Designing Highly Available Repositories for Heterogeneous Sensor Data in Open Home Automation Systems*

Roberto Baldoni, Adriano Cerocchi, Giorgia Lodi,
Luca Montanari, Leonardo Querzoni

Dipartimento di Informatica e Sistemistica "A. Ruberti"
Sapienza Università di Roma - Rome, Italy
{baldoni|cerocchi|lodi|montanari|querzoni}@dis.uniroma1.it

Abstract. Smart home applications are currently implemented by vendor-specific systems managing mainly a few number of homogeneous sensors and actuators. However, the sharp increase of the number of intelligent devices in a house and the foreseen explosion of the smart home application market will change completely this vendor centric scenario towards open, expandable systems made up of a large number of cheap heterogeneous devices. As a matter of fact, new smart home solutions have to be able to takle with scalability, dynamicity and heterogeneity requirements. In this paper we present the architecture of a basic building block, namely a distributed repository service, for smart home systems. The repository stores data from heterogeneous devices deployed in the house that can be then retrieved by context aware applications implementing some home automation functionalities. Our architecture, based on a DHT, offers a completely decentralized and reliable storage service able to offer complex query functionalities.

1 Introduction

Thanks to recent progresses in the area of wired and wireless networking, sensor networks, networked appliances and embedded computing, all the enabling technologies needed to develop the vision of smart automation in home environments seems to be available. Despite this fact, currently available smart home applications are mainly represented by complex prototypes that still face a long way before reaching the status of commercial products.

Existing applications are developed primarily with proprietary technology and seem to lack a long-term vision of evolution and interoperation. The future market for smart home applications will comprise a wide variety of devices and services from different manufacturers and developers. We must therefore achieve platform and vendor independence as well as architecture openness before smart homes become common places.

Future open smart home applications will be ready for the market only if they will meet the following requirements:

* The work described in this paper was partially supported by the EU Projects SM4All, SOFIA and eDIANA.

Scalability - current applications are limited to a small number of devices, but an open architecture would open the market to many different vendors offering a wider selection of devices, thus raising the current limit to hundreds or even thousands of devices per home;
Dynamics - while current applications are mainly based on cabled devices installed by experts, we envisage a future where new devices can be added to existing environments in a plug-and-play fashion and where wearable devices can follow users and join the home environment only when the user enters it. In this scenario, future applications must be able to tolerate or even leverage the dynamic environments where they will be required to run;
Heterogeneity - a large base of available devices offered by different vendors will clearly increase the heterogeneity of the environments causing interoperability issues and requiring new approaches for resource sharing and scheduling;
Reliability - as the users will start to put confidence in smart home applications, the reliability aspects of these applications will gain more importance, requiring the definition of new techniques to guarantee their correct behaviour.

In this scenario, a fundamental building block is represented by a repository where devices (e.g., sensors, actuators etc) store data that can be retrieved either by the devices themselves or by some context-aware application for further processing. This paper describes the design of a scalable and reliable repository well suited to smart home applications. To this aim, the repository is implemented in a fully distributed fashion. Processes constituting the repository can be deployed on various devices located in the house offering sufficient computational and storage resources. TVs, smart phones, PCs, refrigerators etc can be example of such devices. Processes cooperate in a peer-to-peer fashion to implement storage and query functionalities in a dynamic, scalable and reliable way. In order to fit the heterogeneity of data that could be stored in the repository these functionalities are realized through a mapping component able to efficiently store and organize pieces of data in order to facilitate their search and retrieval.

The rest of this paper is organized as follows: Section 2 introduces the architecture of the repository detailing its internal components and explaining how data can be stored and retrieved from it; Section 3 introduces an application scenario that shows a possible usage of the repository in a realistic setting; Section 4 describes related works in this field of research and, finally, Section 5 concludes the paper.

2 Architecture of the Repository

This section is devoted to describing the architecture of our repository; however, before delving into its details, we will briefly introduce the reader to the smart home environment where our repository will be deployed.

2.1 The smart home environment

Traditionally, smart home solutions (e.g. [1,5,6,2]) were provided by one vendor, using a single standard for communication, often choosing a closed one, and were expensive. In the future of this area we envisage a scenario where a single home will host a large (up to hundreds) number of devices coming from different vendors, with different hardware, communication interfaces, and operating software, that will interoperate and cooperate to offer complex services to house residents [9]. The cooperation among widely different technologies will be guaranteed through the use of middleware platforms [7] and interconnection standards [3,4] able to hide to software developers the complexities stemming by the unavoidable differences among communication standards.

Devices interacting within the home environments will be characterized by different capabilities and different available resources: dumb temperature/light sensors, automatic blinds, phones, light switches, home appliances, media centers, PCs, etc. Most of them will offer different kinds of services like reading the current temperature value or showing a high-def movie on a TV. Some of them, arguably the powerful ones, will be able to make part of these resources available to the system to offer storage space or computing power. These resources will be used to build and offer to house inhabitants complex services that could not be offered by devices working alone in a "digitally closed" world. In the following we will assume that all these devices are able to communicate using a common middleware infrastructure.

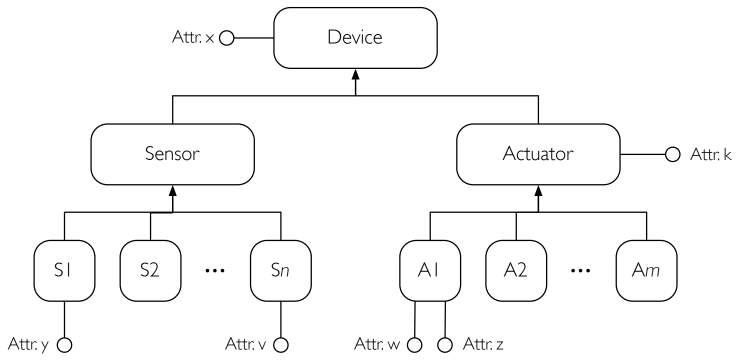

Fig. 1. General descriptive schema of the environment.

We also assume that all devices and services running in the system agree on a common schema representing the environment. This schema contains an organized description of all the devices present in the environment together with the services they offer and the state they maintain. Such shared schema could be queried and then used to automatically compose services offered by different devices. Figure 1 depicts a possible general schema for an environment hosting a number of heterogeneous sensor and actuator devices; note that the schema defines a hierarchy of elements through IS-A relationships; every element can contain a set of attributes used to describe its state (for example a temperature sensor could have an attribute "current_temperature"). The schema also describes the types and admitted values for every attribute it defines. The schema can be represented within the architecture using a markup language like OWL [8]. In this paper we assume that the schema is given and static and that is known by all devices in the system. Devices produce data represented by XML documents whose format respects the schema; we can imagine a document as a set of nested tags specifying completely the device where the data has been originated and the content of the data itself. Using a common format to describe data is fundamental to support operations among heterogeneous sources.

2.2 Architecture overview

In this section we present the architecture of our repository service. The service is provided by a distributed set of processes, all running the same software component, that cooperate to provide the required functionalities: (i) storage of data provided by devices and (ii) retrieval of data matching queries possibly issued by devices or other software components. These processes can exchange messages using the communication primitives provided by the underlying middleware; here we do not make any specific assumptions on these primitives as a simple TCP/IP-like channel would perfectly suit our needs.

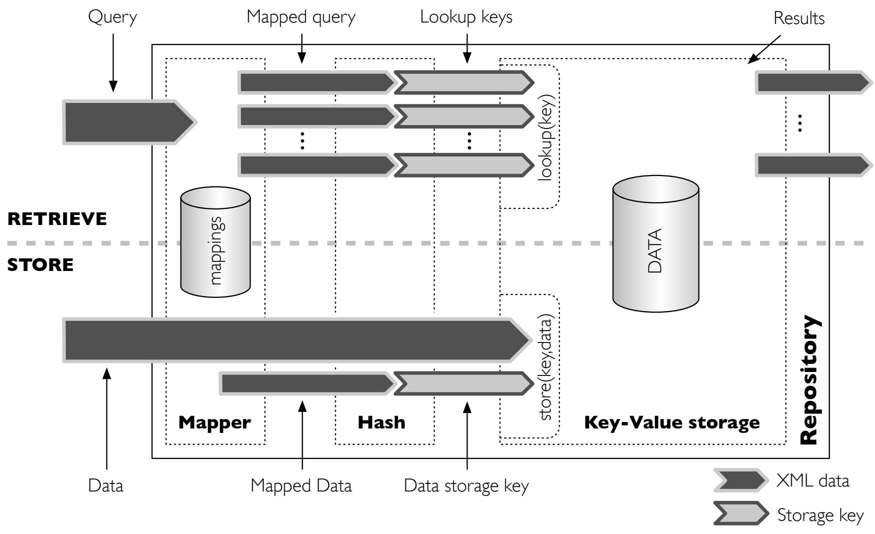

Fig. 2. Architecture of the repository. The figure also shows main operations that can be executed on it: data storage (bottom half) and retrieval (top half).

Figure 2 depicts the internal architecture of the repository service. It consists of three main components: the *Mapper*, a *hash function* and a *Key-value storage*. Arrows in the picture depict data as it flows through the repository; the upper half of the picture shows what happens when an external component queries the repository for some data, while the bottom half shows the repository actions when some data provided by a device must be stored.

Data provided to the repository is stored in a key-value storage component. This component provides a simple interface: a *store(key,data)* primitive that is used to store a document *data* with an identifier *key*, and a *lookup(key)* that returns all data associated to identifier *key*. Keys are usually represented by strings of bits. A key-value storage represents a very simple and flexible method to store large amounts of data; thanks to its simplicity it can easily be distributed to improve its scalability. However, the interface it provides limits users to *exact-match* like data searches. This severely reduces the expressiveness of queries that could be issued to the repository; in this way it would impossible to issue a query like “retrieve all data from devices that sensed a temperature higher than 21°C in the last three hours”. Such kind of complex queries are quite common in smart home scenarios because knowledge about a specific service context must often be built from scratch without the need to retrieve the complete environmental context (e.g. I don’t want to know the temperature in every room of the house if the service will be delivered only in the kitchen). To increase the flexibility of queries issued to the repository we introduce the Mapper component whose goal is to decouple interactions with the key-value storage by automatically mapping complex queries to set of keys. Mappings must be defined such that, if some data generated by a device matches a query, then both the data and query are mapped to two sets of keys with non empty intersection. This *intersection property* guarantees the correct behaviour of the data retrieval functionality. Mapping is realized by transforming original data and queries in their mapped versions and then hashing such documents to obtain keys that can be used to access the key-value storage.

2.3 The key-value storage

This functional component can be embodied by any service able to store data associated to and identifier key. Many different technologies can be used to implement this service. Given the need to run the repository on a possibly large number of devices that could be added or removed at runtime we advocate the usage of a Distributed Hash Table (DHT). DHTs implement a simple key-value storage as a completely distributed and self-organizing system of processes running on different hosts. A DHT is able to change its composition at run-time by allowing new processes to join the storage system o gracefully removing nodes that silently left it. Data stored within the DHT is automatically moved and replicated to adapt to the new system configuration and to resist to unexpected changes. Moreover, thanks to their completely decentralized functionalities, DHT are able to scale to a large number of processes and fairly balance the load among them. Past research on DHTs mainly focused on developing systems for large scale wide area network settings [12]; however, given their nice properties with respect to reliability, scalability and ability to autonomously react to changes in the system, we think DHTs could perfectly fit out smart home scenario.

2.4 The Mapper

Each query issued to the repository is an XML document with the same format used by devices to store their data, but where attributes can contain data or complex operators. These operators can be simple $>=<$ for numerical values, regular expressions for strings or a composite construct with various constraints. When a query reaches the Mapper component these complex constraints expressed on various attributes are used to derive from the original query a set of mapped queries; each of these mapped queries has a different value specified for every attribute. The translation from an attribute constraint to one or more specified values is executed using a mapping for the attribute. Mappings for the various attributes are retrieved from a local storage inside the mapping component.

Mappings must be defined for every attribute associated to every element of the schema. A mapping is built by partitioning the set of all valid values for an attribute in subsets and electing a representative value for each of them. This operation is quite intuitive if we consider a numerical attribute with values bounded by *min* and *max*; all the values in this interval can be partitioned in continuous interval and the first value of each interval can be elected as a representative. For example, the following mapping can be considered for a temperature sensor with an attribute "current_temperature" whose values are floating point numbers bounded by -10 and 60:

/Device/Sensor/Temperature/current_temperature $\Rightarrow [-10, 0, 10, 20, 30, 40, 50]$

Mappings are strictly related to the schema describing the environment, therefore we assume that they are defined at startup time and remain unchanged at runtime. Issues related to runtime updates to mappings will be considered in future work.

2.5 Data storage and retrieval

Let us now detail how data produced by devices can be stored and then retrieved from the repository. For the sake of simplicity we will first show query management. When a query is issued to the repository it is first passed to the Mapper component.

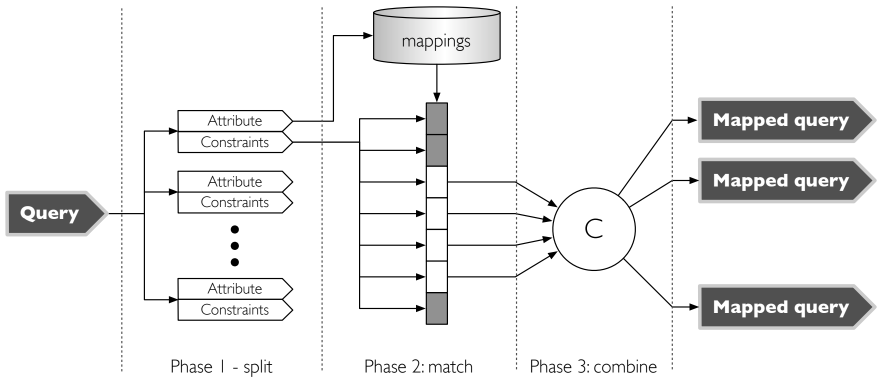

Fig. 3. Query management within the Mapper component.

Figure 3 shows how a query is managed within the Mapper component. The Mapper splits it into its components, i.e. the attributes and their constraints (phase 1). It then checks its content one attribute at a time and retrieves the corresponding mapping from the internal storage. Constraints associated to the attribute are then matched (phase 2) against the sets of values contained in the mapping: all intervals that do not match the constraints are discarded (grey intervals in the figure). When all attributes have been considered the mapping component builds the XML documents representing the mapped queries by combining all the representative values from intervals that have not been discarded in the previous phase for all the different attributes (phase 3); each mapping query contains for each attribute of the original query one of the representative values.

Mapped queries produced by the Mapper are then passed to the hash component. This component simply implements an hashing function that returns a string of bits for any incoming mapped query. These strings are the keys that must be used to invoke the lookup procedure on the key-value storage component. If the storage component is implemented as a DHT the hashing function can be the same function provided by the DHT implementation, otherwise any collision-resistant function will fit. For each key passed to the storage, a resulting document will be returned. All these data constitute the response to the query.

The management of new data submitted to devices for storage is equivalent to query management. The main difference is that data provided by devices will contain values for every attribute, i.e. no constraints are admitted in these documents. When this data is passed to the Mapper component it decomposes it into its components and then checks its content one attribute at a time and retrieve the corresponding mapping from the internal storage. When the value of an attribute is matched against the sets of values defined by the mapping one and only one set is positively matched; this is due to the fact that a mapping is a complete partitioning of the value space defined by the attribute. After all attributes have been matched, the corresponding representative values are combined to obtain the mapped data; note that, since a single representative value can be returned by the matching phase for each defined attribute, only one instance of mapped data will be created in this process.

Mapped data produced by the Mapper is then passed to the hash component to obtain the data storage key associated to the submitted data. Submitted data and the corresponding key are then used as parameters of the storage component's store primitive.

Clearly, the matching between a query and the mappings for the attributes it contains can generate false positives in the result sets, i.e. data that was mapped to one of the keys associated to the query but that does not satisfy the constraints defined in the query. The amount of false positives depends on the granularity of mappings defined for the various attributes: the more value sets are defined in the mapping for a specific attribute, the less will be the probability of false positives caused by constraints expressed on that attribute. False negatives are not possible as long as the intersection property is satisfied for all mappings.

3 Application Scenario

Our example is based on a single house, and within this house we will focus our attention only on three locations: the kitchen, the dining room and the toilet. Figure 4 shows of the left plan the positioning of actuators (represented by triangles), temperature sensors (represented by circles) and light sensors (represented by squares). A fourth symbol is used to denote devices (like the PC or the TV set) whose computational and storage resources are sufficient to host an instance of the repository. Obviously, the set of devices in a real scenario would easily be larger than the one presented here; we decided to limit both the number and the different types of devices to improve the readability of this example.

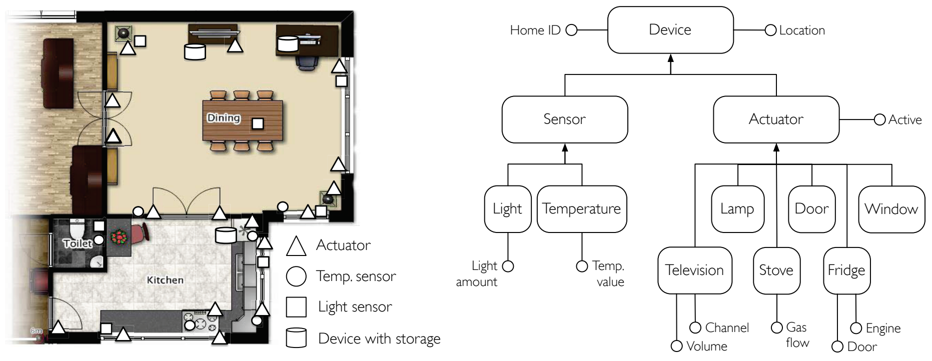

Fig. 4. Example of a smart home and its corresponding schema of the environment.

We consider two types of devices that can produce data: sensors (light and temperature) and actuators (that open and close windows and doors, turn on and off lights, the refrigerator, the tv and the heating system).

Starting from these considerations, a schema of the environment can be produced (right half of Figure 4). The schema describes any element in the environment as a subtype of the *Device* type. Different types are characterized by different attributes. Some attributes, like *Temp_value* or *Gas_flow* for the temperature sensor and Stove type respectively, are bounded numerical values; other attributes, like *Door* for the Fridge type are simple boolean values; finally, attributes like *Channel* for the Television type are enumerated text values. Regardless of the specificities of the considered scenario, each attribute in the schema is supposed to be detailed with its value type and possible bounds or valid value sets.

Mappings definition - Given the schema of figure 4 a mapping for each attribute must be defined in the Mapper component. Let consider the attributes *Temp_Value* and *Light_Amount* for the two sensor types. Assume that temperature values are bounded by 10 and 40 degrees Celsius and grouped in intervals 5 degrees wide. The Mapper internal storage unit will contain an entry like the following one[1]:

$$\text{/Device/Sensor/Temperature/Temp\_value} \Rightarrow [10, 15, 20, 25, 30, 35]$$

[1] In this example we use a compact representation of intervals for numerical attributes in the mapping. This compact representation is subject to variations depending on the specific attribute type

With respect to the *Light_Amount* attribute, we can imagine that we are interested only in four value intervals: dark, soft light, normal light and strong light. A light sensor returns values expressed in lumen, thus we have to discretize the great set of possible light amounts in four coarse intervals. Therefore, the Mapper will contain an entry like the following one:

$$\text{/Device/Sensor/Light/Light\_amount} \Rightarrow [0, 1000, 5000, 20000]$$

An enumeration attribute like *Location* has a predefined set of acceptable values; in our example it could have values "Kitchen", "Dining Room" and "Toilet". In this case grouping the values in sets for the mapping is useless and a 1-to-1 mapping can be considered.

Op 1: temperature data update - Each device producing data knows the data schema and this lets it produce well-formed data. An example of how data produced by a temperature sensor could be represented is shown in the listing 1.1; in this case the data is an instance of the environment schema.

Listing 1.1. Representation of data produced by a temperature sensor

```
<Device>
  <Home_ID datatype="integer">1</Home_ID>
  <Location datatype="enumeration">Kitchen</Location>
  <Sensor>
    <Temperature>
      <Temp_value datatype="float">26.5</Temp_value>
    </Temperature>
  </Sensor>
</Device>
```

This listing represents data produced by a temperature sensor located in the kitchen of out home that is identified by *Home_ID* 1. When this data is passed to the Mapper component it considers the lists of attributes it contains and retrieves the corresponding mappings from the internal storage. In this case the mappings are:

$$\begin{aligned}\text{/Device/Home\_ID} &\Rightarrow [1]\\ \text{/Device/Location} &\Rightarrow [\text{“Kitchen”, “Dining Room”, “Toilet”}]\\ \text{/Device/Sensor/Temperature/Temp\_value} &\Rightarrow [10, 15, 20, 25, 30, 35]\end{aligned}$$

The *Home_ID* and *Location* attributes are thus mapped to their corresponding values. The *Temp_value* attribute is mapped to the representative value 25 as the real value 26.5 is included in the range $25 - 30$. Now that the mapper knows the values for the mapped attributes it can build the mapped data:

Listing 1.2. Mapped data

```
<Device>
  <Home_ID datatype="integer">1</Home_ID>
  <Location datatype="enumeration">Kitchen</Location>
```

```
    <Sensor>
      <Temperature>
        <Temp_value  datatype="float">25</Temp_value>
      </Temperature>
    </Sensor>
  </Device>
```

The mapped data is then used to calculate through the hash function the key representing the sensor data in the key-value storage component. Note how all data generated by temperature sensors located in the kitchen of house 1 and whose sensed value is in the range $25 - 30$ are stored with the same key.

Op 2: querying temperature data - Suppose now that an external software component needs to interrogate the repository and obtain data from all temperature sensors in the house whose last reading reported a temperature value greater than 25.5°C. This kind of query could be useful for example to control the heating subsystem in order to maintain a constant temperature in the house.

The query looks like the following:

Listing 1.3. A query submitted to the repository

```
<Device>
  <Home_ID  datatype="integer">1</Home_ID>
  <Location  datatype="enumeration">*</Location>
  <Sensor>
    <Temperature>
      <Temp_value  datatype="float">>25.5</Temp_value>
    </Temperature>
  </Sensor>
</Device>
```

Note how values for some attributes have been substituted by constraints; a star "*" wildcard is used to match any possible value of the attribute meaning that the query is not interested in restricting the search to a specific location, while the constraint ">25" limits the range of temperatures that are returned.

The mapper receives the query and splits it into its components. It retrieves the mappings for all three attributes and starts to match the contraints against the intervals. All intervals in the mapping associated to attribute *Location* is matched given the "*" constraint. The *Home_ID* attribute defined with value 1 leads to a 1-to-1 mapping. Finally, the constraint ">25.5" defined for attribute *Temp_value* is matched against the intervals contained in the corresponding mapping; a match is positive if there is a non empty intersection between the set of values defined by the contraint and the set of values contained in one of the intervals defined by the mapping; the matching operation thus returns values 25, 30 and 35.

All these matched values are combined in all possible ways to obtain the mapped queries. One of the 9 mapped queries generated by the Mapper component is the one presented as listing 1.2. This mapped query is indistinguishable from the mapped data we showed in the previous section. This means that this query generates at least one key that corresponds to the key previously generated to store the sensor data. This is correct, indeed, as the sensor data perfectly matches the query. Note that a a slightly different query, e.g. a query requiring all data from temperature sensors exposing a value greater than 29°C would have returned the same mapped query. In this case the previously stored sensor data would constitute a false positive returned in the response due to the lack of precision in the attribute mapping.

4 Related Work

Even if the possibility to store and retrieve data is fundamental in all smart home applications, to the best of our knowledge issues related to the design of embedded repositories for such kind of applications have been hardly takled in the State of the Art. Probably this is due to the fact that previous works in this area often considered simple centralized approaches to the problem. To find some informations about the distributed storage problem in this kind of environment, we have to look works addressing problems related to context-awareness, that typically need to access a reliable repository.

Schmidt in [13] explains that context-aware systems are computing systems that provide relevant services and information to users based on their situational conditions. This status of the system must be stored reliably in order to be queried and explored. From this point of view the availability of a reliable distributed repository can be very useful to a context-aware systems deployed in a smart home. In our work we focused on how such reliable distributed repository could be realized.

Khungar and Riekki introduced Context Based Storage (CBS) in [11], a context aware storage system. The structure of CBS is designed to store all types of available data related to a user and provide mechanisms to access this data everywhere using devices capable to retrieve and use the information. CBS provides a simple way to store documents using the context explicitly provided by the user. It allows users to retrieve documents from a ubiquitous storage using the context related directly to the document or context related to the user that is then linked to the document through timestamping methods. The main difference between CBS and our system is that in CBS special emphasis is given to group activities and access control, since CBS is designed for an ubiquitous environment. In our system rights are not considered since we assume that within a closed home environment the set of users is well known and security is tackled when trying to access the system. Another great difference regards the way data is stored: the storage system in our architecture is completely distributed and built to provide reliable storage and access to devices deployed in the environment.

All the previous solutions assume the presence of a central database, a storage server that in same cases could be overloaded. In [14] the authors show a solution targeted at enhancing the process of data retrieval. The system is based on the idea of *prefetching* data from a central repository to improve responsiveness to user requests. The solution proposed in this paper tries to overcome these problems using a scalable distributed approach where all participating nodes are able to provide the same functionalities.

A solution similar to the one proposed in this paper has been previously adopted in the field of large scale data diffusion to implement a content-based publish/subscribe system on top of a DHT [10]. The main difference between the two approaches is in the way data is accessed: while publish/subscribe systems assume that queries (subscriptions) are generated before data (publications) is diffused, our system targets a usage pattern closer to the way classical storage systems are accessed.

5 Conclusions

This paper presented the architecture of a distributed repository for smart home application environments characterized by the presence of a large number of heterogeneous devices. The repository bases its reliability and scalability properties on an underlying DHT that is used to store and retrieve data. The limitations imposed by the DHT lookup primitive is solved by introducing a mapping component able to correctly map queries and matching data. The authors plan to start experimenting this idea through an initial prototype that will be adopted for testing purposes. Aim of these test will be to evaluate the adaptability of the proposed architecture to different applicative scenarios. A further improvement plannes as futur work will consist in modifying the system in order to automatically adapt at run-time the mappings definition in order to reduce the number of false positives returned by the repository as response to queries without adversely affecting its performance.

References

1. AMX - `http://www.amx.com/`.
2. BTicino "My Home" - `http://www.myhome-bticino.it/`.
3. KNX - `http://www.knx.org/`.
4. LonWorks - `http://www.echelon.com/`.
5. Lutron Electronics Co., Inc. - `http://www.lutron.com/`.
6. Philips Dynalite - `http://www.dynalite-online.com/`.
7. The UPnP forum - `http://www.upnp.org/`.
8. Web Ontology Language (OWL) - `http://www.w3.org/2004/OWL/`.
9. Smart Homes for All: An embedded middleware platform for pervasive and immersive environments for-all. EU STREP Project: FP7-224332, 2008.
10. Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 437–446, 2005.
11. Sharat Khungar and Jukka Riekki. A context based storage system for mobile computing applications. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):64–68, 2005.
12. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.
13. A. Schmidt. *Ubiquitous Computing - Computing in Context*. PhD thesis, Ph.D dissertation, Lancaster University, 2002.
14. Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *Proceedings of the 33rd USENIX Technical Conferencee*, pages 377–390, 2008.