP server receiving notifications from Orion. Likewise, a client Node.js application generates the above sample city data entries, e.g., temperature change events. As shown, for the smaller number of data items, both approaches behave similarly. When the number of IoT data entries increases we observe a constant lower notification delay (around 20 ms) for CARIoT thanks to distributing data items between district-specific entries. Note that the main part of the notification delay in this experiment is due the communication with the remote Orion server. This experiment shows that CARIoT does not impose any additional overhead when the service data access model is converted to a SAT over the Orion context framework. Therefore, along with the CARIoT’s other design advantages, its structured way of organizing IoT services will induce no additional overhead to state-of-the-art IoT data processing models.

6.2 Timeliness in Service Access

We evaluate the efficiency of the proposed approach in the context of the Fujisawa application. Since the key performance metric in our work is timeliness, the evaluation is mainly focused on notification delays in the following scenarios: i) the efficiency of hierarchical service notifications in CARIoT as compared to the flat service access model in Orion, and ii) tree-level delay: evaluating the behavior of service notification as contextual hierarchies grow in depth. For both of these scenarios, we use the local Firebase server\(^1\), rather than the cloud-based Firebase\(^2\). The reason for this choice is to assess the above performance metrics on a desktop machine as a worst case setting.

The first evaluation experiment is devoted to investigating the efficiency gained by designing the service access model according to the CARIoT’s hierarchical approach, rather than the existing flat model of Orion. To do this experiment, we consider a scenario in which the context node \(t_i\) of a given SAT will subscribe to events generated by its descendants. The height of \(t_i\) is ten, meaning that the number of edges to a resource node is ten. We intend to measure the notification delay within \(t_i\) when its descendants generate events of interest (e.g., adding or updating a service data items). Then, we convert the given SAT to the equivalent flat Orion model and measure the notification delay similarly. For conversion, each path in the SAT will be described as a string value in Orion which is the concatenation of node names in the path, e.g., \(t_i.t_{i+1}.t_{i+2}....\). In this way, we will be able to measure the efficiency of CARIoT as its data structure is converted to the conventional flat model of Orion.

Figure 14 depicts the imposed notification delay as the number of continuous event notifications (adding a data item to a node or updating a data item) increases from 0 to 2000 in CARIoT and Orion. We have developed a client Node.js application for inserting sample city data items (e.g., temperature of a location) and updating them.

\(^1\)https://www.npmjs.com/package/firebase-server

\(^2\)https://www.firebase.com
In the case of SmartSantander, the reported simultaneous updates are 300 data items per second [11], thereby, we perform the experiment for up to 2000 updates per second as the worst case scenario. The events are equally distributed among all descendants of $L_i$. In the case of CARIoT, we observe that the notification delay is a bit higher for low number of event updates, while it becomes constant for larger numbers. The reason for this behavior is that the underlying data storage, Firebase, performs better under higher loads. Nevertheless, as shown in the graph, the notification delay is quite low and it does not increase as the number of continuous update events to the RAT increases. In the case of Orion, we observed almost equal delay for low and high numbers. However, it induces higher notification delay compared to CARIoT, in addition to the fact that implementing the hierarchical notification abstraction in Orion requires manual creation of tree paths as discussed above.

The other performance metric is related to the depth of a resource in the SAT. The evaluation goal is to find out if hierarchical context levels to access a resource can influence the notification delay. To this end, we extend the five-level tree in Figure 10 to 20 levels and measure the notification delay when listening to a change at the first level up to the 20th level. Figure 15 illustrates the obtained results. As shown, higher context levels lead to slightly longer delay, though the observed delays are negligible. The reason for this overhead is that when data is fetched for a node in Firebase, all of its child nodes will be fetched as well. Again, it should be noted that this result is obtained on the local Firebase server as the worst case scenario. In the cloud version of this storage platform, the trend of the notification delay in Figure 15 will become quite flat, though Firebase allows us to nest data up to 32 levels deep.

7 RELATED WORK

In recent years, several attempts have been made to integrate smart things to web-based platforms, such as proprietary RESTful application servers and later to cloud-based platforms. While early work in this area is mainly focused on technological integration challenges for making such an integration happen efficiently and easily, recent cloud-based solutions aim to facilitate wide-scale adoption and integration of IoT systems, beside cloud’s benefit in terms of performance, scalability, etc. In this section, we study both categories above from the view point of IoT service provisioning, as well as the performance figures considered in the context of this work.

The emergence of lightweight RESTful web services triggered the initial thoughts about developing tiny web servers on sensor nodes and other resource-constrained embedded systems, yielding the notion of Web of Things. In this view, the central idea is to introduce any component of an application that is worth being uniquely identified and linked to as resources [19]. Based on this view, frameworks have been proposed to build applications on top of services offered by smart things within the framework of web of things. A more general viewpoint to such frameworks has resulted in the creation of cloud-based IoT applications. Compared to the approaches based on Web of Things, cloud-based solutions emphasize on the management and composition of IoT services, as
well as their performance at a huge scale. However, in existing frameworks, the way to structure and get access to IoT services in the Cloud is not sufficiently studied. As mentioned before, this is of particular importance with respect to the huge number of IoT devices integrated to a given cloud platform.

IoT domain models which primarily address the issue of IoT device, service and resource modeling, are somewhat relevant in the context of this work. Besides their main goal of modeling, the secondary objective is to tackle the challenge of selecting an appropriate service which satisfies user’s requirements from a huge number of IoT services. Jine et al. [26] propose a physical service model to rate candidate physical services according to a user requirement using the proposed QoS rating functions and then select an appropriate physical service. In [49], a multidimensional resource model is proposed for dynamic resource matching in IoT, motivated by the fact that, with significant number of resources, selecting appropriate resources is time-consuming. Both of these works use only properties of devices and sensors to describe resources, which is different from the broader hierarchical context-based viewpoint we have adopted.

With respect to the middleware solutions for the integration of IoT and the Cloud, recently some efforts have been made to effectively manage various heterogeneous components and services that compose an IoT application in order to achieve seamless integration of the physical world and the virtual one. Boman et al. [7] have developed a generic IoT middleware system that leverages a cloud storage service for sharing and consumption of IoT data and an open source sensor data management system (GSN) for data acquisition from IoT devices. In [30], IoT PaaS is proposed—a cloud platform that supports scalable IoT service delivery, allowing IoT solution providers to efficiently deliver and continuously extend their services. This is achieved through the concept of virtual vertical IoT solutions and domain mediation to leverage computing resources and middleware services on cloud and provide domain-specific control applications, respectively. Thus, the efficiency and scalability they promise is based on the traditional benefits of cloud integration.

Some work has focused on semantic web technology to model and structure the IoT data and sensor data. The reason for using ontological abstraction is to optimize resource utilization in collecting, storing and processing data. Jie et al. [31] use ontologies to capture the hierarchy and equivalency of sensor information. They proposed a hierarchical information structure and a semantic service programming model to perform optimization for resource-aware execution of composite software services. In [3], an IoT virtualization framework is proposed to support sensor event processing and reasoning through a semantic overlay of the underlying IoT cloud. To dynamically sense and respond to IoT events, the authors follow event-driven service-oriented principles. Such modeling approaches are aimed to cover divers and different semantical concepts of IoT systems, with special focus on relating and linking IoT data in IoT domains. The CAT and the SAT may be similar to an ontology and the associated individuals (instances), respectively, however their real-time data processing features and the associated context-driven notification distribution are the unique features of CARIoT, not addressed in semantic modeling approaches. In theory, it might be possible to use ontologies instead of the CAT and the SAT to model the target IoT domain, but they include several data conceptualization and formalization aspects that are not useful for context-based IoT service access. To conclude, we believe that semantic models can serve as a complementary source of information for designing the CAT.

Considering context processing frameworks, a number of approaches have been proposed to address context data distribution and dissemination, and ontology-based and conceptual context processing [9, 10, 39, 45, 48]. In many of works in this category, the main focus is on how to structure context information or how to efficiently disseminate the context information in ubiquitous environments. While our framework is similar to some of them in terms of hierarchical structuring of IoT context information [13, 45], CARIoT is concerned with addressing distributed context query processing and context-aware service provisioning.

Some efforts have been made to use the Linked Data concept [22] to model and organize IoT data, and link related sensor data sets. In [37], an approach is proposed to publish sensor data
as linked data to enable dynamic discovery, integration, and querying of heterogeneous sensor data sources. Recently, the linked data concept has been used for designing Cloud data storages to store IoT data, metadata, and historical sensor readings, using Linked Stream Middleware [28]. OpenIoT is a recent framework, in this category, which provides a middleware platform enabling the semantic unification of diverse IoT applications in the Cloud [42]. LSM in OpenIoT transforms the data from virtual sensors into Linked Data stored in RDF (Resource Description Format), which is queried using SPARQL. Queries refer typically to sensor metadata and historical sensor readings and SPARQL provides the interface to issue these types of queries. Additionally, OpenIoT supports discovering and collecting data from mobile sensors through a proprietary publish/subscribe middleware. In [6], a semantic modeling framework is proposed to annotate streaming sensor data, based on the idea of linked data. The main focus of works in this category is to create efficient and flexible schemes for describing the sensor streams and their attributes, while optimized for storage and query processing purposes.

8 CONCLUSIONS

Efficient context-driven and real-time provisioning of data-centric IoT services is becoming an important design aspect of large-scale IoT cloud applications, such as smart cities. In this paper, we have proposed a software framework, called CARIoT, for efficient design of real-time cloud-based IoT services. The main idea behind the design of CARIoT is structuring the description of data-centric IoT services in a hierarchical model, called the Service Access Tree (SAT), based on different levels of context granularities of the end-user application. Each node of a SAT represents a service delivery, processing and notification point for its own children, accessible by other nodes and third-party applications. CARIoT promises flexibility and generality by providing different levels of context-based abstractions in providing IoT services and their real-time data. We have developed CARIoT on Firebase—a popular cloud-based real-time data storage and application platform. As part of our future work, we aim at investigating further the design details of the proposed framework (e.g., multiple applications with different event subscription criteria) and enabling autonomous creation and maintenance of the SAT using semantical description of IoT resources and services.

REFERENCES


, Vol. 1, No. 1, Article 1. Publication date: January 2018.

Received March 2017; revised August 2017