Searching in a Web-based Infrastructure for Smart Things

Simon Mayer
Institute for Pervasive Computing
ETH Zurich, Switzerland
simon.mayer@inf.ethz.ch

Dominique Guinard*
Institute for Pervasive Computing
ETH Zurich, Switzerland
dom@guinard.org

Vlad Trifa*
Institute for Pervasive Computing
ETH Zurich, Switzerland
trifa@acm.org

***Abstract*—Given the expected high number of accessible digitally augmented devices and their communication requirements, this paper presents our work on creating a Web-based infrastructure for smart things to facilitate the integration, look-up, and interaction with such devices for human users and machines. To exploit the locality of interactions with and between smart things, the proposed infrastructure treats the location of a smart thing as its main property and is therefore structured hierarchically according to logical place identifiers. We discuss the infrastructure's look-up mechanism that leverages Web patterns to foster scalability and load balancing and features an advanced caching mechanism that greatly reduces the response time and number of exchanged messages. These properties are demonstrated in an evaluation in a simulated smart environment.**

I. Introduction

The miniaturization of embedded systems allows computers with wireless communication technologies to be integrated into an ever-increasing number of everyday objects that are thereby transformed into *smart things* [1]. Such isolated smart devices already can provide useful services to human users (e.g., Nike+[1]). However, the real potential of embedding smart things in our everyday environments lies in the interconnection of the services they offer, an idea at the core of the *Internet of Things* (IoT). This development provides new opportunities within communication networks that will not only contain "traditional" data like images and textual content, but also enable the interaction with physical objects (e.g., environmental sensors, mobile phones, etc.) and the possibly real-time data and functionality they offer. Physical items are no longer disconnected from the virtual world but rather become accessible through computers and other networked devices [1]. It is expected that this development will significantly affect our daily lives as we will be able to use ubiquitous computing devices for interacting with the real world from almost anywhere, at any time. For instance, smart electricity meters will enable us to query our environment for its current electricity consumption [2], have the system propose ideas for saving energy, and immediately implement our decisions by configuring smart thermostats [3]. It will probably also become possible to query search engines for the location and state of many physical things [1]. Eventually, we expect that some things will be able to communicate, analyze, decide, and act by themselves and thereby provide an invisible background assistance that could make life more enjoyable, entertaining, and also safer.

While the focus of the IoT is on connecting things on the network-level, the focus of the *Web of Things* (WoT) is the application-level connectivity of smart things: This project aims at making smart things first-class citizens of the World Wide Web and therefore usable like any other hyperlinked data [4]. Thereby, common Web tools (e.g., Web browsers) can be applied to real-world objects and widely deployed and accepted protocols and standards known from the classical Web like Uniform Resource Identifiers (URIs) and hyperlinks can be used to interconnect physical devices. Furthermore, using the Web protocol as opposed to using lower-level Internet protocols when connecting smart real-world devices is that one inherits many of the mechanisms that made the Web scalable and successful like caching, load balancing, indexing, and searching as well as the stateless nature of the HTTP protocol.

As the sheer number of queries and commands that will be produced when integrating a very large number of digitally augmented devices has the potential of slowing or even overloading the Internet, we believe that it is necessary to develop a distributed management infrastructure as architectural backbone for environments of smart devices. Such an infrastructure should support the description, discovery, look-up, and interaction of smart devices and their associated sensing and actuation services for human users and machines. It should allow smart things to communicate and cooperate in extensive settings regarding their spatial distribution as well as their sheer number. A major challenge associated with the task of building such an infrastructure is the expected very large number of networked devices: An Internet of Things is expected to have a larger overall scope than the conventional Internet of computers [1], which renders a centralized solution undesireable if not impossible. We therefore aim for a distributed infrastructure to administer Web-enabled devices that is designed with scalability as prime objective. Regarding smart things properties, we have identified their *location* as their key dynamic feature that is important for both, end-users and the management infrastructure itself. Furthermore, we expect smart things to interact frequently with other things

* Current affiliation: EVRYTHNG Ltd., Zurich, Switzerland.

[1] `apple.com/ipod/nike/`

in their immediate environment (i.e., within a room, floor, or building) and thus exhibit a certain degree of locality. A searching mechanism for smart things should leverage this property for load balancing and to avoid message overhead. In summary, we believe that the following factors are crucial for the success of an infrastructure for smart things:

- *Scalability*: The system must be designed in a way that enables the administration of the expected high numbers of interacting smart things and the communication tasks to be carried out in such environments.
- *Location and Load Balancing*: Smart devices are expected to more often interact with other things in their vicinity and thus exhibit a certain degree of locality. An infrastructure for smart environments should exploit this property and thus avoid global routing whenever possible. Furthermore, the broad availability of information about the location of smart things is a key requirement for enabling context-aware services.
- *Self-management*: The system must be built such that manual administration effort is kept to a minimum. This involves automating the discovery of smart things by the infrastructure as well as minimizing manual configuration and maintenance work on the infrastructure itself.
- *User-friendliness*: The system should expose easily understandable interfaces for both, human users and machines, to search for services provided by smart things.

A fundamental issue in order to make the Web of Things widely usable is to facilitate Web-based interactions for both end-users and WoT-integrated devices. In densely populated smart environments, it will get increasingly difficult for computers as well as human users to find services provided by smart things in a fast, reliable, and user-friendly way. This paper therefore focuses on a look-up mechanism for smart things that lets its users "query the real world" and is incorporated in an infrastructure for smart devices. The task of finding relevant smart things is significantly more complicated than searching for documents, not only because smart things should be identified according to dynamic, contextual information but also due to the lack of a uniform way of describing the things, their properties, and the services they offer: A smart thing does not necessarily express its functionality such that it may be found by traditional search engines that are geared towards finding textual documents.

The rest of this paper is structured as follows: In Section II, we describe several motivational examples for smart device infrastructures. We give an overview of our proposal for a Web-based distributed management infrastructure for smart things in Section III and briefly discuss its hierarchical structure that reflects the logical location of its constituents as well as the discovery system and the embedding of semantic information in smart things' representations. In Section IV, we discuss the look-up service of the proposed infrastructure in greater detail. In particular, we describe the different types of queries it offers and the keyword matching and query routing mechanisms. We also present an evaluation of the infrastructure with respect to searching for smart things (Section V). Finally, we give an overview of related work in Section VI and present a conclusion and prospects for future work in Section VII.

II. Motivation

In this section, we discuss three possible application scenarios where a location-based smart device infrastructure could support human clients and computer programs to find and use services in their surroundings.

a) Universal User Interfaces: An infrastructure that keeps track of smart devices and services that are accessible at a specific place (e.g., home automation systems, smart appliances, or public services like ticket machines) could support human users when using these resources. For instance, a user could carry a mobile device that would contact the infrastructure to find services in the surroundings and then render a user interface to make them intuitively usable.

b) Body Sensor Networks: A smart device infrastructure could offer benefits to body sensor network (BSN) applications, where patients are allowed to freely live their daily lives while being monitored over an extended timeframe (cf. [5]). In such a setting, an infrastructure like the one proposed in this paper could support the interaction of the BSN with sensors deployed in unknown environments. For instance, it could help to feed temperature data from the current location of a monitored individual into the BSN to enable better interpretations of the BSN's own measured data.

c) Robotic Devices and Smart Environments: Finally, an infrastructure like the one proposed here could support robots to interact with smart environments that provide sensing and actuation services. As an example, rescue robots could use readily deployed sensors to acquire information about the (possibly dangerous) surroundings they navigate, for instance by using sensors in a room to ascertain whether it is safe to enter that room.

III. A Web-based Infrastructure for Smart Things

In this section, we give an introduction to the design of our prototype implementation of a Web-based infrastructure for smart devices that aims at consolidating the aforementioned goals of scalability, location-awareness, self-management, and user-friendliness. An initial prototype of this infrastructure that enables the registration of Web resources as well as its basic searching and messaging mechanisms are discussed in [6]. In this paper, we present the extensions that we have made to the searching mechanism, and in particular the adoption of a resource-oriented view on queries (Section IV) that enables an automatic load-balancing mechanism among infrastructure nodes and also allows for advanced caching of queries to save system resources.

Due to the high degree of locality of interactions of human users with smart things and of interactions between smart things, we propose to make the location of a smart thing one of its key dynamic properties and to use logical place identifiers for structuring the infrastructure's nodes. Our proposed system therefore consists of a hierarchy of interconnected

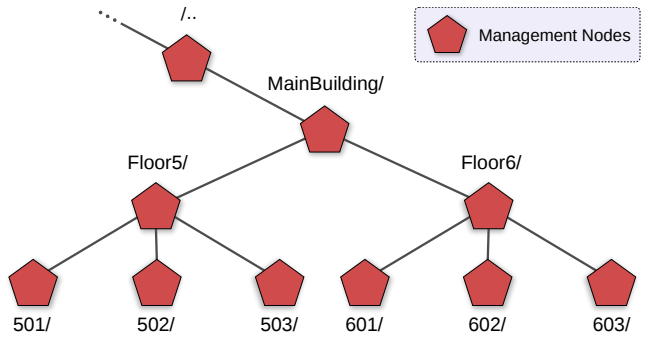

Fig. 1. An example for a distributed smart things management infrastructure whose nodes are hierarchically structured according to logical place identifiers: Nine nodes on two floors with three offices each in the building *MainBuilding*.

management nodes that is structured according to the logical identifiers of the places it covers. To give an example, the node responsible for managing smart things located in room *502* at floor *5* in building *MainBuilding* would have as (transitive) parents the management nodes for *../MainBuilding/Floor5/* and *../MainBuidling/* (Fig. 1). In this setting, only the nodes at the lower levels would have to be "embodied", for instance within wireless routers or network-attached storage devices, in order to establish the physical link to smart things [6].

An important decision when designing the infrastructure was to restrict interactions to direct communication between neighboring management nodes in the tree structure. This guarantees the scalability of the infrastructure as nodes can remain ignorant about the actual hierarchy depth. Furthermore, as smart things will most often interact with other things in their immediate surroundings, such a structure directly mitigates the problem of load balancing as nearby devices are able to discover each other by only querying their local management node. Still, by routing queries between nodes, the infrastructure enables searching on a global scope. Finally, our proposed system is based on widely accepted Web standards and protocols for reasons of scalability, application-level interoperability, and transparency (e.g., human-readable formats).

Regarding the discovery of smart things and their registration with management nodes, we use different lightweight markup languages like Microdata[2] and Microformats[3] for describing smart things' services and interfaces. To support humans and machines in finding and utilizing smart things services, our Web-enabled devices expose their functionality using semantic markup embedded in their representations. This metadata includes static properties of the thing (e.g., its name, category, brand, or a unique identifier) as well as dynamic information like a thing's location or current sensor readings. In particular, the location of a smart thing is used to register it with the management node responsible for its location, irrespective of which node discovers the device. Additionally, things should embed a description of their program interface to facilitate service integration across devices. The chosen markup language should be easy to use in order to allow the fast and simple annotation of services. Importantly, the discovery process of new smart things should be designed in an open and extensible fashion to guarantee compatibility with future description mechanisms. Our infrastructure makes use of a service that is designed with extensibility as primary objective and enables the on-line integration of parsing strategies for semantic markup during operation of the system [7].

Additionally to providing smart things discovery on a semantic level, another task of an infrastructure for smart devices is to enable the *network-level* discovery of physically connected items: After establishing the network presence of a device, the infrastructure should take care of analyzing and semantically identifying the smart thing (cf. [6]).

Regarding self-management, the proposed infrastructure features a self-stabilization mechanism: Based on the (preconfigured) logical locations of management nodes, the infrastructure is able to recover from temporary node failures, where, eventually, the original structural configuration is reestablished (cf. [6]). The entire topology of the infrastructure is induced by the logical locations of the infrastructure nodes. While we do not anticipate frequent changes in this topology, the self-stabilization mechanism would react to such changes by rearranging the nodes to reestablishing a correct topology.

IV. Smart Things Look-Up

In this section, we present the proposed infrastructure's look-up service and in particular the available types of queries, the keyword matching mechanism to identify corresponding resources, and the routing of queries within the infrastructure. Our querying mechanism makes extensive use of the location-based hierarchical structuring of the management nodes when processing queries: Clients may submit queries that, together with the information used to identify corresponding resources, also include spatial information to specify query scopes. Additionally, we have defined several types of queries that allow leveraging scoping information to increase the system's querying performance, especially regarding very deep and/or broad hierarchical structures of nodes in environments with large numbers of registered smart things.

A. Query Types

We propose several query types to let clients choose the scope of their queries, i.e. which nodes of the infrastructure to include when searching for resources. We defined four query types to be used for searching (cf. Fig. 2 for an overview). For a discussion of how the different types are routed internally, refer to Section IV-C.

1) Exhaustive Queries EXQ: The scope of an exhaustive query is the entire subtree of the queried node. The answer will contain all resources that correspond to the query parameters found locally at the queried node or at (transitive) child nodes.

2) Cardinality Queries (CAQ_k): When a client wants to find exactly k resources that match its query, the addressed node triggers a cardinality query with parameter k.

[2] w3.org/TR/microdata/
[3] microformats.org

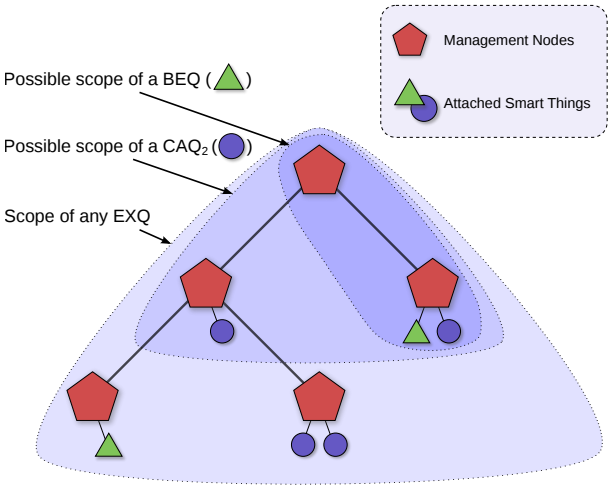

Fig. 2. Query types: EXQ (EXhaustive Query) - search the entire subtree of a node. CAQ_k (CArdinality Query) - search for k corresponding resources within the subtree of a node. BEQ (Best-Effort Query, $BEQ \equiv CAQ_1$) - search for a single corresponding resource within the subtree of a node.

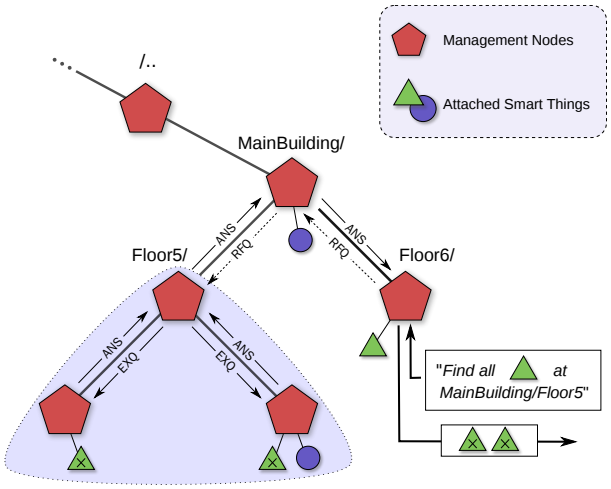

Fig. 3. To trigger an EXQ at location `../MainBuilding/Floor5`, a client contacts its local management node at `../MainBuilding/Floor6`. Internally, the query is routed as an RFQ.

3) Best Effort Queries (BEQ): A special case of the cardinality query is the BEQ which corresponds to a CAQ_1, i.e. to a query that only retrieves the first matching resource that is encountered.

4) Request For Query (RFQ): $RFQs$ are a special type of query that enables clients to search at locations other than that of the initially queried infrastructure node, for instance to involve nodes on higher levels of the infrastructure or to restrict a query to a specific subtree of the infrastructure (cf. Fig. 3). An infrastructure node creates an RFQ internally whenever it receives a request to search at a different location than its own by wrapping the query and relaying it to an attached node that is "closer" to the location given in the query. For instance, a node at *../MainBuilding/Floor6/* that receives an exhaustive query for location *../MainBuilding/Floor5/* will locally decide on creating an RFQ and routing it to its parent node at *../MainBuilding/* which will in turn relay the query to *../MainBuilding/Floor5/*. This node will then trigger the original request locally and return the answer(s) (ATQ). This process happens transparently such that, for the client, it looks like its request is processed locally at the queried node. $RFQs$, as well as the other types of queries, can be processed by only relying on information about the local node and its immediate neighbors and does not require any global information to be distributed among the management nodes.

B. Query Parameters

The proposed system offers multiple querying interfaces, where clients can find resources by providing keywords, resources' unique identifiers, or other pieces of information about the required resource like their offered REST interface (cf. [6]). To use such information for searching implies that the resources expose the information. For instance, to search by REST interface, a resource could embed a machine-readable description of this interface in the `hRESTS` Microformat (cf. [8]). To extract this information from resources and store it locally in a structured way, we use an extensible semantic resource discovery service described in [7].

Here, we describe the system's keyword-based interface that represents an interface mainly for human clients, but can also be used by machines. The other interfaces can be used in a similar way by specifying, for instance, a unique identifier instead of the keywords. Every query consists of at most three parts: The *keyword(s)* to identify corresponding resources, a *location specifier* to route queries to appropriate nodes and a *cardinality specifier* to set the number of desired answers. New queries are created by clients using HTTP `GET` requests that include a single URL-encoded string as query description. An example query for three resources at location *../MainBuilding/Floor5* that match the keywords *Temperature* and *Sensor* is given here:

```
http://host.net/?q=3x+Temperature+
Sensor+at+MainBuilding%2FFloor5
```

Upon receiving a query, it is parsed to extract the required parameters (in the above case using the regular expression `[0-9]*x` for the cardinality specifier and the token `at` for identifying the location specifier) and an appropriate JSON-serializable query object is created to be routed within the infrastructure.

We preferred to use keywords as using these has, mainly due to the popularity of Web search engines, for many users become the most intuitive way of searching for information. To find resources that match a given set of keywords locally at a node, we use an approximate string matching algorithm based on a measure called *Dice's coefficient* that considers the number of equivalent character bigrams as a fraction of the total contained character bigrams when comparing the keyword(s) to resource information like the resource name, its unique identifier, description, category, or brand. We have extended this algorithm with different weighing factors for

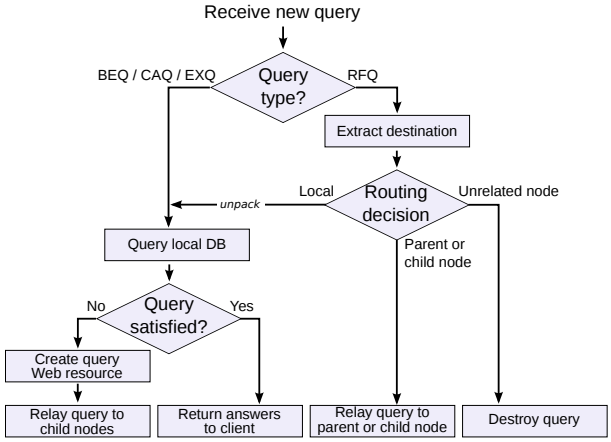

Fig. 4. Overview of the processing of received queries.

matching different kinds of information – for instance, a correspondence with the resource's unique ID is considered a more valuable match than the comparison to the resource's description. By using this metric instead of comparable string matching algorithms like the edit distance or the longest common substring, we found that the searching becomes more robust regarding, for instance, typographical mistakes or keywords in different languages and in general produces very satisfying results for resource lookup by human users. For machines, we recommend the use of structured queries to better specify the desired resource.

C. Query Processing

As mentioned earlier, received queries are parsed to query objects. The internal representation of a query contains, as mandatory parameters, the unique query ID, the URL of the node that initiated the query as well as its type, where the specification of a positive integer is mandatory for $CAQs$. Additionally, queries may contain optional parameters to, for instance, piggy-back structural or management information. After parsing, queries are routed within the infrastructure to obtain answers in the form of registered resources that correspond to the queries' parameters. In this section, this process is described in detail where we first consider the handling of RFQs and then describe the routing mechanism for the other types of queries (see Fig. 4 for an overview). Throughout this discussion, note that queries are only routed locally and that no global knowledge about the infrastructure is implied anywhere in the query handling mechanisms. Rather, all nodes have to merely be aware of their immediate child nodes and their parent node.

1) Routing of $RFQs$: A node that receives an RFQ first analyzes its location parameter in order to determine how to route the request. From the data in the location field, the node is capable of deciding whether the query's destination is the processing node itself in which case the query is unpacked, transformed to the matching query type, and triggered locally. Else, if the management node determines that the query's destination node is contained within its own subtree, it selects that node among its children which is most appropriate to handle the query (i.e., the node whose location better matches the query destination than the processing node). Otherwise, i.e. if the processing node determines that it is a sub-node of the queried one or if the destination node is in a different subtree altogether, the query is relayed to the node's parent. If the query's destination node does not match the processing node's location at all, the query is destroyed.

2) Routing of $BEQs$, $CAQs$, and $EXQs$: When an infrastructure node receives a CAQ, EXQ, or BEQ or transforms an RFQ to one of these locally and cannot satisfy the query itself by delivering enough corresponding smart things from its local database, it starts to collect answers from its (transitive) child-nodes. As soon as enough answers have been collected, they are merged and the composite answer is relayed to the inquirer (i.e., a client or the infrastructure node that relayed the RFQ). To propagate a query across nodes in the infrastructure, the local node immediately creates a local Web resource at the URL `localhost/queries/{queryID}` as a representation of the query and then relays the query to its children which, after doing a local look-up, forward it to their children. The proposed querying mechanism thus features a *resource-oriented view* on querying: Queries are embodied as resources that can be extended by submitting answers and can be queried for the number of answers still due. This helps to reduce answer times to queries, the number of messages induced by single queries, and with load balancing. We describe this mechanism in greater detail in the following by discussing the three main purposes of query resources:

First, any infrastructure node in the query scope aware of smart things that correspond to the query can `POST` that information to this URL as (partial) answer. The local node registers to the resource and thus receives these answers. As soon as the query is satisfied (i.e., if, for a CAQ, enough answers have been received), it destroys the query resource and relays the answers to the inquiring client. For EXQs, where the triggering node does not know in advance when all nodes in its subtree have answered, the query resource features an explicit notification mechanism where information regarding pending answers from child nodes is stored: Whenever a node forwards a query to one of its children, it registers this child as *pending* with the query. This flag is removed again when the child answers such that, as soon as a query has no more pending answers, it may be finalized by its issuing node. The same mechanism is necessary to avoid stalling when the cardinality of a CAQ is higher than the total number of corresponding resources in the system. The explicit query resource thus leads to a *decoupling of infrastructural nodes* with respect to queries: Child nodes merely have to accept query notifications rather than directly obtaining answers from their subtrees, which would induce a wave-algorithm-like propagation of the queries within the infrastructure. Furthermore, the proposed mechanism leads to the infrastructure reporting resources from nodes in the order of the nodes' response times. This enables an automatic load balancing mechanism as overloaded nodes' resources will be reported less frequently when processing queries and thus used less. If a whole subtree of the system

is under heavy load, this whole hierarchy will be excluded from subsequent queries. Note that this does not influence the quality of responses as, in such a scenario, a client has explicitly asked for, e.g., "Three temperature sensors on the 5th floor of building *MainBuilding*" and thus is not interested in where exactly (i.e., from which office on the 5th floor) the delivered sensors are situated. Rather, it enhances the search mechanism as answers from nodes at higher hierarchical levels with a more general scope will automatically be preferred: If, for the aforementioned query, the node responsible for the 5th floor of building *MainBuilding* has registered temperature sensors, these will be delivered to the client rather than considering answers from room-specific nodes, as the query can be served from the local database. Finally, note that, for $CAQs$ of fixed cardinality, the response time will decrease on average when increasing the number of resources per node until a single node is able to fully serve the query.

Second, if the local node receives another query that is equivalent to or overlapping with the first (e.g., if a BEQ with identical keyword(s) is received subsequently to an EXQ), the node does not forward the query but rather registers to the first query's resource. This system thus represents an augmented *query caching mechanism* as overlapping queries may have different scopes and/or types: If a node detects that a query that is already being routed within the infrastructure is overlapping with a newly received query, there is no need to forward the new query. This mechanism can greatly reduce query answer times and the number of messages in the system (cf. Section V), especially when the queried node is under heavy load from concurrent clients.

Third, query resources can themselves be contacted to detect the query's current state. This is helpful whenever a node that receives a query can deliver local corresponding resources as answers: When `POST`ing this information to the query resource, the resource's response contains the number of answers that are still required to fulfill the query. The local node can use this information to determine if the query should be further relayed to its child nodes which again saves resources.

V. Simulation Environment and Evaluation

We evaluated the discussed querying mechanism using a deployment of the infrastructure with nine nodes (one on building-level, two on floor-level, six on office-level, cf. Fig. 1 in Section III) that handled a total of 600 simulated sensors of different types (e.g., temperature, electricity consumption, ambient light) which were registered uniformly at random with the six office-level nodes. We used the *apachebench*[4] tool with different queries (a BEQ, a CAQ_{30}, a CAQ_{200}, and an EXQ). All measurements were performed *locally*, on a single machine running all nodes in parallel, to avoid distortion of the results.

[4]`httpd.apache.org/docs/2.0/programs/ab.html`

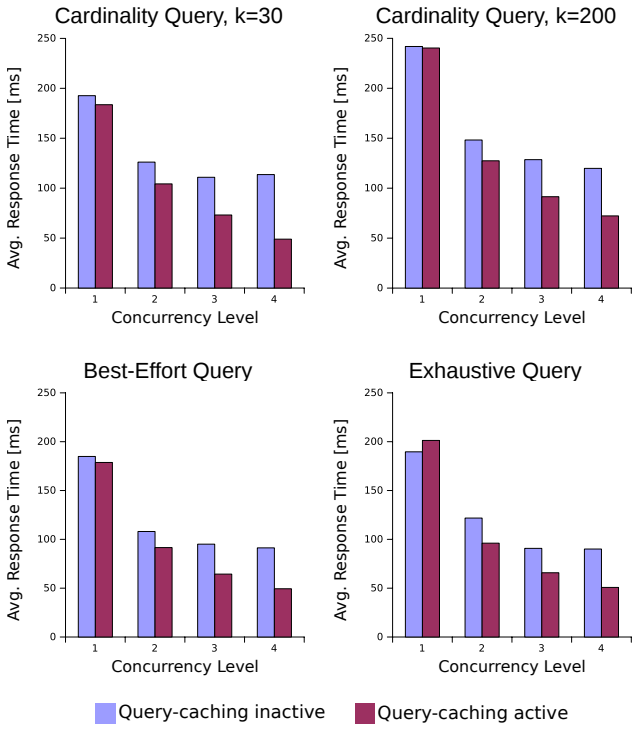

Fig. 5. Simulation results for comparison of active vs. inactive query-caching for different types of queries and concurrency levels.

A. Evaluation Setup

To simulate the smart devices that are registered to the infrastructure, we used a custom-developed simulation environment for Web-enabled devices. In this environment, Web resources can be created that simulate arbitrary sensors (supporting HTTP `GET` requests) and actuators (supporting HTTP `GET` and `PUT` requests). The simulated resources support representations in the form of `text/html`, `application/json`, `application/xml`, and `text/plain`, where representation templates can be configured. Every resource can simulate desired response behavior regarding its reliability and response time by setting parameters for a statistical model. Sensors can furthermore be configured to deliver values according to a statistical model and can also be set to fetch their result from a real device. The simulation environment has been created to deploy large numbers of Web resources and we have tested it for up to 100000 resources without performance issues. Resource templates are available to facilitate the deployment of common sensors, however a user may configure the environment to simulate arbitrary, custom resources.

B. Results – Query Caching

To evaluate the query-caching mechanism, we performed, for each of the aforementioned query types, 1000 requests with varying concurrency levels. The expectation for this test is that a "baseline"-system with query-caching turned off is increasingly outperformed by an otherwise equal system with query-caching turned on with increasing concurrency level. Our evaluation supports this observation, where the response time of the baseline-system is up to twice as high as that of

the caching system for the CAQ_{30}-case with 4 concurrent requests (Fig. 5). The caching system is faster in almost all cases and its advantage strongly increases with the chosen concurrency level. Furthermore, we can see that the BEQ performs only slightly faster than the CAQ_{30}, which is due to the fact that every office-level node knows about more than 30 resources and a CAQ with that cardinality therefore does not induce any overhead when compared to a BEQ (which is essentially a CAQ_1). Finally, we see that the EXQ performs remarkably fast when compared to the other types of queries. This is due to the different internal handling of $EXQs$ – these queries are forwarded by each node as fast as possible as it is well-known that they need to be propagated to all child nodes in any case. For $CAQs$, on the other hand, the local node also considers preempting the query before forwarding it to its child nodes. This advantage of $EXQs$ regarding their handling, however, vanishes when triggered in deeper hierarchies and/or when additional resources are registered with intermediate nodes (i.e., in this scenario, floor-level nodes). Altogether, the obtained simulation results support our expectations and demonstrate that the described query-caching is a valuable mechanism for the scalable routing of queries in such infrastructures.

C. Results – Load balancing

We furthermore tested the infrastructure's load balancing abilities mentioned in Section IV-C: For a BEQ with the keyword *temperature* at location *MainBuilding*, the infrastructure usually (i.e., in more than 90% of the cases) delivered a sensor in one of the rooms on floor 5. However, when we started to flood the node at *../MainBuilding/Floor5* with queries, the same BEQ returned almost exclusively (more than 96% of the cases) sensors in the rooms located on floor 6 due to the increased response time of the nodes in the subtree rooted at *../MainBuilding/Floor5*. Note that, as the BEQ's scope is *MainBuilding*, answers from both locations (i.e. rooms on floor 5 and rooms on floor 6) are equally valid.

VI. Related Work

A large part of the World Wide Web's success stems from its scalable architecture, generic interfaces, and loosely coupled components. In its idealized form, we refer to this architecture as *Representational State Transfer* (REST) [9]. In REST, the primary abstraction of objects that provide information and functionality are *resources* that are identified in a uniform way using Uniform Resource Identifiers (URIs) (cf. resource-oriented architecture). These objects can be queried and manipulated using a limited and fixed set of verbs (in the Hypertext Transfer Protocol HTTP, these are `GET`, `POST`, `PUT`, `DELETE`, etc.) that have generally understood semantics (e.g., for HTTP, `GET` is considered *safe*, i.e., free of side-effects). Messages that are exchanged between client and server are self-descriptive and their structure is considered *common knowledge*, which is supported by a content negotiation mechanism used to determine the concrete resource *representation* that is transmitted between communication endpoints. Although HTTP was designed as an application protocol with particular focus on scalability, many Web applications use it only as a transport protocol and therefore only utilize a fraction of its functionality: For instance, Web applications that rely upon SOAP or WSDL use only the HTTP `POST` operation to perform application programming interface (API) calls on URI-identified endpoints and do not expose the manipulated resources themselves. Practices like these prevent to take full advantage of the Web architecture because they neglect the semantics of the interaction verbs: In the mentioned example, a `GET` could be used to signal that the interaction is *safe* and therefore cacheable. The WoT seeks to fully leverage the Web's architectural principles and widely accepted protocols, standards, and mechanisms (e.g., caching) to foster device interoperability, facilitated user interaction (via browsers), scalability, and openness. We believe that the Web can act as a common ground for enabling smart things to interact with each other, with traditional Web services, and with humans. Furthermore, it could be utilized as a lightweight approach that would enable the creation of physical mashups (i.e., of applications emerging from the crossover of multiple services) on top of smart things.

A promising solution to create WoT discovery and look-up services that users can query to find devices, data, and services, are semantic technologies. A prominent approach to put forward common formats for the integration and combining of data and its relationship to real world objects is the Semantic Web [10], the most advanced concretization of which is based upon RDF. Approaches for enriching HTML documents with semantic metadata most notably include Microformats that aim at re-using existing HTML/XHTML tags for attaching semantics to data on the Web, and Microdata that uses new tag attributes to achieve the like. The implementation of these semantic annotation formats is currently gaining momentum as several companies, including Google, Bing, and Yahoo! have started to define vocabularies (e.g., `data-vocabulary.org`) and use them in their products. In our concrete use case of supporting the look-up of smart things, we imagine these technologies to be utilized by devices to expose their static and dynamic properties in a machine- and human-readable form. This would allow for centralized or distributed smart things registries like the one described in this paper to easily detect and classify devices and make them discoverable for potential clients.

Middleware solutions that leverage description languages for facilitating the management and interconnection of smart things have been proposed, for instance in [11], where a central argument for using semantic technologies is that these would not only facilitate the discovery but also the behavioral control and coordination of heterogeneous components. Similarly, in [12], the authors propose a discovery, querying, and selection framework for WS-* and RESTful web services where resources expose their APIs using machine-readable formats. This framework supports Microformat-based markup and Web Services Description Language (WSDL) documents and federates them in a meta description format that is accessible to

external clients through a (WS-*) resource discovery service. Their system, however, represents a centralized solution that does not exploit properties of smart things interaction like locality and does not allow for the location-based look-up of devices. A survey of sensor-actuator networks along with a resource repository implementation is presented in [13]. This repository lets Web-enabled things be discovered using a tag-based approach while publishing an OpenSearch[5] document to describe its resource retrieval capabilities. It thus features a support infrastructure for smart things like the one discussed in this paper where resources are registered and exposed to clients, but, like [12], makes use of a centralized resource repository. *Dyser* [14] is a search engine for the real-time Internet of Things which uses statistical models to make predictions about the state of its registered resources when a user submits a query. These predictions are used to establish a ranking that determines the order in which resources are contacted to find out whether their current state matches the query. Our system does not contact the registered resources for every query but rather acts as a broker between clients that wish to interact with smart things and the devices, where this interaction is currently fully left to the client.

VII. Conclusions

In this paper, we described how a distributed management infrastructure could support human users and machines in finding smart devices and described our prototype implementation of such a system. We argue that, for reasons of scalability and interoperability, it would be beneficial to base the infrastructure on widely accepted Web standards and protocols. Due to the locality of smart things interactions, we proposed to make the location of a smart thing its key dynamic property and to use logical place identifiers for hierarchically structuring the infrastructure's nodes. On top of this architecture, we described a scalable look-up mechanism that treats queries as resources and features a caching system which significantly reduces response time and the number of messages required when searching for devices. The proposed mechanism also has benefits regarding load balancing, as queries are automatically routed to not further strain overloaded parts of the infrastructure. We demonstrated these properties in an evaluation that involves multiple infrastructure nodes and several hundreds of simulated Web-enabled sensors distributed among these nodes.

One main concern with the proposed location-based overlay is to establish standard naming conventions to avoid problems in the hierarchical routing algorithms. Furthermore, the system is dependent on a mechanism that enables the correct assignment of logical place identifiers to smart things – in our current implementation, this information is statically assigned to the devices and embedded in their Web representations as semantic markup such that the infrastructure's discovery service is able to interpret it. Regarding future work, we plan to implement an indoor localization system also on the smart devices and send location updates to the infrastructure to enable the roaming of devices between infrastructure nodes. We furthermore plan to extend device descriptions with semantic information about the services and data they offer and use this information to enhance the infrastructure's querying mechanism. Finally, we plan to integrate the proposed infrastructure with the Dyser real-time search engine [14] and incorporate its ranking system into our querying mechanisms.

[5] opensearch.org

Acknowledgments

This work was supported by the Swiss National Science Foundation under grant number 134631.

References

[1] F. Mattern and C. Floerkemeier, "From the Internet of Computers to the Internet of Things," in *From Active Data Management to Event-Based Systems and More*, ser. LNCS, K. Sachs, I. Petrov, and P. Guerrero, Eds. Springer, 2010, vol. 6462, pp. 242–259.

[2] D. Guinard, M. Weiss, and V. Trifa, "Are you Energy-Efficient? Sense it on the Web!" in *Adjunct Proceedings of the 7th International Conference on Pervasive Computing*, Nara, Japan, May 2009.

[3] W. Kleiminger, C. Beckel, and S. Santini, "Opportunistic Sensing for Efficient Energy Usage in Private Households," in *Proceedings of the Smart Energy Strategies Conference 2011*, Zurich, Switzerland, Sep. 2011.

[4] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, "From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices," in *Architecting the Internet of Things*, D. Uckelmann, M. Harrison, and F. Michahelles, Eds. Springer, 2011.

[5] C. Seeger, A. Buchmann, and K. Van Laerhoven, "An Event-based BSN Middleware that supports Seamless Switching between Sensor Configurations," in *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*. New York, NY, USA: ACM, 2012, pp. 503–512.

[6] V. Trifa, D. Guinard, and S. Mayer, "Leveraging the Web for a Distributed Location-aware Infrastructure for the Real World," in *REST: From Research to Practice*, E. Wilde and C. Pautasso, Eds. Springer, 2011.

[7] S. Mayer and D. Guinard, "An Extensible Discovery Service for Smart Things," in *Proceedings of the 2nd International Workshop on the Web of Things*, San Francisco, USA, Jun. 2011.

[8] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML Microformat for Describing RESTful Web Services," *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 1, pp. 619–625, 2008.

[9] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[10] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 34–43, May 2001. [Online]. Available: http://www.scientificamerican.com/article.cfm?id=the-semantic-web

[11] A. Katasonov, O. Kaykova, O. Khriyenko, S. Nikitin, and V. Y. Terziyan, "Smart Semantic Middleware for the Internet of Things," in *Proceedings of the 5th International Conference on Informatics in Control, Automation and Robotics*, 2008, pp. 169–178.

[12] 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," *IEEE Transactions on Services Computing*, vol. 3, no. 3, pp. 223–235, Jul. 2010.

[13] V. Stirbu, "Towards a RESTful Plug and Play Experience in the Web of Things," in *IEEE International Conference on Semantic Computing*, Los Alamitos, CA, USA, Aug. 2008, pp. 512–517.

[14] B. Ostermaier, K. Römer, F. Mattern, M. Fahrmair, and W. Kellerer, "A Real-Time Search Engine for the Web of Things," in *Proceedings of the 2nd IEEE International Conference on the Internet of Things*, Tokyo, Japan, Nov. 2010.