2012 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 7-12, 2012. Vilamoura, Algarve, Portugal

A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation

Tim Niemueller
Knowledge-based Systems Group
RWTH Aachen University
Aachen, Germany
niemueller@cs.rwth-aachen.de

Gerhard Lakemeyer
Knowledge-based Systems Group
RWTH Aachen University
Aachen, Germany
gerhard@cs.rwth-aachen.de

Siddhartha S. Srinivasa
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA
siddh@cs.cmu.edu

***Abstract*— During operation of robots large amounts of data are produced and processed for instance in perception, actuation, or decision making. Nowadays this data is typically volatile and disposed right after use. But this data can be valuable and useful later. Therefore we propose a database system that taps into common robot middleware to record any and all data produced at run-time. We present two examples using this data in fault analysis and performance evaluation and describe real-world experiments run on the domestic service robot HERB.**

I. Introduction

Autonomous mobile robots produce an astonishing amount of *run-time data* during their operation. Data is acquired from sensors and actuator feedback, processed to extract information, and further refined as the basis for decision making or parameter estimation. In today's robot systems, this data is typically *volatile*. It is generated, used, and disposed right away. However, some of this data might be useful later, for example to analyze faults or evaluate the robot's performance. A system is required to *store* this data as well as enable efficient and flexible *querying mechanisms*.

In particular, such a data store must have the following capabilities: (C1) ability to store any and all data produced on the robot in real-time; (C2) powerful retrieval features to query specific data; (C3) integration with typical robot middleware; (C4) no or minimal configuration; (C5) easy adaptation to evolving data structures; (C6) distributable among multiple robots and off-board machines; (C7) independence towards robot platform and software context.

Motivated by these requirements, we propose an architecture for data storage and retrieval based on MongoDB [1], a state-of-the-art document-oriented, schema-less database. MongoDB groups related value fields into documents and stores them without predefined or enforced data schemas. By giving up strong ACID properties (atomicity, consistency, isolation, and durability) prevalent in competing relational systems, MongoDB can create choices that can focus on availability and speed first and consistency second [2]. Combined with flexible data structure definitions, we can store any and all robot run-time data at very high data-rates in real-time (C1). For its ability to query depending on arbitrary combinations and conditions on fields and by supporting the MapReduce [3] paradigm, MongoDB provides excellent retrieval capabilities (C2). MongoDB documents have a striking similarity to data structures used in robot middleware allowing for a one-to-one mapping to MongoDB documents (C3). Given the middleware's capability to list all existing communication channels, by tapping those channels we can virtually eliminate the need for configuration (C4). Because no particular schema is enforced, evolving data structures, that are prevalent in robotics due to the high speed of innovation and development, can be accommodated (C5). MongoDB was designed to be highly scalable, supporting replication as well as sharding where documents only exist on a subset of all database instances (C6). We have implemented generic recording facilities for the well-known robot operating systems ROS [4] and Fawkes [5]. Run-time generated data is stored by tapping the communication middleware, listening for exchanged messages without requiring modifications to any existing component. The stored data is independent of the particular robot system (C7).

For a variety of problems such a data store can be useful. It can provide, for example, input for reinforcement learning during robot idle times, or a robust and persistent world model to store acquired information for a longer time [6]. A particular example we briefly present is an approach to use the database for *systematic manual fault analysis*. Another example we detail is the application in *performance evaluation*, especially for the case that the relevant parameters or the time range of interest are not known a-priori. Recently, a related cross-platform data store was proposed to compare simulated and real-world results in robotic surgery [7].

We make the following contributions: 1) a generic robot database, which fits naturally with robot middleware systems to record any and all data generated at run-time, providing the ground for many new applications; 2) its application in fault analysis, using a Data–Information–Knowledge hierarchy as a data flow model for robot data processing, and in performance evaluation using the MapReduce paradigm.

We are excited about this research direction. We believe that it lays the foundation for organizing the diverse, large-scale data that is produced by modern autonomous robots. In the future we imagine robots that share common databases among each other or with nodes in cloud-computing infrastructure leveraging its vast amounts of processing power and data [8] to cope with future robotic challenges.

978-1-4673-1736-8/12/S31.00 ©2012 IEEE

II. Generic Robot Database

In this section we describe the chosen database system, how it fits nicely with typical robot middleware, and how data is stored and queried.

Relevant Features of the Database MongoDB

MongoDB is a document-oriented, schema-less database which fits particularly well with robot middleware.

Document-oriented means that key-value pairs (fields) are grouped into entities called document. The keys are names used to access a certain value as well as selectors to retrieve a particular document. The values are of basic types like numbers or text, or nested documents. For example, in Figure 1 the content-related keys "frame", "child_frame", "translation", and "rotation" represent the location and orientation of a child coordinate frame with respect to another. The frame and child_frame keys reference text values holding the name of the frames, while the translation and rotation fields contain sub-documents consisting of number values representing Cartesian translation and rotation. Readers familiar with ROS might see the striking similarity to its transform message type. Indeed this very similarity is a strength of the document-oriented nature which we are going to exploit later on. Since there is a one-to-one mapping between the two it is very easy to store and retrieve communicated data. A similar argument applies to the Fawkes framework.

Documents are *schema-less*, meaning there is no a-priori declaration or enforcement of a particular structure. This is quite contrary to classic relational database management systems where semi-static schemas to which applications have to comply are enforced by the database. Documents in MongoDB tend to have structure derived from the stored data, the schema is hence implicitly and dynamically defined by the application. For instance, even though the schema for the document in Figure 1 has not been explicitly defined, other transforms will likely have the same structure. Documents of a similar or the same dynamic structure are typically grouped into *collections*. Then, the application has a first frame of reference of what to expect from documents from a particular collection. But the schema is not enforced, and this property is particularly useful for fast changing robotic applications using a database. As we often see, data structures used for component interaction and communication are more or less frequently modified. Were the schema enforced, changing the type of a value or adding or removing keys could pose a problem. In the case of a schema-less database, we can store these documents of similar (but still different) structure in the same collection. The differentiation can then be shifted towards the application, either defining a query in a way that only certain forms of the document are retrieved, or formulating appropriate checks in the processing code. MongoDB also provides *capped collections* which are collections of a fixed maximum size. When the maximum size is reached, new records replace the oldest entry. When continuously logging, such a specified maximum can prevent accidentally exceeding the available storage capacity.

```
{
  _id: "4f55e28ffa24ebb2e469d331",
  __recorded: "2011-11-11T17:17:17Z .2342",
  frame: "/rx28/tilt",
  child_frame: "/kinect/image",
  translation: { x: 0.1, y: 0.2, z: 0.3 },
  rotation: { x: 0.0, y: 0.0, z: 0.0, w: 1.0 }
}
```

Fig. 1. MongoDB document example for a coordinate transform

Indexing is a database feature particularly relevant for query performance. MongoDB maintains indexes over collections for specified combinations of document fields. If new applications create new query patterns, corresponding indexes can easily be generated as needed.

One of the original intents to create document-oriented stores was to ease *replication*. Denormalization of documents and hence creating independence between documents makes this very efficient. In our particular case of a robot database we imagine that replication could be useful for applications like a shared, distributed world model among a group of robots, more than failure safety during database downtime. Nevertheless, this feature is outside the scope of this paper, as it would also require to consider the connections between consistency, availability, and partitioning (cf. CAP conjecture [2]). For now, we focus on the single-robot case allowing us to ignore these questions.

Robot Software Context

Robot systems often employ software frameworks and middleware for inter-component and tool communication. The proposed database architecture has been implemented in two such frameworks, ROS and Fawkes.[1] The particularly interesting aspect is the communication middleware.

ROS [4] uses a messaging middleware. Topics are registered with a central broker. Each topic has a message type defined hierarchically of elementary data types such as numbers and text strings or other message types (without cyclic dependencies). Subscribers of a topic connect to all publishers communicating in a peer-to-peer fashion.

Fawkes [5] uses a hybrid blackboard-messaging middleware. Shared memory areas (interfaces) are created labeled with a type which defines their data structure. Interface types are shallow and consist of elementary types such as numbers or text strings. Data is then exchanged via read and write transactions on a central blackboard. Messaging is used for command queuing.

In the following we focus on ROS for reasons of brevity.

MongoDB Recording and Robot Middleware

Both ROS and Fawkes already provide data recording facilities. However, these tools store data in binary streams to a file, copying chunks of volatile memory to disk. This can be very useful for certain tasks like recording sensor data for later repeated playback to test processing applications, but they lack advanced querying features and platform independence which are important for many applications.

[1]Source code and documentation for ROS and Fawkes are available at `http://www.fawkesrobotics.org/projects/mongodb-log/`

```
db.behavior.find(
  {behavior: "grab", object: "bottle",
   status: "failed"}
  ).sort({__recorded: -1}).limit(1)

db.tf.find(
  {frame: "/kinect/depth",
   child_frame: "/object/bottle",
   __recorded: {$gte: start, $lte: end}})
```

Fig. 2. MongoDB example queries for the latest failed attempt to grab a bottle and coordinate transforms in a defined time range

A requirement for acceptance of an online storage system operated continuously along the robot software is *no or only minimal configuration*. We can achieve this by exploiting two properties of the middleware. First, message types can be introspected at run-time, meaning that we can iterate over the fields of the structure and their respective values. Second, both frameworks provide a method to retrieve a list of existing topics or interfaces. Hence, to start recording all data automatically, we get this list and start a recording agent for each entry. On each message for a topic or update for a blackboard interface received, the agents can iterate over the structure's fields and create a corresponding MongoDB document, which is then stored in the database.

Specifically, in the case of ROS we have created a generic tool which can record any and all messages transmitted via topics. A master instance spawns one recording node per topic. Each node connects to all publishers of the specified topic and stores all received messages in the database. Some optimizations are described in Section IV.

Data Inquiry and Retrieval

A particularly important feature is *efficient and yet flexible* data inquiry and retrieval. The binary data stores provided by ROS and Fawkes allow for a sequential advance or search through the recording only.

By using MongoDB, we gain the ability to formulate complex queries based on arbitrary document fields, even on the pure existence of such. By using indexes, these queries are very fast compared to sequential searches. In Figure 2 we show two example queries. The first retrieves the document which contains information about the last failed execution of grabbing a bottle. The second extracts all positions of a bottle which were recorded in a specified time range, for example during the time the failed behavior was executed.

MongoDB supports the MapReduce paradigm [3] which is suitable for condensing answers out of large data sets. Based on a set of input documents, determined by a regular query, the two higher order functions map and reduce are used to process the data. First, map applies a given function to each of the input documents emitting tuples of a grouping key and a transformed, extracted, or otherwise preprocessed document. Then the reduce function is applied to merge sets of emitted documents with the same grouping key. The resulting value can be input for other invocations and the process continues until a single result record emerges. The reductions are necessarily independent of each other and can therefore be easily parallelized. MongoDB supports running

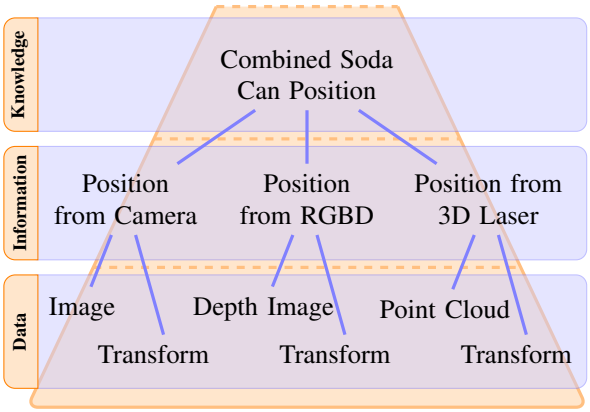


Fig. 3. Knowledge tree for soda can object position

a finalize function on the reduction result for each key, for example to compute one-step values like averages. Not only is MapReduce used to boost query performance, but it is also an intuitive way of processing data for certain applications. An example is provided in Section III (Performance Evaluation).

III. Applications in Fault Analysis and Performance Evaluation

Many applications of a generic robot database are conceivable. We now give two examples we implemented employing the database in fault analysis and performance evaluation.

Data-driven Fault Analysis

In this section we describe a novel approach to perform a systematic guided fault analysis by inspecting data recorded at run-time. We look at the frequent and time consuming subclass of robot failures caused by wrong data, for instance by an error in a software component, or by unforeseen or low-quality input from sensors or actuator feedback. The goal is to identify the component or sensor which caused the bad data. Considering the large amount of data robot systems produce nowadays, we strive for a solution to increase development and debugging efficiency by reducing the amount of irrelevant data to look at and making relevant data appear more prominent in our search.

We model the data flow using the Data-Information-Knowledge-Wisdom (DIKW) hierarchy [9], [10]. The intent is to understand the emergence of Actionable Knowledge [11], that is Knowledge the robot can use to make decisions and act upon to achieve its tasks [12]. We classify content of the robotic mind into a hierarchy. Typical robot data processing pipelines build trees in a *bottom-up* fashion. We will now explain this process along Figure 3, showing the establishment of a belief of the position of a soda can. As a start, *Data* is acquired, that is color and depth images, a point cloud, and the associated transforms with respect to a common coordinate frame. Actuators like a pan-tilt unit for the camera might also contribute. The Data is processed to extract useful *Information* combining the transforms with the positional information from the sensors, e.g. the position in the depth frame. Obstacle subtraction using voxel grids might also be on this level. The Information is further refined to

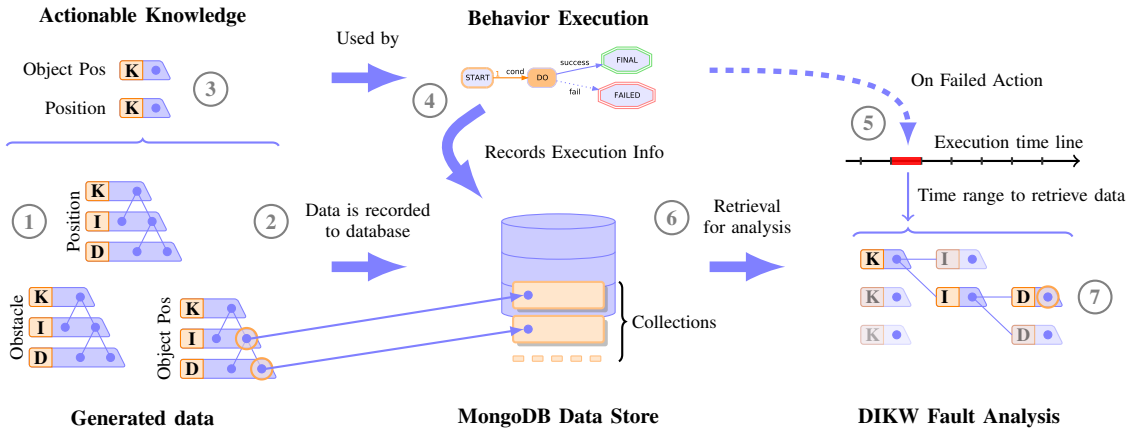


Fig. 4. Online data storage and post mortem fault analysis using the database

Knowledge by means of a weighted average of the extracted positions to gain higher accuracy. Note that this is only one particular way to gain Knowledge. Other methods might include tracking objects over time or combining Information with background knowledge, for example grasps that can be applied to an object. Ultimately the robot bases behavioral decisions on subsets of its Knowledge. For different behaviors this subset might be specific and we call this *Actionable Knowledge* which depends on the particular behavior. In the background of the figure we depict the typical DIKW pyramid. For this paper we conclude our hierarchy at the Knowledge level. In future work this might be extended, for example to consider machine learning results used by the robot as Wisdom. We also understand that for contemporary robot systems the extracted Information is often used without further refinement, therefore we do not insist on or proclaim a well-defined boundary between Information and Knowledge in all situations. With regard to the desire to primarily explain *Actionable* Knowledge, we take a pragmatic approach and allow promotion of Information to Knowledge without further processing.

Now imagine that the robot is to grab a soda can and fails, e.g. its grasp is off by a few centimeters. We assume that this is caused by an error somewhere along the data processing path. To analyze this kind of error, we employ a *top-down* guided search through the DIKW hierarchy based on the data stored in the database. The goal is to minimize the amount of data we need to look at. We classified all the data that is produced, extracted, and refined by the robot a-priori into the Data, Information, and Knowledge levels and modeled their relations. In our experiments this was a manual step, it might pose an interesting research question to find a way to do this classification automatically.

Robots need some kind of behavior execution system, for example a system like the Lua-based behavior engine [13]. It provides a reactive layer which takes commands from a higher level (deliberative) agent and instructs and monitors low-level components like locomotion. Consider Figure 4 which sketches analysis of the fault while grasping the soda can. Several sensor processing components analyze the scene and build up knowledge trees (1). All of this runtime data is continuously stored in the database (2). The relevant Actionable Knowledge for grasping the cup are the object's and the robot's positions (3). This is used by the behavior component to make decisions and instruct the lower level components. It also records additional information like behavior execution start and end times to the database (4). Now the grasping fails and we are going to analyze the cause of the problem. The recorded behavior information allows to create a timeline and query the time range of the failed behavior (5). For this particular time range we now query the recorded Knowledge from the database (6). Then we make use of the DIKW hierarchy to guide our search for the reason of the failure (7). We start with the soda can position and analyze its credibility and plausibility. To quickly verify data from the database, visualization tools could be useful, for example like ROS' rviz replaying the scene at the time of the error. We advance to the Information level and retrieve the separate positions, from which we find that the RGBD and laser range finder data agree, but the monoscopic camera deviates. Hence we descend further into the camera's Data layer. Looking at the recorded images and bounding boxes of the detected object we quickly see that they match. Consequently we investigate further Data and look at the transformation, for example using the second query from Figure 2. We see that the expectation and the observed values do not match. Finally, we find that the camera mount has shifted over time, for example because a screw got loose. A barely visible horizontal shift of the camera by only 3 ° can already cause a 5 cm positional error at 1 m distance to an object.

Even though the root cause of the problem is simple, consequences can be severe and investigation can be tedious. Incorrect data which propagates and cascades through the system and manifests in failed robot behaviors is the particular class of errors we strive to make easier to diagnose using the proposed guided approach. In our example we could prune the depth and point cloud sub-trees of the knowledge tree. We have presented a problem which is persistent. But often problems cannot be as easily reproduced. In both cases, a continuously recording database which provides flexible query abilities supports faster post mortem diagnoses.

```
map = function () {
  emit(this.behavior,
    {successes: this.success ? 1 : 0,
     fails: this.success ? 0 : 1,
     duration: this.duration,
     dursq: this.duration * this.duration });
}
reduce = function (key, values) {
  var result = { successes: 0, fails: 0,
                 duration: 0, dursq: 0 };
  values.forEach( function(value) {
    result.successes += value.successes;
    result.fails += value.fails;
    result.duration += value.duration;
    result.dursq += value.dursq;
  });
  return result;
}
finalize = function (key, value) {
  var N = (value.successes + value.fails);
  value.duration_avg = value.duration / N;
  value.duration_dev =
    Math.sqrt((value.dursq / N) -
      Math.pow(value.duration_avg, 2));
  return value;
}
```

Fig. 5. MongoDB MapReduce performance query example

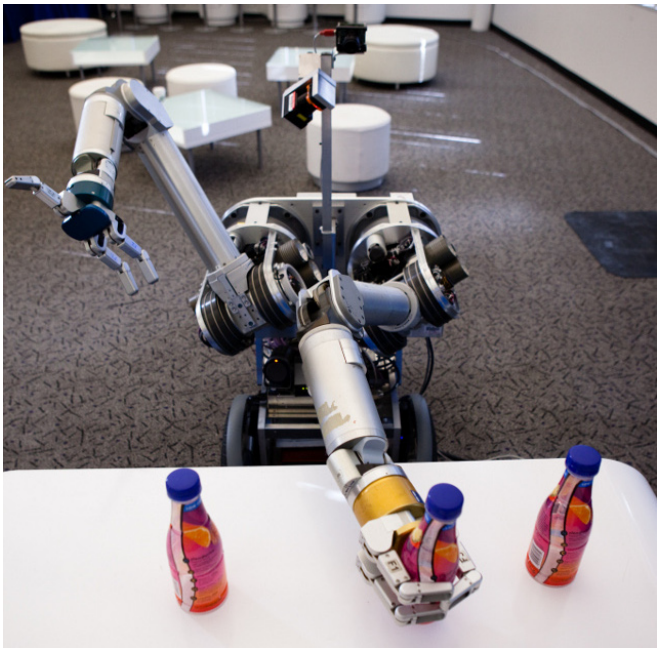


Fig. 6. HERB 2.0: A bimanual mobile manipulator developed at the Personal Robotics Lab at Carnegie Mellon University

Performance Evaluation

Performance evaluation means to analyze criteria that determine how good the robot is at accomplishing its task. As a simple example we will set the goal to analyze, for the task to grab a soda can and bring it to a human, the success rate of the last month and the average time taken.

As performance evaluation often includes aggregation of data, the MapReduce paradigm proves useful for this task. We extract information about all behaviors executed in a certain time range and count the number of successes and failures as well as the average time the behavior took and its standard deviation. In Figure 5 the corresponding MapReduce query is shown. First, map emits a count 1 or 0 for success and failure depending on the behavior's outcome, as well as duration and squared duration information (for deviation). The reduce function is then applied, using the name of the behavior as the key to group the emitted records, to compute cumulative values for each behavior.

After reduction has completed, one result record for each behavior containing the accumulated data is stored. For each such record the finalize function is called once, enriching it with average duration information and its standard deviation. The resulting records are then stored in a new collection for retrieval.

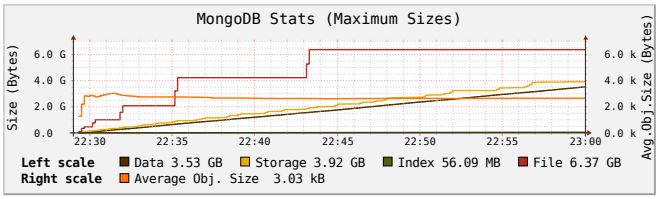
IV. Experiments and Database Evaluation

Experiments were conducted on the domestic service robot HERB [14] depicted in Figure 6, using ROS. Its sensors are an RGBD and monoscopic camera as well as a rotating 3D laser range finder mounted on a stationary mast. It features two 7 DoF arms for bimanual manipulation and a wheeled base. Three high-performance mobile workstations with hyper-threaded quad-core processors make up the primary compute capacity. Each laptop features 8 GB of RAM. For the experiments, HERB's task was to bring bottles from a table to a human user.

The generic recording facilities for ROS were written in Python to exploit message introspection which is unavailable in the C++ API. However, deserialization of the network streams into usable representations can be expensive in Python depending on the message type and size (cf. benchmark below). Hence, for message types like images or transforms which can grow quite large or are sent at a very high frequency, specialized loggers have been written in C++. In our experiments, the recording started with only minimal configuration which excluded duplicate topics, e.g. to avoid storing the uncompressed image of an already recorded compressed image. The behavior system was extended to directly record start and end times, outcome, and potential error messages to the database in one document per executed behavior for more convenient queries.

The system supports automated generation of performance graphs like the ones in Figure 7 using RRDtool [15]. The upper graph shows the almost constant growth of the database during operation. The growth rate on HERB typically is about 120 MB/min. At peak times up to 500 MB/min were recorded. Although this was caused by too high update frequencies of some components, it shows that the system can even cope with such a high throughput. The lower graph shows the number of insert operations performed during normal operation which averages at about 4300 inserts/min. To overcome current resource limitations (due to slow Python ROS message deserialization, see below) a distinguished logging machine was used. With today's high network bandwidth capabilities and transparent middleware networking support this is feasible.

Figure 8 shows the results of benchmarks performed on an Intel E6750 with 8 GB of RAM and a local hard disk.



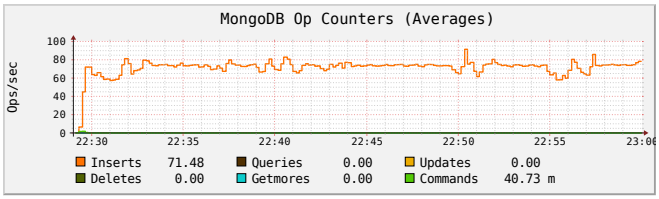
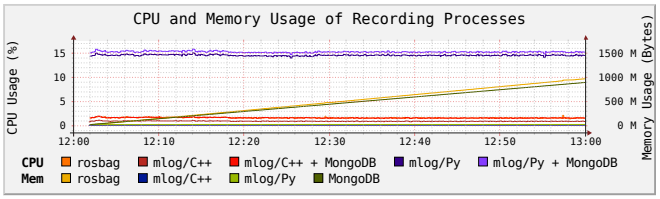


Fig. 7. RRD graphs showing database growth and operation counters



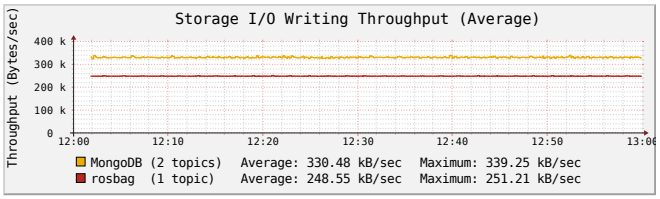


Fig. 8. Logger benchmarks showing CPU/memory usage and output rates

A single topic was recorded using ROS' native rosbag, the generic Python MongoDB logger as well as an optimized C++ MongoDB logger, all running at the same time with messages produced at 100 Hz and a size of about 480 B each. In the upper graph the rosbag CPU line (orange) is almost hidden behind the accumulated C++-Logger/MongoDB line (red), showing that both approaches cause a similarly low CPU load. The C++ logger averages at 10 MB memory usage and the Python logger at about 35 MB. The Python logger also requires an order of magnitude more CPU time, as seen by the purple lines. This is a problem of ROS message parsing in Python, which is computationally expensive. A solution could be to implement C++ ROS message introspection and a generic C++ logger. Since this already exists in Fawkes it does not suffer from this problem. The lower graph shows the throughput overhead of MongoDB of about 30 %. Note that we recorded two topics to MongoDB, but only one using rosbag. A considerable overhead is produced by rosbag storing the type definition with each message.

Error scenarios were investigated using the approach outlined in Section III. The ability to query specific data sets, especially when referencing a time range from a failed behavior, helped to decrease investigation time. However, it became apparent that future work needs to address that visualization is necessary to understand the data more quickly.

V. Conclusion

In this paper we have presented a system based on the highly scalable, document-oriented, schema-less database MongoDB which is able to record the data generated at run-time for later use and to fulfill the required capabilities stated in Section I. The system was run in real-world experiments on the domestic service robot HERB producing large amounts of data handled at typical rates of about 120 MB/min, and at peak rates of 500 MB/min. Synthetic benchmarks demonstrated the system's low overhead.

We have provided two specific application examples. For one, the recorded data is used in post mortem fault analysis. For another we described how to use the data to retrieve quantitative performance data of the robot's behavior showing the value of the MapReduce query paradigm.

Future work might go towards better visualization support for even easier fault analysis, or automatic diagnosis or DIKW classification of run-time data. It is also worthwhile to investigate cloud-computing integration for various robot tasks, for example by distributing the database.

Acknowledgments

This work was partly supported by the Quality of Life Technology Center at The Robotics Institute of the Carnegie Mellon University. T. Niemueller was partly supported by the Intel Summer Fellowship 2010 and by the German National Science Foundation (DFG) with grants GL747/9-5 and LA747/18-1. We thank the anonymous reviewers.

References

[1] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*. O'Reilly, 2010.
[2] E. Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed," *IEEE Computer*, vol. 45, no. 2, 2012.
[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. of the ACM*, vol. 51, 2008.
[4] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.
[5] T. Niemueller, A. Ferrein, D. Beck, and G. Lakemeyer, "Design Principles of the Component-Based Robot Software Framework Fawkes," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2010.
[6] M. Ullah, F. Orabona, and B. Caputo, "You live, You learn, You forget: Continuous Learning of Visual Places with a Forgetting Mechanism," in *Int. Conf. on Intelligent Robots and Systems (IROS)*, 2009.
[7] Y. Gao, M. Sedef, A. Jog, P. Peng, M. Choti, G. Hager, J. Berkley, and R. Kumar, "Towards validation of robotic surgery training assessment across training platforms," in *International Conference on Intelligent Robots and Systems (IROS)*, 2011.
[8] E. Guizzo, "Robots With Their Heads in the Clouds," *IEEE Spectrum*, vol. 48, no. 3, 2011.
[9] R. L. Ackoff, "From Data to Wisdom," *Journal of Applied Systems Analysis*, vol. 16, 1989.
[10] J. Rowley, "The wisdom hierarchy: representations of the DIKW hierarchy," *Journal of Information Science*, vol. 33, no. 2, 2007.
[11] C. Argyris, "Actionable knowledge: Design causality in the service of consequential theory," *Journal of Applied Behavioral Science*, 1996.
[12] A. Silberschatz and A. Tuzhilin, "What makes patterns interesting in knowledge discovery systems," *IEEE Trans. on Knowledge and Data Engineering*, vol. 8, no. 6, 1996.
[13] T. Niemueller, A. Ferrein, and G. Lakemeyer, "A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao," in *RoboCup Symposium 2009*, 2009.
[14] S. S. Srinivasa, D. Berenson, M. Cakmak, A. Collet, M. R. Dogar, A. D. Dragan, R. A. Knepper, T. Niemueller, K. Strabala, M. Vande Weghe, and J. Ziegler, "HERB 2.0: Lessons Learned From Developing a Mobile Manipulator for the Home," *Proceedings of the IEEE*, vol. 100, no. 8, 2012, in press.
[15] T. Oetiker, "RRDtool," http://oss.oetiker.ch/rrdtool/.