ROS Meets Cassandra: Data Management in Smart Environments with NoSQL

André DIETRICH[a], Siba MOHAMMAD[b], Sebastian ZUG[a] and Jörg KAISER[a]
University of Magdeburg, Germany

Abstract. Distributed and smart environments can be seen as a distributed storage for data, information, and knowledge. Thus, one of the key challenges for future smart environments is the organization, access, and querying of this distributed storage, while allowing entities to dynamically access both real-time and historical data. A relatively new approach for data management, the NoSQL (Not only SQL) databases, promises data-model flexibility, high scalability, and availability without the overhead in the fully fledged traditional Relational Database Management Systems (RDBMS)s. In our work, we exploit the previous benefits of NoSQL databases by integrating Cassandra into the Robotic Operating System (ROS). We evaluated our approach in two scenarios; within a realistic robotic exploration and with a pessimistic benchmark using randomly generated data.

Keywords. data management, smart environment, sensor data, robotics, cloud

Introduction

A large part of research for smart environments copes with interconnecting entities by developing communication middleware protocols (e. g., IO- Link[1]), and standards (e. g., `IEEE 1451` [1]). Interoperability is reached by transmitting data, whereas everything else, such as analyzing and processing of data is left out to the application. To enable autonomous systems in smart environments to cooperate, serve tasks, and solve problems, sharing real-time data is simply not sufficient enough and can be seen as the "tip of the iceberg". Whereas access to real-time data is necessary for applications in smart environment, a wide range of these applications requires access to historical data. Examples of such applications are mining of sensor logs to detect unusual patterns, analysis of historical trends, and post-mortem analysis of particular events. Thus, the archival storage of past sensor data is becoming more important; which requires the use of a more sophisticated data management.

One of the approaches for storing data in smart environments uses relational databases (RDBs). An example of such systems is the Monitoring System Toolkit (MOST) [2], which uses MySQL. Another example uses PostgreSQL in [3] to store sen-

[a]A. Dietrich, S. Zug and J. Kaiser are with the Department of Distributed Systems, at the Otto-von-Guericke-Universität Magdeburg in Germany; E-mail `{dietrich, zug, kaiser}@ivs.cs.uni-magdeburg.de`
[b]S. Mohammad is with the Database and Information Systems Group at the Otto-von-Guericke-Universität Magdeburg in Germany; E-mail: `siba.mohammad@iti.cs.uni-magdeburg.de`
[1]Enables the interconnection of intelligent sensors & actuators in automation systems: `www.io-link.com`

sor readings and in the MavHome-Project [4] for data mining. Various improvements on the RDB structure were made such as in [5]. An extended survey on RDB systems integrated into the software of control systems for industrial applications is given in [6].

However, conventional systems often fail to meet the increasing scalability, availability, and real-time requirements of applications. Therefore, other systems that leverage the problems of classical RDBs in smart environments have evolved. An example of such a system is the Universal Industrial Databases (UNIDB) [7]. It sets on the top of standard SQL servers, and is used for long-term storage of real-time data in industrial automation systems. Another example is GaianDB, a distributed federated Database (DB) on top of multiple other RDBs, that uses biologically-inspired self-organization principles to minimize management [8].

Another approach tries to leverage the centralized DB notion in total and thus leaves out data storage and processing to where it comes from, i. e., to the sensor nodes. One of the most prominent DB examples for distributed sensor networks is TinyDB [9] and COUGAR [10]. An overview on such approaches is given in [11]. However, all of these approaches are specialized on long-term measurements with sensor networks with limited power, memory, computation, and communication capabilities.

A smart environment with appearing and disappearing entities, overlapping workspaces, changing tasks, etc. can be seen as some kind of distributed mind that is continuously producing new data, information, and knowledge. Systems in such environments will have to continuously adapt to these changes, and therefore will require holistic access to this distributed mind. Thus, one of the main problems that will arise in future is to store, organize, and enable access to all that data, information, and knowledge (other problems are discussed in [12]). But at the moment, we even lack of sharing data. Data is either transmitted directly (raw or in specialized formats and protocols) or stored (within files, DBs, or knowledgebases) and abstracted (e. g., converted to specific datasets, 3D models, maps, action sequence, etc.) by every entity itself, without having the possibility to dynamically access even historical measurements or abstracted data. Furthermore, we require new methods to request and filter data, because the pure amount of data generated by new types of sensors is too large to be handled within the main memory.

For the aforementioned reasons, we suggest the combination of dynamic properties of smart environments with distributed DBs. By connecting entities within DB clusters, we allow holistic access to every entity's "mind". We also obtain the possibility for querying and filtering of historical data. By integrating Cassandra [13] into the Robot Operating System (ROS) [14], we enable a system to dynamically access both real-time and historical data. Therefore, every entity hosts a local Cassandra DB instance, which is individually filled and updated for its own purposes. But, it can also be queried from foreign entities within the cluster and vice-versa.

The rest of this paper is organized as follows. Within the next section, we will give a brief overview on related work. After that, in Section 2, we explain in details our system approach followed by Section 3 which illustrates the evaluation in two scenarios. We finish the paper with insights into possible future work directions.

1. Background and Related Work

The Robot Operating System (ROS) is a framework for the development of robotic applications. It offers services that deal with hardware abstraction, device control, mes-

sage passing between processes, and package management for commonly-used functionalities. And it has to become a standard in the scientific robotics community. Entities (nodes) in a ROS publish/subscribe-network exchange real-time data via logical channels (so called topics). Messages are therefore standardized and described by a language-neutral interface definition language (cf. Lis. 2).

Selecting an appropriate data management system is a tough challenge. The authors of [15] examined what might be the most appropriate DB to store sensor data. They argued that it is more advantageous to use NoSQL DBs than to use classical RDBs. Three DB systems were therefore selected for further investigation: PostgreSQL, Cassandra, and MongoDB. Their evaluation revealed that Cassandra is the best choice, when dealing with large and critical sensor applications or when scalability is important. MongoDB performs best for small or medium sized non-critical sensor application and when write performance is relevant. In [16,17], NoSQL DBs (especially Cassandra) were identified as ideal memory management systems for future robotic applications. They can handle terabytes of data and their timestamp mechanism allows querying and retrieving current data without additional efforts. Sensor data and all other messages in ROS are strictly typed, so that extensible record stores are the best choice. Therefore, and because of the other advantages, such as horizontal and vertical partitioning, scalability, and fast access, we decided to use Cassandra for our approach. Cassandra also offers interesting features, such as extended querying using CQL[18], its data model supports super columns for columns nesting, data can be marked with a time-to-live (and thus also be forgotten after a certain time), and it is under continuous development.

To the best of our knowledge, there is currently no standard approach or project that combines distributed DBs with smart environments or robotic applications, except `warehousewg`[2]. `warehousewg` is a ROS package that uses MongoDB to provide a persistent DB layer for ROS. It allows to store ROS messages in binary format and to associate each message with metadata that can be indexed and queried. However, the binary stored messages cannot be directly queried. Querying messages is only enabled by manually attaching metadata, which inflates the storage space consumption, requires additional efforts and foreknowledge, such as what attributes are relevant for the application, what might be interesting in the future, and what is the format of a message, etc.

2. A System for Data Management in Smart Environments

In this part, we describe the architecture of our approach. The system is segregated into two major parts. The first one abstracts the DB management, whereas the other enables data access in a ROS typical manner. A simplified draft of the system-architecture and its classes is given in Figure 1. The whole library as well as the evaluation (see Section 3) were programmed in `Python`[3]. The project is freely available under the GNU public license and free for download on `http://ros.org/wiki/cassandra_ros`.

The class `Cassandra` uses the pycassa32-library to implement standard functionalities, such as connecting to a cluster, maintaining connection pools, or creating and deleting column families, etc. `RosCassandra` is the topic management system, abstracting and hiding all CassandraDB related stuff with an interface designed to be as close as

[2] `http://ros.org/wiki/warehousewg`

[3] Python Programming Language: `www.python.org`

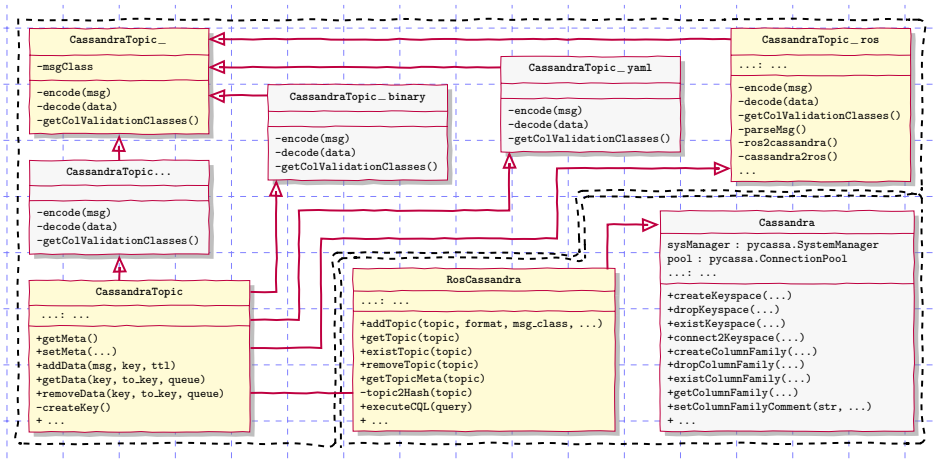


Figure 1. Simplified UML class diagram of the current implementation divided into topic handling (left) and DB management (right) classes

possible to the common ROS ideology. Thus, messages are stored and queried in their typical manner, without bothering about message conversion or DB related issues; an example is given in Lis. 1. Column families are applied as (topic-) containers, storing one ROS-message per row. Since column family names are limited to a maximum size of only 48 B, whereas topic names easily exceed this limit, hashes of topic names are applied as column family names. The real topic names as well as other metadata, such as the ROS message format, type of primary key (e. g., a timestamp, a hash value, or a part of the message itself), the storage format (binary, yaml, json, string, encoded ROS, etc.), are stored during creation time of a `CassandraTopic` within the comment field of every column family.

Listing 1: Minimal source-code example

```
rosCas = RosCassandra(host, port)          # connect to Cassandra
rosCas.connectToKeyspace(keyspace)
rosCas.addTopic('topic', format='ros')       # create a new topic
casTopic = rosCas.getTopic('topic')
casTopic.addData(msg)                      # add and get a message
msg = casTopic.getData(key)
```

Storing metadata within the comment field has several advantages compared to the usage of a separate column family. To begin with, it does not affect any column family structure and is easy to extend and interpret. It also does not require any further replication or synchronization. Furthermore, this kind of information storage is used for topic identification, which allows topics to exist in parallel with other column families within the same keyspace. Thus, requesting for available topics or other metadata is done via retrieving and parsing all column family comment fields.

2.1. Translation of Messages

As mentioned before, our system allows storing in various formats, such as string, json, ros, or binary. As depicted in Figure 1, the parents of `CassandraTopic` de-

fine the storage format, in other words, how messages are translated into a format that is storable in CassandraDB. A class therefore just has to implement the methods `encode(msg)`, `decode(data)`, and `getColValidationClasses()` from the abstract class `CassandraTopic_`. The last method is only called during the creation of new `CassandraTopic` (and thus, a new column family) and is used to define the format of every column. In some cases this is done statically, if the whole message is stored as once by using just one column, like in `binary`- (`BYTES_TYPE`) or in `string`-format (`UTF8_TYPE`). Column validation classes are required for building secondary indexes on columns. The usage of such single column formats enables fast read/write access but it also diminishes possibilities for filtering and querying.

Class `CassandraTopic_ros` for example enables to fully exploit CassandraDB's querying methods on secondary indexes (by using CQL, `pycassa.index`, etc.). Therefore, the definition formats of ROS-messages (cf. Lis. 2), are parsed and translated from a nested message structure into a list of columns. Furthermore, primitive ROS types[4], are translated into CassandraDB data types[5], to define column validation classes. Lis. 3 shows the resulting translation for the message definition of `geometry_msgs/TransformStamped`[6]. The fact, that the originally tree structure is transferred into column names, allows to query or to filter messages with a similar syntax as it is used to access message objects from a programming language like `C++` or `Python`. We demonstrate this in detail in Section 3.1.1.

Listing 2: Example of a ROS nested message definition for type `geometry_msgs/TransformStamped`

```
std_msgs/Header header uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/Transform transform
  geometry_msgs/Vector3 translation
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion rotation
    ...
```

Listing 3: ROS encoded version of `geometry_msgs/TransformStamped` into CassandraDB columns

```
header.seq: INT_TYPE
header.stamp: DATE_TYPE
header.frame_id: UTF8_TYPE
child_frame_id: UTF8_TYPE
transform.translation.x: DOUBLE_TYPE
transform.translation.y: DOUBLE_TYPE
transform.translation.z: DOUBLE_TYPE
transform.rotation.x: DOUBLE_TYPE
transform.rotation.y: DOUBLE_TYPE
transform.rotation.z: DOUBLE_TYPE
transform.rotation.w: DOUBLE_TYPE
```

2.2. Accessing and Querying

We support two different flavors for this. On the one hand, data can be directly accessed with the help of `CassandraTopic` and its methods, such as `addData`, `getData`, etc. (cf. Figure 1). Messages of one topic are automatically converted into different Cassandra formats and vice versa. On the other hand, querying over multiple topics (column families) requires more dedicated methods than just key value requests. In class `RosCassandra`, we offer method `exequteCQL` that gets a string (CQL statement) as input, translates topic names into their column family names (hashes), and executes the query using Cassandra's Python driver for CQL. The returned result is an array of values. An example of both query methods is given in Section 3.1.1.

[4]List of primitive ROS types: `www.ros.org/wiki/msg`

[5]List of CassandraDB's data types: `www.datastax.com/docs/0.8/ddl/column_family`

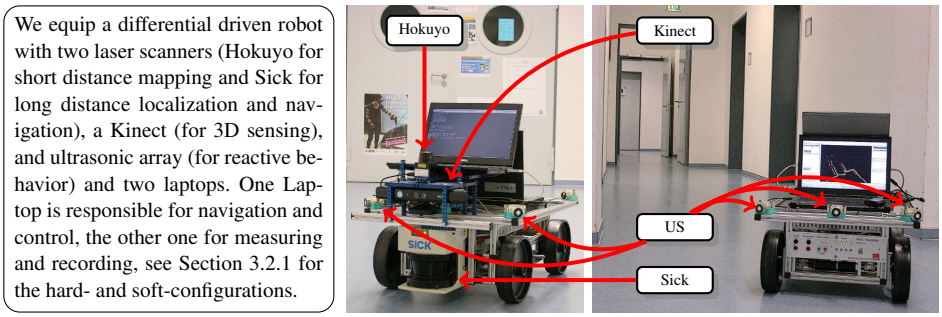
[6]Commonly used for maintaing relationship between different coordinate frames (cf. Section 3.1.1).

3. Evaluation

The evaluation is divided into two parts. In the first, we apply our approach in a realistic scenario; with real robots and sensors. In the second part, we perform a pessimistic comparison of different system for data storage and querying, using artificial sensor data.

3.1. Scenario

Inspired by our previous evaluation in [19], we chose a simple and common robot exploration scenario as described in Figure 2. A mobile robot is instructed to survey a certain area. The collected data of this trail (ca. 18 GB) and the information are extracted and analyzed with the help of our system. To visualize this part of the evaluation and the achieved results, we created uploaded some videos to our YouTube-channel at `www.youtube.com/ivsmagdeburg`, screenshots are depicted in Figure 3.



(a) Description (b) Front with Sensors (c) Back with Corridor

Figure 2. Setup of the robotic platform that was used for exploration task

3.1.1. Storyboard

This should serve as an example of several applications within distributed smart environments and how to query them. A cube (with markers) is randomly placed within our setup in which another application is interested. It could therefore connect to all camera streams and analyze them. A topic containing a camera stream can be easily identified by analyzing the metadata of each column family (cf. Lis. 1):

```
for topic in rosCas.getAllTopics():
    if topic.getMeta['msg_class'] in ['Image', 'CompressedImage']:
        ...
```

The cube in our example was detected within one of the analyzed video streams, gained from the exploration robot, as depicted in Figure 3a. There is a point in time (`t1`) when the cube was firstly detected within the stream and another point in time (`t2`) when it was last seen. These values are gathered from the primary keys of the stored camera images. As mentioned before, adding additional metadata to a stored topic is not a big deal. Every column family is tagged with the `nodeID` of the producing entity, to ease a nodes identification, which can be queried as follows:

```
nodeID = topicCamera.getMeta['nodeID']
```

Now that the nodeID (Who) and the time frame (t1, t2) (When) are known, it is possible to use simple CQL statements to clarify where these pictures were recorded. The position of an object relative to another object or to a coordinate origin is published via tf[7], which are stored in queryable ros format (cf. Lis. 3). The operational area of the robot, at a certain point in time, can now be deduced with the following statements:

```
stmt = 'select ... translation.x from tf where ... child_frame_id=' \
       + nodeID + ' and KEY > ' + str(t1-5) + ...
x_list = rosCas.exequteCQL(stmt)
x_min = min(x_list)[0]; x_max = max(x_list)[0]
```

The upper example shows a simplified version, where all x values are queries that belong to certain node and that were stored between t1 minus 5 s and t2 plus 5 s (the additional seconds are simply used to enlarge the horizon of the map, which will be generated for this area with the help of the other sensor measurements). The executeCQL method of class RosCassandra was used, because plain topic (column family) names like tf (transformation) have to be substituted by their hash values, as mentioned earlier. The result is an array that can easily be filtered to determine the min and max values of the x and y coordinates.

In the same way, it is now also possible to query tf about other systems/nodes that were located in the same area or to retrieve a map (see Figure 3c). Of course, we are aware of the sensor systems, which were mounted on the robot, but it is also possible to determine relevant sensor systems by querying tf, and to search for nodes whose parent (frame_id) is equal to the exploration robot's nodeID. Replaying these sensor readings is quite simple, as listed below and depicted in Figure 3b. It should be remarked that the results of method getData are always returned with the correct ROS format, so that they can be used and interpreted immediately:

```
hokuyo=rosCas.getTopic('/hokuyo_scan').getData(key=t1-5,to_key=t2+5)
kinect=rosCas.getTopic('/depth/points').getData(key=t1-5,to_key=t2+5)
type(hokuyo[0])  # sensor_msgs/LaserScan
type(kinect[0])  # sensor_msgs/PointCloud2
```

Now think of another robot that reaches the mentioned environment and has to grab the cube. It is able to access the existing measurements and generate an appropriate map of the environment to fulfill individual tasks. In our example, the robot generates an occupancy grid, based on the laser scans. Afterwards, it filters the Kinect 3D measurements, related to possible obstacles in the environment which are not detectable by the laser scanner close to the ground. The filter criteria depend on the height of the robot. This Information (contour of chairs and desks) is included in the basic map. Figure 3d illustrates this process that results in a robot specific representation of the environment.

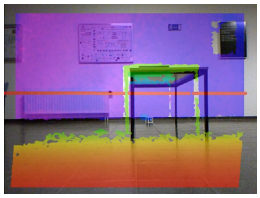
3.2. Benchmark

Within the following subsection, we compare our solution against two other solutions for sensor data storage that are commonly used in ROS-applications (rosbag[8] and mongo_ros). In our evaluation, we consider a complex message structure to test the ef-

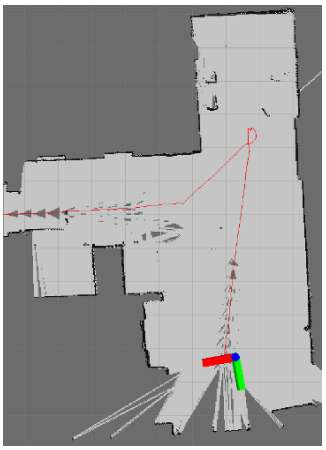
[7] ROS transformation package (cf. Lis. 2): ros.org/wiki/tf

[8] A library that enables to store and access serialized messages in a file: www.ros.org/wiki/rosbag

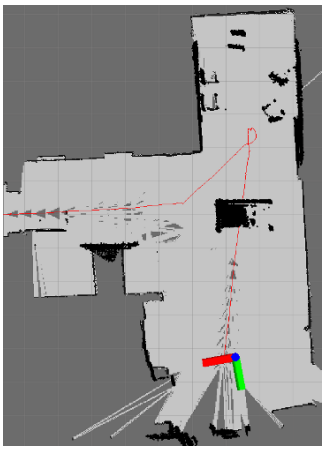
(a) Detected cube[9]



(b) Overlayed sensor data[10]



(c) Map based on laser scans only[11]



(d) Map with selected Kinect data[12]

Figure 3. (a) and (b) depict a basic replay of ROS sensor data, stored within Cassandra, (c) and (d) show occupancy grid maps of the the same operational area; note the differences in the center of each map, caused by a table in the scene, visible only to Kinect

fect of different data types on the query performance. We therefore applied a specific ROS message, containing 5 different data types (i. e., 8 bit, 16 bit, 32 bit, 64 bit, and random arbitrary size (ranging from 1 B to 200 B)). Furthermore, we examine the memory usage and storing performance in relation to the number of messages. We used the term "pessimistic" because all values were generated randomly, making them difficult to index.

3.2.1. Configuration

All tests run on the same PCs with the configurations listed in Table 1. Every utilized software system was executed with its standard-settings. Only Python and Python-APIs were used to exclude the possible impact of different programming languages on the performance tests. The use of compression may lead to some performance improvements. However, taking into consideration the random nature of data, this improvement will be too small and can be neglected. Consequently, we did not apply compression of messages in any of the systems for our experiment.

3.2.2. Writing (Effect on Size & Speed)

In this first test, we examine the storage and time consumptions related to the number of messages. Fluctuations in the size of a message are only caused by the random sized element. However, these random length values are equal for all DBs, due to the usage of a certain seed-value. The consumed storage space for the tested systems with and without indexing is depicted in Figure 4a.

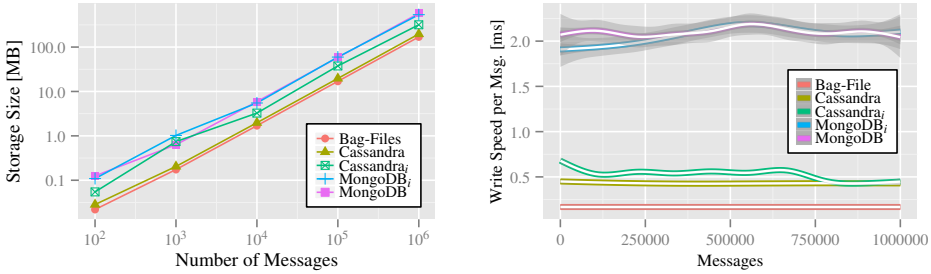
All systems show a "nearly" linear behavior in their usage of storage. It is not surprising that `rosbag` shows the best results. Two things are remarkable in this evalua-

[9] `www.youtube.com/watch?v=lQczBtVmomc`
[10] `www.youtube.com/watch?v=tfczj1jb3B4`
[11] `www.youtube.com/watch?v=czLQ-yxBYC4`
[12] `www.youtube.com/watch?v=y6LqLNB4VDk`

Table 1. Soft- & Hard-Configuration.

Java-Version:			1.7.0_07		Kubuntu		
OpenJDK Runtime Environment:			IcedTea7 2.3.2		Version:		12.04 (precise)
OpenJDK Server VM:			build 23.2-b09, mixed mode		Kernel:		3.2.0-32-generic-pae
Python:	2.7.3	Cassandra:	1.1.3		ROS:		Fuerte
pycassa:	1.6.0	CQL:	1.0.10		MongoDB:		2.0.4
CPU:		Intel® Core™ i5-540M		Disk:		Samsung SSD PM800 2.5	
speed:	2.53 GHz	cores:	2	size:	256 GB	write:	185 MB/s
cache:	3072 kB	threads:	4	cache:	128 MB	read:	220 MB/s
Memory size:		3.74 GB					

tion. First, our ROS-Cassandra implementation consumes just a little bit more storage (without secondary indexes) than `rosbag` does. Even with the usage of indexes, Cassandra requires less storage than MongoDB. This fact can be easily explained by the ROS-MongoDB implementation. To be able to store and thus to restore the original message, they have to be stored in a binary format. Furthermore, to be able to query for specific messages, meta-information has to be generated and stored. Whereas, our implementation is able to parse every message-format and thus to generate specialized columns. The Second thing to be noticed is the consumption of storage shows a similar pattern as the consumption time, as depicted in Figure 4b.



(a) Relative storage consumptions for different amounts of messages

(b) Writing time for a single message relative to the number of messages already stored

Figure 4. Overall writing performance, whereby *i* denotes the application of indexes

In both diagrams `rosbag` shows the best results, closely followed by Cassandra, but also the writing time for indexed columns in Cassandra decreases similarly as to the trend in storage usage. Furthermore, MongoDB also shows equal patterns, divergences at the beginning and then a complete overlapping. The oscillations in the measurement courses of MongoDB can be explained by its tactics of memory pre-allocation, where multiple files are used for data storage. Once a file is running out of size the next one is generated, with a larger file size. In summary, MongoDB performs bad in comparison to Cassandra and `rosbag`, while Cassandra shows for both tests results close to `rosbag`.

3.2.3. Reading (Keys & Data Types & Complex)

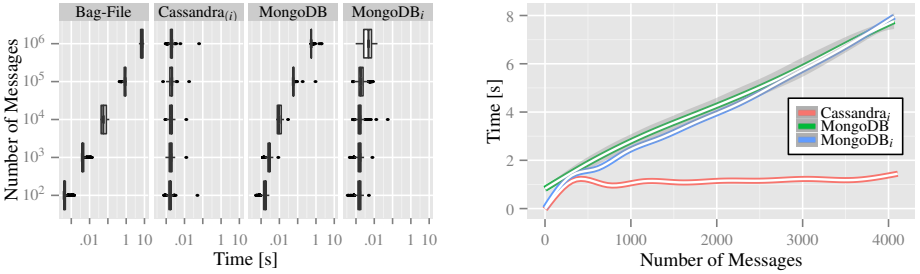
This benchmark is divided into two parts. In the first part, we investigate how fast key-value request performs (response time) in relation to different DB sizes. In the second part, we investigate how good every system performs for complex requests (filtered with random generated “WHERE-clauses”).

Response time is measured as the duration between the submission of a request and the reception of message in the correct data format. It is expected that `rosbag` and MongoDB perform better, because messages are stored by them in a binary format and do not have to be reassembled, as it is done by our Cassandra implementation. Because a timestamp is added to every message by each system, this timestamp was chosen as a primary key. The results of 1000 key-value request according to different DB size are presented as boxplots in Figure 5a; all axes with a logarithmic scale. All keys were randomly chosen. In `rosbag` they were randomly chosen from the last 10 % of stored data. The reason for this is the sequential storage and search of data within bagfiles, which would otherwise lead to uniform distributed results (concerning the response time).

As Figure 5a shows, response time of `rosbag` is directly (linear) affected by the size of stored messages. This is caused by the fact that all messages are loaded into the main memory and then searched sequentially. At the same time, Cassandra shows nearly no effect where the median response time of 1.71 ms for a DB containing 10^2 messages increase to 2.03 ms for a DB containing 10^6 messages.

Since primary keys are always indexed by Cassandra, there was no difference between the results for indexed and non indexed columns. In contrast to Cassandra, MongoDB showed different response times for indexed and non indexed columns. Using indexes, MongoDB could gain a decrease of the median response time falling from 551.75 ms to 4.84 ms for a DB storing 10^6 messages. Nonetheless, this decreased time is twice as long as Cassandra's response time.

For the last evaluation, we created queries with random length and random terms. The used version of CQL supports queries composed of the operators : "=", "<" , ">", "<=", ">=", and "*and*", while at least one "=" operator has to be used. The results of this evaluation are presented in Figure 5b. In contrast to MongoDB and Cassandra, the time for querying bagfiles was not affected by the number of returned results, but instead affected by the number of stored messages within the DB, resulting in the following median durations: 10^2:0.0054 s, 10^3:0.0546 s, 10^4:0.5521 s, 10^5:5.52 s, and 10^6:56.43 s.



(a) Response time for random key-value requests

(b) Execution duration related to the # of messages

Figure 5. Overall reading performance, whereby *i* denotes the application of indexes

Figure 5b also reveals that the influence of returned messages on MongoDB, whether indexed or non-indexed, is much higher than on Cassandra. Cassandra indeed seems also to be linearly affected by the number of returned messages but the duration increases much more slowly. Returning 1000 messages took 1.55 s while returning 4000 messages required 1.72 s only.

3.3. Discussion

With the evaluation scenario, we tried to demonstrate that sensory or actuator data in a complex environment is more than just a single value labeled with a key and a timestamp, as it is traditionally treated [15]. On the one hand, sensor data can be a complex composition of many different values, where only few values might be interesting. On the other hand, the same data can be interpreted in many different ways, according to the task and the context.

By applying different (key-value, range, and select) queries combined with advanced algorithms for marker detection, localization, and map building it was possible to identify relevant data and to deduce all required information (i. e., the generation of an occupancy grid map for a certain area and height from different sensor values and types). But this was only possible due to contextual knowledge about the task, algorithms, and data types. Accessing and requesting data in future smart environments will require novel kinds of querying methods (in terms of explorative search) as well as an intelligent combination with various algorithms to interpret sensor data in different ways.

The results reveal that Cassandra seems to be an ideal DB system for sensor data in smart environments, although it was originally intended to serve for another purpose. Even with our conversion methods for messages, storing data consumes only a little more memory than it is required to save it to a file. The same is also true for the amount of time that is needed to store data. But this tiny overhead pays off, if data needs to be queried. Cassandra showed significantly better results for both single key-value requests and complex queries where the performance was nearly not affected by the size of the database, in contrast to all other solutions.

There is one current drawback, the complexity of queries is restricted by using CQL and require further filtering within the program code. Fortunately, the development of Cassandra is an ongoing process with improvements and new functionalities, so that this drawback may be remedied in the next versions.

4. Future Work

Currently our system supports only the storage of raw data, such as sensor data or control commands, but there is much more data present in distributed smart environments. Any kind of data might be necessary for an application and thus needs to be organized, accessed, and queried. To put all available information about robots, sensors, and local environments into a spatial context and thus, to keep track of local changes in smart environments, we developed a hierarchical concept in [20]. But, as pointed out by the evaluation, location is only one possible context, others are infrastructure, safety and security requirements, time, physical conditions, etc. Therefore, we need to develop new concepts for data organization, representation, and querying that are able to cope with new requirements in distributed smart environments. In our next steps we will extend our approach and integrate methods for storing and accessing any kind of abstract data like maps, 3D models, trajectories, etc. as well as different types contextual information. A first step in this direction is also the development and application of new programming paradigms[13] for 3D environments.

[13] See the project-site of our new querying language SelectScript:
`https://pythonhosted.org/SelectScript_OpenRAVE/`

Acknowledgement

This work is (partly) funded and supported by the German Ministry of Education and research within the project ViERforES-II (grant no. 01IM10002B) and by the EU FP7-ICT program under the contract number 288195 “Kernel-based ARchitecture for safetY-critical cONtrol” (KARYON).

References

[1] Song E., Lee K. Understanding IEEE 1451-Networked smart transducer interface standard-What is a smart transducer? *IEEE Instrumentation & Measurement Magazine*, 2008, 11(2), 11–17.

[2] Zach R., Glawischnig S., Hönisch M., Appel R., Mahdavi A. MOST: An open-source, vendor and technology independent toolkit for building monitoring, data preprocessing, and visualization. In: *eWork and eBusiness in Architecture, Engineering and Construction (ECPPM)*. CRC Press, 2012.

[3] Youngblood M., Cook D., Holder L. Seamlessly Engineering a Smart Environment. In: *International Conference on Systems, Man and Cybernetics (SMC)*. IEEE, 2005. 548–553.

[4] Cook D.J., Youngblood G.M., Heierman III E.O., Gopalratnam K., Rao S., Litvin A., Khawaja F. MavHome: An Agent-Based Smart Home. In: *1st IEEE International Conference on Pervasive Computing and Communications (PerCom)*, volume 3. IEEE Computer Society Press, 2003. 521–524.

[5] Zach R., Schuss M., Bräuer R., Mahdavi A. Improving building monitoring using a data preprocessing storage engine based on MySQL. In: *eWork and eBusiness in Architecture, Engineering and Construction (ECPPM)*. CRC Press, 2012.

[6] Zolotová I., Flochová J., Ocelíková E. Database Technology and Real Time Industrial Transaction Techniques In Control. *Journal of Cybernetics and Informatics*, 2005, 5, 18–23.

[7] Research C.S., Center D. Universal industrial database v1.3. [WWW] `www.industrial-database.com`. (accessed 09.10.2013).

[8] Bent G., Dantressangle P., Vyvyan D., Mowshowitz A., Mitsou V. A Dynamic Distributed Federated Database. In: *Proc. 2nd Ann. Conference International Technology Alliance (ITA)*, 2008.

[9] Madden S., Franklin M., Hellerstein J., Hong W. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 2005, 30(1), 122–173.

[10] Bonnet P., Gehrke J., Seshadri P. Towards sensor database systems. In: *Mobile Data Management*. Springer, 2001. 3–14.

[11] Bin Ahmad M., Asif M., Islam M., Aziz S. A short survey on distributed in-network query processing in wireless sensor networks. In: *1st International Conference on Networked Digital Technologies (NDT)*. IEEE, 2009. 541–543.

[12] Remy S., Blake M. Distributed Service-Oriented Robotics. *IEEE Internet Computing*, 2011, 15(2), 70–74.

[13] Lakshman A., Malik P. Cassandra - A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010, 44(2), 35–40.

[14] Quigley M., Conley K., Gerkey B., Faust J., Foote T., Leibs J., Wheeler R., Ng A. ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, volume 3, 2009.

[15] van der Veen J., van der Waaij B., Meijer R. Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual. In: *Proc. of the 5th Intl. Conference on Cloud Computing (CLOUD)*. IEEE, 2012. 431–438.

[16] Vijaykumar S., Saravanakumar S. Future Robotics Memory Management. *Advances in Digital Image Processing and Information Technology*, 2011, 315–325.

[17] Vijaykumar S., Saravanakumar S. Future Robotics Database Management System along with Cloud TPS. *International Journal on Cloud Computing: Services and Architecture (IJCCSA)*, 2011, 103–114.

[18] DataStax. CQL 3 Language Reference. [WWW] `http://www.datastax.com/docs/1.1/references/cql/index`. (accessed 09.10.2013).

[19] Zug S., Penzlin F., Dietrich A., Nguyen T.T., Albert S. Are Laser Scanners Replaceable by Kinect Sensors in Robotic Applications? In: *IEEE International Symposium on Robotic and Sensors Environments (ROSE 2012)*. Magdeburg, Germany, 2012.

[20] Dietrich A., Zug S., Mohammad S., Kaiser J. Distributed Managment and Representation of Data and Context in Robotic Applications. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014. (Accepted).