A Real-Time Search Engine for the Web of Things

Benedikt Ostermaier*, Kay Römer†*, Friedemann Mattern*, Michael Fahrmair‡ and Wolfgang Kellerer‡

*Institute for Pervasive Computing, ETH Zurich, Switzerland
†Institute of Computer Engineering, University of Lübeck, Germany
‡DOCOMO Euro-Labs, Munich, Germany

***Abstract*—The increasing penetration of the real world with embedded and globally networked sensors leads to the formation of the *Internet of Things*, offering global online access to the current state of the real world. We argue that on top of this real-time data, a *Web of Things* is needed, a software infrastructure that allows the construction of applications involving sensor-equipped real-world entities living in the Internet of Things. A key service for such an infrastructure is a search engine that supports lookup of real-world entities that exhibit a certain current state as perceived by sensors. In contrast to existing Web search engines, such a real-world search engine has to support searching for rapidly changing state information generated by sensors. In this paper, we show how the existing Web infrastructure can be leveraged to support publishing of sensor and entity data. Based on this we present a real-time search engine for the Web of Things.**

I. Introduction

The physical world is being increasingly penetrated by embedded sensors that are connected to the Internet and the World Wide Web, making it possible to observe an ever-increasing proportion of the world with minimal delay using a standard Web browser. For example, sensor networks are deployed to monitor the environment, mobile phones are equipped with an increasing number of sensors that can be used as mobile sensor nodes [1], and indoor plants are enabled to publish their current state using Twitter [2]. Services like Microsoft SenseWeb and pachube.com strive to simplify this process by providing platforms for publishing structured sensor data on the Web. Additionally, real-time data regarding the state of the real world is also published by various services: Examples include traffic status integrated with Google Maps, public transport operators who provide RSS-feeds of incidents and delays, or Bicing [3], a public bicycle-sharing system in Barcelona, Spain, that provides the number of bicycles available at each rental station in real time on the Web. Extrapolating this trend into the future, we should soon be able to infer the state of many real-world *entities* (people, places and things) in real time by analyzing the output of associated sensors and publishing this information on the Web.

We believe that these trends are precursors of a Web of Things [4] which will extend the original document-centric Web, making it a universal interface for the real world by giving real-world entities a Web presence that can be accessed using lightweight APIs (typically based on the REST principle [5]). These representations include the current state of those entities (*empty* room, *sleeping* person, *quiet* restaurant) as perceived by embedded sensors. However, in contrast to visions of a *Sensor Web*, which offers access to raw output of sensors, users of the Web of Things will most likely not be interested in searching for *sensors* with a specific raw *reading*, but for *entities* of the real world with a specific current *state*. For example, instead of searching for loudness sensors with a current reading below 30 dB, we expect that users will rather be interested in searching for *places* which are *quiet*.

As for today's Web, a key service for the Web of Things will be a *search engine* that allows to search for real-world entities with certain properties. While the traditional Web is dominated by static or slowly changing, unstructured content being manually typed in by humans, a key feature of the Web of Things is rapidly changing, structured content being automatically produced by sensors. Thus, a search engine for the Web of Things has to support searching for structured and rapidly changing content, which is a key challenge given that existing Web search engines are based on the assumption that most Web content changes slowly, such that it is sufficient to update an index at a frequency of days or weeks. This is clearly insufficient for the Web of Things where the state of many real-world entities changes within minutes or even seconds.

The contribution of this paper is a real-time search engine for the Web of Things, addressing the key challenge of scalable search for rapidly changing content while leveraging existing Web infrastructure. Essentially, our search engine called *Dyser* supports the search for real-world entities with a user-specified current state. For example, Dyser could be used to search for rooms in a large building which are currently occupied, for bicycle rental stations which have currently bikes available, for currently quiet places at the waterfront, or for current traffic jams in a city. Essentially, we turn the algorithmic foundations of our previous work [6] into a running system – presenting its design, implementation, and evaluation.

II. Problem Statement and Approach

The key challenge that needs to be addressed in constructing a search engine for the Web of Things is the anticipated huge size and extreme dynamics of the search space. Extrapolating the current trend of instrumenting objects, places, and even people with sensors several years into the future, we can expect that the Web of Things may contain orders of magnitude more sensors than currently existing Web pages. Moreover, the output of these sensors is highly dynamic. In contrast, the large majority of today's Web is static in the sense that Web pages are changed at time intervals orders of magnitude longer than the update rate of sensors. Thus, traditional indexing approaches are insufficient as an index would be outdated as soon as it has been constructed.

There are two fundamental approaches to construct a search engine for the Web of Things. With a *push* approach, sensor output is proactively pushed to a search engine, such that the search engine can resolve queries based on that data. With a *pull* approach, only upon a user entering a query the search engine sends the query to the sensors to pull the relevant data.

In the Web of Things we can expect substantially more sensors than users typing queries, and sensors would produce data at a much higher rate than users can type queries. Hence, the pull approach can be expected to generate a substantially smaller communication volume between sensors and search engine than the push approach. Still, pulling all sensors upon each query would not scale. Therefore, in previous work we have proposed an approach called *sensor ranking* [6], which for a given query computes the probability that a sensor produces the sought output at the time of the query by using indexed *prediction models*. Sensors are then pulled in decreasing order of probability until enough matches have been found, thus spending effort first where it counts. Here, we also exploit the fact that users of search engines are typically *not* interested in all results, since these are typically far too many to check manually.

The main assumption in our approach is that there are enough sensors which offer a sufficient level of predictability. While this may not be the case for arbitrary sensors, many phenomena in the real world feature periodic characteristics, especially those related to people. For example, in a person's life, daily, weekly and yearly cycles can usually be identified. Recent research (e.g., [7], [8]) has shown promising results regarding the predictability of human behavior. For this reason, we focus on sensors related to human behaviors.

In the following subsections we formally define our system model and detail the basic operation of the search engine, summarizing the main results from our previous work [6] to make the paper self-contained.

A. System Model

Formally, a sensor s is a function

$$s : T \mapsto V \qquad (1)$$

where T denotes real time and V the set of possible sensor values. V is assumed to be a finite set of discrete states that an entity can be in (e.g., a room entity could include an occupancy sensor that can yield one of two values "occupied" or "empty"). Note that the mapping of raw sensor readings to discrete states may introduce subjectivity, as it often requires interpretation of the sensor data. We let the operator of the sensor implement this mapping, as he has comprehensive information about its context. The modality of the mapping is beyond the scope of this paper.

Each sensor is associated with a type and optionally further structured meta information such as a location. For example, the following function can be used to obtain the type $\in Y$ of a sensor:

$$\text{type} : S \mapsto Y \qquad (2)$$

A prediction model for a sensor s is a function

$$m_{s,t^0,t^1} : T \times V \mapsto [0,1] \qquad (3)$$

The parameter $t^0 \in T$ refers to the time of the first considered sensor reading, $t^1 \in T$ refers to the time when the model has been constructed, meaning that all sensor values $s(t^0 \leq t_i \leq t^1)$ have been available for the construction of the model. The idea behind constraining the construction of the prediction model to the time window $[t^0, t^1]$ is that sensor values from the distant past are typically bad indicators for the future output of the sensor. Also, using a time window instead of all past data typically reduces the resource consumption (i.e., execution time and memory footprint) of the model construction.

Given a point in time $t > t^1$ and a sensor value $v \in V$, $m_{s,t^0,t^1}(t, v)$ is an estimate of the probability that $s(t) = v$ holds. We call $t - t^1$ the *forecasting horizon*.

An entity $e \in E$ is associated with one or more sensors,

$$\text{sensors} : E \mapsto \mathcal{P}(S) \qquad (4)$$

whereas we assume that each sensor of an entity e has a distinct type.

B. Basic Operation

Given the above defintions, we can now outline the basic operation of the search engine. At regular intervals, the search engine crawls the Web of Things. For each visited entity e_i, the search engine downloads and indexes the structured meta information including the prediction models of all $\text{sensors}(e_i)$ associated with that entity.

A basic search operation issued at time t would specify the current state of sought entities at time t, where the current state of an entity is represented as a set of sensor types $y \subset \mathcal{P}(Y)$ and a mapping function $\text{val} : Y \mapsto V$ that maps sensor types in y to the requested values. That is, an entity matches the search if for each type $y_i \in y$ the entity contains a sensor s of that type with $s(t) = \text{val}(y_i)$.

To perform a search operation, the search engine first fetches entities e_i from the index which contain sensors of requested types. Next, for each fetched entity e_i a probability p_i is computed that the entity matches the search. As the search is conjunctive (e.g., the entity must contain a matching sensor for each type and value specified in the search), p_i equals the product of the prediction probabilities of the individual sensors (assuming independent sensors):

$$p_i := \prod_{s \in \text{sensors}(e_i) \wedge \text{type}(s) \in y} m_s(t, \text{val}(\text{type}(s))) \qquad (5)$$

Next, entities e_i are sorted by decreasing values p_i. Beginning with the entities with the largest p_i, the sensors of the entities are consulted to check whether indeed $s(t) = \text{val}(\text{type}(s))$ holds. Entities where all requested sensor states match are returned as search results until enough matching entities have been found.

Note the difference between sensor ranking and the relevance ranking of traditional search engines. The former tries to optimize the performance of the search engine, while the latter tries to optimize user satisfaction. As detailed in Sects. III and IV, we combine both in Dyser: sensor ranking for the internal search process, and relevance ranking when displaying the results to the user.

C. Prediction Models

We now sketch three examples for concrete prediction models for use in Dyser, more details can be found in [6].

Our simplest prediction model computes the fraction of the time during which the sensor output equals the sought value v within the time window $[t^0, t^1]$. For example, if the sensor output was v during the whole time window $[t^0, t^1]$, then the probability computed by the above prediction for value v equals 1. Note that the output of the prediction model is independent of the actual point in time t of the search. We call this model the *aggregated prediction model* (APM).

A more elaborate model would take into account the time t of the search. For this, we exploit the assumption that there is a dominant period length L after which the sensor output is likely to repeat, where L is either known in advance or derived by a spectral analysis of past data. For example, it is reasonable to assume that the occupancy pattern of a room is likely to repeat every week, that is, L equals one week. Here, the computed probability that a given room is empty on Monday noon equals the fraction of Monday noons contained in the time window $[t^0, t^1]$ at which the room was empty over all Monday noons contained in that time window. As this model assumes a periodic process with a single period, we call this model the *single period prediction model* (SPPM).

The previous model ignores the fact that sensor output is often the result of many periodic processes with different period lengths. For example, a meeting room may host a group meeting every Monday and a general assembly every first Tuesday of each month. Here, we have two periodic processes with period lengths of one week and one month, respectively. To support such multi-period processes, we use the following approach. In a first step, we discover so-called periodic patterns in the time window using a variant of an existing algorithm [9]. As a result, we obtain a list of periodic patterns of the form (l, o, w, p), where l is the period length of the pattern, o is an offset in the period such that the sensor output $s(kl+o)$ equals w for integer values k with probability p. For example, the pattern (one week, 1, occupied, 0.5) means that every second day in a week (i.e., Tuesday) a room is occupied with probability 0.5. To make a prediction, we first filter all patterns that match the search time t, that is we keep a pattern (l, o, w, p) if and only if there exists some integer k such that $kl + o = t$ and $w = v$, where v is the sought sensor value. Among all remaining patterns we compute the maximum probability and output it as the prediction. We call this model the *multi-period prediction model* (MPPM).

III. Design

The basic algorithmic approach sketched above now needs to be mapped to a concrete system design and implementation. In particular, we have to design basic abstractions for the Web of Things to represent real-world entities and their states. On top of these, we need to define the system architecture for the search engine. One key goal herein is to retain the *open* and *loosely coupled* architecture of the current Web such that everybody can introduce one's own search engine for things – rather than producing a closed system under exclusive control of one party, thus hampering scalability and sharing.

In the Web of Things, entities of the real world should therefore be represented as Web resources, accessed using HTTP, and should provide (among others) an HTML representation. This allows a seamless integration into the existing Web infrastructure, making it possible to utilize existing applications and services with Web-enabled things [5].

Our search engine for the Web of Things follows the basic approach of existing Web search engines: it builds up an index of all relevant pages and offers a simple search language to find indexed entities. Note that our pull-based approach does not require sensor or entity publishers to register with Dyser, as pages are found by following hyperlinks. While our search engine indexes representations of sensors and entities, including their structured data, it does *not* rely on the last known reading of a sensor for facilitating the search by the current state of an entity. Instead, indexed prediction models are utilized to efficiently resolve a search request as outlined in Sec. II.

Note that published prediction models might affect privacy when used with certain sensor installations: they reveal a "big picture" of the sensed phenomenon at once, which would otherwise have to be composed by an attacker himself, by periodically collecting sensor readings over a longer period of time. However, this issue is beyond the scope of this paper.

A. Overview

An overview of the system architecture is depicted in Fig. 1. There, *sensors* are connected to *sensor gateways* which are in charge of creating prediction models and publishing sensor information on the Web. Note that sensor gateways may also be implemented as processes running directly on the sensors, if resources permit.

As an example for a real-world entity, we consider a meeting room, represented by an HTML page. This page does not only contain static information like a textual description, but also dynamic information about its real-world state. This information is gathered from an associated sensor which detects whether the room is currently occupied. This sensor is also represented by an HTML page, which contains besides unstructured text also structured meta data about the sensor.

These embedded sensor data can be identified by the indexer, which periodically crawls the Web in order to build an index of relevant pages. Essential information about entities and sensors, including their prediction models, are then stored in the index. When a user issues a search request (in this case "`room ifw occupancy:empty`"), the *resolver* is in charge of handling its execution. For this, it will first reduce the result set to entities which match the static part of the search term ("`room ifw`") and feature the requested sensor type(s) ("`occupancy`"), by querying the index. In a second step, the dynamic part of the search request is handled by executing the prediction models of the specified sensor types for the specified sensor states ("`empty`") and the current time. The probabilities determined by the prediction models of the sensors are then combined to an overall probability for each entity according to Eq. 5. This probability reflects how likely it is that all of the entity's sensors currently monitor the states posed in the search request. Beginning with the entity which has the highest probability, the sensors of each entity are then

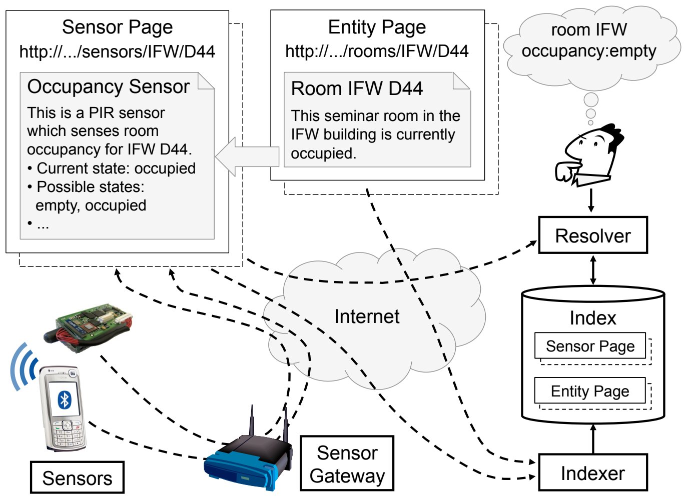
Sensor Page
http://.../sensors/IFW/D44
Occupancy Sensor
This is a PIR sensor which senses room occupancy for IFW D44.
• Current state: occupied
• Possible states: empty, occupied
• ...
Entity Page
http://.../rooms/IFW/D44
Room IFW D44
This seminar room in the IFW building is currently occupied.
room IFW occupancy:empty
Resolver
Index
Sensor Page
Entity Page
Internet
Indexer
Sensor Gateway
Sensors

Fig. 1. Overview of the system architecture

```
<div class="sensor">
  <div class="id">OccupancySensor7</div>
  <div class="location">IFW D44</div>
  <div class="type">Occupancy</div>
  <div class="currentState">occupied</div>
  <div class="possibleStates">
    <div class="state">empty</div>
    <div class="state">occupied</div>
  </div>
  <div class="predictionModelType">
    SinglePeriodPredictionModel
  </div>
  <div class="predictionModel">
    <!-- JSON serialization, omitted for
       readability reasons -->
  </div>
</div>
```

Fig. 2. HTML source of prototypical sensor microformat.

```
<a href="http://example.org/sensors/OccupancySensor7"
   rel="http://dyser.org/sensor">occupied</a>
```

Fig. 3. HTML source of an entity page including a sensor. The hyperlink text *occupied* is not evaluated by the search engine.

contacted to gather their actual state, in descending order of their overall probabilities. As soon as enough hits are found, this process is stopped and the results are returned to the user. Note that our search engine will only return results that match the query, even if the indexed prediction models are inaccurate, which could happen for newly added sensors, for example. Inaccurate prediction models will, however, degrade the performance of the search engine. To sum up, prediction models are periodically *created* by a sensor gateway, based on a set of recent sensor states, periodically *indexed* by Dyser, at a rate of days to weeks, and *evaluated* by Dyser during the resolution of a search request.

B. Integration with the World Wide Web

As all Web resources, sensors and entities should be identified by a URL and accessed using HTTP. While there may be multiple suitable data formats, we focus on HTML, as it can be directly viewed in Web browsers, is indexed by existing search engines, can be hyperlinked with other HTML documents, and is well-known to today's Web users. We denote an HTML page that represents a sensor as a *sensor page*, and an HTML page that represents a real-world entity as an *entity page*.

In order to be able to display sensor-specific information to the user and at the same time provide this data accordingly structured for the indexer, we follow the concept of *microformats* [10]. By "abusing" certain portions of HTML markup, microformats provide the possibility to semantically tag information included in HTML pages, while still giving the authors complete freedom on how that data is displayed.

C. Search Language

As we expect that the search for entities of the real world, based on their dynamic state, will be as common as the search for Web pages or images is today, it is important that the usability of Dyser is comparable to that of today's popular search engines. In particular, the search language used to construct search requests should be usable by the average search engine user without great learning efforts. We believe that query languages like SQL or SPARQL do not fulfil this requirement and are also oversized for our needs. Hence, we aim to gently extend the keyword-based search language utilized by the majority of today's Web search engines. There, one can already include structured data in a search request by using attributes, which are (name, value) pairs denoted as "`name:value`". In order to be able to search for sensor data, we introduce additional attributes to the search language which allow the specification of sensors and their current readings. For example, "`people:few loudness:quiet`" constitutes a search term with two dynamic attributes, *people* and *loudness*. The set of available attributes is defined by the sensor types indexed by the search engine.

D. Sensor Gateways

A *sensor gateway* connects sensors to the World Wide Web and makes their captured states and additional meta-information available on the Web via HTTP. Sensor gateways are also in charge of creating and publishing the prediction models for the sensors they administrate. They may furthermore offer additional functionality, like providing access to archived sensor states. While sensor gateways may run on the sensing devices itself, there may be technical or administrative limitations which require the use of dedicated gateways.

In most cases, sensors do not output a high-level state directly, but capture low-level data from which the high-level state of an entity has to be inferred, which can be performed by the sensor gateway. For example, the GSN middleware [11] may be used for that purpose as it offers a homogeneous interface to a large variety of sensors and can also perform complex data-stream processing.

IV. Implementation

In this section, we show how we leverage existing Web infrastructure in order to realize our real-time search engine for the Web of Things. The implementation of our prototype of Dyser is based on Java and PHP, using web services for communication between components.

A. Sensor and Entity Pages

As there is currently no microformat for sensor information, we designed a prototypical microformat for this purpose, which is depicted in Fig. 2. Semantic tagging is achieved by utilizing dedicated labels for Cascading Style Sheet (CSS) classes within HTML element tags, e.g., `class="sensor"`. Note that although the depicted example solely uses `<div>` elements, sensor information can be specified using any HTML element, given that it supports the `class` attribute.

In order to create a sensor page, one has to define the following structure: an enclosing HTML element is required whose class attribute is set to `"sensor"`. Inside this element, information regarding the sensor is defined using further HTML elements, whose enclosed text defines the value for the aspect specified by the CSS class names: `id` is the (local) identifier for the sensor at its sensor gateway, `location` provides information about the location of the sensor, `type` identifies the type of the sensor, and `currentState` lists the state the sensor currently detects. `possibleStates` includes a list of `states`, which together specify the list of states this sensor can perceive. In `predictionModelType`, the utilized type of the prediction model is specified, while in `predictionModel`, a serialized version of the prediction model is stored, using the JSON format.

To adapt the visual appearance of a sensor page, one can select the appropriate HTML elements and also define the according CSS classes. As CSS allow not only to change font style and color, but also to hide complete elements or to prepend or append specified text, the embedding of sensor information does not need to have an influence on the depiction of the according sensor page.

In order to create an entity page, one has to include at least one hyperlink to a sensor page, which has to follow a particular syntax: The attribute *rel* needs to be set to the specific URL *http://dyser.org/sensor* in order to denote that the corresponding hyperlink is a semantic link between an entity and a sensor page. Crawlers parsing HTML pages and following the embedded links utilize this information to detect entity pages and their associated sensors. An example of such a hyperlink is depicted in Fig. 3.

Note that in our current approach, sensor types and their states are just text labels, which can be specified at will by sensor publishers. This simplistic approach is intended to provide an open and flexible approach for publishing sensor data. In order to be able to provide well-defined and global semantics for sensor types, one could enhance the concept by outsourcing the definition of a sensor type, including its possible states and further specifications to a separate document. A sensor page would then use a hyperlink to the specification of the according sensor type.

B. Sensor Gateway

The sensor gateway was implemented in Java and features a SOAP Web service which provides access to information regarding the administrated sensors. To facilitate testing of our prototype, the sensor gateway automatically generates both a sensor and an entity page for each sensor it is in charge of and automatically publishes them using a REST interface. The current state of a sensor is modeled as a separate resource below the URL of the sensor: For example, the current state of the sensor *http://example.org/sensors/occupancy42* could be accessed at *http://example.org/sensors/occupancy42/currentstate* (note that the suffix to the base URL of the sensor matches the name of the respective section of the microformat on purpose). This greatly reduces the overhead of resolving the current reading of a sensor, as only the current state is transmitted instead of the complete sensor page.

Our sensor gateway does not only support physical sensors, but also allows the creation of virtual sensors, based on recorded log files or methods generating synthetic sensor data, for example. If not done by the sensor itself, the sensor gateway will infer a state based on recent readings of a sensor. For each sensor, a history of its perceived states is stored and utilized to create a prediction model.

C. Prediction Models

Besides the time-independent, single-period and multi-period model presented in Sec. II-C, we also implemented a random prediction model, as a baseline for evaluation. This model outputs random probabilities, which change for each time slot, state, and sensor. It is used to simulate the lookup of current sensor readings in random order.

One important aspect is an efficient representation of the models with respect to memory footprint, as the models need to be transmitted between the sensor gateway and the search engine and stored in the index. For APM, we only need to transmit one probability value for every possible output state of a sensor. For the other models, we transmit the discretized output of the model for every possible state for a certain forecasting horizon.

D. Search Engine

Like the sensor gateway, the search engine was implemented in Java and features a simple SOAP Web service to pose search requests. This Web service is wrapped by a PHP script, which provides an HTML front-end for entering search requests and displaying their results. Since we cannot expect that users are aware of all possible sensor types or of all possible states of a sensor, we provide an auto-suggest mechanism which helps the user to complete the search term by suggesting possible matches. Besides the front-end, the search engine consists of three main components:

1) Indexer: We implemented two indexers in our prototype: The first one is using a third-party Web search engine like Google in order to find all entity pages. For this, all entity pages must contain a "magic" string of characters. By searching for this magic string with Google, all entity pages can be found. However, it may take several days or even weeks until Google's Web crawler visits a page and includes it in Google's index, which is impractical for experimental purposes. For this reason, we include an alternative indexer which contacts sensor gateways directly, using the provided Web services, in order to obtain the URLs of all sensor and entity pages. The pages found by either of those methods are then downloaded, parsed and the contents are stored in the index.

2) Index: In our prototype, the index is implemented as a relational database, which is accessed using the JDBC interface, thus allowing for a large variety of different database implementations. We are currently using the MYSQL database with the InnoDB engine as storage backend. In order to speed up the evaluation of the prediction models at query time, we *materialize* the outputs of all indexed prediction models in the database for a given forecasting horizon. That is, the database does not contain the prediction models, but the discretized output of the prediction models (i.e., probability values) for a certain forecasting horizon. Finding the entity with the highest probability of matching sensor outputs is thus realized as an efficient database lookup operation.

3) Resolver: The resolution of a search request is implemented as a chain of filters over sets of entity pages as follows:

i) The given search term is parsed and separated into a static part (i.e., static keywords referring to the entity page) and a dynamic part (i.e., referring to the current output of sensors associated with an entity page).
ii) A third-party search engine like Google is used to find entity pages matching the static part of the search request. For this, a magic character string contained in every entity page is appended to the static part of the search request. As a side effect, we also obtain a *relevance* ranking for each entity page from Google. As for the indexer, we also implement an alternative approach which does not rely on Google, but queries the index of Dyser directly.
iii) Each entity page found in the previous step is checked whether it includes sensors of all types requested in the dynamic part of the search request. Entities which do not contain all requested sensor types are removed from the result set. For the remaining entities, the overall probability that they match the dynamic part of the search is then determined by querying the index holding the materialized prediction models.
iv) The entity pages produced by the previous step are then considered with decreasing probability of matching. For each entity, the sensors associated with that entity page are contacted and their current values retrieved. If the current state of a sensor does not match the state requested in the search term, the entity is removed from the list of results. To speed up processing, multiple sensors are contacted in parallel, using a pool of threads. This process stops when enough matching entities have been found.
v) Finally, all matching entities are sorted according to their relevance ranking obtained during step ii) and presented to the user.

To avoid the overhead of generating large intermediate lists of entity pages, the above steps are performed in a pipelined fashion. As soon as a certain number of entity pages are produced by one of the above steps, they are passed on to the next step. Only if not enough matching entity pages are generated in the last step, the previous steps are triggered recursively to generate more input.

V. Evaluation

We evaluated the performance of our prototypical search engine in terms of the communication overhead and latency. To obtain repeatable results, we used a realistic data set that is replayed by the sensor gateway instead of real sensors. The data set has been captured from the *Bicing* service, a short-term bicycle rental service in the city of Barecelona, Spain, which contains 385 sensors that measure the number of available bicycles at the different rental stations. This data set represents an exemplary case for our system as the data is heavily affected by the behavioral patterns of people in everyday life.

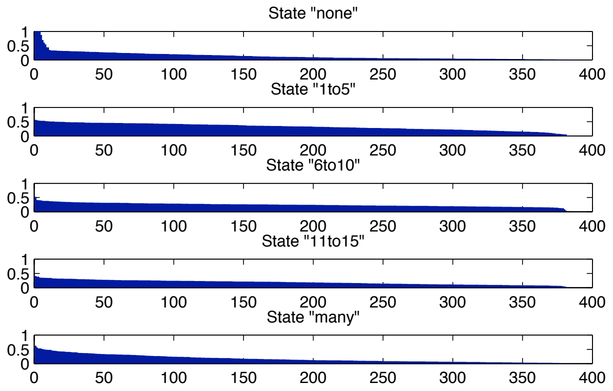

Fig. 4. Fraction of time during which each sensor reads a certain state. Sensors are sorted by decreasing fraction of time on the abscissa.

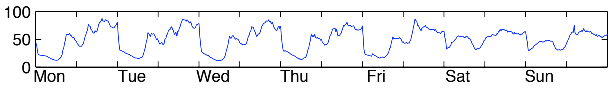

Fig. 5. Average number of sensors reading *none* over the course of a week.

A. The Bicing Service

Bicing [3] was started in March 2007 and currently operates about 6000 bicycles, which can be rent at and returned to any of the existing stations distributed throughout the city. Unlike commercial bicycle rental services, *Bicing* aims at extending the public transport service in Barcelona. This is not only reflected by the large number of bicycles and fully automated rental stations, but also by the pricing scheme, which enforces short hire periods.

B. Data Set

Bicing provides an interactive map on its homepage, which displays all bicycle stations, including the number of available bicycles and free slots at each station. Using a simple script, we fetched the HTML code of this page every 5 minutes between January and May 2009. The raw data was then processed into a single log file, resulting into a total of 385 stations. Each of those stations results in a sensor measuring the current number of available bicycles. To mimic high-level states of entities, the numbers were mapped to one of six discrete states *none* (no bikes available), *1to5* (1...5 bikes available), *6to10*, *11to15*, and *many* (more than 15 bikes available). To limit the amount of data, three consecutive time slots of 5 minutes each were averaged to a single time slot of 15 minutes.

The aggregated distribution of sensed states is displayed in Fig. 4. For each possible state, the list of sensors is sorted according to the fraction of time that they yield the respective state, in descending order. For example, for the state “none”, there exist a few sensors which always yield this state, while the vast majority of sensors will output this state less than 50% of the time. Fig. 5 visualizes the average number of sensors reading state *none* over the course of the week as an example, showing clear daily and weekly patterns.

C. Setup

We utilized the search engine implementation with virtual sensors that replay the Bicing data set described above. The time window determining the amount of past sensor data to be used for the creation of the prediction models was set to

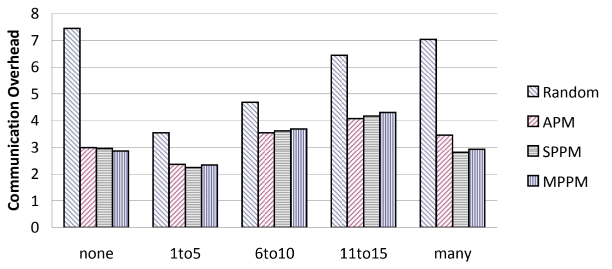

Fig. 6. Communication overhead when searching for the different states.

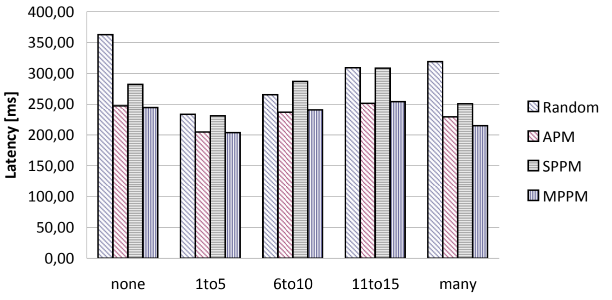

Fig. 7. Latency when searching for the different states.

8 weeks, and the size of the forecasting horizon was set to one week. The length of a time slot was set to 15 minutes and the number of required search results was set to 20. In order to perform the simulation, we used a PHP script which accessed both the sensor gateway and the search engine using the provided SOAP web services. All processes were running on the same local machine, which features two dual-core Intel Xeon Core 2 CPUs running at 2,66 GHz. The maximum number of threads for the resolver was set to 8.

Starting on March 1st 2009, all prediction models are created and indexed by the search engine. Then, a query is posed for each possible state and the outcomes are recorded for later analysis. After all states have been queried at the given point of virtual time (i.e., simulation time), the sensor gateway and search engine are instructed to advance their virtual time by one time slot (i.e., 15 minutes) and search requests for all possible states are posed again. If the forecasting horizon of one week is reached, all prediction models are recreated at the sensor gateway and re-indexed by the search engine. This process is continued for 3 months, until the end of the simulation is reached on June 1st 2009.

We consider two metrics. Firstly, the *communication overhead* is the number of contacted sensors L_n divided by the number of requested results n for a given query. A communication overhead of 1 is an optimal result, indicating that no non-matching sensors were contacted. Secondly, we consider the *latency* from issuing a query until the requested number of matches has been found and returned to the user.

D. Results

Figure 6 shows the average communication overhead when searching for the different states. We note that there is a considerable difference in the overhead caused by the random prediction model and the other prediction models. This is as expected and shows the improvement of sensor ranking over a naive approach, which would contact sensors in arbitrary order until enough results are found. Compared to the other prediction models, the aggregated prediction model produces good results despite the fact that it does not take the factor time into account. This can be explained by two reasons: First, as can be seen from Fig. 4, there is a significant number of sensors which read the searched state 40% - 50% of the time, for example, considering the state "1to5". In this case, this should result in an expected average communication overhead of 2 - 2.5, which is confirmed by our simulations. The second reason are irregularities in the simulation data: although one can identify periodic patterns in the simulation data, these are often disturbed with outliers. This might partly be attributed to the simulation data, which is inherently ordered and was mapped to unordered states. A small change in the underlying raw sensor data (e.g., the number of free bicycles changing from 10 to 11) may provoke a change of the deduced state. Prediction models which rely on periodicities in the data are susceptible to these imperfections.

Figure 7 depicts the average latency when searching for the different states, showing a similar trend as the communication overhead. However, the differences between the different models and the improvement over *Random* are smaller, as the latency does also include overhead for datebase lookups which are shared by all prediction models and which result in a constant latency baseline for all models. This baseline makes up about half of the total latency (about 160 ms for *none* and about 120 ms for all other states), showing potential for further improvements in our prototype. The remainder of the latency is predominantly caused by remote sensor readouts. Note that the latter heavily depends on the parallelism of remote sensor readout operations. In our setup, where the sensor gateway and the search engine are executing on a single computer, this parallelism is limited by the number of CPU cores and hence the latency figures can be considered a worst case that is unlikely to occur in a real deployment where sensor gateways are distributed over many computers, resulting in higher parallelism despite longer average latency for a single remote access to a sensor due to higher round-trip times in the global Internet. A notable artefact in the results are the relatively high latencies for the single-period prediction model. Based on the communication overhead figures, we would expect similar values for all three prediction models. Analysis indicates that this is an implementation-specific problem caused by memory management issues in the Java virtual machine.

VI. Related Work

We identify two major subfields of related work: recently emerging real-time Web search engines, and search engines specifically designed for networks of sensors.

A. Real-Time Web Search

The concept of real-time Web search engines has recently gained momentum, also for established search engines and social networks [12]. Twitter, for example, introduced Twitter Search (search.twitter.com). Users can search a vast number of Twitter messages using keywords and get the latest results in real time. Messages are pushed to the Twitter service, where they are archived. Twitter also provides a feed of all public Twitter messages called the *firehose* [13], which is updated in real time. Selected partners can build applications on top of this data stream. For example, Google and Bing offer real-time search based on Twitter's firehose data.

Facebook provides an integrated search engine which allows searching in the various status updates of its members. There is also a central feed of all status updates (visible to everyone) of its members, which can also be searched using Bing.

OneRiot (www.oneriot.com) is a real-time search engine that focuses on links shared by users of social community sites such as Digg and Twitter. It restricts its index to these shared sites, thus focusing on search results that are currently considered relevant by users of these communities.

Technorati.com is a search engine for blogs. According to the site, Technorati "indexes millions of blog posts in real time and surfaces them in seconds." Initially, users were able to provide a hint to the search engine when they updated their blogs, using a dedicated API call – a so-called RPC ping. However, the site notes that it is no longer using these hints, claiming that "more than 90% of the pings we received were spam and non-blogs".

An approach used in all of the outlined examples is the limitation of the search space: Twitter has strict limits not only on the allowed size of messages but also on the allowed publishing rate of messages. OneRiot focuses on a small subset of the Web, which is currently popular among users of social networks. Technorati only considers blogs, which are a small subset of the Web. A second approach is to utilize user-specified hints, which may affect the order and frequency in which sites are re-indexed. This is performed by OneRiot and was utilized in a simpler variant by Technorati. Finally, Twitter is using a centralized approach – all data is stored at Twitter's servers, thus the service has a real-time view of all published messages. These techniques would not scale to the huge and highly dynamic search space of the Web of Things.

B. Sensor Search Engines

The basic idea behind Snoogle [14], Microsearch [15], and MAX [16] is that sensor nodes attached to real-world objects carry a textual description of that object in terms of keywords. For example, a node attached to a book would contain the keyword "book". Users then have the opportunity to find real-world objects matching an ad hoc query consisting of a list of keywords. The system would return a ranked list of the top k entities matching this query. All these systems are not applicable to the Web of Things as they do not support search for dynamic content.

In Distributed Image Search [17], camera sensors are considered. A user can pose a query by specifying an image and the system returns sensors that captured similar images. Results are ranked and the top k most relevant matching sensors/images are returned to the user. While the system supports search for dynamic content, all queries are pushed to all sensors, which does not scale to a global Web of Things.

GSN [11] is a system for Internet-based interconnection of heterogeneous sensors and sensor networks, supporting homogeneous data-stream query processing on the resulting global set of sensor data streams. However, the selection of sensors is based on static meta data describing the sensors, not on their dynamic output. SenseWeb [18], a system similar to GSN, also provides for sensor discovery based on static meta data of the sensors, including geographical locations of the sensors. As in GSN, keyword-based search over static meta data is supported. In addition, geospatial queries are supported to discover sensors in a certain geographical region typically described by a polygon, assuming static sensor locations.

VII. Conclusions

Founded in the increasing trend of enriching real-world entities with embedded and globally networked sensors, we proposed the augmentation of the existing Web with representations of these real-world entities – thus offering online and real-time Web access to the state of the real world. We presented the design and implementation of Dyser, a real-time search engine that enables finding real-world entities that exhibit a certain state at the time of the query. We studied the performance of Dyser on a real-world data set containing data from 385 sensors over a period of 5 months.

An important thread for future work is the distribution – where several instances of the search engine take care of certain subsets of entity and sensor pages – and parallelization of the search engine – where the search engine itself and its index are distributed over multiple computers. The former is expected to offer a significant speedup for the execution of multiple simulateneous queries, while the latter can be expected to also speed up the execution of a single query.

References

[1] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell, "Sensing Meets Mobile Social Networks: The Design, Implementation and Evaluation of the CenceMe Application," in *SenSys 2008*, pp. 337–350.

[2] "Botanicalls." [Online]. Available: www.botanicalls.com

[3] "Bicing." [Online]. Available: www.bicing.com

[4] D. Guinard, V. Trifa, and E. Wilde, "Architecting a Mashable Open World Wide Web of Things," ETH Zurich, Tech. Rep. CS-663, 2010. [Online]. Available: www.vs.inf.ethz.ch/publ/papers/WoT.pdf,

[5] E. Wilde, "Putting Things to REST," UC Berkeley School of Information, Tech. Rep. 2007-015, November 2007.

[6] B. M. Elahi, K. Römer, B. Ostermaier, M. Fahrmair, and W. Kellerer, "Sensor Ranking: A Primitive for Efficient Content-Based Sensor Search," in *IPSN 2009*, pp. 217–228.

[7] N. Eagle and A. Pentland, "Eigenbehaviors: identifying structure in routine," *Behavioral Ecology and Sociobiology*, vol. 63, no. 7, pp. 1057–1066, 2009.

[8] C. Song, Z. Qu, N. Blumm, and A. L. Barabasi, "Limits of Predictability in Human Mobility," *Science*, vol. 327, no. 5968, pp. 1018–1021, 2010.

[9] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid, "Using Convolution to Mine Obscure Periodic Patterns in One Pass," in *EDBT 2004*, pp. 605–620.

[10] "Microformats." [Online]. Available: http://microformats.org

[11] K. Aberer, M. Hauswirth, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks," in *MDM 2007*, pp. 198 – 205.

[12] A.-M. Corley, "Real-Time Search Stumbles Out of the Gate," IEEE Spectrum, February 2010. [Online]. Available: http://spectrum.ieee.org/telecom/internet/realtime-search-stumbles-out-of-the-gate

[13] B. Stone, "Twitter and XMPP: Drinking from The Fire Hose," July 2008. [Online]. Available: http://blog.twitter.com/2008/07/twitter-and-xmpp-drinking-from-fire.html

[14] H. Wang, C. C. Tan, and Q. Li, "Snoogle: A Search Engine for Pervasive Environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1188–1202, 2010.

[15] C. C. Tan, B. Sheng, H. Wang, and Q. Li, "Microsearch: When Search Engines Meet Small Devices," in *Pervasive 2008*, pp. 93–110.

[16] K.-K. Yap, V. Srinivasan, and M. Motani, "MAX: Human-Centric Search of the Physical World," in *SenSys 2005*, pp. 166–179.

[17] T. Yan, D. Ganesan, and R. Manmatha, "Distributed Image Search in Camera Sensor Networks," in *SenSys 2008*, pp. 155–168.

[18] A. Kansal, S. Nath, J. Liu, and F. Zhao, "SenseWeb: An Infrastructure for Shared Sensing," *IEEE MultiMedia*, vol. 14, no. 4, pp. 8–13, 2007.