Real-Time Search for Real-World Entities: A Survey

Kay Römer, Benedikt Ostermaier, Friedemann Mattern, Michael Fahrmair, Wolfgang Kellerer

***Abstract*—We are observing an increasing trend of connecting embedded sensors and sensor networks to the Internet and publishing their output on the Web. We believe that this development is a precursor of a *Web of Things*, which gives real-world objects and places a Web presence that not only contains a static description of these entities, but also their real-time state. Just as document searches have become one of the most popular services on the Web, we argue that the search for real-world entities (i.e., people, places, and things) will become equally important. However, in contrast to the mostly static documents on the current Web, the state of real-world entities as captured by sensors is highly dynamic. Thus, searching for real-world entities with a certain state is a challenging problem. In this paper we define the underlying problem, outline the design space of possible solutions, and survey relevant existing approaches by classifying them according to their design space. We also present a case study of a real-world search engine called *Dyser* designed by the authors.**

I. Introduction

Today, increasing numbers of sensors and sensor networks are being connected to the Internet and the World Wide Web, making it possible to observe an ever-increasing proportion of the real world with minimal delay using a standard Web browser. While webcams may currently be the most popular sensors on the Web, services like pachube.com and Microsoft SenseWeb [1] offer APIs for publishing structured sensor data in real-time, such as energy consumption, for example. Additionally, real-world services are starting to publish real-time data relevant to their operation on the Web. One example of this is Bicing [2], a public bicycle-sharing system in Barcelona, Spain, where users can see the number of bicycles available at each rental station in real-time on the Web.

We believe that these trends are precursors of a *Web of Things* [3], [4], which will extend the original document-centric Web, making it a universal interface for the real world by giving real-world objects and places a Web presence that can be accessed using lightweight APIs (typically based on the REST principle). Representing real-world objects – including attached sensors and actuators – as Web resources gives rise to a range of novel application scenarios. For example, users might create private mash-ups, combining novel real-world services offered by sensor-augmented objects with existing Web services. An example of this might be indoor plants that send a message to their owner as soon as attached sensors report a critical state.

K. Römer is with the Institute of Computer Engineering, University of Lübeck, Germany and with the Institute for Pervasive Computing, ETH Zurich, Switzerland. Email: roemer@iti.uni-luebeck.de.

B. Ostermaier and F. Mattern are with the Institute for Pervasive Computing, ETH Zurich, Switzerland. Email: {ostermaier, mattern}@inf.ethz.ch.

M. Fahrmair and W. Kellerer are with DOCOMO Euro-Labs, Munich, Germany. Email: {fahrmair, kellerer}@docomolab-euro.com.

Just as the search for documents (Web pages, videos, blog entries, etc.) on the Web has become one of its most popular services, we expect the search for real-world entities to become equally important. The vast number of sensors that will be connected to the Web, the anticipated frequency of changes in sensor readings, and the requirement to search for the real-time state of real-world entities would all place huge demands on a real-world search engine. In this paper we explore the design space of real-world search engines and survey existing approaches. Most of the systems surveyed do not use Web protocols and standards, but their concepts could easily be translated into a Web language.

Because existing approaches do not support searching for entities by their current state in a way that could be scaled up to a global Web of Things, we conclude this paper with a description of our own recently developed system called Dyser [5], [6]. In Dyser, real-world entities (persons, places, or things) are represented by virtual counterparts in the Web that can be accessed using their unique URL – similar to the well-known Cooltown project [7]. Entities have dynamic properties that are gathered by associated sensors or deduced from their readings. For example, a public place may have a property that reflects how busy it is, which is deduced from Bluetooth readings of pedestrians' mobile phones [8]. Users can search the Web for entities not only by using keywords referring to their static descriptions, but also (and more importantly) by specifying dynamic properties that entities have to fulfill at the time of the query. For example, users could search for nearby bicycle rental stations that had enough bikes available, or for picnic places by the lake that were currently quiet.

II. The Problem and its Design Space

We consider the problem of *entity discovery*, i.e., the process of finding real-world entities with a given dynamic state. The problem refers to four architectural elements: (1) real-world entities, (2) sensors associated with them which perceive their state, (3) users posing queries to search for real-world entities with a certain state, and (4) a search engine that accepts queries and returns references to entities matching the query. In practice, it is often sufficient to return a subset of (the most relevant) entities matching the query.

Note that this problem is related to, but different from, data stream query processing as implemented in TinyDB [9], for example. With data stream query processing, queries are posed to process sensory data streams generated by a *given set* of sensors monitoring the state of an entity. Our problem, in contrast, is concerned with the *discovery* of entities and their sensors. Data stream processing queries require communication with *all* sensors (to send them queries or to retrieve their output data streams), while discovery typically requires

communication with only a small subset of all discoverable sensors and hence could potentially be scaled up to function in a global network of sensors. A small set of discovered entities or sensors might then form the input for a data stream query processing task.

Systems addressing the entity discovery problem vary across a number of different dimensions, which form the design space of our problem. In the context of this paper we consider a subset of important design space dimensions, which are detailed below. In the remainder of the paper we will refer to these dimensions to characterize existing systems that address our problem.

a) Query Type: We distinguish between two query types: *ad hoc* queries, which are one-time queries where a result is returned immediately and the query is then completed. In contrast, *continuous* queries are active for a period of time and matches are returned while the query is active.

b) Query Language: We distinguish between *keyword*-based query languages where the user provides a list of keywords to be matched and *complex* languages, which support more general predicates regarding the state of sought entities.

c) Query Scope: Some search engines are designed for small *local* domains, whereas other systems scale up to a *global* Web of Things.

d) Query Time: Queries can refer to the *real-time* state of sought entities and/or the *historical* state of entities.

e) Query Accuracy: The algorithms used to match queries with entities may be *exact* in the sense that entities returned in response to a query are guaranteed to match the query, whereas for *heuristic* algorithms returned entities may not actually match the query perfectly.

f) Query Content: This dimension of the design space defines the rate of change of the sought data. We identify three relevant intervals on this scale. Queries can either refer to *static meta data* concerning the entity (such as the entity type), to slowly-changing *pseudo-static meta data* such the entity's owner, and/or the *dynamic content* (i.e., output) of sensors describing an entity's state. It might be appropriate to identify further sub-intervals on this scale such as slowly changing dynamic content, but no systems have yet been explicitly designed for these points in the design space.

g) Entity Mobility: Some systems assume that entities are *static* (i.e., don't change their location), whereas other systems also support *mobile* entities.

h) Entity State: The state of an entity can be described either by a finite, discrete set of *categorial* states (e.g., "hot", "cold"), or by a value on a *continuous* scale (e.g., degrees Celsius).

i) Target Users: A discovery system may be either designed for *domain experts* (who could use the system to construct an application such that the end user does not directly interact with the discovery service) or for direct use by *end users*, similar to Web search engines.

Note that the above-mentioned dimensions are not all completely independent of each other, and that further dimensions may be defined that are not considered relevant to the focus of this paper.

A practical real-world search engine for end users would, of course, be a major breakthrough. Such a search engine would have to support global-scale systems, allow queries across both static meta data and dynamic content, and support mobile entities.

III. Fundamental Strategies

Despite the large design space, many existing systems rely on a small set of fundamental techniques and strategies to implement entity discovery. In this section we outline these mechanisms. The descriptions of actual systems in the subsequent section reference these mechanisms.

a) Push and Pull: Entities produce data describing their state, while users produce queries describing sought entities and the data produced by them. Discovery in a distributed setting requires that data and queries meet somewhere. This may be achieved by entities *pushing* data proactively towards the users, such that the data meets the queries in the user's system, where queries can be executed locally. Entities may also store data locally, which means that queries have to be sent to the entities to *pull* the data. With *hybrid* approaches, the data is pushed part of the way towards the users and has to be pulled from there by queries (cf. mediators below). Also, some data (e.g., that which is frequently queried) may be pushed to users, while other data (e.g., that which is rarely queried) remains at the entity and has to be pulled from there. In a global Web of Things, pulling might make more sense given that there will probably be far more entities than human users, and entities typically change their state more frequently than users produce queries.

b) Publish/Subscribe: For continuous queries, it is beneficial to establish explicit relations between entities and users such that data from entities has to be pushed only to users with potentially matching queries, or queries have to pull potentially matching entities only. Publish/subscribe is a common technique where users issue subscriptions to certain entities or data, such that only data matching the subscription is delivered to the user.

c) Mediators: A mediator is a conceptual element that sits logically between entities and users. Mediators typically maintain an aggregate view of entities, such that a query sent to the mediator can be answered without pulling data from all the entities. Mediators may be implemented in a centralized or distributed fashion (e.g., as a hierarchy of mediators, where a super-mediator has an aggregate view of its sub-mediators).

d) Inverted Index: Mediators maintaining an aggregate view of a set of entities need to be able to efficiently look up entities that might be in a certain state specified by a query. An inverted index is a data structure that allows such an efficient lookup of entities producing certain data.

e) Compression: Compression may be used to reduce the amount of storage or communication required for data and queries. For example, mediators often store a compressed aggregate view of the data from many entities. In the case of lossless compression, the system's basic operation is unaffected. However, the use of lossy compression results in an approximate view only. In this case, the query results are either

heuristic, or the approximate view may be used to identify a subset of entities or users that have to be considered by subsequent push or pull operations to obtain exact results. An extreme case for lossy compression, for example, is to communicate or store just one bit of information indicating whether the data produced by an entity has changed, such that pull operations have to consider changed entities only.

f) Models: A model makes it possible to infer information about a user or entity without actually communicating with it. Entities may use a model of a user to decide whether the user might be interested in an entity and then push data to interested users only. Users may apply a model of an entity to decide whether they might be interested in the entity and then pull data from interesting entities only. A model is often constructed from past information, e.g., a statistical model of an entity's state may be constructed from that entity's previous states. As with compression, model-based techniques may either lead to heuristic query resolution or the models may be used to identify a subset of entities or users that have to be considered by subsequent push or pull operations to obtain exact results.

g) Scoring and Ranking: Scoring is concerned with giving entities a scalar value proportional to their relevance to a query. Ranking is concerned with sorting entities according to their score. Scoring and ranking are used for two purposes. Firstly, to present users with those entities that seem to be most relevant first. Secondly, scoring and ranking may be used internally to increase efficiency by expending effort (e.g., pushing or pulling) on top-ranked entities first. Normalizing scores is an important issue for enabling a direct comparison of entities. In keyword-based searches, for example, the overall frequency of keywords could be taken into account such that frequent keywords are given a lower weighting than rare keywords.

h) Top-k Query: It is often sufficient to return a subset of the k (most relevant) entities matching a query rather than the complete set of matching entities. It is sometimes possible to find these top k entities without considering all entities, resulting in increased efficiency compared to brute-force approaches that first find all matches and then return only a subset of them.

IV. Survey

In this section we survey prominent systems that support the discovery of real-world entities. While some of them have been explicitly designed with entity discovery in mind, others support the discovery of sensors with certain properties, but their underlying mechanisms would also apply to entity discovery. Yet another class of systems is concerned with real-time searches for dynamic documents in the current Web – a problem closely related to discovering entities based on their current state. We classify the systems according to their design space in Table I. For each system, we first describe the specific problems they solve, then review the technical approach, and finally discuss their strengths and weaknesses.

A. Snoogle/Microsearch

The basic idea behind this system [10], [11] is that sensor nodes attached to real-world objects carry a textual description of that object in the form 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 (a parameter that can be specified as part of the query) entities matching this query.

The system is based on a two-tiered hierarchy of mediators. Mediators in the lower tier are called *Index Points* (IP) and each IP maintains an aggregate view of all sensors in a certain geographical area such as a room (that is, every sensor is assigned to a unique IP). The single mediator in the top tier is called the *Key Index Point* (KeyIP) and maintains an aggregate view of the whole network. Sensors push changed textual descriptions to their IP, and IPs push these on to the KeyIP. Users send queries either to an IP to query entities in the geographical area of this IP, or to the KeyIP to perform a global search. Both IPs and the KeyIP maintain an inverted index that allows efficient lookups to be performed of sensors that contain a certain keyword. Much of the research effort is devoted to maintaining this index in slow, page-structured flash storage on typical sensor nodes.

Mobile sensors are supported by implementing a hand-over protocol between IPs. Basically, IPs detect the presence of sensors by means of periodic beacon messages, update their index if a sensor (dis)appears, and inform the KeyIP. There is also support for changing object descriptions by pushing these changes to the IP and KeyIP. To compress communication, a Bloom filter is used to represent a set of keywords. Essentially, this is a bit vector representation where a keyword is entered by setting n bits in the vector to "1". A set of n hash functions is used to compute the indices of these n bits from the keyword. To check if a keyword is contained, all these n bits are examined to see whether they are set to "1". Note that this may result in false positives, where a keyword appears to be contained but is actually not. False negatives are impossible.

When resolving a query, sensors are ranked according to how many of the sought keywords they contain. To normalize the ranking across different IPs, the total frequency of a keyword in all sensors belonging to an IP is incorporated into the ranking. When performing a local query, a query message is sent directly to an IP. Based on the inverted index, the IP computes a ranked list of matching sensors and returns the top k ones. For a global query, a message is sent to the KeyIP. To compute the global top k matching sensors without obtaining complete ranked lists of matching sensors from all IPs, the KeyIP proceeds in an incremental fashion. First, it sends a query to all IPs asking for the local top-ranked sensors. By sorting this list, the KeyIP can obtain the global top-ranked sensor. To obtain the second-ranked sensor, the KeyIP sends a query to all IPs to return only sensors with a score that is higher than the score of the second-ranked sensor in the global sorted list. The results are then inserted into the global sorted list and the second-ranked sensor in this list is returned to the user. To obtain the global third-ranked sensor, the KeyIP

Dimension	Snoogle	MAX	OCH	DIS	GSN	SenseWeb	RT Web Search	Dyser
Query type	ad hoc	ad hoc	continuous	both	ad hoc	ad hoc	ad hoc	ad hoc
Query language	keywords	keywords	keywords	image	keywords	keywords+geo	keywords	keywords
Query scope	local	local	global	local	global	global	global	global
Query time	real-time	real-time	real-time	both	real-time	real-time	(near) real-time	real-time
Query accuracy	heuristic	heuristic	heuristic	heuristic	exact	exact	exact	exact
Query content	pseudo static	pseudo static	dynamic	dynamic	static	static	dynamic	dynamic
Entity mobility	mobile	mobile	mobile	mobile	mobile	mobile	—	mobile
Entity state	categorial	categorial	categorial	continuous	categorial	both	text (categorial)	categorial
Target users	end users	end users	end users	end users	experts	experts	end users	end users

TABLE I
CLASSIFICATION OF SYSTEMS SUPPORTING ENTITY DISCOVERY ACCORDING TO THEIR DESIGN SPACE DIMENSIONS.

queries all IPs for sensors with a score higher than the third-ranked sensor in the global sorted list and so on, until k sensors have been returned to the user.

Key limitations of this approach are as follows. Firstly, it only supports searches for pseudo-static meta data. Although there is a mechanism to deal with changing meta data, every change requires an immediate push to the IP and KeyIP, an approach that is not scalable. Secondly, the basic approach is not suitable for global searches due to the centralized nature of the KeyIP that maintains a complete view of the whole network, where the KeyIP pulls all IPs for every global query. While the distributed top-k algorithm may result in a reduction in the total payload data transmitted, it results in a high number of (smaller) messages. Thirdly, query results are approximate (i.e., may contain sensors that do not actually match the query) due to the use of Bloom filters.

B. MAX

MAX [12] follows a similar model to Snoogle. Tags are attached to real-world objects storing a textual description of these objects. Users can pose queries containing a list of keywords to search for tags holding one or more of these keywords. MAX then returns a ranked list of the top k matching tags.

One of the goals of MAX is that users can easily locate objects. For that, a three-tiered hierarchy of mediators is used. At the lowest level, *substations* represent immobile objects such as tables or shelves, on which mobile tagged objects can be placed. On a middle tier, *basestations* represent a geographical space such as a room containing multiple substations. At a top level, the MAX server represents the entire space covered by the system. Knowing to which substation and basestation an object belongs, it is easy to locate where an object has been left. In a prototype implementation, RFID tags are used to tag objects with a short communication range, whereas substations and basestations are sensor nodes and the MAX server is a workstation computer.

Query resolution is pull-based. The MAX server contains a directory of basestations and associated spaces, so users can select which spaces/basestations to query. The query containing a list of keywords is then sent to all selected basestations. The basestations broadcast the query to all substations within communication range, and the substations broadcast the query to all tags within communication range. The tags then count how many keywords in the query match, and the substations pull this information from the tags and forward a list of tags along with their match count to the basestation. The basestations sort the tags according to their counts. They then pull the complete tag descriptions for the top k tags from the substations. Finally, the basestation returns the descriptions of the top k tags to the MAX server and the user.

This pull-based approach is very good at dealing with tag mobility and changes to tag contents as no indexes are maintained that would require updating. Although the system was designed for pseudo-static meta descriptions, it would be possible to extend it to include content-based searches. However, the downside is that queries are very costly as they have to be sent to every single substation and tag. The resulting limited scalability makes MAX inappropriate for global networks.

C. Objects Calling Home (OCH)

This system [13] allows users to query the current location of lost real-world objects (such as umbrellas or keys). For this purpose, objects are tagged with a device holding the *identity* of the object (i.e., not a description of the object as in other systems). Mobile object sensors are used to detect the presence and identity of such objects. In a prototype, mobile phones equipped with Bluetooth are used as object sensors and objects are tagged with small Bluetooth modules. A user can query an object with a certain identity. As a result, the system will return the approximate location of the lost object if it can be found. Along with the query, the user can specify a timeout t and a budget q. As it may take some time before a mobile object sensor approaches the lost object, a continuous query is installed that is terminated after t time units. The budget q constrains the total number of messages the system may send while searching for the object.

At first sight, this may look like a very different problem from those tackled by the other systems as we are searching for the location of objects with a given identity, whereas the other systems allow us to find the identity of objects matching a given description. However, we can treat the above scenario as a content-based search problem as follows. We can consider an object sensor as a sensor that outputs the identifier of the tag that last entered its vicinity. That is, the co-domain of an object sensor is the set of all possible object identifiers. Our problem then consists of finding a sensor that currently outputs

a given object identifier. While this is certainly a specific form of content-based sensor search, it cannot be generalized to cover arbitrary sensors.

The key issue addressed by OCH is that queries should not be sent to all object sensors for scalability reasons and also to save phone batteries. This is achieved by using a model of the sought object. This model associates a probability with locations, meaning that the object currently has a certain probability of being at a certain location. This model is constructed using a number of heuristics. For example: (1) The object is probably at a location that is close to the location where it was last seen. (2) The object is probably at a location that has been recently visited by its owner. (3) The object is probably at a location where the owner spends a large part of his or her time.

A user performing a query can associate priorities with the above heuristics. A query containing the identity of the sought object, the above priorities, timeout t, and budget q is sent to a centralized mediator. The mediator uses the model of the sought object to create a ranked list of locations at which the object has a high probability of being located. Note that this step does not involve any communication with object sensors. Now the mediator selects object sensors in the proximity of locations in the ranked list. Then, the mediator subscribes to these object sensors from top to bottom in the ranked list until either one of the subscribers reports to the mediator that the sought object has been detected or until the communication budget q is exhausted. The subscriptions remain active until either the sought object has been found or the timeout t expires.

A key advantage of the above approach is that it can be scaled up for very large networks, as typically only a small proportion of all object sensors are contacted to resolve a query. However, unless the budget is set to infinity (in which case all mediators will eventually be pulled), there is no guarantee that the sought object will be found even if it is actually detected by an object sensor. While the system implements a form of content-based sensor search, the models it uses are very specific to the problem of finding objects belonging to a user and cannot be extended to other types of sensors. Furthermore, computing the models involves an overhead. For example, an object sensor carried by the object owner has to detect where objects were last seen, and the locations of object sensors have to be tracked. Fortunately, operators know the cell IDs of mobile phones anyway, so that in principle the approximate route taken by mobile phones can be obtained at a low cost. Finally, the system supports and in fact benefits from having mobile sensors, since mobility increases spatial coverage and hence the probability of finding a sought object.

D. Distributed Image Search (DIS)

This system [14] involves camera sensors. Users can pose queries by specifying an image and the system returns details of sensors that have captured similar images. Results are ranked and the top k most relevant matching sensors/images are returned to the user. The system supports both continuous real-time queries and historical ad-hoc queries, for which purpose sensor nodes store historical images. DIS could be used to search for places with a certain visual state.

To avoid transmitting images, the system uses a feature extraction mechanism (i.e., an extreme form of lossy compression) that maps an image to a set of *visterms* (similar to keywords) describing the image. This mapping is a two-stage process, which first maps an image to several hundred continuous 128-dimensional vectors using the SIFT algorithm. Note that SIFT vectors are mostly invariant to viewpoint changes, some occlusions, and lighting effects such as specularities. As the set of SIFT vectors is still large, a second stage maps SIFT vectors to descriptions of image features. For this purpose, the SIFT algorithm is run on a large pre-recorded image database, and the resulting SIFT vectors are clustered. Each resulting cluster is then represented by an average SIFT vector and a visterm. The resulting dictionary is stored in the flash memory of sensor nodes as a read-only data structure and used to map SIFT vectors to visterms.

When a sensor node captures a new image, it is converted into a set of visterms by running the SIFT algorithm and using the dictionary. The resulting visterms are then stored in an inverted index (also in flash memory) for efficiently looking up images that match certain visterms. Much effort is spent in devising efficient data structures to work around the constraints of flash memory.

To perform a query, the user sends the reference image to a centralized mediator. The mediator maps the image to a set of visterms and sends the visterms to all sensors. The sensors use their inverted index to look up images containing at least one of the query visterms. For these images, a normalized score is computed quantifying how well the query visterms match the image visterms, taking into account the overall frequency of visterms. From the ranked list, the identifiers of the top k ranked images are returned to the mediator. The mediator then computes a global ranked list, pulls thumbnails of the global top k images from the sensors and displays them to the user.

A key limitation of the above approach is that it is not scalable for large networks as every query has to be sent to all sensors. Also, the search is approximate as image matching is based on feature vectors. Although the system has been designed for image searches, the problem is mapped to keyword searches, so some of the mechanisms may be applicable to other types of sensors that can also be mapped to keywords. Content-based search and mobility are supported as sensor data is not pushed to the mediator, but instead local computations are performed when a new image is captured.

E. Global Sensor Networks (GSN)

GSN [15] is a system for the Internet-based interconnection of heterogeneous sensors and sensor networks, supporting homogeneous data-stream query processing on the resulting global set of sensor data streams. In the context of this survey, we are not interested in the actual data stream processing, but in the discovery of sensors that form the input to a data-stream processing task.

A key abstraction in GSN is a *virtual sensor* that represents either a physical sensor or a virtual entity that takes as input

one or more data streams from other virtual sensors and processes them to produce a single output data stream. GSN supports the discovery of virtual sensors and their interconnection to form new virtual sensors.

Each virtual sensor in GSN has a unique identifier and can be annotated with meta data to describe the sensor. Such meta data may be used, for example, to specify the location or the type of sensor. To discover a virtual sensor, one can either specify the identifier of the virtual sensor being sought, or provide meta data describing the sensor. In the latter case, a keyword search is performed to find matching sensors.

The actual discovery is performed using a peer-to-peer approach. The meta data (i.e., key-value pairs) for the virtual sensors is stored in a peer-to-peer directory, supporting scalable discovery in large networks [16]. Virtual sensors themselves are hosted in a so-called GSN container, which is essentially a process executing on a computer connected to the Internet. These GSN containers also form a peer-to-peer overlay.

GSN also supports mobile sensors, provided they implement a self-description mechanism known as Transducer Electronic Data Sheet (TEDS) as specified by the IEEE 1451 standard. When such a sensor enters the communication range of a GSN base station, the TEDS is downloaded and an appropriate virtual sensor is instantiated using the information in the TEDS. Likewise, if the sensor leaves, the virtual sensor is deleted.

A key limitation of the sensor discovery mechanism in GSN is that it does not support content-based discovery. While data stream processing could be used to find sensors with a given current output value, this would require communication with all discoverable sensors, which would certainly not scale up for use with large numbers of sensors.

F. SenseWeb

SenseWeb [1] also provides for sensor discovery based on static meta data relating to sensors, including the geographical locations of the sensors. In addition to keyword-based searches of the meta data, geospatial queries are supported for discovering sensors in a certain geographical region typically described by a polygon. Sensor readings are streamed to sensor gateways, which register with the SenseWeb core and provide a SOAP-based API for retrieving sensor information and readings. SenseWeb maintains a central repository of sensor meta data. However, sensor readings are not streamed from the sensor gateways to a central sink. For this reason, SenseWeb presently only supports searches of static meta data, not of current sensor readings. SenseWeb also includes an application that provides a geographical representation of registered sensors on a map called SenseMap.

Due to the fact that sensors and their locations are registered in a central repository, sensor mobility would result in a significant overhead when updating the repository. Instead, SenseWeb introduces the concept of a mobile proxy representing a virtual sensor that is positioned at a fixed location. The proxy dynamically binds to a real sensor that happens to move to the location covered by the proxy. In this way, the central repository does not need to support mobility directly.

G. Real-Time Web Search Engines

Real-time search engines for the Web support searches for dynamically changing content and have recently gained momentum, also for established search engines and social networks [17]. Although they do not directly support entity discovery, the underlying mechanisms could be adopted to search for entities with a given dynamic state. In this section, we introduce three examples and outline the concepts involved.

Twitter is a company that provides a real-time, web-based, public messaging system (a so-called *micro-blogging* service), which is currently very popular on the Web. The idea is to provide people with the possibility of saying what they are currently doing by using textual messages limited to 140 characters. Twitter has recently introduced its own search engine, 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, updated in real-time, using the XMPP protocol [18]. Selected partners can build applications on top of this data stream.

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". Until recently, 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".

One approach used in all of the outlined examples is that of limiting the search space. Twitter has strict limits not only for the permitted size of messages but also for the permitted rate of publishing messages. OneRiot focuses on a small subset of the Web that is currently popular among users of social networks. Technorati only considers blogs, which are a 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 uses a centralized approach – all data is stored on Twitter's servers, so the service has a real-time view of all published messages.

While the concept of Twitter and its search engine could also be utilized for publishing sensor readings and searching for them in real-time, we would question whether this concept, which requires a central instance aware of all current Twitter messages, could be scaled up to the dimensions of an anticipated Web of Things. When compared to human-generated messages, the volume of sensor-generated content is expected to be magnitudes higher, both in the number of sensors and in the frequency of updates. For the same reason, we doubt whether the concepts utilized by OneRiot

and Technorati are appropriate for enabling real-time searches in the upcoming Web of Things.

V. Dyser – A Case Study

In this section, we introduce *Dyser*, our prototype of a search engine for the real world. After discussing the concepts involved, we present an evaluation based on the results of a simulation using real-world data. Finally, we outline the prototypical implementation.

A. Overview

Dyser is a prototypical search engine for the Web of Things, developed by the authors of this article. It allows real-world *entities* (i.e., people, places, and objects) to be searched by their *current state*. As the expected audience of our search engine consists of average Web surfers rather than domain experts, we argue that users will most likely not be interested in searching for *sensors* with a specific reading, but for *entities* in the real world with a specific current state. So rather than searching for *loudness sensors* with a current reading below 30 dB, we assume that users will instead be interested in searching for *places* that are currently *quiet*. The state of an entity is determined by the current states monitored by its associated sensors. The state a sensor outputs is inferred from its readings.

In our model, there are two key elements: *sensors* and *entities*. Each sensor and each entity has a virtual counterpart, a Web resource, identified by a URL and accessible using HTTP. For all of these Web resources, there is always an HTML representation, which we call the *sensor page* and the *entity page* respectively. In addition to unstructured text they also contain structured information, for example, the type of sensor or its possible readings. The relationship between sensors and entities is *many-to-many*, i.e., one sensor can be associated with multiple entities, and one entity can be associated with multiple sensors.

The proposed search language is quite simple and based on that of popular Web search engines. The user specifies a list of keywords and properties that have to be fulfilled by possible results. For example, when looking for quiet Italian restaurants, we could search for "`italian restaurant loudness:quiet`". The first two elements of the query are static keywords, which have to appear on the entity page. The last element of the query specifies a (dynamic) property that is determined automatically by an associated loudness sensor. Dyser will return a ranked list of entities that currently match the search term. From there, the user may browse to the corresponding entity pages and in turn to the associated sensor pages, for further information. A classification of Dyser according to our design space dimensions is depicted in Table I.

An overview of the architecture is shown in Fig. 1, in the context of a user searching for free rooms in the IFW building. Dyser consists of a *resolver*, which handles user queries according to the process outlined in Section V-B, an *index*, in which indexed meta data on sensors and entities is stored, and an *indexer*, which crawls sensor and entity pages.

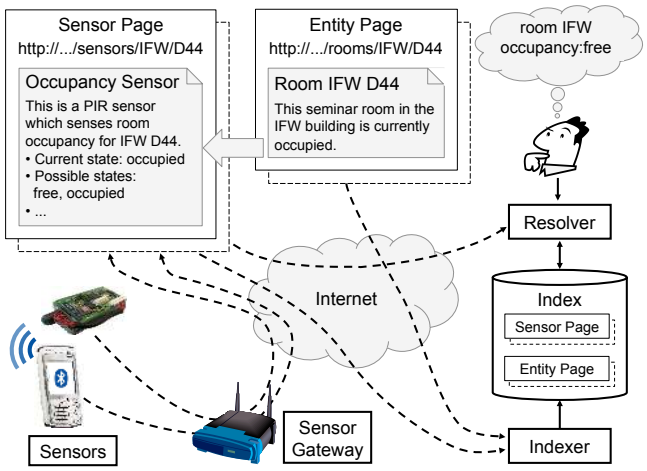

Fig. 1. Architecture of Dyser

As the index contains only pseudo static meta data, it could be mirrored to enable scalability.

B. Approach

Although we have built a central index consisting of static meta data for sensors and entities, we do not rely on a central sink to which the readings of all the participating sensors are streamed in real-time. This means there is no global view of the world and thus to answer a query, the search engine needs to contact relevant sensors (i.e., sensors that could possibly read the searched state) at the time a query is posed, in order to determine whether they currently match the query. Entities whose associated sensors do not read the searched state are excluded from the result set, which Dyser returns to the user as soon as enough matches have been found. Note that indexing current sensor readings is not an option, as the index would be outdated as soon as it was built due to the anticipated frequency of changes in sensor readings.

Even though this query resolution process can be optimized by contacting multiple sensors in parallel, it is not scalable with respect to network traffic – given that the number of possible results is significantly larger than the number of actual results, many sensors would be contacted unnecessarily by the search engine when processing a single query. However, instead of contacting the sensors in an arbitrary sequence, we could contact them in an order that reflected the probability that they currently matched the query. We call this approach *sensor ranking* and argue that it enables scalability, provided we can order the list of relevant sensors sufficiently well. For this, we need to be able to estimate the probability of a sensor matching a query with sufficient accuracy.

We used prediction models for this purpose, which return the probability of a sensor reading a specific value at a given point in time. For a given search term, the relevant sensors are contacted in descending order of their prediction model outcomes. Provided that the prediction models map reality sufficiently accurately, it should be possible to significantly reduce the number of sensors contacted per query. Prediction models are created by sensors or by sensor gateways

and are published on the corresponding sensor page. Dyser periodically indexes sensor and entity pages, including their relations, much like current Web search engines do. The published prediction models are then copied to Dyser's index, making their evaluation a local operation. Although this index is centralized in our current prototype, it could easily be distributed, for example by allocating sensors of the same type to the same sub-index. As queries contain the sensor type being searched for, we can easily find the sub-index that holds the models relevant for a query.

The integral assumption in our approach is that there are enough sensors that 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, daily, weekly and yearly cycles in people's lives can usually be identified. Recent research [19], [20] has shown promising results regarding the predictability of human behavior.

To sum up, Dyser performs the following steps at query time in order to process a search request for k results:

1) All entities that do not match the given keywords or sensor types are filtered out. The remaining entities are ranked according to their expected relevance to the user. The process starts with the first x entities.
2) For the entities currently considered, the probability that the entity currently matches the searched state is computed. For each entity, the prediction models of its associated sensors are evaluated, using the requested states. The probability of an entity is then computed as a function of these outcomes.
3) The considered entities are re-sequenced (in descending order of probability).
4) Beginning with the top-ranked entity, the search engine contacts the entity sensors to check whether their values actually match the searched state. Only entities that fulfill the search request are added to the result set.
5) If enough (i.e., k) matching entities have been found, the list of results is re-sorted according to the relevance criterion in 1) and returned to the user. If not, the process continues from step 2) with the next x entities.

Note that we are using two different ranking approaches: *sensor ranking*, which affects the order in which sensors are contacted in order to test whether they are actually reading the searched state; and *relevance ranking*, which adjusts the order in which elements are displayed to the user. We need to combine both these approaches – sensor ranking for efficiency and relevance ranking to fulfill the expectations of the user. However, this combination is admittedly a trade-off between efficiency and relevance.

C. Prediction Models

In this section we give three examples of concrete prediction models for use in the search engine. For this, we first state the required definitions.

A sensor s is a function

$$s : T \mapsto V \tag{1}$$

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 "free"). We do not assume that elements of V follow some order, as we also want to support sensors that sense inherently unordered phenomena, e.g., sensors that return the current activity of a user.

Each sensor is associated with a type and optionally further meta-information such as a location. A prediction model for a sensor s is a function

$$P_{s,t^0,t^1} : T \times V \mapsto [0,1] \tag{2}$$

The parameter $t^0 \in T$ refers to the time at which the first sensor reading was considered, $t^1 \in T$ refers to the time at which the model was constructed, meaning that all sensor values $s(t^0 \leq t_i \leq t^1)$ were 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 poor indicators of a sensor's future output. Also, using a time window instead of all past data typically reduces the resources (i.e., execution time and memory footprint) required to construct the model.

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

Probably the most simple prediction model computes the fraction of the time during which the sensor output equals v within the time window $[t^0, t^1]$, that is:

$$P_{s,t^0,t^1}(t,v) = \frac{1}{t^1 - t^0} \int_{t^0}^{t^1} \chi(s,q,v)\mathrm{d}q \tag{3}$$

where $\chi(s,t,v)$ is an indicator function that returns 1 if the sensor s reads v at time t, and 0 otherwise:

$$\chi(s,t,v) = \begin{cases} 1 & : \quad s(t) = v \\ 0 & : \quad \text{else} \end{cases} \tag{4}$$

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 model equals 1. Note that the output of the prediction model is independent of the actual point in time t of the search. Hence, 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 our assumption that there is a dominant period length L after which the sensor output is likely to repeat. 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. Assuming[1] that the time window size $t^1 - t^0$ is an integral multiple of L, i.e., $NL = t^1 - t^0$ for integral N, and that the forecasting horizon $t - t^1$ is smaller than L, we can perform a prediction as follows:

$$P_{s,t^0,t^1}(t,v) = \frac{1}{N} \sum_{1 \leq i \leq N} \chi(s, t - iL, v) \tag{5}$$

[1]Note that these limitations can be easily removed, but result in a somewhat less obvious formula.

Here, we consider all points in time t' contained in the time window that have the same *phase* as t with respect to period length L, i.e. $t' \equiv t \mod L$ such that $t' \in [t^0, t^1]$. Under the assumptions stated above, this is the case for $t' = t - iL$ for all $1 \leq i \leq N$. The output of the prediction model equals the fraction of the instances of t' for which $s(t') = v$ among all t'. As this model assumes a periodic process with a single period, we call this model the *single period prediction model* (SPPM). Note that a spectral analysis of the sensor data or a periodicity indicator as given in [21] can be used to automatically derive the dominant period length L for a given data set $s([t^0, t^1])$. Alternatively, L could be derived from domain knowledge.

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 on the 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 periodic patterns in the time window using a variant of an existing algorithm [22]. 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, and 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, 2, occupied, 0.5) means that every second day of the week (i.e., Tuesday) the probability of a room being occupied is 0.5.

To make a prediction, we first filter all patterns that match the search time t, that is, we retain 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. Assuming N such patterns exist, we perform a prediction as follows:

$$P_{s,t^0,t^1}(t, v) = \max_{1 \leq i \leq N} p_i \tag{6}$$

where p_i is the probability of periodic pattern i. In order to mitigate the effects of sudden changes in periodic patterns in the data set, we could specify a lower and upper bound (ω_{min} and ω_{max} respectively) for the number of instances of periodic symbols considered. We call this model the *multi-period prediction model* (MPPM).

D. Adjustment Process

During experiments with our system using realistic data sets we observed that sudden changes in the periodic processes underlying the data (e.g., the end of a semester where room occupancy patterns in a university change drastically) typically cause sensors to be consistently misranked until the old data is shifted out of the time window, resulting in the decreased performance of the search process. Outdated, erroneous or malicious prediction models may also have the same effect.

Using the prediction models introduced in the previous section, a search engine would compute a rank list S^Q containing m sensors s_i sorted by decreasing probability of matching a query for sensor value v_Q at time t_Q. In the best case, all sensors actually matching the query would be at the top of the rank list, while all non-matching sensors would be at the bottom. Imperfect rankings result if a sensor s_i is misranked for one of the following reasons. Firstly, s_i matches the query but is ranked lower than other sensors that do not match the query. Secondly, s_i does not match the query but is ranked higher than other sensors that do match the query. Formally, we can measure this ranking error for sensor s_i by counting the number of non-matching sensors ranked higher than a matching sensor s_i, or by counting the matching sensors ranked lower than a non-matching sensor s_i. The sign of the following metric indicates whether s_i should have been ranked lower (< 0) or higher (> 0).

$$\mathrm{re}(s_i, v_Q, t_Q) = \tag{7}$$
$$\begin{cases} -|\{s_j \in S_k^Q | j > i \wedge s_j(t_Q) = v_Q\}| : s_i(t_Q) \neq v_Q \\ \quad |\{s_j \in S_k^Q | j < i \wedge s_j(t_Q) \neq v_Q\}| : s_i(t_Q) = v_Q \end{cases}$$

where S_k^Q represents the ranked list of sensors that were contacted for a given query Q in order to provide k results:

$$S_k^Q = \begin{cases} S^Q & : \quad M_k = 0 \\ s_1 s_2 s_3 ... s_{M_k} & : \quad else \end{cases} \tag{8}$$

The rank M_k of the kth matching sensor is defined as

$$M_k = \begin{cases} \underset{1 \leq i \leq m}{\arg min} (\chi(s_i, t_Q, v_Q) * i) & : \quad k = 1 \\ \underset{M_{k-1} < i \leq m}{\arg min} \ (\chi(s_i, t_Q, v_Q) * i) & : \quad else \end{cases} \tag{9}$$

The adjustment process modifies the probabilities computed by the prediction models using the above ranking error $\mathrm{re}()$ such that systematic misranking is corrected. Essentially, we introduce a control loop where the ranking error of s_i controls the ranking of s_i in future queries such that the future ranking error is reduced. For this, we compute an adjustment term $\mathcal{AT}$ for each sensor s_i and sought value v_Q at query time t_{Q_j}:

$$\mathcal{AT}(s_i, v_Q, t_{Q_j}) = \mathcal{AT}(s_i, v_Q, t_{Q_{j-1}}) + \frac{\mathrm{re}(s_i, v_Q, t_{Q_j})}{|S_k^Q|} \tag{10}$$

where the number of sensors contacted $|S_k^Q|$ is used to normalize the ranking error. The probability estimate for a sensor s_i holding a state v_Q at the time $t_{Q_{j+1}}$ of the subsequent query is then modified using the above adjustment term:

$$\hat{\mathrm{P}}_{s_i}(t_{Q_{j+1}}, v_Q) = \mathrm{P}_{s_i}(t_{Q_{j+1}}, v_Q) + \mathcal{AT}(s_i, v_Q, t_{Q_j}) \tag{11}$$

The rank list resulting from these adjusted probabilities is then in turn used to update the adjustment terms according to (10). To avoid the use of outdated adjustment values, $\mathcal{AT}$ is reset to zero when a sensor has not been queried for some time or when a sensor's prediction model is updated.

The adjustment process is most effective if the query is executed frequently. However, if the query is executed infrequently, then its contribution to the overall performance of the search engine will be small anyway. Note that this approach can be used with any prediction model.

E. Evaluation

To assess the effects of sensor ranking and the adjustment process, we performed various simulations on a real-world data set. The evaluation presented here was conducted in

Matlab and addresses the effects of sensor ranking and the adjustment process, without considering the integration of these approaches into Dyser. This means that the simulations ignore the influence of a relevance ranking of results and do not consider multiple sensors per entity.

1) Data Set: In the simulations outlined below, we utilized a data set gathered from *Bicing*, a public bicycle-sharing service located in Barcelona, Spain [2], as it contains many geographically distributed sensors whose output is related to people's behavior. Bicing was started in March 2007 and currently operates about 6000 bicycles, which are distributed across 400 stations throughout the city. Bicycles can be rented from and returned to any of the existing stations using an automated process. Unlike commercial bicycle rental services, Bicing aims to expand the public transport service in Barcelona. This is not only reflected by the large number of bicycles and rental stations, but also by a maximum permitted rental time of 2 hours. Bicycles provided by Bicing have no locks and can therefore be stored safely only at Bicing stations.

For each rental station, Bicing publishes on its homepage the current number of bicycles available and the current number of free return slots. We retrieved this data every 5 minutes, using a simple script. The readings we gathered were then cropped to include only data from January to May 2009, and processed in order to generate a single log file. Stations that entered into service during this period were excluded from the data set, so only stations that were in service for the total 5 month period of time were considered. This resulted in a total number of 385 stations. For our simulations, we considered virtual sensors that reflected the number of bicycles available at each rental station. For that we transformed the data into discrete states using a simple mapping scheme: 0 available bicycles were mapped to the state *none*; 1-5, 6-10 and 11-15 available bicycles were mapped to the states *1to5*, *6to10* and *11to15* respectively; more than 15 available bicycles were mapped to the state *many*; and erroneous readings were mapped to the state *unknown*. To limit the amount of sensor data, three consecutive time slots of 5 minutes each were mapped to a single time slot of 15 minutes for the resulting log file.

Fig. 2 visualizes the number of sensors reading a certain state during the course of the week, averaged out over the total data set. For example, it shows that there were on average about 50 sensors that read the state "none" at noon on Tuesdays. One can clearly identify typical daily and weekly patterns, e.g., for the state "none".

2) Error Metric: In order to be able to assess the performance of our approach, we measured the *communication overhead* in our simulations, which is the number of contacted sensors M_k divided by the number of requested results k for a given query. A communication overhead of 1 is an optimal result, indicating that no non-matching sensors were contacted. When the number of requested matches cannot be provided, the communication overhead is undefined and will not be considered when computing the average communication overhead. We formalized the communication overhead as

$$o(t,v,k) = \begin{cases} undefined & : \quad \sum_{i=1}^{m} \chi(s_i,t,v) < k \\ \frac{M_k}{k} & : \quad \text{else.} \end{cases} \tag{12}$$

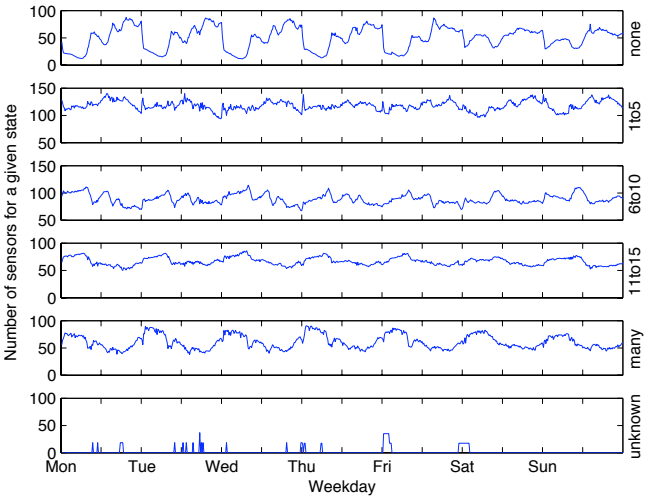

Fig. 2. Average number of sensors reading a given state vs. weekday

where t represents the query time, v the sought state, m the total number of relevant sensors, and k the number of requested results.

3) Simulation Setup: The simulations we conducted considered queries in a time frame from March 1st to May 31st 2009. The first two months (January and February) of the data set were used to construct the initial prediction models. The forecasting horizon was set to one week, and the size of the time window for the prediction models was set to 8 weeks. All prediction models were periodically re-created when their forecasting horizons were reached. Thus, prediction models "aged" during the course of a simulated week. Search requests for all possible sensor states were placed every 15 minutes throughout the simulation period. The number of required results k was set to 20. All simulations were conducted both with and without the adjustment process.

The period length for the single-period prediction model (SPPM) was set to one week, as this turned out to be the dominant period length. For the multi-period prediction model (MPPM), ω_{min} was set to 4 and ω_{max} was set to 8. As a baseline, we included the results from a prediction model that outputs random values, resulting in a randomized sensor ranking that changes per time slot and sensor state.

4) Simulation Results: The average communication overhead determined in the simulation runs is depicted in Figs. 3 and 4. Note that the state "unknown" is omitted from the discussion, since it always results in an optimal overhead of 1 for all considered prediction models. This is because either all sensors sense the state "unknown" or none of them do. As we only computed the overhead for situations where there were enough results, we only considered the former situation, which always resulted in an optimal ranking.

For both graphs, we can see that the average communication overhead for the individual states varies across all considered prediction models. One factor that affects this variance is the average number of sensors reading the searched state. A larger percentage of matching sensors usually results in fewer sensors needing to be contacted in order to find the required

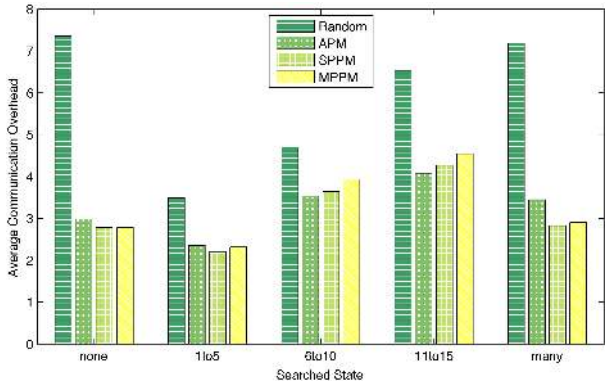

Fig. 3. Average communication overhead without adjustment process

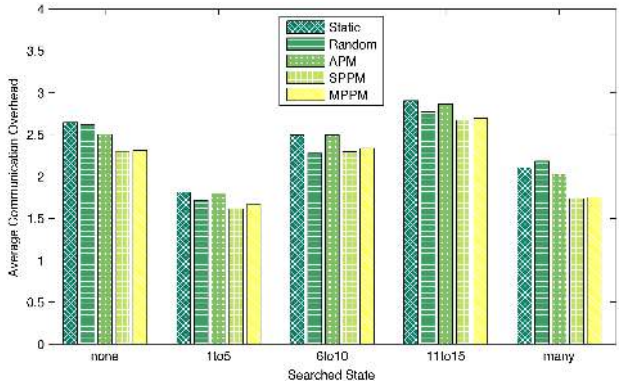

Fig. 4. Average communication overhead with adjustment process

number of results. As we can see from Fig. 2, on average the state "1to5" is read by the sensors most frequently. This is also the state with the smallest communication overhead. The correlation between the average communication overhead and the average number of matching sensors can be seen best when comparing the data from Fig. 2 with the results of the random prediction model in Fig. 3. However, when considering an actual prediction model, the "predictability" of a certain state naturally influences the communication overhead. For example, although the state "none" has a low average number of matching sensors, its communication overhead is quite low compared to the other states when the adjustment process is not taken into account. This can be attributed to its periodic nature, which can be identified in Fig. 2.

a) No Adjustment Process: When the adjustment process is not taken into account, we see that for each state 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 that sensor ranking is better than the naive approach of contacting sensors in an arbitrary sequence until enough results are found. Note that the random prediction model produces random but different rankings per time slot and searched state, so we can rule out the effects of poorly selected random numbers. Compared to the other prediction models, the aggregated prediction model yields good results despite the fact that it does not take the time factor into account. This can be explained by two reasons. First, an analysis of the simulation data reveals that there are a significant number of sensors that read the searched state 40% to 50% of the time, for example, considering the state "1to5". In this case, this should result in an expected average communication overhead of 2 to 2.5, which is confirmed by our simulations. The second reason is irregularities in the simulation data: although we can identify periodic patterns in the simulation data, these are often disrupted by outliers, which makes the patterns imperfect. This could partly be attributed to the data set, which was mapped to a small number of relatively coarse states for the simulation. A small change in the underlying raw sensor data (e.g., a change in the number of free bicycles from 10 to 11) may thus provoke a change in the deduced, coarser state. Prediction models that rely on periodicities in data are susceptible to these discretization effects. Additionally, in our model, we do not look at ordered states, which further reduces the information content of the data set.

b) With Adjustment Process: When looking at the simulation that utilized the adjustment process (AP), we see that all the results benefit from this approach (Fig. 4). While the largest average communication overhead without considering the AP is about 7, this falls to about 3 when utilizing the AP. This can be explained by the feedback loop we introduced, which decreases the prediction results of sensors ranked too high and increases the prediction results of sensors ranked too low. We also included a static "prediction model" that produces an arbitrary but constant ranking of the sensors, in order to better assess the effects of the adjustment process.

Comparing the different prediction models reveals that the average prediction overhead of the static and random prediction model is only slightly worse than that of the other prediction models. This finding (prediction models that do not predict meaningful values result in good rankings) may appear counter-intuitive. To understand it, remember that the adjustment process works best if executed frequently. Since we executed it for every query in our simulation, we updated it every 15 minutes per state, and reset it when re-creating the prediction models. For the static prediction model, this resulted in behavior comparable to caching – sensor predictions are adjusted to reflect the recent state. If the states sensed by the sensors are not changing too frequently, the adjustment process is able to model a ranking and therefore compensate for the lack of an actual prediction model. However, this does not explain why the random prediction model with adjustment process performs even slightly better than a static ordering of the sensors. An analysis of the simulation results revealed that, with the adjustment process, there were some sensors that were not considered at all when using a static ranking. However, when using a ranking that varies over time (for example, like the random prediction model does) then all sensors will be considered by the adjustment process, which increases the probability that sensors which continuously read the searched state over a longer period of time will be found and adjusted "upwards". Thus, in this case, the random permutations actually help the adjustment process. The adjustment process generally seems to increase the stability of the top k sensors

of a ranking, which results in a lower ranking error, provided that the sensor states do not change too rapidly.

5) Summary: Based on our simulation results, we identified two major findings. First, that the greater effort introduced by more sophisticated prediction models does not necessarily lead to improved sensor rankings. Second, that the adjustment process not only significantly improves the sensor ranking, but also alleviates the influence of the prediction models on the sensor ranking. The best results are obtained by combining the single-period prediction model with the adjustment process. However, we should point out that these findings are based on the simulation data considered and might not be directly generalizable to other scenarios.

F. Prototype

We carried out a two-part prototypical implementation of the concepts outlined in this case study. First, a sensor gateway that connects sensors to the Web, computes prediction models, and publishes automatically generated sensor and entity pages. Second, our search engine *Dyser*, which indexes sensor and entity pages, and which provides a web-based frontend and an API for searching entities by their current state in real-time.

In order to be able to publish structured data, we used so-called *microformats* [23] on the sensor pages. These are a way of publishing structured data using plain HTML, such that the content can be displayed in a standard Web browser and can be easily parsed by the search engine to extract the structured data. The association between an entity and its sensors is created by utilizing specially crafted hyperlinks, placed on the entity page, which link to the corresponding sensor pages. To save query time, we materialized crawled prediction models, i.e., we evaluated them at indexing time for all possible states and all possible time slots within their prediction horizons and stored the results in our index. A ranking of the results based on expected relevance was provided by utilizing Google's ranking of results. The sensor and entity pages therefore contained a special keyword ("magic string"), which enabled us to find all sensor and entity pages on the Web. When Dyser processes a query, it will first craft a search term for Google by using only the keyword-based parts of the query, plus the magic string. The returned list of results is then used as described in Section V-B. Since we cannot expect users to remember sensor types and their possible readings, we also included in our real-world search engine a recommendation mechanism that suggests possible sensor types after typing the first few letters of the sensor name and also lists all possible states for a selected sensor.

VI. Conclusion

The emerging trend of publishing real-time sensor data on the Web opens up a wide variety of novel application scenarios. One application, which we detailed in our paper, is a search engine for the real world which allows users to search for real-world entities with a given state. We provided a survey of existing approaches to this problem, including a classification of the systems in the design space for real-world search engines. The paper concluded with a detailed case study of Dyser, which utilizes a novel approach for searching the real world in real time. Evaluation results presented in this context indicate that periodic patterns in sensor readings may be exploited to significantly decrease the communication overhead of a real-world search engine.

References

[1] 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.

[2] "Bicing," http://www.bicing.com.

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

[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: http://www.vs.inf.ethz.ch/publ/papers/WoT.pdf,

[5] B. Ostermaier, B. M. Elahi, K. Römer, M. Fahrmair, and W. Kellerer, "Poster Abstract: Dyser – Towards a Real-Time Search Engine for the Web of Things," in *SenSys '08: Proc. 6th Conference on Embedded Networked Sensor Systems*. ACM, 2008, pp. 429–430.

[6] B. M. Elahi, K. Römer, B. Ostermaier, M. Fahrmair, and W. Kellerer, "Sensor Ranking: A Primitive for Efficient Content-Based Sensor Search," in *Proc. 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2009, pp. 217–228.

[7] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic, "People, Places, Things: Web Presence for the Real World," *Mobile Networks and Applications*, vol. 7, no. 5, pp. 365–376, 2002.

[8] "Reality Mining." [Online]. Available: http://reality.media.mit.edu

[9] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[10] H. Wang, C. C. Tan, and Q. Li, "Snoogle: A Search Engine for Pervasive Environments," *IEEE Transactions on Parallel and Distributed Systems*, 2009, http://doi.ieeecomputersociety.org/10.1109/TPDS.2009.145.

[11] C. C. Tan, B. Sheng, H. Wang, and Q. Li, "Microsearch: When Search Engines Meet Small Devices," in *Pervasive '08: Proc. 6th International Conference on Pervasive Computing*. Springer, 2008, pp. 93–110.

[12] K.-K. Yap, V. Srinivasan, and M. Motani, "MAX: Human-Centric Search of the Physical World," in *SenSys '05: Proc. 3rd Conference on Embedded Networked Sensor Systems*. ACM, 2005, pp. 166–179.

[13] C. Frank, P. Bolliger, F. Mattern, and W. Kellerer, "The Sensor Internet at Work: Locating Everyday Items Using Mobile Phones," *Pervasive and Mobile Computing*, vol. 4, no. 3, pp. 421–447, 2008.

[14] T. Yan, D. Ganesan, and R. Manmatha, "Distributed Image Search in Camera Sensor Networks," in *SenSys '08: Proc. 6th Conference on Embedded Networked Sensor Systems*. ACM, 2008, pp. 155–168.

[15] K. Aberer, M. Hauswirth, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks," in *Mobile Data Management (MDM'07)*, Mannheim, Germany, May 2007.

[16] ——, "Middleware Support for the "Internet of Things"," in *Fachgespräch Sensornetze*, Stuttgart, Germany, 2006.

[17] 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

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

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

[20] J. Reades, F. Calabrese, A. Sevtsuk, and C. Ratti, "Cellular Census: Explorations in Urban Data Collection," *IEEE Pervasive Computing*, vol. 6, no. 3, pp. 30–38, 2007.

[21] A. Raveh and C. S. Tapiero, "Periodicity, Constancy, Heterogeneity and the Categories of Qualitative Time Series," *Ecology*, vol. 61, no. 3, pp. 715–719, 1980.

[22] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid, "Using Convolution to Mine Obscure Periodic Patterns in One Pass," in *Proc. 9th International Conference on Extending Database Technology (EDBT)*. Springer, 2004, pp. 605–620.

[23] "Microformats," http://microformats.org.

Kay Römer is currently an associate professor of computer science at the Institute for Computer Engineering of University of Lübeck, Germany. After completing his studies in computer science at the University of Frankfurt/Main, Germany in 1999, he joined ETH Zurich as a research assistant, where he obtained his Ph.D. in 2005 with a thesis on wireless sensor networks. From 2005 to 2009 he led the research activities related to wireless sensor networks in the Distributed Systems Group at ETH Zurich. Kay Römer's current research interests encompass networked embedded systems, in particular wireless sensor networks, pervasive computing, and the Internet of Things.

Benedikt Ostermaier was awarded a master's degree (Diplom-Informatiker) in computer science by the Technische Universität München (TUM), Germany in 2005. His master's thesis addressed security in vehicular ad-hoc networks and was conducted in cooperation with BMW Group Research and Technology. Benedikt Ostermaier joined Prof. Mattern's research group at ETH Zurich in 2006 and is currently working towards his Ph.D. His research interests include search engines for the real world, mobile phones, and the Web of Things.

Friedemann Mattern is a Professor of Computer Science at the Institute for Pervasive Computing of ETH Zurich, Switzerland. He received a Masters Diploma from the University of Bonn, and a Ph.D. from the University of Kaiserslautern, Germany. Before joining ETH Zurich in 1999, he was a faculty member of Saarland University in Saarbrucken and of TU Darmstadt, Germany. Prof. Mattern is a member of the editorial board of several scientific journals and book series, has initiated and chaired a number of international conferences, published more than 150 research articles and books, and is involved in various research projects, typically in cooperation with industrial partners. He is a member of the National Academy of Sciences Leopoldina and of acatech, the German Academy of Science and Engineering. His main research interests are distributed systems, ubiquitous computing, and the Internet of Things.

Michael Fahrmair was awarded Dipl.-Ing. (M.S.) and Dr. (Ph.D.) degrees in computer science by the Technische Universität München (TUM), Germany, in 1999 and 2005 respectively. He joined DOCOMO Communications Laboratories Europe in 2006 to work in the Ubiquitous Networking group. He is currently working as a Senior Researcher in both the Smart and Secure Services and the Research Planning and Promotion group. His main interests are ubiquitous mobile service platforms and Rich Communication Suite (RCS).

Wolfgang Kellerer (Member, IEEE) was awarded Dipl.-Ing. (M.S.E.E.) and Dr.-Ing. (Ph.D.) degrees by the Technische Universität München (TUM), Germany, in 1995 and 2002 respectively. He is currently Director and Head of the network research department at NTT DOCOMO's European research laboratories in Munich, Germany. His research interests include mobile networking, QoE-based resource management, service platforms, peer-to-peer and sensor networks. Before joining DOCOMO Euro-Labs, he was a member of the research and teaching staff at the Institute of Communication Networks at TUM, Germany. In 2001 he was a visiting researcher at the Information Systems Laboratory of Stanford University, California, US. For his outstanding scientific publications he received the German ITG Award in 2008.