**RC25564 (WAT1512-016) December 9, 2015**
**Computer Science**

# IBM Research Report

## Proceedings of the 9th Advanced Summer School on Service Oriented Computing

**Johanna Barzen[1], Rania Khalaf[2],**
**Frank Leymann[1], Bernhard Mitschang[1], Editors**

[1]University of Stuttgart
Germany

[2]IBM Cloud
One Rogers Street
Cambridge, MA 02142-1203
USA

# The 9th Advanced Summer School on Service-Oriented Computing

June 28 – July 3
Hersonissos, Crete, Greece

The 9th Advanced Summer School on Service-Oriented Computing (SummerSOC'15) continued a successful series of summer schools that started in 2007, regularly attracting world-class experts in Service-Oriented Computing to present state-of-the-art research during a week-long program organized in four thematic tracks: Formal methods for SOC; Computing in the Clouds; Elasticity in the Cloud; and Emerging Topics. The advanced summer school is regularly attended by top researchers from academia and industry as well as by graduate students from programs with international acclaim, such as the Erasmus Mundus International Master in Service Engineering.

During the morning sessions at SummerSOC renowned researchers gave invited tutorials on subjects from the themes mentioned above. The afternoon sessions were dedicated to original research contributions in these areas: these contributions have been submitted in advance as papers that had been peer-reviewed. Accepted papers were presented during SummerSOC. These papers have been edited and integrated into the volume included in this Technical Report. Furthermore, PhD students had been invited to present the progress on their theses and discuss it during poster sessions; some of the extended abstracts of these posters are included in this Technical Report, too.

Johanna Barzen, Rania Khalaf, Frank Leymann, Bernhard Mitschang
Editors

## Content

# Incremental Data Transformations on Wide-Column Stores with NotaQL

Johannes Schildgen and Stefan Deßloch

University of Kaiserslautern, Germany
`{schildgen,dessloch}@cs.uni-kl.de`

**Abstract.** Wide-column stores differ from relational databases in terms of their data model, schema flexibility and query method. NotaQL is a transformation language for wide-column stores that is powerful and easy to use. Complex transformations are expressible in only two or three short lines of code, and can be executed in an incremental fashion. In this paper, we present a NotaQL transformation platform that makes each table transformation self-maintainable. This means, a transformation can reuse its former result and only needs to analyze the delta since the last computation.

**Keywords:** NoSQL, transformation, language, wide-column stores, incremental, cloud middleware, analytics

## 1 Motivation

| ROW_ID | information | | | children | |
|---|---|---|---|---|---|
| | born | cmpny | salary | Susi | John |
| Peter | 1967 | IBM | €50000 | €5 | €10 |
| | born | cmpny | salary | Susi | John |
| Kate | 1968 | IBM | €60000 | €20 | €0 |
| | born | school | | | |
| Susi | 1989 | Eton | | | |
| | born | school | | | |
| John | 1991 | Eton | | | |

**Fig. 1.** Person table with a children graph and amounts of pocket money

The idea of storing data records in a column-wise instead of row-wise fashion is older than thirty years [8]. It enables a good compression and fast aggregation. Google BigTable [7], HBase [3] and Cassandra [2]—which are the most prominent examples for wide-column stores in the NoSQL era—differ from classical column-oriented databases. They are distributed systems and add new concepts like column families. When we take a look at HBase, a table is only defined by a table name and its column families. The columns are not known at table-creation time, so every *Put* operation can set arbitrary columns. This enables new possibilities that are not possible in relational database systems, for example storing heterogeneous rows. The table in Figure 1 has two column families. The first holds some information of a person. Note that different rows can have different columns in this column family. The second column family models a graph structure. Each column can be seen as a key-value pair where the key is the name of a person's child and the value is the amount of pocket money they receive.

It stands to reason that SQL is not well-suited for querying these kind of databases. Different from relational databases, the columns are not known in advance and column names can be seen as part of the data, not metadata. To make complex queries and transformations on HBase, a simple *CRUD* (Create, Read, Update, Delete) API or MapReduce [9] can be used. So, if a user wants to calculate the average salary per company, one has to develop a Java program that scans the whole table, skips rows where the salary column doesn't exist, collects all values per company and in the end, compute the average value per company. With the usage of MapReduce, this task can be divided into a Map and Reduce function and a framework handles the whole computation over a large cluster of compute nodes.

In the given example table, there is a row-id to identify each row. Every HBase table has such an identifier, and based on this, rows are distributed in the cluster. There always is an index on the row-id, but other secondary indexes are not supported. That is why many transformations need to read the whole data. These long-running transformations, which are often executed periodically (e.g. once per day), can be accelerated by executing them in an incremental fashion. With this, a transformation only needs to read the result from the former run as well as the changes in the input to compute the new result. To build an incremental MapReduce job, the following things have to be developed by hand: First, an input format that reads both the former result and the changes since the former computation. Second, a component that distinguishes between insertions, deletions, updates and a former result. And third, an output format that updates the rows in the output table.

We developed the language *NotaQL* to define incremental transformations on wide-column stores. In Section 2, we show that this language is easy to use and powerful. Projection, Selection, Aggregation as well as complex graph and text algorithms can be expressed in just two or three short lines of code. NotaQL is well-suited for schema-flexible tables and allows to define transformations from metadata into data level and vice versa. In Section 3, we present the challenges of an incremental computation and how we made NotaQL transformations self-maintainable. After the presentation of performance results in Section 4, there is a discussion about related work in Section 5 and a conclusion in Section 6.

## 2 NotaQL Transformations

In this section, we give a brief overview of the NotaQL language. As our focus is on the implementation and incremental maintenance of query results, a more thorough discussion is beyond the scope of this paper, but can be found in [24].

### 2.1 The Language NotaQL

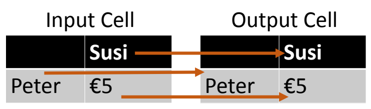
Apache HBase—and most other wide-columns stores as well—was designed for simple scalability. An easy-to-use query language or a query optimizer is not in the scope of this project. There are frameworks that can be used on top of HBase to query and transform tables (see Section 5), but they are either based on SQL—which is not well-suited for wide-column stores—, or they introduce a new complex language. So, for relational-like tasks like Projection, Selection, Aggregations and Joins, an SQL-based framework can be used, and for more complex tasks, one either has to develop a complex transformation program by hand, using MapReduce or another framework.

We analyzed dozens of typical MapReduce jobs for transformations on wide-column stores and found out that they all share a common pattern. With approximately one hundred lines of code in Map and Reduce functions, a user basically defines only one thing: How to map rows from an input table to an output table. With the transformation language *NotaQL*, one can simply define these mappings in two or three short lines of code:

| | |
|---|---|
| `IN-FILTER: ...,` | (optional) Which rows to filter? |
| `OUT._r <- ...,` | How to compute the new row-id for the output rows? |
| `OUT.xyz <- ...;` | How to compute the value of a column called `xyz`? |

The execution of a NotaQL script works as follows: Each row which satisfies the row predicate given in the optional `IN-FILTER` clause will be split into its cells. Each of these which satisfies an optional cell predicate can produce new output cells. If one output cell consists of multiple values, these values are grouped and aggregated to become the final value of that cell.

Instead of a fixed column name (`xyz`), one can make use of a `$`-expression to derive the column name from other values. To introduce the syntax, Figure 2 shows the simple example of copying a whole HBase table. The row-id of the output row is set to the input row-id. And for each cell in the input, a new cell is produced that has the same column name and the same value.



```
OUT._r <- IN._r,
OUT.$(IN._c) <- IN._v
```

**Fig. 2.** Table copy with NotaQL

In the given example, the variables `IN._r`, `IN._c` and `IN._v` were used to access the row-id, column name and value of an input cell. A table can be seen as a large set of input cells and a NotaQL script can be seen as a definition of how to construct an output cell based on an input cell.

```
IN-FILTER: born>1950,
OUT._r <- IN._r,
OUT.salary <- IN.salary,
OUT.cmpny <- IN.cmpny
```

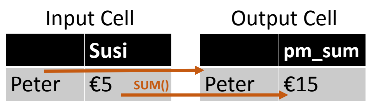
**Fig. 3.** Row predicate

```
OUT._r <- IN._r,
OUT.$(IN._c?(@='€5')) <- IN._v
```

**Fig. 4.** Cell predicate

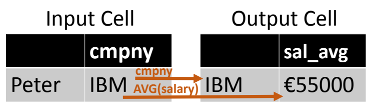
Often, not the full input should be read, but only those rows that satisfy a predicate. Figure 3 shows the usage of a *row predicate* for performing a selection using an `IN-FILTER`. Only rows where a column `born` is present and its value is greater than 1950 are read, all others are skipped. Furthermore in this example, a projection is made by not mapping all cells, but only those with the column name `salary` and `cmpny`. Figure 4 shows how one can use a *cell predicate* in NotaQL. It starts with a `?` and it can be placed after `IN._c` or `IN._v`. The cell is only considered when the predicate is evaluated to true. The `@` symbol can be used to access a value if the column name is unknown. So, the transformation in Figure 4 copies all cells from the input table into the output table which hold the value `'€5'`. These value-based projections are not possible in SQL.

In the transformations above, the value for an output cell (i.e. an `(OUT._r, OUT._c)` pair) was unique. If more than one value is mapped to the same cell, an aggregation function must be used. The transformation in Figure 5 computes a horizontal aggregation over all values in the column family `children`. So the result is a table with one column `pm_sum` that holds the sum of all pocket-money amounts for the person with the given row-id.



```
OUT._r <- IN._r,
OUT.pm_sum <- SUM(IN.children:_v)
```

**Fig. 5.** Aggregation: SUM



```
OUT._r <- IN.cmpny,
OUT.sal_avg <- AVG(IN.salary)
```

**Fig. 6.** Aggregation: AVG

Vertical aggregations—which are well-known from SQL—aggregate the values of a given column over multiple rows in the input. Figure 6 shows how to compute the average salary per company. As the output row-id is a company name and the output column is fixed (`sal_avg`), there can be multiple values in each cell. The function `AVG` computes the average of these values. Other aggregate functions are `SUM`, `MIN`, `MAX`, `COUNT` and `IMPLODE` (for text concatenation).

Figure 7 shows a simplified BNF of NotaQL. Example scripts for more transformations as well as for graph and text-analysis algorithms can be found in Figure 8.

$$
\begin{aligned}
\langle\text{NotaQL}\rangle &\models [\text{IN-FILTER: } \langle\text{predicate}\rangle,]\langle\text{rowspec}\rangle, \langle\text{cellspec}\rangle(, \langle\text{cellspec}\rangle) * [;]\\
\langle\text{rowspec}\rangle &\models \text{OUT.\_r <- } \langle\text{vdata}\rangle\\
\langle\text{cellspec}\rangle &\models \text{OUT.}(\langle\text{colname}\rangle \mid \$(\langle\text{input}\rangle)) \text{ <- } (\langle\text{vdata}\rangle \mid \langle\text{aggfun}\rangle(\langle\text{vdata}\rangle))\\
\langle\text{input}\rangle &\models (\text{IN.\_r} \mid \text{IN.}[\langle\text{colfamily}\rangle :](\text{\_c} \mid \text{\_v}) \mid \text{IN.}\langle\text{colname}\rangle)[?(\langle\text{predicate}\rangle)]\\
\langle\text{vdata}\rangle &\models \langle\text{input}\rangle \mid \langle\text{const}\rangle \mid \langle\text{vdata}\rangle(+ \mid - \mid * \mid /)\langle\text{vdata}\rangle\\
\langle\text{aggfun}\rangle &\models \text{COUNT} \mid \text{SUM} \mid \text{MIN} \mid \text{MAX} \mid \text{AVG}\\
\langle\text{const}\rangle &\models \text{'}(A \dots Z \mid a \dots z \mid 0 \dots 9) + \text{'} \mid (0 \dots 9) +\\
\langle\text{colname}\rangle &\models [\langle\text{colfamily}\rangle :](A \dots Z \mid a \dots z \mid 0 \dots 9) +\\
\langle\text{colfamily}\rangle &\models (A \dots Z \mid a \dots z \mid 0 \dots 9) +\\
\langle\text{predicate}\rangle &\models (\langle\text{colname}\rangle \mid @ \mid \text{col\_count}([\langle\text{colfamily}\rangle]))[\langle\text{op}\rangle(\langle\text{colname}\rangle \mid \langle\text{const}\rangle)]\\
&\quad \mid (\text{NOT} \mid !)\langle\text{predicate}\rangle \mid \langle\text{predicate}\rangle(\text{AND} \mid \text{OR})\langle\text{predicate}\rangle\\
\langle\text{op}\rangle &\models = \mid != \mid < \mid <= \mid > \mid >=
\end{aligned}
$$

**Fig. 7.** NotaQL Language Definition (simplified)

| Transformation | NotaQL | SQL |
|---|---|---|
| Table copy | `OUT._r <- IN._r,` `OUT.$(IN._c) <- IN._v` | `INSERT INTO out` `SELECT * FROM in` |
| Projection, Selection | `IN-FILTER: born > 1950,` `OUT._r <- IN._r,` `OUT.salary <- IN.salary` | `SELECT salary FROM in` `WHERE born > 1950` |
| Drop Projection | `OUT._r <- IN._r,` `OUT.$(IN._c?(!salary))<- IN._v` | —not possible— (only with FISQL [28]) |
| Horizontal Aggregation | `OUT._r <- IN._r,` `OUT.sum <- SUM(IN._v)` | —not possible— (unknown column names) |
| Vertical Aggregation | `OUT._r <- IN.cmpny,` `OUT.a <- AVG(salary)` | `SELECT cmpny,AVG(salary)` `FROM in GROUP BY cmpny` |
| Metadata ↔ Data | e.g. Reverse Web-Link Graph: `OUT._r <- IN._c,` `OUT.$(IN._r) <- IN._v` | —not possible— (unknown column names) |
| PageRank | `OUT._r <- IN.edges:_c,` `OUT.alg:PR <-` `SUM(alg:PR/COL_COUNT('edges')` | —not possible— ("number of columns being not null" is impossible to express) |
| Breath-First Search to compute distances in a graph | `IN-FILTER: dist,` `OUT._r <- IN.edges:_c,` `OUT.alg:dist <-` `MIN(alg:dist+1)` | —hard— (recursion needed) |
| Word Count | `OUT._r <- IN._v.split(' '),` `OUT.count <- COUNT()` | —hard— (string table functions needed) |
| Term Index | `OUT._r <- IN._v.split(' '),` `OUT.$(IN._r) <- COUNT()` | —hard— (string table functions needed) |
| TF-IDF | `OUT._r <- IN._r,` `OUT.$(IN._v.split(' ')) <-` `COUNT();` `OUT._r <- IN._r,` `OUT.max <- MAX(IN._v);` `OUT._r <- IN._c?(!max),` `OUT.$(IN._r) <- IN._v / max` | —hard— (string table functions needed) |

**Fig. 8.** Some NotaQL transformation scripts

## 2.2 A MapReduce-based Execution Platform for NotaQL

We decided to use the Hadoop framework for our NotaQL execution platform because of its scalability, failure handling, and its support for input and output formats for HBase. Furthermore, a NotaQL script can easily be divided into one Map and one Reduce function. Thus, every transformation can be executed in one single MapReduce job.

Figure 9 shows the NotaQL Map function. The Map input is one row from the input table. This row is skipped when a row predicate is not met. Otherwise, each cell in the row will produce intermediate-key-value pairs if the given cell predicate is satisfied. The map-output key consists of both the row-id of the output cell and its column name. Figure 10 shows that in the Reduce function, all values for an output cell are aggregated using the given aggregation function. If there is no such function, a cell value has to be unique. Otherwise, an exception will be thrown.

One alternative approach is the usage of only the row-id as map-output key. In this case, each Reduce function is responsible for a whole row and it has to aggregate all values for each distinct column qualifier separately. While this requires the Reduce function to group the values for each column, this approach reduces the data transfer between Map and Reduce nodes compared to the cell–wise-grouping approach because each row-id arrives only at one Reduce function. Furthermore, in the Reduce phase only complete rows with all their columns are written to the output table. This is more efficient than cell-wise writing. Nevertheless, we currently did not implement the cell-wise writing because the incremental execution is more easily comprehensible when cell grouping is done by the Hadoop framework and not by the NotaQL Reduce function.

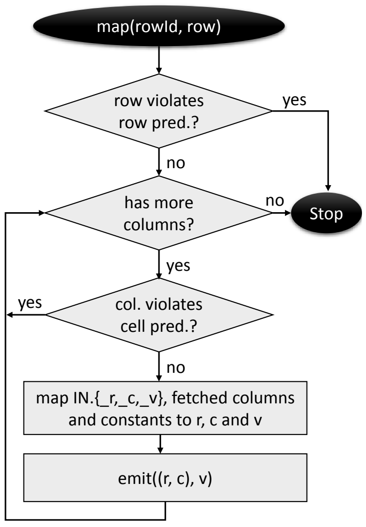


**Fig. 9.** NotaQL Map function

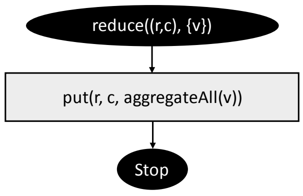


**Fig. 10.** NotaQL Reduce function

A Combine function can accelerate the computation by executing the aggregate function of the subset of values for an output cell on a Mapper node. Then, the Reduce function only needs to complete the aggregation. For most functions, the Combiner can be used because of the associativity of the aggregate operation. This means that cell values form a Monoid. One exception is the `AVG` function. For example, one Mapper node produces ten Map-output values and computes the average with the Combine function. Another Mapper node produces only one value. The correct average is the sum of the eleven values divided by eleven. Instead, the Reduce function divides the sum of two already computed average values by two. To solve this problem, a data type that stores the average value as a fraction of sum and count can be used.

## 3 Self-Maintainable Transformations

One goal of NotaQL is the support for self-maintainability. When executing a query periodically—for example once per hour—the changes since the previous computation can be aggregated on the former result to avoid a full recomputation. Nearly every block in figures 9 and 10 leads to challenges regarding an incremental computation. In the following, we list the problems and show how to solve them.

map(rowId, row)

A self-maintainable MapReduce job needs a special input format with support of change-data capture (CDC). Different from the non-incremental execution, the Map input is not the complete input table, but only the rows that changed. In HBase, each cell is annotated by a timestamp of when it was written. Our solution to support change-data capture in a wide-column store is a timestamp-based approach. Only those rows which were added, or those which got new columns, or those whose column values changed are used as map input. These cells are annotated by a flag *inserted*. If a column value changed since the last computation, an additional map-input cell with a *deleted* flag is produced that holds the former value. HBase provides a Delete option to delete a complete row from a table. As it is actually deleted, it does not appear in a table scan and the former value is lost. So, instead of deleting a row, an application has to set each column to null (an empty byte array). Then, the NotaQL framework flags each cell with a *deleted* flag. Different from an update, no *inserted* cell is produced because the new value is null.

Our approach uses a so-called Overwrite Installation [16, 15, 25] (see Figure 11) in which the full former result is also provided as input for the Map function.

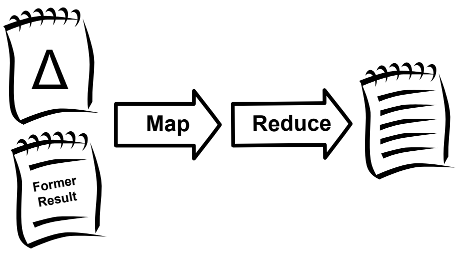


**Fig. 11.** Incremental MapReduce - Overwrite Installation

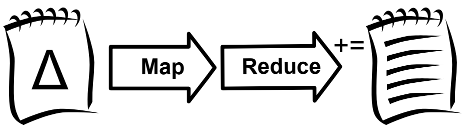
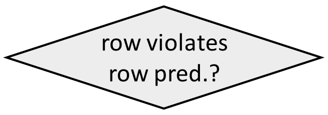


**Fig. 12.** Incremental MapReduce - Incremental Installation

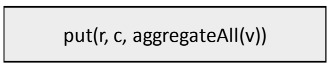
Example: Peter's company changes from IBM to SAP. The Map function will be called $n + 1$ times. Once for each of the $n$ rows in the former result (for example the salary sums per company) and once with the row `Peter`. The latter one consists of one *inserted* cell `(Peter, cmpny, SAP)` and one *deleted* cell `(Peter, cmpny, IBM)`.

Figure 12 shows an alternative to the Overwrite Installation, called the Increment Installation[16, 15, 25]. Here, the input is only the delta. The results of a computation are written into the output table in an incrementing way. This means, if the aggregation function SUM is used and the computed value of an output cell `(r,c)` is 5, the current value of the cell `(r,c)` has to be read, increased by 5 and written again. Experiments in [25] showed that this approach is faster for a small number of changes in the base data. If the changes are high, the Overwrite Installation is faster because it writes the output sequentially, not randomly. We decided not to use the Increment Installation, because it only works for NotaQL scripts that aggregate values either with a COUNT or SUM function. Simple table transformations without an aggregation, as well as computations of averages, minimums and maximums are only possible in the Overwrite-Installation approach. In the future, the execution framework can support both approaches and choose the Increment Installation if COUNT or SUM is used, and if the number changes in the base data are below a specific threshold (e.g. 1%).



In the non-incremental case, a row will be skipped as if it would not exist when it violates the row predicate. But in the incremental case, it can happen that the row predicate was satisfied on the former execution and—due to a change—now it is violated. When Peter's company changes from IBM to SAP, a query execution "Who works at IBM?" cannot skip Peter, but it has to delete him from the former result.

Deletions and changes in base data, and the violation of predicates can lead to changes in the output. This problem is well-known in the area of incremental MapReduce and can be solved by letting the Map output values form an Abelian group [16, 15]. This means, each Map-output value $v$ has an inverse element $v*$ that compensates the effect of an aggregation: $v \circ v* = e$ ($e$: identity element). The incremental Map function for supporting self-maintainable NotaQL queries inverts map-output values by setting a mode to them. There are four possible modes: *inserted*, *deleted*, *preserved*, *oldresult.*



In the Reducer, each aggregation function treats values depending on their mode. For example, the sum function multiplies each *deleted* value by minus one. So, when there are three Reduce input values, an inserted 5, a deleted 2, and an old result 115, the new sum is $5 + (-1) \cdot 2 + 115 = 118$.

| value `v` is... | **SUM** | **COUNT** | **AVG**=`sum/count` |
|---|---|---|---|
| ...inserted | `res+=v` | `res++` | `sum+=v; count++` |
| ...deleted | `res-=v` | `res--` | `sum-=v; count--` |
| ...preserved | - | - | - |
| ...an old result | `res+=v` | `res+=v` | `sum+=v.sum; count+=v.count` |

**Fig. 13.** Treating value modes in aggregate functions

Table 13 shows that for calculating an average incrementally, a special data type is needed to store the average as a fraction $\frac{sum}{count}$. This type—and for the functions SUM and COUNT the real numbers—forms an Abelian group under the given aggregation function. In contrast to that, the functions MIN and MAX only form a monoid. Here, there is no inverse element and so it is not possible to support deletions when these functions are used. When €60000 is the highest salary value for people working at IBM and the salary of the person with this value decreases, it is not possible to derive the new maximum without caching all the other values.

map IN.{_r,_c,_v}, fetched columns and constants to r,c and v

In typical timestamp-based change–data-capture approaches on wide-column stores[13], unchanged cells are not part of the input. But, depending on the NotaQL script, unchanged cells have to be read as well: First, to check row and column predicates which are defined on those unchanged columns and second, to fetch values of unchanged columns. When Peter's salary did not change, but his company affiliation does, and the query is "salary sum per company", the result of his new and his old company has to be adjusted. So, our change–data-capture implementation skips all rows without any column changes and processes the other rows with all important columns. As mentioned before, each cell will be flagged and now we introduce a new flag *preserved* for unchanged columns only for predicate checks and fetch operations.

## 4 Performance

We performed experiments on a six-node cluster (Xeon Quadcore CPU at 2.53GHz, 4GB RAM, 1TB SATA-II disk, Gigabit Ethernet) running Hadoop and HBase. Therefore, we executed two modified TPC-H queries plus one graph algorithm, and we performed the following experiments on each of those:

1. Measure the overhead of a NotaQL transformation versus a natively coded Hadoop job;

2. Compare a full recomputation with an incremental computation using a primitive timestamp-based change–data-capture approach;
3. Compare the results of 2. with an optimal change–data-capture approach to show the potential of the incremental computation.

The first experiment was to compare the runtimes between MapReduce jobs which are natively coded in Hadoop and NotaQL jobs. A NotaQL transformation cannot be faster than an optimally coded MapReduce job. But in our relational and graph examples, the runtimes have not been slower, too (see Figure 16). That means, writing a three-line NotaQL script instead of coding a complex Hadoop job is simpler and this job is executed without a noticable overhead.

Figure 15 shows our three benchmark transformations, two modified TPC-H [27] queries and one graph algorithm.

| **Transformation** | **NotaQL** | **SQL** |
|---|---|---|
| TPC-H 1 | `OUT._r <- IN.L_RETURNFLAG,`<br>`OUT.sum <- SUM(IN.L_QUANTITY)` | `SELECT L_RETURNFLAG,`<br>`SUM(L_QUANTITY) FROM`<br>`LINEITEM GROUP BY`<br>`L_RETURNFLAG` |
| TPC-H 6 | `IN-FILTER: L_QUANTITY < 24,`<br>`OUT._r <- 'all',`<br>`OUT.sum <- SUM(L_EXTENDEDPRICE)` | `SELECT`<br>`SUM(L_EXTENDEDPRICE)`<br>`FROM LINEITEM WHERE`<br>`L_QUANTITY < 24` |
| Reverse<br>Web-Link Graph | `OUT._r <- IN._c,`<br>`OUT.$(IN._r) <- IN._v` | —not possible— |

**Fig. 14.** Transformation scripts used in our experiments

The experiments show that incremental queries are faster than a full recomputation because the former result can be used. In our test dataset between two computations 40 MB (0.1%) data changed. The non-incremental TPCH query 1 lasts 94 minutes, the incremental one is 13 minutes faster. The most expensive part of our incremental computation was the change-data capture (CDC). For our experiments, we used a simple timestamp-based approach. But as HBase does not provide efficient time-range scans, with other approaches and in other database systems, there is a lot more optimization potential. This can be seen from the black bar in the chart. The input of that job were three HBase tables: one for inserted, one for deleted rows and one with old results. In our tests, the first two tables were filled by doing change-data capture manually. But those tables can be filled by the application which writes the base data or with the use of database triggers as well. Then there is a small overhead for every write in the data, but queries and transformations can be executed very quickly.

Another NotaQL job transforms a graph. For every page in the German Wikipedia, a row in the input table exists with its title as row-id. For each outgoing link, there is a column with the title of the linked page as column name.

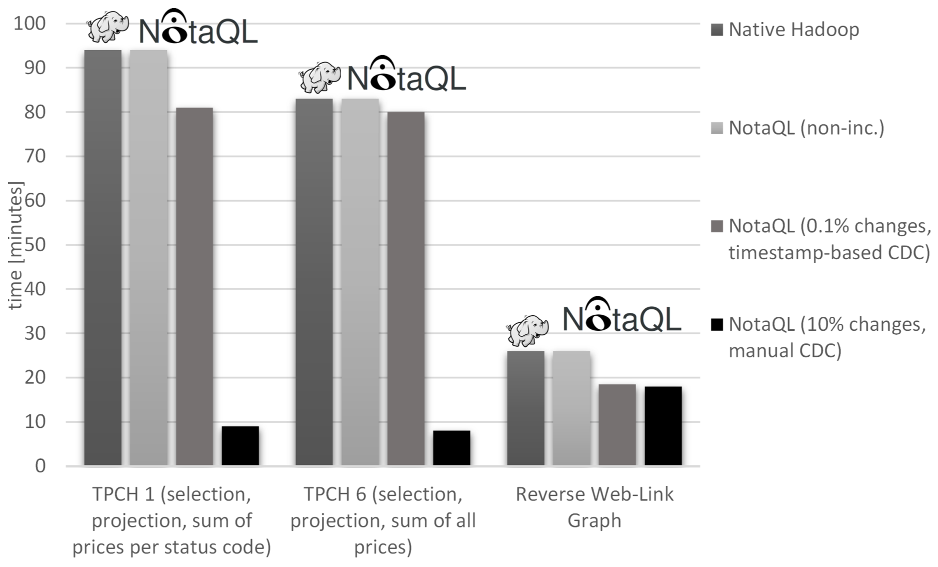


**Fig. 15.** Runtimes of Hadoop and NotaQL jobs (in minutes)—simplified TPCH queries[7] on 40GB lineitem table, graph algorithm on the full German Wikipedia link graph.

Our NotaQL job reverts this graph structure by computing the incoming links for each page. In our experiment, the manual and timestamp-based change-data capture are about the same speed because the most effort is reading the complete old result. Different from the TPCH queries, in this algorithm no aggregation and data compaction is performed.

## 5 Related Work

As there is no built-in query language in HBase, different approaches exist that allow access and transformations on HBase tables. Some of them were developed to bring back SQL into the NoSQL database. *Apache Hive* [26, 12], *Phoenix* [4] and *Presto* [22] and *PostgreSQL Foreign Data Wrappers* [10] work all in a similar way. First, a user has to define a mapping between an existing HBase table and a new SQL table using an `CREATE TABLE` statement. Thus, the columns of the table have to be known in advance. After that, one can use SQL to query, modify and transform data in the underlying HBase table. These approaches are easy to use and the user does not need to learn a new language. JDBC can be used on top of these frameworks to develop applications that read and write HBase. These approaches are good for relational queries with filters and joins. But they rely on a fixed schema and cannot transfer metadata into data and vice versa. Horizontal aggregations over multiple columns as well as value-based column filters are not possible in SQL, and therefore not possible in these approaches.

Google BigQuery [23] is the publicly-available version of Dremel [18]. The authors recommend to use MapReduce to perform data transformations and use

BigQuery to access and analyze the results in the generated output. In BigQuery, the language SQL is used. So, it can be seen as similar to the approaches in the previous paragraph. When using BigQuery on an output table of a NotaQL transformation, one can benefit from BigQuery's high performance and from NotaQL's simplicity and self-maintainability.

Other languages for wide-column stores are more powerful but less user-friendly. With *MapReduce* [9], arbitrary complex transformations can be defined. But often, a hundred lines of code need to be written. There are frameworks like *Incoop* [5] and *Marimba* [25] that simplify the development of incremental MapReduce jobs. They detect changes in the input data and update the output in an incremental fashion. A query language named *Jaspersoft HBase QL* [14] can be used to query data in Java applications via a JSON API. Here, more complex queries are possible. Different than SQL, it is well-suited for wide-column stores, but harder to use. Even a simple query with a single column-value filter needs about ten lines of code. In NotaQL it is only three short lines.

*Pig Latin* [20] is another language that allows complex transformations over different data sources. Pig scripts consist of multiple statements and are therefore closer to a programming language than to a query language. *Nova* [19] is a workflow manager for Pig programs and allows an incremental execution of those. Nova keeps track of deltas in the input datasets and triggers the execution of Pig programs either when new data arrives, when a task changes its state, or time-based. Currently, there is no support for wide-column stores. The only supported input format is the Hadoop file system HDFS. Different from our approach, a user has to make their workflows self-maintainable. In NotaQL, every transformation can be executed in an incremental way. Not the user, but the NotaQL transformation platform cares about change-data capture and the recomputation of aggregated values.

Based on the bag relational algebra, *DBToaster* [1] translates SQL view definitions on relational database systems into triggers. After an insert, delete or update operation in a base table, a trigger fires an update in the materialized view. This trigger-based approach is similar to NotaQL's timestamp-based change-data capture for self-maintainable views and avoids full recomputations. But, NotaQL updates the target table batch-wise, DBToaster on every single change. DBToaster supports higher-order incremental view maintenance, so it does not only capture the delta but also the delta of the delta. This leads to a high performance for queries that are based on multiple base tables. As NotaQL transformations have only one input table, first-order deltas are sufficient here.

Sawzall [21] is a programming language used by Google to define log analytics on large CSV files. The programs are executed as a MapReduce job. Our NotaQL language is partially inspired by Sawzall, but Sawzall can only process files, not tables.

With Clio [11], one can perform a schema mapping from different source schemata into a target schema using a graphical interface. Clio creates views in a semi-automatic way which can be used to access data from all sources. This virtual integration differs from our approach because NotaQL creates materialized views. Clio can only map metadata to metadata and data to data. There is no possibility to translate attribute names into values and vice versa. There are language extensions for SQL to allow these translations on relational databases, like SchemaSQL [17] and FISQL [28]. In [6], a copy-and-paste model is presented to load data from different sources into a curated database. Curated databases are similar to data warehouses but here, it is allowed to modify data in the target system. A tree-based model is used to support operations from SQL and XQuery as well as copying whole subtrees. The language presented in that paper also contains provenance functions to find out by which transaction a node was created, modified or copied. Although the language is very powerful, it does not support aggregations, unions and duplicate eliminations because in these cases, the origin of a value is not uniquely defined.

## 6 Conclusion

In this paper, we presented NotaQL, a transformation language for wide-column stores. The strengths of NotaQL are its expressive power and simplicity. Typical data processing tasks with filters and aggregations can be defined in just two or three short lines of code. Our language is well-suited for NoSQL databases that have no fixed schema. Metadata can be transformed into data and vice versa. This allows the definition of complex log, text, and graph algorithms. We showed that NotaQL transformations can be executed within the MapReduce framework in an incremental fashion. The fact that NotaQL transformation are self-maintainable allows incremental updates whenever there are changes in the base data. Our experiments showed that this incremental computations of NotaQL transformation scripts have a high potential for a much faster execution compared to a full recomputation.

## References

1. Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.
2. Apache Cassandra. `http://cassandra.apache.org/`.
3. Apache HBase. `http://hbase.apache.org/`.
4. Apache Phoenix - "We put the SQL back to NoSQL". `http://phoenix.incubator.apache.org/`.
5. Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.
6. Peter Buneman and James Cheney. A copy-and-paste model for provenance in curated databases. *Notes*, 123:6512, 2005.
7. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

8. George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.
9. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, pages 137–150, 2004.
10. Foreign data wrappers - PostgreSQL wiki. `https://wiki.postgresql.org/wiki/Foreign_data_wrappers`.
11. Mauricio A Hernández, Renée J Miller, and Laura M Haas. Clio: A semi-automatic tool for schema mapping. *ACM SIGMOD Record*, 30(2):607, 2001.
12. Hive HBase Integration. `https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration`.
13. Yong Hu and Stefan Dessloch. Extracting Deltas from Column Oriented NoSQL Databases for Different Incremental Applications and Diverse Data Targets. In *Advances in Databases and Information Systems*, pages 372–387. Springer Berlin Heidelberg, 2013.
14. Jaspersoft HBase Query Language. `http://community.jaspersoft.com/wiki/jaspersoft-hbase-query-language`.
15. Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Can MapReduce learn form Materialized Views? In *LADIS 2011*, pages 1 – 5, September 2011.
16. Thomas Jörg, Roya Parvizi, Hu Yong, and Stefan Dessloch. Incremental Recomputations in MapReduce. In *CloudDB 2011*, October 2011.
17. Laks VS Lakshmanan, Fereidoon Sadri, and Iyer N Subramanian. SchemaSQL-a language for interoperability in relational multi-database systems. In *VLDB*, volume 96, pages 239–250, 1996.
18. Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Communications of the ACM*, 54(6):114–123, 2011.
19. Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki BN Rao, Vijayanand Sankarasubramanian, Siddharth Seth, et al. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1081–1090. ACM, 2011.
20. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.
21. Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.
22. Presto - Distributed SQL Query Engine for Big Data. `http://prestodb.io/`.
23. Kazunori Sato. An inside look at google bigquery. *White paper, URL: https://cloud.google.com/files/BigQueryTechnicalWP.pdf*, 2012.
24. Johannes Schildgen and Stefan Deßloch. NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores. In *British International Conference on Databases - BICOD 2015*, 2015.
25. Johannes Schildgen, Thomas Jörg, Manuel Hoffmann, and Stefan Deßloch. Marimba: A framework for making mapreduce jobs incremental. In *2014 IEEE International Congress on Big Data*. IEEE, 2014.
26. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

27. TPC-H (ad-hoc, decision support) benchmark.
`http://www.tpc.org/tpch/`.
28. Catharine M Wyss and Edward L Robertson. Relational languages for metadata integration. *ACM Transactions on Database Systems (TODS)*, 30(2):624–660, 2005.

# A Service-oriented Approach for Improving Quality of Health Care Transitions

Marina Bitsaki[1,] Yannis Viniotis[2],

[1] Computer Science Department, University of Crete, Greece
bitsaki@tsl.gr
[2] Department of Electrical and Computer Engineering, North Carolina State University, North Carolina
candice@ncsu.edu

**Abstract.** Current studies show that patients' readmission rate in hospital can be significantly reduced by providing transition of care services. Offering services supported by information technology may help to address the gaps in communication with patients and quality of patients' transition from the hospital to another healthcare setting. In this paper, we propose and analyze a service system for managing patient transitions and investigate ways to improve its performance in a cost-effective way.

**Keywords:** Care transition • e-health services • matching problem

## 1 Introduction

Hospital readmission rates are frequently used as a measure to evaluate the performance of healthcare organizations. In a recent study [1], it was found that almost one fifth of the Medicare beneficiaries in the United States who are discharged from a hospital are re-hospitalized within 30 days, and 34% are re-hospitalized within 90 days. Inadequate health literacy, unrecognized cognitive impairment, severity of illness, lower quality of care in the initial admission are some of the factors that contribute to re-hospitalization [2], [3]. Current studies show that patients' readmission rate in hospital can be significantly reduced by providing transition of care services based on community support programs, patients' training, medication reconciliation, coordination across care settings and others to improve health outcomes and achieve cost reductions [4], [5], [6] .

In this paper, we propose a novel approach for managing patient transitions and investigate ways to improve its performance in a cost-effective way using a technology-based system that provides a standardized communication and information exchange between healthcare settings. We define and analyze a service system that models the interactions between healthcare systems, patients, insurance companies, engineers supporting technology to facilitate the transition of patients. The main objective of the proposed approach is to improve the performance of the service system in terms of cost of care, patient satisfaction, and readmission rate. In this paper, we deal with the problem of minimizing the overall cost incurred by a transition between two health care facilities provided that patients' preferences and economic constraints are satisfied. The remainder of the paper is structured as follows: in Section 2, we define the service network and present the research objectives. In Section 3, we provide the mathematical framework to the problem defining the total cost and the various constraints facing the transferring and the receiving care facilities. In Section 4, we provide concluding remarks and steps for future work.

## 2 Medical Problem Definition

Transition of care refers to the transition of patients from a primary, expensive facility to a secondary, less expensive facility that provides medical support to ensure a successful recovery. The transferring party provides medical records and instructions to the receiving party. The receiving party monitors predefined patient characteristics and evaluates the health status of the patient. The communication and the coordination of services between the two health care providers are performed by an intermediary who is responsible for a successful transition. The intermediary chooses the proper receiving provider based on specific criteria, and provides electronic health record technologies to gather, share and exchange information.

We model the above situation as a service system that describes the interconnections among humans (doctors, patients, engineers), organizations (hospitals, insurance companies, data management service companies) and technology (cloud environments, data repositories, sensors, etc.). The system aims to facilitate the transition of patients in a cost effective way and at the same time improve their medical status.

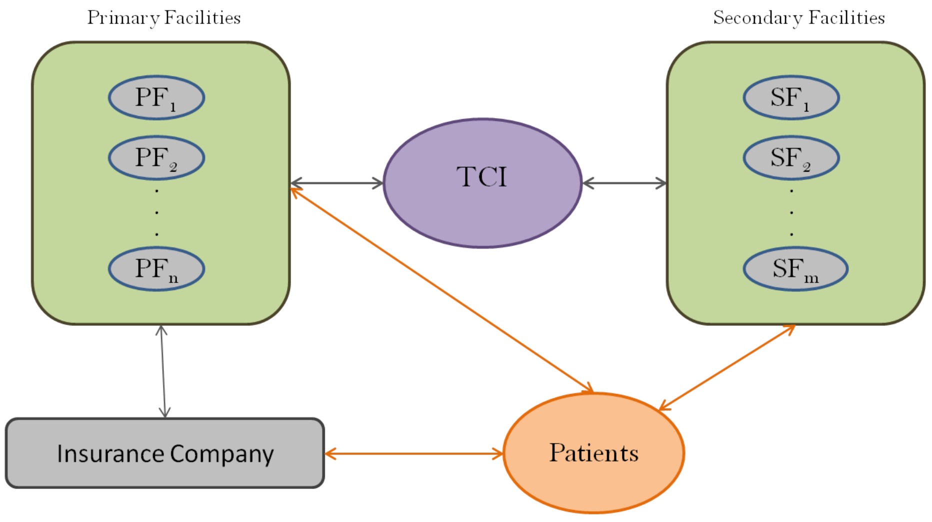


**Fig. 1.** The service system of transition of care

The components of the proposed service system are the entities that interact with each other and the relationships that enable the flow of information, services and revenues among them. We define the following types of entities (Fig. 1):

- A number of primary, expensive facilities (PF). They provide a healthcare service to patients.
- A number of secondary, less expensive facilities (SF). They provide a healthcare service to patients.
- Transition of care intermediary (TCI). TCI handles the logistics of the patient transfer from the primary to the secondary care facility. It also matches PFs to SFs. There might be more than one competing TCIs.
- Patients. They use health services provided by PFs and SFs.
- Insurance company. It covers hospitalization costs.

The formulation of the service system and the interactions among the above entities depend on the way the intermediaries match PFs to SFs. We consider the following variations:

- Match PFs to SFs statically, on a per facility basis. A PF may be matched to one or multiple SFs (all patients of this PF are transferred to the chosen SFs) and an SF may be matched to one or multiple PFs (The SF receives patients from the chosen PFs).
- Match PFs to SFs dynamically, on a per patient basis.
- Match PFs to SFs on a per disease basis. In this case SFs are specialized with patients of a common disease.

We can use the following payment models to specify the flows of revenues over the relationships among the entities of the service network:

- Fee-for-service model. An insurance company pays PF, SF, TCI, on a per patient basis. There is no limit on the cost (per patient).
- Bundled payments model (ACO). The insurance company pays one provider (typically PF) to insure a population of potential patients. The PF pays the SF and TCI, either on a per patient basis or as a bundled payment. There are cost limits on a per patient basis as well as per disease basis.

The performance of the service system is measured through a number of Key Performance Indicators (KPIs). We are interested in the total cost of care per patient, the patient satisfaction, the readmission rate from the secondary facility back to the primary facility that reflects the quality of service provided by the secondary facility and the market share each intermediary gains in a competitive environment. The above KPIs are affected by a number of factors such as the matching of PFs to SFs, the services provided by the various entities and the use of infrastructure. For example, taking into account patient preferences on the matching of PFs to SFs increases patient satisfaction, considering geographical constraints reduces costs, providing high quality care services reduces the readmission rate.

IT infrastructure used by TCIs plays a significant role in the formulation of the relationships among the entities within the service network. Therefore, it provides a good solution to the way patients are transferred from primary to secondary facilities. Indicatively, IT infrastructure provides the electronic medical records (EMR) of patients, sensors and an automated process of the transition of patients. EMR is shared by all interested parties consistently such that everyone has an updated version any time this is required, reducing significantly the readmission rate. Sensors are placed in the patient rooms in order to watch and record any change in the behavior/status of the patient or of the room conditions. This information is incorporated in the EMR of the patient so that new instructions are given for her/his medical care, increasing their satisfaction and reducing the readmission rate. The automation in the transition process facilitates the transfer of patients and reduces costs or errors in handling the patients.

The main objective of the transition of care service system described above is to provide a cost-effective way for patients' transitions within healthcare organizations improving their health status and reducing readmission rates. In particular, the analysis of the proposed service system is meant for solving the following business problems:

- What is the optimal matching of PFs and SFs such that the overall cost is minimized provided that patients' preferences and economic constraints are satisfied?
- What are the optimal strategies of the entities within the service system such that the performance of the service system is improved in terms of customer satisfaction or readmission rate?

In the next section we formulate the optimization problem that corresponds to the first of the above questions.

## 3 Mathematical Problem Formulation

In this section, we address the case in which there is one transition of care intermediary (TCI) aiming at statically matching primary facilities (PFs) to secondary facilities (SFs) on a per facility basis. A PF may be matched to one or multiple SFs (all patients of this PF are transferred to the chosen SFs) and an SF may be matched to one or multiple PFs (the SF receives patients from the chosen PFs). The solution will be implemented on a per patient basis; backup SF will be chosen by solving the same problem with the first SF removed. We adopt the bundled payment model (see Section 2) in which the insurance company pays each PF that in turn pays an amount per patient to each SP it is associated with. The objective of the TCI is to choose the matching of each PF to a set of SFs so as to minimize the total cost of PFs.

Let the set $PCF$ of primary facilities contain $I$ facilities, $PCF = \{PF_1, \dots, PF_I\}$ and the set $SCF$ of secondary facilities contain $J$ facilities, $SCF = \{SF_1, \dots, SF_J\}$. Within these two sets, facilities may be grouped into subsets (e.g., primary facilities that belong to a for profit organization). Care facilities have certain properties, as we describe below:

1. Secondary care facilities $SF_j,\ j \in \{1, \dots, J\}$
   (a) $RR_j$: the readmission rate; we assume that $0 \leq RR_j \leq 1$.
   (b) $\bar{C_j}$: the cost per patient that $SF_j$ charges; we assume that $\bar{C_j} > 0$.
   (c) $PSI_j$: the patient satisfaction index; we assume that $0 \leq PSI_j \leq 1$.
   (d) $PSC_j$: the patient capacity; we assume that $PSC_j > 0$.
2. Primary care facilities $PF_i,\ i \in \{1, \dots, I\}$
   (a) $PPS_i$: the patient population size; we assume that $PPS_i > 0$.
   (b) $GC_i$: the geographical constraints; we assume that $GC_i$ is a subset of the set $\{1, \dots, J\}$.
   (c) $MCP_i$: the maximum cost per patient; we assume that $MCP_i > 0$.
   (d) $MRR_i$: the maximum tolerable readmission rate; we assume that $0 \leq MRR_i \leq 1$.
   (e) $MPSI_i$: the minimum tolerable patient satisfaction index; we assume that $0 \leq MPSI_i \leq 1$.

The variable of choice in the minimization problem is the matching of the primary care facility $PF_i,\ i \in \{1, \dots, I\}$ to a set of secondary care facilities $SF_j,\ j \in \{1, \dots, J\}$. More specifically, let $M_i$ denote the set of $SF_j$ that will receive patients from $PF_i$. $M_i$ is a nonempty subset of $\{1, \dots, J\}$. Let $M$ denote the matching of all primary care facilities to secondary ones. The variable of choice is defined as $M = \{M_1, M_2, \dots, M_I\}$.
The objective function in the minimization problem is the total cost associated with choice $M$. It is defined as follows: let $C_i(M_i)$ denote the cost associated with primary care facility $PF_i$ when patients from this facility are transferred to choice $M_i$ and $C(M)$ denote the total cost associated with choice $\{M_1, M_2, \dots, M_I\}$. We have:

$$C_i(M_i) = \frac{1}{|M_i|} \sum_{j \in M_i} \bar{C_j} \ , \tag{1}$$

$$C(M) = \sum_{i=1}^{I} C_i(M_i) \ . \tag{2}$$

The constraints to be satisfied are the following:

$$M_i \in GC_i, \forall i \in \{1, \dots, I\} \ , \tag{3}$$

$$RR_j \leq MRR_i,\ \ \forall j \in GC_i, \forall i \in \{1, \dots, I\} \ , \tag{4}$$

$$PSI_j \geq MPSI_i,\ \ \forall j \in GC_i, \forall i \in \{1, \dots, I\} \ . \tag{5}$$

The set of choices satisfying the constraints is denoted as $\mathcal{M}$. The problem is formulated as follows:

$$\min_{M \in \mathcal{M}} C(M) \text{ such that} \tag{6}$$

$$C_i(M_i) \leq MCP_i, \forall i \in \{1, \dots, I\} \ . \tag{7}$$

### 3.1 A Numerical Example

In this section, we present a simple example and provide a solution to the corresponding mathematical problem defined in Section 3. We consider $I = 3$ primary care facilities and $J = 5$ secondary care facilities. The properties of primary and secondary care facilities are given in Table 1 and Table 2 respectively.

**Table 1.** Properties of primary care facilities

| $PF_i$ | $GC_i$ | $MCP_i$ | $MRR_i$ | $MPSI_i$ |
|---|---|---|---|---|
| 1 | {1,2,3} | 1100 | 0.8 | 0.8 |
| 2 | {2} | 1100 | 0.8 | 0.8 |
| 3 | {1,2,3,4,5} | 1000 | 0.7 | 0.6 |

**Table 2.** Properties of secondary care faciities

| $SF_j$ | $\overline{C}_j$ | $RR_j$ | $PSI_j$ |
|---|---|---|---|
| 1 | 1100 | 0.7 | 0.9 |
| 2 | 1000 | 0.8 | 0.8 |
| 3 | 1000 | 0.8 | 0.7 |
| 4 | 1000 | 0.7 | 0.8 |
| 5 | 900 | 0.6 | 0.8 |

The set of choices $M$ without taking into account the constraints defined in Equations (3), (4) and (5), consists of $(2^5 - 1)(2^5 - 1)(2^5 - 1) = 29791$ elements. The set $\mathcal{M}$ of choices $M$ satisfying the above constraints consists of $(2^2 - 1)(2^2 - 1) = 9$ elements (the feasible set of $SFs$ for $PF_1$ is {1,2}, for $PF_2$ it is {2}, and for $PF_3$ it is {4,5}). The solution to the problem defined in Equations (6) and (7) is then given by the following matching: $PF_1 \rightarrow SF_2,\ PF_2 \rightarrow SF_2, PF_3 \rightarrow SF_5$ (that is, $M = \{\{2\},\{2\},\{5\}\}$ with total cost $C(M) = 2900$.

## Conclusions

Transition of care services are used to improve hospital quality care in order to prevent readmission. In the current work, we define and analyze a service system that models the interactions of healthcare systems, patients, insurance companies, engineers supporting technology to facilitate the transition of patients. We formulate a cost minimization problem to find the optimal matching of primary care facilities to secondary ones. In the future we intend to use game theoretic tools to model the interactions of entities within the proposed service system and calculate optimal strategies for improving the performance of the service system in terms of patient satisfaction and readmission rate.

## References

1. SF Jencks, MV Williams, and Coleman EA, "Rehospitalizations among patients in the Medicare fee-for-servise system," *N Engl J Med*, vol. 360(14), pp. 1418-1428, 2009.
2. A Chugh, MV Wiiliams, J Grigsby, and E Coleman, "A better transitions: improving comprehension of discharge instructions," *Front Health Serv Manage*, vol. 25(3), pp. 11-32, 2009.
3. NI Goldfield et al., "Identifying potentiallly preventable readmissions," *Health Care Finance Rev* , vol. 30(1), pp. 75-91, 2008.
4. M D Loguel and J Drago, "Evaluation of a modified community based care transitions model to reduce costs and improve outcomes," *BMC Geriatrics*, vol. 13(94), 2013.
5. R Voss et al., "The care transitions intervention : translating from efficacy to effectiveness," *Arch Intern Med*, vol. 171(14), pp. 1232-1237, 2011.
6. BW Jack et al., "A reengineered hospital discharge program to decrease rehospitalization: a randomized trial," *Ann Intern Med*, vol. 150(3), pp. 178-87, 2009.

# C2C: An Automated Deployment Framework for Distributed Applications on Multi-Clouds

Flora Karniavoura, Antonis Papaioannou, and Kostas Magoutis

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Heraklion 70013, Greece
{karniav,papaioan,magoutis}@ics.forth.gr

**Abstract.** The Cloud Application Modeling and Execution Language (CAMEL) is a new domain-specific modeling language (DSL) targeted to modeling applications and to supporting their lifecycle management on multiple (heterogenous) cloud providers. Configuration management tools that provide automated solutions to application configuration and deployment, such as Opscode Chef, have recently met wide acceptance by the development and operations (or DevOps) community. In this paper, we describe a methodology to map CAMEL models of distributed applications to Chef concepts for configuration and deployment on multi-clouds. We introduce C2C, a tool that aims to automate this process and discuss the challenges raised along the way suggesting possible solutions.

**Keywords:** Cloud computing, Application modeling, Configuration management

## 1 Introduction

In the era of cloud computing, applications are benefitting from a virtually inexhaustible supply of resources, a flexible economic model, and a rich choice of available providers. Applications consisting of several software components or services typically need to be deployed over multiple underlying technologies, often across different cloud providers. To bridge across different cloud environments, tools based on model-driven engineering principles are recently gaining ground among developers and operations engineers. TOSCA [1], CloudML [5] and CAMEL [18] are three recently introduced model-driven approaches used to express application structures and requirements, and to manage application deployments over time.

An important tool in the hands of application developers and operations engineers is the ability to maintain a detailed recording of software and hardware components and their interdependencies in an infrastructure, in a process known as *configuration management* (CM) [13]. An effective CM process provides significant benefits including reduced complexity through abstraction, greater flexibility, faster machine deployment and disaster recovery, etc. There are numerous configuration management tools from which the most widely known are:

Bcfg2 [3], CFEngine [6], Chef [7], and Puppet [17]. Each of these tools has its strengths and weaknesses [20] [9]. A CM solution is often combined with provisioning and deployment tools.

In this position paper we bridge the gap between application models (which are typically declarative expressions of application state) and configuration management tools (imperative procedures for carrying out CM actions) using CAMEL and Chef as specific instances of the two approaches. We introduce *CAMEL-to-Chef* (or C2C for short), a new methodology for the deployment and configuration of applications expressed in CAMEL across multi-cloud environments.

## 2 Background

### 2.1 CAMEL

Cloud Application Modeling and Execution Language (CAMEL) [18] is a family of domain-specific languages (DSLs) currently under development in the PaaSage EU project [15]. CAMEL encompasses DSLs covering a wealth of aspects of specification and execution of multi-cloud applications. CloudML, one of the DSLs comprising CAMEL, is used to describe the application structure and specify the topology of virtual machines and applications components. Below we describe key modeling elements that CAMEL shares with CloudML.

- *Cloud*: a collection of virtual machines (VMs) offered by a cloud provider
- *VM type*, *VM instance* : a VM type refers to a generic description of a VM, while an instance of a VM type refers to a specific instantiation of a VM, including specific configuration information.
- *Internal component* : a reusable type of application component, whereas an *internal component instance* represents an instance of an application component. The description of an application component stays at a generic level while the specification of its respective instances involves particular configuration information.
- *Hosting*, *Hosting Instance* : a hosting relationship between a host VM and a component of the application, or between two application components.
- *Communication*, *Communication Instance* : a dependency relationship between two application components or component instances.

CAMEL is under development at this time and thus constantly evolving. The changes that have been brought into CAMEL since we started the C2C project have so far been dealt with with just minor changes at the model parsing phase and have not resulted in drastic changes in the fundamentals of our approach. Future changes in CAMEL could be dealt with existing technologies that address the co-evolution of models [14].

### 2.2 Opscode Chef

Chef is a configuration management tool created by Opscode [7]. Following an infrastructure-as-code approach, Chef uses *Recipes*, configuration files written in

Ruby that describe the actions that should be performed on a node in order to bring it to its desired state. Related recipes are stored in *Cookbooks*. Users can store and write Cookbooks at the Chef repository in their local workstation, from where they can also interact with the Chef server. Every machine-node that is managed by Chef has a *run-list*, which is the list of recipes that will run on it at the time of the next Chef client run. We should note that dependencies between cookbooks are handled automatically by the Chef server, which is also responsible for various other tasks like run-list and cookbook storing.

Chef brings in a number of benefits. It offers automated and reusable solutions for the configuration and deployment of applications and a lot of ready-to-use, publicly available Cookbooks via the Chef repository, also known as *Chef supermarket* [8]. One of the strongest aspects of Chef is its active and constantly evolving community. The Chef community consists of people of various backgrounds and expertise that contributes to the creation and improvement of a large set of Cookbooks covering a wide range of software components.

## 3 Related work

Application modeling is becoming increasingly popular nowadays due to the complexity and increased needs of distributed applications. A recently introduced modeling approach covering the description, deployment, and lifecycle management of distributed applications is TOSCA [1]. Perhaps closest to our approach is a recent paper on cloud service orchestration using TOSCA, Chef and openstack [11] uses Chef as a deployment tool for applications defined as TOSCA models. "Deployment artifacts" are defined at the time of model creation for each component, stating which Cookbook recipe(s) should be used to deploy them. Deployments take place on openstack and various Chef functions are triggered using the knife-openstack client [12]. The differences between this work and ours are (1) the fact that we use CAMEL instead of TOSCA to model our applications, and (2) we automatically derive information from CAMEL models to achieve deployment with Chef in *multi-cloud* environments. The CAMEL model does not need to contain information about the recipes needed for each component, although we describe this as an alternative technique in Section 6.

CloudML [5] also offers a deployment and lifecycle management mechanism [4] by associating each deployable component with a pointer code responsible for its deployment. The deployment process is restricted to scripted commands and does not involve the usage of Chef.

## 4 The C2C methodology

### 4.1 Architecture

Figure 1 depicts the overall architecture of our system. C2C comprises of three major modules: i) the model parser ii) the VM manager and iii) the Chef instructor. The *model parser* analyses the input application model, extracts the necessary information and prepares the input for the other C2C modules. In more details, it forms a list containing the VMs that will be used for the application deployment. In addition it prepares the input of the Chef instructor module which is a list containing the software components that comprise the application along with their hosting and communication relationships. The *VM manager* module is responsible for the provisioning of the VMs and the installation of the Chef client in each one. The *Chef instructor* manages the deployment of the application software components on the appropriate VM, indicated by the hosting instances. As a first step, it collects all the necessary Cookbooks by searching at the Chef workstation or on the Chef's community repository [8]. Next it forms the run-list of each node in order to install the application components. The Chef instructor derives the order in which the components should be installed as well as the node that will host each one by analysing the communication and hosting relationships among the components.

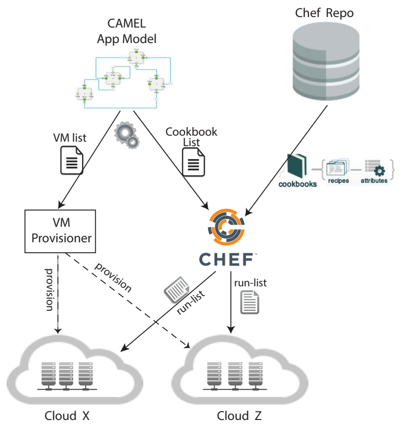


**Fig. 1.** System architecture

We follow this modular approach because it allows us to split the functionality of our tool and minimize the amount of effort needed in case of a component update (e.g. if we want to support more Cloud providers we just update the VM manager component). The model parser was implemented using the Java compatible CAMEL API library. The multi-Cloud provisioning logic we embedded in VM manager uses the third party library of *Apache JClouds* [10] and the official *Azure Java sdk* [2] API to operate across the different Cloud architectures of Openstack, Flexiant, Amazon EC2 and Microsoft Azure.

## 4.2 Mapping of concepts

In this section we describe the necessary CAMEL attributes that are used by the model parser in order to prepare the input of the other C2C modules. The

*VM instance* properties contain all the necessary information for the VM manager to provision the required resources for the application deployment. The Chef instructor uses the names of the *Component instances* in order to identify the corresponding cookbooks. The *hosting instance* relationship between a (software) component instance and an VM instance indicates that the corresponding cookbook should be added to the run-list of the node. On the other hand if a component instnace A is hosted in component instance B, then the cookbook corresponding to B should also be included in the run-list of node that will host component A. The deployment order of the components is derived based on the the *hosting instances* and the *communication instance* between component instances. For example if component X communicates with component Y then the deployment of Y should precced the deployment of X.

## 5 Use case

We demostrate our systems functionality using the distributed SPEC jEnterprise2010 benchmark [19] as a case study. SPEC jEnterprise2010 is a full system benchmark that allows performance measurement of Java EE servers and supporting infrastructure. The SPEC jEnterprise2010 application requires a Java EE application server and a relational database management system.

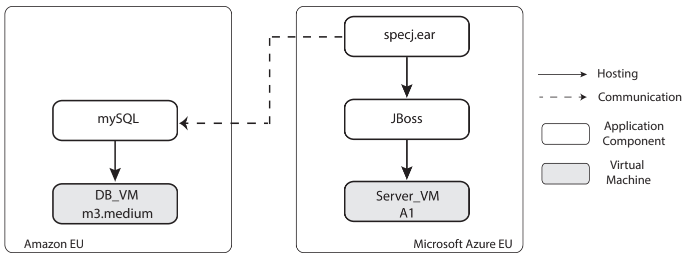


**Fig. 2.** Spec jEnterprise2010 application structure

We model the SPEC jEnterprise2010 application using three software components corresponding to the business logic of the application, the application .ear, the application server and the RDBMS [16]. These components are instantiated as a specj.ear, a JBoss application server and a MySQL database. Figure 2 presents the application structure and the deployment scenario of SPEC jEnterprise2010. The solid line arrows indicate the hosting relationships between VMs and application components as well as the communications among software components. The dashed line arrows represent the communications between the application components. In this scenario we demonstrate a cross-Cloud deployment of the application (different application components are deployed on different Cloud platforms).

At the first step the model parser instructs the VM manager to provision a m3.medium VM on Amazon EC2 platform and an A1 VM on Azure. Then the Chef instructor fetches the necessary cookbooks of MySQL and JBoss from the Chef supermarket. On the other hand we provide our custom cookbook for the deployment of the application logic (specj.ear) in our local workstation. The hosting and communication relationships between software components and VMs suggest the run-lists of Chef nodes. The run-list corresponding to the DB_VM contains the cookbook of MySQL while the run-list of Server_VM contains the cookbooks of and specj.ear. The deployment of DB node precedes the server node according to the Chef instructor logic described in section 4.

## 6 Challenges

Our aim for the C2C methodology is to be as automatic as possible, however there are challenges to achieve this that we discuss in this section along with possible solutions.

A key challenge in the C2C methodology is to decide automatically what is the correct Chef cookbook to use for a particular software component. In our automatic implementation we assume that there is a match between a component's name and the name of the suitable cookbook for it – however this need not always be true.

If we assume that components in CAMEL models are named using some variation of the name of the software component that they model, C2C could map them to cookbooks whose names most closely match the name of the software component (e.g., exhibit minimal lexicographical distance from it). This solution could be error-proof if the creator of a CAMEL model is aware of the Chef cookbook it wants to map the component to and thus names the component using the exact name of the Chef cookbook.

However even if the right cookbook for a component is discovered, the problem has not been solved. The difficulty now lies in distinguishing the right recipe for the desired task, between all recipes in the cookbook. In most cases the "default" recipe, present in all cookbooks, is responsible for the basic cookbook task (in most cases, installation) but this does not apply to every cookbook. Usually, recipe names are quite descriptive but not to an extent that could lead to efficient recipe selection.

A solution to this problem could be an appropriate naming scheme for cookbook recipes. Firstly, unique keywords such as "install", "update" or "start" should be used for basic tasks implemented by recipes. Cookbook recipes could simply include annotations stating which of these tasks each one of them implements. Current recipes do not provide this kind of information but it could be retrofitted or overlaid on recipe metadata based on user feedback: cookbook users could report on the task each recipe they use performs, and this information could later be used to help C2C automatically choose the right recipe.

Finally, a simple way to address these issues is by declaring the exact cookbook recipes to use in each application component within the CAMEL model (similar to what was proposed in [11]). This solution eliminates the risk of choosing the wrong recipe, albeit at the cost of reduced flexibility.

## 7 Conclusion

In this position paper we introduced C2C, an automated deployment framework for distributed applications on multi-clouds. We showed that the configuration, deployment and lifecycle management of CAMEL applications leveraging the large base of Chef cookbooks is achievable in an automated fashion. We discussed the challenges that stand in the way of full automation with this methodology and proposed different ways to overcome them. Finally, we demonstrated the usability of C2C using the SPEC jEnterprise2010 application as a case study.

## References

1. Oasis: Oasis topology and orchestration specification for cloud applications (tosca)
2. Azure SDK (Accessed 1/2015), `https://github.com/Azure/azure-sdk-for-java`
3. Bcfg2: (Accessed 2/2015), `http://www.bcfg2.org/`
4. Blair, G., Bencomo, N., France, R.: Models@ run.time. Computer 42(10) (2009)
5. Brandtzg, E., Parastoo, M., Mosser, S.: Towards a Domain-Specific Language to Deploy Applications in the Clouds. In: CLOUD COMPUTING 2012: 3rd International Conference on Cloud Computing, GRIDs, and Virtualization (2012)
6. CFEngine: (Accessed 2/2015), `http://www.cfengine.com/`
7. Chef: (Accessed 1/2015), `http://www.getchef.com/`
8. Chef Supermarket: (Accessed 1/2015), `https://supermarket.chef.io/`
9. Delaet, T., Joosen, W., Vanbrabant, B.: A survey of system configuration tools. In: Proceedings of the 24th International Conference on Large Installation System Administration. pp. 1–8. LISA'10 (2010)
10. JClouds: (Accessed 1/2015), `https://jclouds.apache.org/`
11. Katsaros, Menzel, L.: Cloud service orchestration with tosca, chef and openstack pp. 1–8 (2014)
12. Knife-Openstack client: (Accessed 2/2015), `https://docs.chef.io/plugin_knife_openstack.html`
13. Lueninghoener, C.: Getting started with configuration management. ;login: 36(2), 12–17 (2011)
14. Nikolov, N.: Integration and Co-evolution of Domain Specific Languages in Heterogeneous Technical Spaces. Master's thesis, Tilburg University, University of Stuttgart and University of Crete (Jul 2014)
15. PaaSage EU FP7 project: (Accessed 2/2015), `http://www.paasage.eu/`
16. Papaioannou, A., Magoutis, K.: An architecture for evaluating distributed application deployments in multi-clouds. In: Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on. vol. 1 (Dec 2013)
17. Puppet: (Accessed 2/2015), `http://www.puppetlabs.com/`
18. Rossini, A., Nikolov, N., Romero, D., Domaschka, J., Kritikos, K., T., K., Solberg, A.: Paasage project deliverable d2.1.2 - cloudml implementation documentation. Public deliverable (2014)
19. SPEC jEnterprise2010: (Accessed 2/2015), `https://www.spec.org/jEnterprise2010/`
20. Tsalolikhin, A.: Configuration management summit. ;login: 35(5), 104–105 (2010)

# The TPL Mission: We Bring Customized Cloud Technology to Your Private Data Centers
## – Overview/Industry Paper –

Tim Waizenegger[1], Frank Wagner[2], Cataldo Mega[3], and Bernhard Mitschang[4]

[1] Technology Partnership Lab, University of Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany,
tim.waizenegger@ipvs.uni-stuttgart.de
[2] Technologie Transfer Initiative GmbH,
Nobelstraße 15, 70569 Stuttgart, Germany,
frank.wagner@ipvs.uni-stuttgart.de
[3] IBM Deutschland Research & Development GmbH,
Schönaicher Straße 220, 71032 Böblingen, Germany,
cataldo_mega@de.ibm.com
[4] Institute for Parallel and Distributed Systems, University of Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany,
bernhard.mitschang@ipvs.uni-stuttgart.de

**Abstract.** In order to save cost and minimize operational overhead, enterprises seek to outsource data management to cloud providers. Due to security concerns and performance problems, these projects often are not successful or don't perform as expected. The main cost factor in these situations are the operational costs for running and maintaining the data management infrastructure. In the Technology Partnership Lab, we research and evaluate solutions that keep the data and infrastructure on-premise and reduce costs by automating system management and operations, through the use of on-premise cloud-technology. This allows us to avoid security concerns arising from outsourced data, and still reduces cost for the customer. In this paper we present our approach and the underlying hardware and software architecture. We also present an industry example, based on an enterprise content management system, and discuss our experience in running this system as an on-premise cloud application.

**Keywords:** Cloud Computing, Enterprise Content Management, Software-Defined Environment, Service-Topology Modeling

## 1 Introduction

The driving force for outsourcing data management is reducing the operational cost for running the system long-term. Cloud providers can offer low prices because of the economy of scale which works in their favor in two ways. First, they share their infrastructure between large numbers of customers. This practice allows them to utilize the hardware to a high degree, resulting in cost-savings through efficiency. And second, they use automation for provisioning and running a large part of the services and products they offer to their customers. This allows them to solve common tasks and problems only once, and then apply the method to all their customers, resulting in cost-savings through re-use [5,4,1].

The Technology Partnership Lab (TPL) works in the research and development of these cloud-technologies, but applies them to local, on-premise infrastructures. We want to transfer the two main advantages cloud-providers have, infrastructure-efficiency and solution re-use, to on-premise data-centers. With this approach we can also transfer (some of) the cost-savings to on-premise solutions. In addition, we can still keep the benefits of an on-premise solution, compared to off-premise cloud solutions. Data security, privacy and low access times are the most important ones of these benefits. They are still not achievable with cloud-solutions, and are commonly cited as the main reasons against outsourcing to cloud providers. With our on-premise-cloud approach we combine the benefits of both worlds and avoid the major problems they have.

## 2 The Technology Partnership Lab – TPL

The Technology Partnership Lab[5] (TPL) was founded in 2012 in order to bridge the gap between university research, teaching, and concrete industry-related activities. In order to provide a benefit to our industry partners, we offer a portfolio of knowledge, tools, and hardware, as represented by Figure 1. At the same time, we benefit the university and students by bringing new material and knowledge into the teaching, and most importantly by cooperating with students in our projects.

The TPL's concept of an on-premise cloud stands on two pillars: one, the efficient and flexible use of hardware resources, and two, the application and re-use of predefined solutions. Together with partners we evaluate how we can apply our on-premise cloud approach to solve their challenges. We begin this evaluation by setting up the desired system on our TPL test-lab hardware. We then transform the system into an on-premise cloud solution by using the methods described in Section 4. Finally, we test the solution together with our partners on their local infrastructure.

## 3 First Pillar: The on-Premise Cloud Infrastructure

The efficient use of the available hardware resources is one of the cornerstones of cloud computing. Providers analyze the utilization of different customer systems in order to determine how they should be assigned to the physical resources [3]. An optimal assignment will satisfy all the resource requirements of the customer systems, and at the same time use a minimal amount of physical resources. This efficient use of resources allows the cloud providers to offer competitive pricing.

[5] `http://tpl.informatik.uni-stuttgart.de`

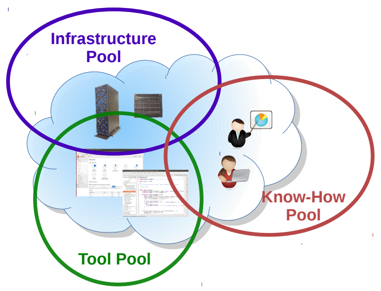


**Fig. 1.** The three TPL-pools

Therefore, we design our on-premise cloud infrastructure to support the same resource optimization.

The TPL operates a test-lab which we designed as a small-scale representation of a typical infrastructure found in most corporations. We use this test-lab hardware to develop and evaluate our software solutions for the on-premise cloud infrastructure. The test-lab consists of an IBM PureFlex high-density server cluster with a flexible 10 GBit network and FibreChannel-attached IBM V7000 storage system (see Figure 2).

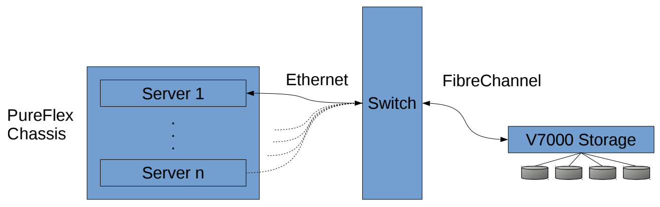


**Fig. 2.** Physical hardware configuration

In order to fulfill the first requirement, efficient use of hardware resources, we operate this hardware according to cloud-computing paradigms and offer Infrastructure-, Platform-, and Software-as-a-Service. We achieve this by running a combination of four software-components, as details in Figure 3.

The *first software-component* for our on-premise cloud infrastructure is the GPFS file system. Using the V7000 storage system, we attach a large disk-volume to all the servers in the cluster simultaneously. The GPFS file system manages and synchronizes access to this shared volume and creates the first Platform-as-a-Service offering, a shared file system.

In order to provide an Infrastructure-as-a-Service offering, we run an OpenStack cluster as the *second software-component* on some of the servers. OpenStack controls hypervisors, virtual network devices and the GPFS file system. It allows creating virtual machines with attached storage, and is able to con-

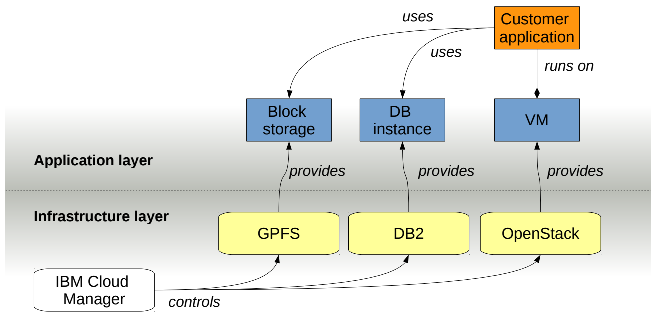


**Fig. 3.** The TPL on-premise cloud stack

nect them in arbitrary, virtualized network configurations. With these first two components we can offer virtualized and isolated environments to run arbitrary application software. The virtualized environment can be fine-tuned to fit the requirements of the specific application, and allows the efficient use of resources.

The *third software-component* is the IBM DB2 PureScale[6] relational database management system. Since we focus on customer applications for data management, we decided to offer database instances as a Platform-as-a-Service in our cloud environment. Without this database service, customer applications would not only need to contain, but also manage and optimize their own database. With this Database-as-a-Service approach, we can operate DB2 in a highly available configuration and also perform optimizations regarding the underlying hardware infrastructure.

Our *fourth software-component*, on the on-premise cloud infrastructure layer, is the orchestration system IBM Cloud Manager. This orchestration system controls the three other software components. It has an infrastructure pattern language as its input, and based on these definitions, uses the services of the other components to set up the complete application. This includes configuring the shared file system, virtual machines and networking as well as database instances and application-software installations within the virtual machines.

With the combination of these four components, we create a Software-Defined-Environment (SDE) on our test-lab infrastructure. With this SDE we can programmatically create and manipulate the infrastructure for our application software [2]. This enables us to achieve the efficient use of resources since we can maximize the utilization on multiple levels: Virtual machines, storage and database instances are all centrally managed and can be assigned to any hardware resource. Without interruption operations, resources can be added and removed to maintain optimal utilization and performance. With this capability, we completed the first step towards an on-premise cloud infrastructure.

[6] `http://www-01.ibm.com/software/data/db2/linux-unix-windows/purescale/`

# 4 Second Pillar: Transforming Abstract Models to a Concrete System Architecture

Cloud providers can offer low prices for on-boarding new customers as well as performing regular operations and maintenance, since they develop the necessary solutions once and re-use them many times. In our on-premise cloud infrastructure, we use the same approach in order to minimize the effort for setting up new applications and maintaining them. We achieve this in our solution by applying the following two concepts: first, abstract modelling and transformation, and second, reusable deployment- and maintenance-artifacts. These concepts are detailed in the following Sections 4.1 and 4.2.

## 4.1 Abstract Modelling and Transformation

The first concept of our second pillar, the abstract modelling and transformation, reduces the effort and cost for setting up new applications. Similar to public cloud offerings, with our on-premise cloud, customers can easily deploy new applications for evaluation, testing or production. Previously, these tasks involved the complicated installation and configuration procedures that drove the migration to cloud offerings in the first place.

Our abstract modelling concept uses a domain-specific language (DSL), which we develop for a specific application domain (here, Enterprise Content Management Systems). This modelling language exposes only a high-level view on the desired system to the customers. It enables them to specify functionalities and requirements without having knowledge of the actual implementation details. We also provide an interpreter for our DSL which contains expert knowledge and experience from the application domain. This interpreter chooses from a set of proven and tested system topologies, and adapts it to the specific requirements which the customer expressed using our domain-specific language. The result is an infrastructure pattern for the Cloud Manager component from Section 3. This pattern contains all the necessary information for setting up a new instance of the application. This includes virtual machine configurations with networking and storage as well as all the application-software components and their installation procedures. Figure 4 details this process.

**Example: An Abstract Pattern for Enterprise Content Management Systems.** The domain-specific language describes the requirements in an abstract way, which is independent of products and technologies. Some requirements are functional, like the requirement to add and view content over the web or to declare content as records and file them to a file plan. Other requirements are non-functional, like the number of users of the system, the amount and frequency of the documents, or the required availability. The functional requirements mandate the application to be used. For example IBM FileNet P8 with IBM Navigator for the basic functionalities, and FileNet Records Manager for records and file plans. The non-functional requirements have a huge impact

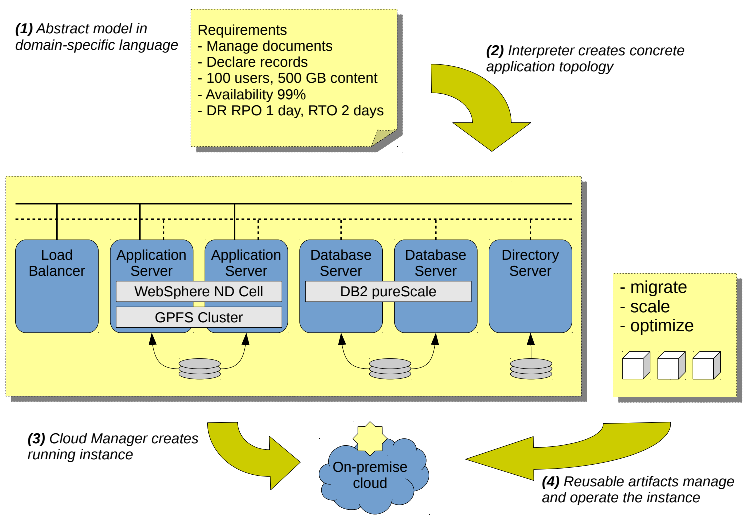

(1) Abstract model in domain-specific language
Requirements
- Manage documents
- Declare records
- 100 users, 500 GB content
- Availability 99%
- DR RPO 1 day, RTO 2 days
(2) Interpreter creates concrete application topology
Load Balancer
Application Server
Application Server
WebSphere ND Cell
GPFS Cluster
Database Server
Database Server
DB2 pureScale
Directory Server
- migrate
- scale
- optimize
(3) Cloud Manager creates running instance
On-premise cloud
(4) Reusable artifacts manage and operate the instance

**Fig. 4.** From an abstract model to a concrete system topology to a running instance

on the infrastructure and the deployment of the applications. A high-availability requirement, or an expected growth of the user base, suggests an application server clustered over multiple nodes, which can be easily extended or reduced. Similarly, the database that manages the metadata can be deployed on a single node, with an associated passive HA node, or in an active-active cluster. And the file system that stores the content can be on a share, exported by a server, or on a clustered file-system [7,6].

The application is then implemented using parameterizable templates for TOSCA or the OpenStack Heat orchestration engine. This template and the scripts it triggers are developed at the TPL and tested on the test-lab infrastructure. They perform the initial deployment of the system, from the provisioning of the nodes, networks and storage, over the installation of application servers and databases to the deployment and configuration of the applications. But operations does not end when the system is deployment. The ongoing maintenance of the system, like applying fix packs and adjusting to changed requirements, must be handled, too. This is where we apply our reusable maintenance-artifacts.

## 4.2 Reusable Artifacts for Maintenance

The second concept of our second pillar, reusable artifacts for maintenance, helps to reduce the effort and costs in operating and maintaining the application. Cloud providers benefit from the economy of scale in their large customer bases. They can offer cheap operations and maintenance per customer, because the procedures only need to be developed once, and can then be applied to all their customers. We use the same approach for our on-premise cloud applications. We develop maintenance artifacts which encapsulate common tasks like data migration, import or application scaling. Such artifacts can also be used for future software updates. We can reuse these artifacts with all customers that use the same basic application in their on-premise cloud. This is possible because all installations are generated from our interpreter (see Section 4.1) and the definitions in the domain-specific language. Therefore, we know all possible variations of the application, and our artifacts always run in a known, and well-defined environment.

## 5 Summary & Future Work

In this paper we have presented the current focus of the TPL: the on-premise cloud infrastructure and the interpretation of abstract application models. These two concepts allow us to bring the efficiency of public clouds to local, on-premise data centers. With the first pillar, the cloud infrastructure, we provide the basis for efficient and automated infrastructure management. At the same time, our second pillar, the abstract modeling and interpretation, allows the efficient management of the application software.

Currently, we work on expanding and evaluating our domain-specific language for ECM systems. In parallel, we are looking for new industry partners to extend and detail our approach for other application domains.

## References

1. Boerner, A., Lebutsch, D., Mega, C., Zeng, C.: Method for determining system topology graph changes in a distributed computing system (May 1 2014), uS Patent App. 14/054,011
2. Breiter, G., Behrendt, M., Gupta, M., Moser, S., Schulze, R., Sippli, I., Spatzier, T.: Software defined environments based on tosca in ibm cloud implementations. IBM Journal of Research and Development 58(2/3), 9:1–9:10 (March 2014)
3. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer Vienna (2014)
4. Mega, C., Waizenegger, T., Lebutsch, D., Schleipen, S., Barney, J.: Dynamic cloud service topology adaption for minimizing resources while meeting performance goals. IBM Journal of Research and Development 58(2/3), 8:1–8:10 (March 2014)
5. Mega, C., Lange, C.: Optimizing resource topologies of workload in the cloud by minimizing consumption and maximizing utilization while still meeting service level agreements. In: 44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland. pp. 873–881 (2014)
6. Mega, C., Wagner, F., Mitschang, B.: From Content Management to Enterprise Content Management. In: für Informatik, G. (ed.) Datenbanksysteme in Business, Technologie und Web. pp. 596–613. Köllen (März 2005)
7. Wagner, F.: A virtualization approach to scalable enterprise content management. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (November 2011)

# Architectural Refactoring for the Cloud: a Decision-Centric View on Cloud Migration

Olaf Zimmermann[1]
[1] University of Applied Sciences of Eastern Switzerland (HSR FHO), Oberseestrasse 10, 8640 Rapperswil, Switzerland
`ozimmem@hsr.ch`

**Abstract:** Legacy systems typically have to be refactored when migrating to the cloud; otherwise, they may run in the cloud, but cannot fully benefit from cloud properties such as elasticity. Architectural refactoring has been suggested as an evolution technique, but is not commonly practiced yet. One difficulty is that many of the involved artifacts are abstract and intangible. We therefore propose a task-centric view on architectural refactoring and introduce a template that identifies architectural smells and architectural decisions to be revisited when refactoring a system architecturally. We also outline an initial catalog of architectural refactorings that can be considered during cloud migration.

**Keywords:** Architectural Decisions, Architectural Patterns, Cloud Computing, Knowledge Management, Reengineering, Software Evolution and Maintenance

Software-intensive systems often have to be reengineered, e.g., due to unpredictable context changes and technology innovations that occur during system lifetime. Many reengineering activities affect the software architecture of these systems. Given the success of code refactoring, it is rather surprising that architecture refactoring [1] is not common yet; other cloud migration techniques have been preferred so far [2].

Our paper proposes architectural refactoring as a novel design technique to evolve architectures and enhance cloud deployability. The paper establishes a task- and decision-centric architectural refactoring template, identifies 30 cloud-specific architectural refactorings and outlines a future practice of architectural refactoring.

In our definition, an *Architectural Refactoring (AR)* is a planned and coordinated set of deliberate architecture design activities that address an architectural smell and improve at least one quality attribute while possibly compromising other quality attributes but leaving the scope and functionality of the system unchanged. An AR revisits certain Architectural Decisions (ADs) and selects alternate solutions to a given set of design problems [3]. Table 1 introduces an AR template that calls out these key elements of an AR, including the ADs to be revisited and tasks to be performed:

**Table 1.** Decision- and task-centric Architectural Refactoring (AR) template [3].

| ***AR Identification*** | How can the AR be recognized and referenced easily? |
|---|---|
| *Context* | Where (and under which circumstances) is this AR eligible? |
| *Stakeholder concerns* | Which non-functional requirements and constraints (e.g., quality attributes) are affected and/or impacted by this AR? |
| *Architectural smell* | When and why should this AR be considered? |
| *Architectural decision(s)* | Typically more than one solution exists for a given design problem. Applying an AR means revisiting one or more ADs; which ones? |

| | |
|---|---|
| *Evolution outline (solution sketch)* | Which design elements does the AR comprise of (e.g., patterns for conceptual ARs, platforms for technology ARs)? |
| *Affected architect-ural elements* | Which design model elements have to be changed (e.g., components and connectors, infrastructure nodes and communication links)? |
| *Execution tasks* | How can the AR be applied (i.e., executed and enacted)? |

Table 2 identifies an initial catalog of candidate cloud ARs in different categories:

**Table 2.** Cloud Architectural Refactorings (ARs).

| **AR Category** | **ARs (1/3)** | **ARs (2/3)** | **ARs (3/3)** |
|---|---|---|---|
| IaaS | Virtualize Server | Virtualize Storage | Virtualize Network |
| IaaS, PaaS | Swap Cloud Provider | Change Operating System | Open Port |
| PaaS | "De-SQL" | "BASEify" (remove "ACID") | Replace DBMS |
| PaaS | Change Messaging QoS | Upgrade Queue Endpoint(s) | Swap Messaging Provider |
| SaaS/application architecture | Increase Concurrency | Add Cache | Precompute Results |
| Scalability | Change Strategy (Scale Up vs. Out) | Replace Own Cache with PaaS Offering | Add Cloud Resource (xaaS) |
| Performance | Add Lazy Loading | Move State to Database | |
| Communication | Change Message Exchange Pattern | Replace Transport Protocol | Change Protocol Provider |
| User anagement | Swap Identity and Access Management Provider | Replicate Credential Store | Federate Identities |
| Service/deployment model changes | Move Workload to Cloud (use XaaS) | Privatize Deployment, Publicize Deployment | Merge Deployments (Use Hybrid Cloud) |

All of these ARs can be represented as instances of the task-centric template from Table 1; e.g., the tasks to introduce a cache include deciding on a lookup key and a cache invalidation strategy. Our current catalog does not claim to be complete; in the future, we plan to document these and additional cloud ARs with the template.

To establish ARs as effective and efficient carriers of software architecture evolution knowledge, guidelines (i.e., principles and practices) for AR capturing and sharing should be developed. These principles and practices should explain how to find good names for ARs, how to phrase architectural smells, how deeply to document ARs, how to group and link ARs, how to reference ADs and tasks (e.g., when instantiating our AR template), and how to apply ARs in agile project contexts.

An open question is how to create, share, review, apply, and revise ARs – are templates and catalogs good enough as knowledge carriers or are tools more appropriate?

## References

1. Stal, M., Refactoring Software Architectures, in: Babar, A., Brown, A.W., Mistrik, I., (eds.), Agile Software Architecture, Morgan Kaufman, 2014.
2. Höllwarth, T. (ed)., Migrating to the Cloud, http://www.cloud-migration.eu/en.html
3. Zimmermann O., Architectural Refactoring – a Task-Centric View on Software Evolution. IEEE Software, vol. 32, no. 2, March/April 2015

# "Let's migrate our enterprise application to BigData technology in the cloud" - What Does That Mean in Practice?

Andreas Tönne

NovaTec Consulting GmbH, Leinfelden-Echterdingen, Germany
andreas.toenne@novatec-gmbh.de

**Abstract.** Many enterprise applications are challenged with increasing data quantities and rates. Enterprise 2.0, Industry 4.0 and the Internet of Things generate huge amounts of data that challenge the architecture and technology of existing enterprise applications. It seems a logical idea to move such applications on a Big Data technology stack. In this paper, we summarize the architectural changes and the often unanticipated consequences for the business requirements that result from the highly distributed nature of Big Data architectures. The examples are based on an eleven man year migration project from a Java EE stack to a cloud-native microservices architecture.

**Keywords:** Microservice, Big Data, Java EE, NoSQL, Consistency

## 1 Introduction

In this paper, we summarize the migration of a Java EE based enterprise application to a cloud-ready Big Data architecture. We focus on the consequences of the architectural change for the application requirements and thus for the business stakeholders.

Migrating existing enterprise applications to a Big Data stack is certainly not the premiere application of Big Data. Usually Big Data is more likely to be associated with analytics, monitoring or as a more fashionable data warehouse solution. However with the diminishing boundaries of enterprises, driven by Enterprise 2.0[1], Industry 4.0 [1] or the Internet of Things[2], the sheer amount of data, its rate and heterogeneity mean that most enterprise applications do face Big Data already. The interesting question is, what does it take to migrate an existing enterprise application architecture to a technology stack that can handle it properly? And what are the consequences?

[1] Enterprise 2.0 is the adoption of Web 2.0 tools and concepts to the enterprise to allow for example the collaboration of employees, partners, suppliers and customers.

[2] IoT is the concept of enabling everyday physical objects to have an identity (IP adress) and communicate through the Internet. This means for instance production tools like a laser that report their health status to the service partner but also common household appliances that can be used or controlled over the internet.

## 1.1 The Migration Subject

This paper is based on a Big Data migration project that achieved its "operational" milestone with an effort of eleven man years in 2014. We cannot disclose the exact nature or name of this project as it is an ongoing product development of one of our customers. The author was the architect and development manager for the first release. The migration discussed in this paper provides the business logic and persistence layer for data integration and analysis. At this time, the product has been reviewed, used in prototypes and in medium size proof-of-concept implementations by several global enterprises. It is in production in manufacturing and is planned to be put in production for a B2C task.

The product is a semantic middleware, providing a combination of a ETL (extract, transform, load) process, a semantic analysis of the data and a Big Data scale repository for the results. Data from a variable number of sources is imported using agents in the environment of the data source and converted to a common data model. This data could be structured like a relational database record or an XML file but also unstructured data like a PDF document or plain text, music or video is imported.
Analytics algorithms discover relations between the data records based on their content. One of the simplest of such algorithms detects keyword matches using an inverted index like Elasticsearch [2]. A discovered relation could be for example "my user matches your author". More elaborate algorithms that for instance use business knowledge can be added as plugins. The relations together with statistical information like the relevance of the relation are persisted alongside with the common data model.

The product is used in the general problem area of data integration and solves expensive problems: data harmonization, schema discovery and contextual search. An application example is combining the databases of insurances after a merger. Discovering data commonalities like same customers is a very laborious and error prone process that can be automated with this product.

The original middleware was developed using best practices and a common layered monolithic architecture[3] based on Java EE clustering, a relational database, a full text index for the keyword-related analysis algorithms and a distributed cache to share global statistical information across the cluster nodes. The solution was well received by industry prospects but unfortunately it was not able to deliver the required scaling both in terms of data amounts and speed. The solution has to deliver its business value for huge data amounts (hundreds of terabytes and more) with a multitude of sources and formats that could not be handled effectively with traditional data integration designs. In other words, our customer tried to deliver a Big Data solution on a Java EE technology stack.

[3] An architecture where the functional components like user interface, business logic and persistency are part of a single self-contained application. Dependencies between the internal components are expressed by conceptually arranging them in layers where the high layers may use the lower layers but not vice versa.

Interestingly the inability of the technology stack to scale to the data rates and sizes turned out to be the lesser problem. The more challenging problems that were revealed by the Big Data paradigm change lay in the traditional atomic operations and consistency focused requirements that effectively prohibited scalability by design. We will discuss these in section 3.2.

## 2 Big Data to the Rescue

Big Data is commonly associated with two technology innovations: NoSQL databases, e.g. Cassandra [3] and map-reduce, e.g. Apache Hadoop [4] that are designed for massively concurrent computation. However Big Data is not just a set of technologies but a diagnosis of demanding growing application requirements. These are the three V's in Big Data parlor:

- Volume
- Variety
- Velocity

The volume of data to handle increases rapidly for many enterprises. Modern digital strategies embrace the Internet as the platform of the enterprise and this means the number of data sources and thus the related data volume is growing. But the data volume is not only growing through new data sources. Enterprises also want to utilize already existing data that traditional enterprise applications cannot handle and that is ignored.
If an enterprise increases the number and range of data sources, the variety of data formats also increases. Especially weakly structured or unstructured data like office documents, plain text, multi-media, web pages, Web 2.0 information snippets like tweets, comments, blog posts et cetera are characteristic for a Big Data situation.
Finally the velocity of data changes and the requirement to process these more rapidly in analytics distinguish Big Data from traditional data warehouse approaches.

Forresters definition of Big Data is "The practices and technology that close the gap between the data available and the ability to turn that data into business insight." [5] Big Data technologies were developed to handle one or more of these V's. The Big Data ecosystem is rapidly changing and growing and covers a wide range of problems. Common characteristics are the focus on horizontal scalability[4] and the preference for flexibility over data schemas. Typical areas of Big Data technology are:

**Batch processing**: The map-reduce algorithm that is implemented by Apache Hadoop allows to distribute batch processing tasks over an almost unlimited number of distributed computational nodes. The only requirement is that the tasks can be imple-

[4] A computational system is scaled horizontally by adding resources through new appliances. It literally means to put a new computer next to the existing computers on the table. Alternatively a computational system is scaled vertically if it is replaced by a more powerful system.

mented without synchronization between the concurrent executions. Otherwise the required scalability cannot be achieved by Amdahl's law[6]. Map-reduce was originally developed by Google to handle their massive data processing needs. Apart from Hadoop there are several implementations of map-reduce with special characteristics like Apache Spark [7] that delivers super fast in-memory map-reduce.
**Streaming**: Near real time processing of large data quantities  can be achieved with Apache Storm [8]. One business case for Big Data streaming is the online security check and rating of credit card transactions.
**Analytics**: The task to discover business value opportunities in Big Data and its implementation is called Data Science. Data Science implementations are based mostly on statistical methods like the Bayesian statistics and other machine learning approaches. These can be fully automated like IBM Watson [9] or based on frameworks like Apache Spark ML [10].

The Big Data business values seem to be rather comparable to the characteristics of enterprise applications: large quantities of business data being handled concurrently for a large number of users, integrating data sources and finally supporting the enterprise business goals. However Big Data is a game changer for business.

### 2.1 Paradigm Changes

Big Data is called disruptive for business [11, 12] with a huge transformational power. This does have substantial consequences for enterprise applications; the requirements aspect of this is shown in this paper. Since enterprise application are materializations of business goals and strategies and since business needs to reinvent these in the light of Big Data, it should be obvious that enterprise applications for Big Data scenarios cannot be the same as before.
But in our consulting experience, business stakeholders are allured by the apparent similarities of their traditional data driven architectures and Big Data. It is a common mistake to think of Big Data as simply the extension of their enterprise application architectures that is able to handle more data.

One paradigm change of Big Data that needs to be implemented is the data import itself. Enterprise applications, driven by the problem of sheer data masses, are very selective in what information they actually import and process. Information is filtered for the applications business functionality. Not only is a lot of information lost but a lot of information can also be duplicated if several applications need parts of the same data. Master data management solutions try to centralize these duplications, providing a single source of reference to structured business data.
Big Data to the contrary yields the highest value if no processing, filtering or structuring takes places in the import. The paradigm of Big Data is to offer future business value in today's data. To achieve this, one of the best practices for Big Data is to collect as much data as possible in its raw format in a data lake [13]. A data lake is the schema free storage of original data. The database of a data lake is very often HBase [14], the database used by Hadoop since this allows to analyze the raw data without

prior export. A data lake is filled with all data without any processing like filtering. The promise of Big Data analytics is that it is possible to find valuable correlations in this heap of data with a reasonable effort. In contrast, business intelligence based on a data warehouse requires expensive ETL steps to extract the needed information from the productive databases and store it in the data warehouse.

Another paradigm change that is important for this paper concerns the inherent conflict of consistency versus scalability.

## 2.2 Scalability is the New Ruling Requirement

Consistency is the holy grail of business stakeholders. Data must be accurate, at any time. This is of course an important requirement if customer, production or legal information is concerned. The author would not approve if Amazon uses statistical methods to compute the approximate sum of bills. On the other hand, there are plans at Amazon to use Big Data methods to forecast the customers' needs and ship in advance. In an enterprise, especially with huge data masses and concurrent changes, not everything can be computed 100% accurate and does not need to. Sometimes it is admissible to achieve consistency at a later time and sometimes approximations are good too.

Many applications of Big Data are only possible if scalability becomes the new holy grail of requirements [15] and to that end consistency is necessarily limited. Consistency in a distributed system requires a consensus between the distributed nodes. Such consensus with for example the Paxos protocol [16] limits the horizontal scalability. In practice, a compromise between scalability and consistency has to be achieved.
In our migration example, consistency had no business value if the import and analytics ran unacceptably long. Massive concurrent computation like map-reduce with Hadoop critically depends on horizontal scalability. This scalability can only be achieved if the processed data has a high locality and as part of that, the processing does not entail side effects for the other map and reduce nodes. Map-reduce does not scale if the nodes need to synchronize in any way.

## 2.3 Consistency Challenges

Distributed replicated storage, which is one of the core strategies of Big Data, is often associated with "eventual consistency" [20]. A distributed system is eventually consistent when after a data change at one node there is an unbounded period of time in which the version of the changed data at the other nodes is undefined. The system is inconsistent until the data has been distributed to all nodes.

In a distributed environment, the CAP theorem [17, 18] gives us two practical choices for storing new data[5]. Either we prefer consistency at the expense of scalability and make new data only available after it has been replicated safely on all database nodes. Or we prefer accessibility at the expense of consistency and make the new data available at the rate it is replicated between database nodes. We choose eventual consistency for the paramount importance of scalability in our application. In this case, an enterprise application may yield different (inconsistent) results depending on the time it accesses a particular database node.

This choice of consistency or accessibility does not need to be that strict in practice. NoSQL databases like Cassandra offer various strategies [19] to choose C or A as needed on a per case basis. A common option is to use a quorum of replication nodes that need to be written to before a write is considered complete. This assures data consistency even with a limited failure of nodes. The decision for only eventual consistency of many early NoSQL databases was excused with the CAP theorem but actually it is justified by the better scalability of weaker consistency.

Eventual consistency is aggravated if multiple distributed storages are used. Rather common is the combination of a NoSQL database like Cassandra and a full text index like Elasticsearch. Writes to Cassandra are at least immediately visible at the node we committed to. Data written to Elasticsearch is not immediately visible after a commit, even at the node where the commit occurred[6]. We may observe inconsistencies also between the databases at the same node.

Typical Big Data applications using NoSQL storage and map-reduce follow a write fast, process later pattern. Once the data is written to the data lake, it is rarely changed. An enterprise application on the other hand typically does change its data like customer records, shipping lists, material bills or reports frequently. As we point out below, keeping these changes consistent is the reason for a lot of synchronization and locking. If we omit this to allow scalability at Big Data scale, the time window of inconsistencies is getting much larger. The window of eventual consistency of a NoSQL database replication is typically 10-100 milliseconds to replicate the data in the cluster. With an unsynchronized enterprise application service, this window spans a much longer period: over the concurrent execution of two conflicting service calls, through the database replication and possibly a following correction. This may take several seconds where the inconsistency can cause a lot of damage to business data and decisions.
Migrating an existing enterprise application logic to Big Data has to deal with these paradigm changes and consistency challenges.

[5] In a Big Data scale distributed system we have to expect partitioning at any time. Thus according to the CAP theorem we can only choose either consistency or accessibility in conjunction with partition tolerance.

[6] This is due to the replication strategy of Elasticsearch that involves bulk updates

## 3 Migration Strategy and Consequences

The author started to work with the software company as a consultant when the migration to Big Data technology was inevitable. We started with an in-depth performance and architecture review. The developers had reasonable ideas how to improve their solution and Big Data and Cloud deployment seemed to be mandatory. The changes suggested to us were however not founded on an analysis of the problems with performance and usage data. It appeared the changes were suggested since "the technology looked promising". Big Data projects should never be founded on a technology choice but on business requirements and value. A key takeaway of the migration project was that Big Data is also disruptive for the way we think about requirements and their implementation design. We will discuss this in examples below.

The architecture review was conducted by inspection of the system documentation, a detailed code review and several interview sessions. The review revealed a design for the data import and analysis logic that is rather typical for Java EE. Concurrency control of the import service was delegated to stateless session beans. The import logic was built on several steps like persistency, indexing and analysis that could be rather complex and time consuming. A common persistency layer performed the translation of the data model to a relational database.

The performance analysis was conducted with inspectIT [21], which records time and trace data of Java applications. It showed that in high load situations the server was spending a substantial part of the import time for commits and index updates. This was a serious performance problem independently of the scaling limits of the Java EE stack itself. The reason for this ineffective behavior lied in the business requirements for the import logic. The holy grail of these requirements was "quality of relations" which requires atomicity of the import services to assure consistency. Assuming such atomicity and thereby evading difficult decisions about concurrent computation is rather typical for the business stakeholder in enterprise applications. Java EE and ACID databases taught the stakeholder that atomicity of operations is a reasonable requirement.

Quality of relations could for example be compromised by

- Duplicate relations between two records that are imported in parallel. If they have related contents, they both might write the same relation. E.g. one will write "my author matches your user" and the other will write "my user matches your author". Since such relations are commutative, they are considered a duplicate and an error.
- Deviations of unsynchronized statistical information (e.g. word counts) on the nodes that lead to errors in the relation annotations

The usual ways to achieve atomicity of services are locks, synchronization, uniqueness constraints and very short transactions to reduce commit conflicts. Each is a scalability killer. Measures on a test cluster showed that acquiring a database lock in

Cassandra could take up to 40 milliseconds, due to the necessary consensus between the database nodes. In other words, the throughput of the import was limited to 25 records per second if a database lock is required.

## 3.1 Architectural Changes

The new holy grail of the Big Data based solution is scalability. It does not matter how good the results of the data import are if you cannot achieve them in any reasonable time or if a combinatorial explosion of the computed data exceeds the capacity of the system.

Re-architecting the solution was driven by the goal to achieve scalability in a cloud-native architecture [22]. This means to tailor the architecture towards deployment in the cloud, allowing dynamic changes to its topology of services as needed. Changes to the resource allocations for the data import as well as the other APIs may be needed for a time-of-day or season reason or simply because the mixture of data to import has changed. This lead to a decomposition of the previously monolithic server into independently deployable server components:

1. A database cluster using a NoSQL database together with Elasticsearch for indexing purposes
2. Client API services to access the data model
3. **Import services to receive new or changed data records and process them using analysis plugins**
4. Content agents that are located near the data sources and push changes to the import API
5. Periodic services for cleanup or consistency repairs using Hadoop

The architecture blueprint for the import services was based on the Staged Event Driven Architecture (SEDA) [23]. SEDA is considered one of the predecessors of the microservices architecture that is popular for implementing cloud-native applications. One may say the import service has a microservices architecture borrowing some elements of SEDA like the coupling of services with queues.

A microservice [24] is a standalone self contained implementation of one business capability in one or more matching services. It is independently deployable and uses typically lightweight communication protocols like REST. A microservices architecture is built as the collaboration of several of such small services. It is the radical opposite to a monolithic application architecture that hides the business capability services internally. There is some dispute about the relation of microservices and SOA as discussed in [24]. A SOA service is implemented to break up monolithic architectures and allow better and cheaper integration of applications. Microservices are in our opinion more focused on the project and engineering issues, allowing very small independent projects for the services that are delivered and operated at their own pace.

Each microservice component in our architecture is called a Stage and stages are connected in a defined order by persistent queues. The import workflow is represented by events that carry the data records and further control information that is updated by the stages. The routing of events is controlled by the stages themselves. In terms of enterprise integration patterns, this is the pipes and filter pattern [25].

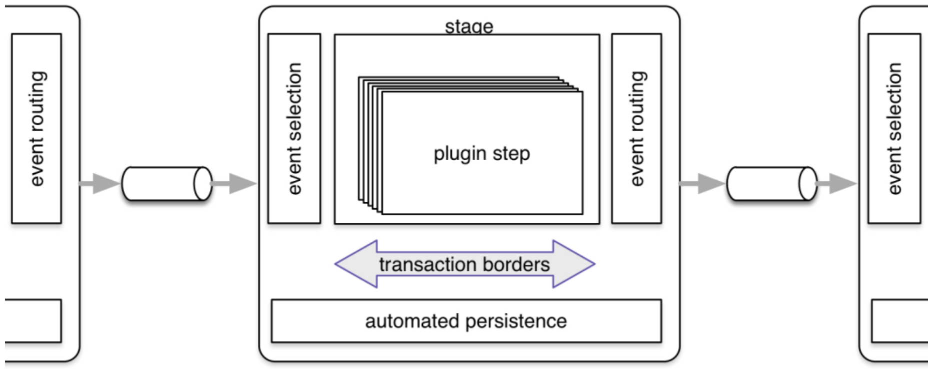


**Fig. 1.** Stage

We choose this model to reflect the fact that the import and analysis plugins do have a defined execution order, have varying resource requirements and may be based on different technologies. We needed a lot of flexibility to setup the import- and analysis chain.
The plugin logic itself might need to be decomposed into steps that take place at different times of the import in different stages. For example, a first step early in the import process to setup base data and statistics and a second step later in the chain of steps to do the actual processing. For this, plugins implement one or more step functions that executed in sequence form the plugin logic. The plugin steps are executed concurrently in a stage using a thread pool. They are purely functional and delegate the persistency of their results to an automated persistence framework of the stage.

This overall architecture allowed us to

1. Group plugin steps by their resource needs
2. Adapt the number of instances of each stage to the current needs
3. Achieve the best possible data locality
4. Use different technology in the stages
5. Have a single source of truth for transaction optimization
6. Arrange the steps in the import process such that concurrency related inconsistencies are minimized

Grouping of steps in a stage allows to control the configuration of the cloud VMs according to the CPU or memory needs. Some of the algorithms are CPU intensive, some are I/O bound waiting for the database while others need lots of memory for their computation. A one size fits all strategy (e.g. putting all plugins in one VM and deploying multiple such VMs) is likely a waste of resources. Being able to deploy a flexible number of such stages also helps to adjust the resource allocation and optimize the throughput of the overall network of stages.

Optimizing for data locality of the database caches in the stages is an optimization strategy that was recommended by the database vendor consultants. Steps that are working on the same type of data should have fewer database cache misses and cache overflows. We could not verify a measureable positive effect of this strategy for the first release though. Our assumption was that our high change rate worked against the database caching strategy.

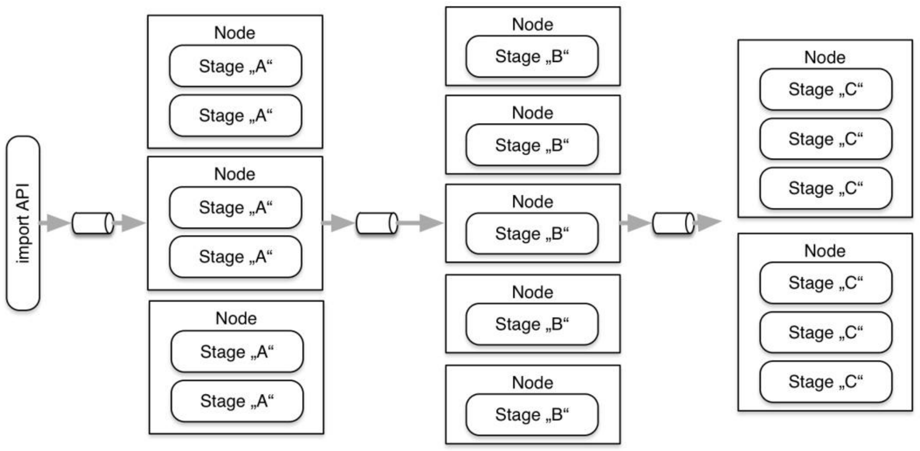


**Fig. 2.** Simple stage / server node topology example without routing

The core stages and most plugins were developed using the same platform technology (Spring Boot [26] on Java SE). We also wanted to be able to include sophisticated technology from third-parties like natural-language processing libraries that could be implemented differently. For these, stages can be also implemented in other languages. The interface contract of stages is only messages carrying events.

This new architecture has proven to be very flexible and scalable while adding only a negligible overhead to the pure plugin execution time. This flexibility came at the expense of a laborious and error prone configuration and deployment process. In the end, each micro service deployment was an individual VM with its individual Spring configuration like queue names or plugin options. A production ready version will have to use Cloud deployment automation processes and technologies. At this time, the product development team is using Ansible Tower [27] with good success.

## 3.2 Consistency Constrained Business Requirements

The fine granular decomposition of the import service into highly concurrent stages and steps puts a spotlight on the consistency focused business requirements. Achieving best quality by enforcing atomicity of the import services was an illusion to start with. Even a strictly sequential execution of the import (one stage with only one thread) will produce different results at different times. The computation of relations is based on global information and only altering the order of the import events in the queue will result in slightly different relations and statistical annotations. Quantifying the analysis quality turned out to be extremely hard. To the business stakeholders, the best strategy for up keeping the quality was to insist on as much constraints and synchronization as possible.

As outlined above, eventual consistency is an intrinsic property of concurrent distributed computing. We cannot avoid short phases of inconsistency without sacrificing scalability. These phases of inconsistency result in noticeable quality defects as illustrated by the following example.

Consider a data source adding two related records, e.g. a technical specification and a construction plan that have a relation based on mentioning the same standards name. These two records shall be imported as updates concurrently and the import will likely happen in different stages on different nodes. There are chances for the following errors:

1. Import and analysis of each record does not see the new contents of the other record. The relation on the commonly mentioned standards name is missed.
2. Since import and relation analysis are separate steps with separate transactions per stage, both records analysis may see the added other record. But each analysis might not see the relation created by the other analysis due to transaction isolation. Without uniqueness constraints on the database, we will get duplicated relations between the two records.
3. Other concurrent imports do not see the added records at the same time due to the replication latency of the NoSQL storage. This will produce uncontrollable errors to the relations created by the other concurrent imports.
4. Delays in the propagation of the full text index will introduce further errors of concurrent analysis steps since their computation may be based on a mismatch of record contents and its outdated index entries (or vice versa, depending on which database was faster).

Most of these inconsistencies must be allowed for some time and compensated by for example Hadoop correction runs. These re-run the analysis of records that were imported in the same time window or clean up duplicate entries. Depending on the frequency of these correction runs, we have to accept longer timeframes of inconsistencies.

Be prepared for lengthy discussions with the business stakeholders that their convenient illusion of atomic operations cannot be kept up in a highly concurrent Big Data solution. The need for scalability will not only affect the existing service algorithms but also their functional properties if the service requirements would have a need for locking or synchronization.

At the time of writing this paper, all of the previous analysis algorithms of the solution were replaced by fundamentally different algorithms with new properties. Some are still based on a "select candidates, filter and persist results" pattern. Others are using predefined index functions or stochastic algorithms.

## 3.3 Data Accuracy

Enterprise applications often need global aggregated information like tallies or statistics. Even for a clustered Java EE server with a clustered database, this can be too time consuming to compute.

Because of the large database sizes and the distributed topology of nodes, this is even more expensive in our case. Consider for example the need to know the number of occurrences of a word in all records.

The NoSQL database Cassandra provides a counter column type [28] that allows concurrent delta changes without the need for locks. These counters are maintained as a journal of changes until the next read access, when the journal is consolidated into the current counter value. This is an expensive operation, especially when the counter value changes frequently (which was the case in our application). One strategy we evaluated was to cache the counter values locally for a duration and then refresh the value from the counter. This requires an error approximation how long the stale counter cache is acceptable which is usually a function on the database size.

Another valid strategy that is also implemented by Elasticsearch for its cardinality aggregation involve the HyperLogLog algorithm [29, 30]. This algorithm approximates such counts with a known precision that is dependent on the memory allocated for its hash sets.

In our opinion, one should always prefer such approximating algorithms with a known error rate over periods of stale data with an unknown error. Unfortunately, be prepared that not mathematically educated stakeholders will have a preference for the latter. They prefer the risk of eventual errors over a constant error rate, even if the eventual errors could be much larger.

## 4 Conclusions

Given the potential disruptive changes to the business requirements, one might question if the migration of traditional enterprise applications to a Big Data stack in the cloud is a good idea in the first place.
However the development of data rates and the increasing demands to include larger numbers of data sources beyond the enterprise boundaries do not give the enterprise much of a chance.

The question is not ***if*** to move to Big Data but ***when and how***?

The initial questions in the introduction were "What does it take to migrate to the technology stack that can handle it (Big Data) properly? And what are the consequences?". From the experience of our project we conclude that it takes a complete re-evaluation of the application requirements about what is still possible and reasonable and what not. And the consequence is likely a completely new application.

You should not try to migrate an existing application to Big Data by simply replacing its persistency layer but you should modify the business capabilities backed by the enterprise application to reflect Big Data and then implement that business' new requirements.

## References

1. Bundesministerium für Bildung und Forschung: Zukunftsprojekt Industrie 4.0, http://www.bmbf.de/de/9072.php, April 9, 2015.
2. Elasticsearch BV: Elasticsearch | Search & Analyze Data in Real Time, https://www.elastic.co/products/elasticsearch, April 9, 2015.
3. Apache: Welcome to Apache Cassandra, http://cassandra.apache.org, April 9, 2015.
4. Apache: Welcome to Apache Hadoop, https://hadoop.apache.org, April 9, 2015.
5. Khatibloo, Brian Hopkins and Fatemeh. Reset On Big Data. Report, Forrester Research, Inc. (2014).
6. Amdahl, G.M., "Validity of the single-processor approach to achieving large scale computing capabilities". In: AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference, pp. 483-485. ACM, New York (1967)
7. Apache: Spark Lightning-fast Cluster Computing, https://spark.apache.org, April 9, 2015.
8. Apache Storm, https://storm.apache.org, April 9, 2015.
9. IBM: IBM Watson, http://www.ibm.com/smarterplanet/us/en/ibmwatson/index.html, May 10, 2015.
10. Pentreath, N.: Machine Learning with Spark. Packt Publishing (2015).
11. Accenture Analystics; Big Success with Big Data. Report, Accenture (2014).
12. McKinsey Global Institute: Disruptive technologies: Advances that will transform life, business, and the global economy. Report, McKinsey Global Institute (2013).

13. Fowler, M.: DataLake. http://martinfowler.com/bliki/DataLake.html, February 5, 2015.
14. Apache: Welcome to Apache HBase, http://hbase.apache.org, May 10, 2015.
15. Vogels, W.: Availability & Consistency. Presentation at QCon 2007.
16. Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM 21, pp. 558-565. ACM, New York (1978).
17. Brewer, E.: Towards Robust Distributed Systems. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC '00), pp. 7- 10. ACM, New York (2000).
18. Brewer, E.: CAP Twelve Years Later: How the 'Rules' Have Changed. Computer Vol. 45, pp. 23-29. IEEE, New York (2012).
19. DataStax: Configuring Data Consistency. http://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html , April 9, 2015.
20. Vogels, W.: Eventually Consistent. Communications of the ACM 52, pp. 40-44. ACM, New York (2009).
21. NovaTec: inspectIT ...because performance matters!, http://www.inspectit.eu, May 10, 2015.
22. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns. Springer, Wien (2014).
23. Welsh, M., Culler D., Brewer, E.: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In: SOSP'01 Proceedings of the eighteenth ACM symposium on Operating systems principles, pp. 230-243. ACM, New York (2001).
24. Fowler, M.: Microservices, http://martinfowler.com/articles/microservices.html, May 10, 2015.
25. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley, Indianapolis (2012).
26. Pivotal Software: Spring Boot, http://projects.spring.io/spring-boot/, May 10, 2015.
27. Ansible: Ansible Home, http://www.ansible.com/home, May 10, 2015.
28. DataStax: What's New in Cassdra 2.1: Better Implementation of Counters, http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters, May 20, 2014.
29. Heule, S., Nunkesser, M., Hall, A.: HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In: EDBT '13 Proceedings of the 16th International Conference on Extending Database Technology, pp. 683-692. ACM, New York (2013).
30. Flajolet, P. et.al.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: AOFA '07: Proceedings of the 2007 International Conference on the Analysis of Algorithms. DMTCS (2007).

# Cloudiator: A Cross-Cloud, Multi-Tenant Deployment and Runtime Engine

Jörg Domaschka, Daniel Baur, Daniel Seybold, and Frank Griesinger

University of Ulm, Institute of Information Resource Management,
Albert-Einstein-Allee 43, 89081 Ulm, Germany
{joerg.domaschka,daniel.baur,frank.griesinger,
daniel.seybold}@uni-ulm.de
http://www.uni-ulm.de/in/omi

**Abstract.** In recent years cloud computing has reached tremendous attention and adoption in both academia and industry. Yet, still vendor lock-in constitutes a severe threat to the actual idea of cloud computing which is to adapt quickly to changing requirements and conditions. This paper addresses these shortcomings and presents Cloudiator, a cross-cloud, multi-tenant tool to deploy applications and redeploy them using different algorithms and heuristics.

## 1 Introduction

In recent years cloud computing has reached tremendous attention in both academia and industry. This evolution has led to a state where the cloud paradigm has reached the mainstream of software development and application operation.

Nevertheless, many issues still have to be considered as unresolved. In particular vendor lock-in and limited auto-scaling capabilities are considered the most pressing and most limiting aspects of cloud computing today [2]. Vendor lock-in avoids an easy migration from one cloud provider to another. It also avoids the parallel use of multiple cloud providers and establishes a technical barrier between operators and providers. Initiatives and standards such as TOSCA [13] go beyond multi-cloud operation in the sense that they support *cross-cloud operation*, i.e. running a single application across multiple cloud providers. On the down-side, they use a deploy-and-forget approach where the deployment of an application is not re-considered after it has been deployed once. Regarding auto-scaling, several approaches exist. Yet, they lack the possibility to define *complex adaptation scenarios* and mostly sacrifice control possibilities for simplicity [10].

This paper addresses these challenges and presents Cloudiator[1], a cross-cloud, multi-tenant tool. In addition to a mere cross-cloud deployment, it supports the `model@runtime` paradigm [12] and also allows redeployment capabilities as well as support for automatic as well as manual adaptations. Regarding auto-scaling support, Cloudiator goes beyond basic threshold-based approaches. In sum, it is a stepping stone for higher-level functionality such as reasoning and optimisation algorithms.

[1] https://github.com/cloudiator/

This document is structured as follows. The next section presents an overview and introduces the terminology used in this paper, describes basic features of CLOUDIATOR, and presents the main system layers. Section 3 sketches the user interactions with the platform, while Section 4 details the individual components and gives an outlook on implementation aspects. Finally, we discuss related work, before we conclude the document with an outlook on future work.

## 2 Overview

In the following, we clarify the terminology on clouds and applications that we use in the remainder of this document. We then introduce CLOUDIATOR's main features as well as the core components and their interplay. We refine the overall architecture in Section 4.

### 2.1 Terminology and System Model

*Cloud platform* refers to a software stack and accordingly the API offered by that stack (e.g. OpenStack Juno). A *cloud provider* runs a cloud platform under a dedicated endpoint/URL (e.g. RedStack). A *cloud* refers to a cloud platform offered by a cloud provider as seen by a tenant. That is, besides the endpoint of the provider, *a cloud* (in contrast to *the Cloud*) is also linked to log-in credentials.

A *cloud application* or *application* for short is a possibly distributed application consisting of multiple interlinked application components. As such, an application is solely a description and does not represent anything enacted. An *application component* or *component* for short is the smallest divisible element of an application. It is the unit of scale and the unit of failure. For illustration consider a blog application that may consist of the three components load balancer, application server together with a servlet, and a database. The *lifecycle handlers* of a component define amongst others how to install, configure, and run the application component, but also how to detect its failure, and how to stop it. Section 3.1 discusses the handlers supported by CLOUDIATOR.

The deployment of an application results in an *application instance*. An application instance for application $A$ is linked to at least one *component instance* $ci^j_{A,i}$ for each component $C_{A,i}$ that belongs to the application. Component instances are created through the lifecycle handlers associated with the respective component. Each component instance is run on particular virtual machine $VM_k$ and multiple component instances can be mapped to the same virtual machine.

Components may be connected with each other using directed *channels*. Connecting two components with a channel imposes that at least one component instance from the source component will interact with at least one instance from the target component. The concrete wiring between the source and target instance is subject to both the deployment and the scaling.

## 2.2 Features, Use, and Access

Cloudiator is a multi-user-capable Web-based software service. Its features can be separated into registries, deployment functionality, automatic adaptation, and the specification of monitoring requirements.

**Registries** The data stored in the registries lay the ground for the management, access to, and comparison of cloud providers. They are essential for the deployment and monitoring features. So far, Cloudiator contains four different registries: *(i)* The *cloud registry* stores offerings of cloud providers. This includes the type of cloud platform offered, the data centres and availability zones offered by that provider, the virtual machine types (flavours) and operating system images available at each of these system levels. Additional geographical location information can be attached to each data centre. *(ii)* The *specification registry* stores abstract properties of cloud providers. This includes generic virtual machine specifications consisting of #cores and amount of RAM. This registry, also supports an operating systems hierarchy that for instance states that `Ubuntu 14.04` belongs to the `Ubuntu` family which in turn belongs to the class of `Linux` operating systems. The entries of the specification registry are linked to these of the cloud registries where applicable. *(iii)* The *credential registry* holds cloud access credentials for each user of Cloudiator needed to access the cloud providers. The kind of credentials stored vastly depend on the underlying cloud platform. *(iv)* The *component registry* contains components which can be assembled to applications. These applications can then be instantiated (cf. Section 3). Each of the registries is multi-tenant, meaning that its entries are bound to a Cloudiator tenants.

**Deployment** The deployment part of Cloudiator features the capability to transform applications specified in the component registry to application instances by providing a *deployment specification* that defines which application shall be deployed and further specifies how many instances of a particular component shall be set up. Further, the deployment description specifies how the component instances shall be grouped on virtual machines and how these virtual machines are supposed to be configured. A deployment specification may be either abstract by specifying hardware requirements such as #cores and an operating system, or concrete by referencing actual flavours or images of a dedicated cloud provider. More detailed information on deployment specification is subject to Section 3.2.

**Adaptation** Adaptation describes the capability of the application to autonomously evolve under changing conditions such as system load. This may be needed when more users access the application instance than anticipated by the deployer. In a cloud environment, the most frequent reaction to such events will be a scale out/in of individual components or groups of components or the scale up/down of individual virtual machines or groups of virtual machines. For that reason, CLOUDIATOR supports an auto-scaling functionality based on the *Scalability Rules Language (SRL)* [6, 10]. In order to realise this functionality, CLOUDIATOR enables the specification of (hierarchical) metrics, conditions on these metrics, and actions to be executed when the conditions are fulfilled. It is important to note that these rules do not have to be provided with the deployment description, but can be added and changed while an application instance is running. The adaptation interface is described in Section 3.3.

**Monitoring** In addition to monitoring information that is collected and evaluated for the adaptation functionality, a user can specify further monitoring requirements. Here, he defines sensors that collect the necessary data and instructions that define how these raw metrics shall be aggregated to higher-level metrics. The data collected there is provided to the clients of CLOUDIATOR via its API. The monitoring interface is also discussed in Section 3.3.

### 2.3 System Components

CLOUDIATOR not only deploys cross-cloud applications on various virtual machines on various clouds, but also installs system components on these virtual machines. Hence, besides the CLOUDIATOR service that is used as an access point for clients (CLOUDIATOR's *home domain*), the platform consists of a distributed system of components. These components are controlled and orchestrated by the components of the home domain as sketched in the following paragraphs.

Once an application is to be deployed, the *Deployment Engine* is responsible for allocating the virtual machines in the multi-cloud environment through the cloud providers' APIs. It also lays the ground for the installation of the components, by installing a *Lifecycle Agent* on each of the allocated virtual machines.

The surveillance components are concerned with monitoring the virtual machine as well as component instances running on them, but also with storing and post-processing the monitoring data. Monitoring data is collected on the virtual machines; aggregation takes place partially on the individual virtual machines and partially in the home domain (cf. Section 4.5). The adaptation functionality makes use of the same monitoring and aggregation system, but additionally comes with an evaluation mechanism that triggers adoptions of the deployment graph of the current application instance when rules are satisfied.

The home domain also stores monitoring data requested by the user as well as information about executed scaling actions and provides this information through the CLOUDIATOR API.

## 3 Cloudiator Interfaces

This section describes the interface of CLOUDIATOR in a simplified manner. The actual system is Web-based and offers a fine-grained REST[2] as well as a browser-

[2] REST is an architecture principle, that describes the transfer of the representational state of an object to be used as a generic interface. This is implemented in HTTP.

based interface. In the following, we represent the external user input coming to the system in coarse-grained JSON-like format. We do not intend to present the entire interface of the platform, but rather to give an impression on what kind of data to exchange for what purpose. The input covers applications, components and lifecycle handlers; deployment plans; and monitoring as well as adaptation.

### 3.1 Applications, Components, and Lifecycle

As described in Section 2.1, deployment happens on the level of applications which in turn consist of components. Components are connected via channels. Deployment has two basic tasks: *(i)* instantiate the individual components in the correct order. This includes the provisioning of the binaries, their configuration, as well as running them. *(ii)* wire component instances.

```
{”name”: ”LB”,
  ”lifecycle” :{
    ”start” :..., ”configure” :..., ...
  },”ports” :[
    {”name” :”out”, ”type” :”out”, ”card” :”1+”}
  ]
}{”name” :”ghost”,
    ”lifecycle” :{...}
    ”ports” :[
    {”name” :”in”, ”type” :”in”, ”card” :”*”},
    {”name” :”out”, ”type” :”out”, ”card” :”1”}
    ]
}
```

**Listing 1.1.** component description

```
{”name” :”blog”,
  ”components”:[
    {”name”: ”DB”, ”order” :1,},
    {”name”: ”LB”, ”order” :1,},
    {”name”: ”ghost”, ”order” :2,}
  ],
  ”wiring” :[
    {”in” :”DB.in”, ”out” :”ghost.out”},
    {”in” :”ghost.in”, ”out” :”LB.out”}
  ]
}
```

**Listing 1.2.** application description

**Component Description** A component description defines a set of lifecycle handlers that describe how to provision the binaries of the component, how to configure it, and how to run it. Other handlers capture the shutdown of this application instance. The lifecycle concept of Cloudiator is heavily influenced by Cloudify[3] and CloudML [7]. The lifecycle can be specified as script files, command line instructions, Chef scripts[4], or Java commands. In addition to that, Cloudiator supports two special handlers: The *start detector* serves the purpose of detecting whether an application has started successfully. It is run after the component instance has been started and is used to determine when it is ready for wiring other instances. Once an application is considered running, the *stop detector* is invoked periodically in order to figure out whether the application has accidentally stopped. Beside lifecycle handlers, a component description defines open ports, that other components can use. Further, it defines

[3] `http://getcloudify.org/`
[4] `https://www.chef.io/chef/`

ports the component will consume from other components. For both, incoming and outgoing ports the cardinality of connections can be defined.

Listing 1.1 shows two components of a three-tier blog application consisting of a load balancer, the ghost blog implementation and a database. The load balancer has a single outgoing port that can connect to several targets. The ghost component has an incoming port that accepts connections from various sources and an outgoing port that targets a single target, the database.

**Application Description** An application description builds a component graph by connecting incoming and outgoing ports of components. Currently, CLOUDIATOR requires that the deployment order of components be specified in the description. Listing 1.2 presents the application description of above blog application. It wires the load balancer with ghost and ghost with the database.

Putting this example to a real set-up, e.g. with an `nginx`-based load balancer, `node.js` as an application server running the `ghost` blog application, and a mySQL database, one faces the situation that the load balancer has to be reconfigured for each new instance of the application server. This is realised by a dedicated lifecycle handler on the load balancer component that is invoked whenever a new down-stream component has been started.

### 3.2 Deployment Interface

The deployment interface is responsible for triggering the deployment of an application within the cross-cloud environment (cf. Section 2.2). Using it, the user specifies the number of virtual machines he wants to start and which application component instances shall be placed on them. To allow isolation between the components of different tenants, each virtual machine can only host component instances of the same tenant (but of multiple applications). For the example from Section 3.1, the user could place one instance of the `nginx` load-balancer on one virtual machine, two instances of the `node.js/ghost` component on two separate machines and the `mysql` database on yet another machine.

For each virtual machine the user has to provide a detailed *deployment specification* depicting which resources will be used to start the virtual machine. For this specification the user has two possibilities: he specifies the concrete resources used (cf. Listing 1.3), meaning that he has to provide the unique identifiers for virtual machine type, image and location. The other possibility is to define abstract resource requirements (cf. Listing 1.4) such as #cores, operating system and constraints, e.g. *should run in Ulm* (geographical requirement).

### 3.3 Scalability Interface

CLOUDIATOR adopts the *Scalability Rules Language (SLR)* [6, 10] for supporting auto-scale. By providing the necessary modelling concepts it gives the developer the ability to specify behavioural patterns and scaling actions. A rule-based approach is supported because of the intuitiveness and simplicity. Yet, SRL still

```
{
  "cloud":"openstack−ulm",
  "flavour":1, # m1.small flavour
  # trusty−server−amd64
  "image":"6cba2fe...",
  "location": {
          "region" :"uni−ulm",
          "av−zone" :"campus−west"
   }
   "instances" :2,
   "application" :"app_name",
   "components" :["comp1", "comp2"]
}
```

**Listing 1.3.** concrete deployment

```
{
  "geolocation":"ulm"
  "hardeware":{
    "cores":1,
    "ram":2048,
  },
  "operatingSystem":{
    "vendor":"ubuntu",
    "version":"14.04",
    "architecture":"amd64"
  },
  ...
}
```

**Listing 1.4.** generic deployment

supports the definition of complex metric-based conditions to trigger an event-driven processing. In order to specify a scalability rule, the user has to define metrics based on raw sensor data and aggregations on these metrics. Aggregation happens based on mathematical functions and time. Listing 1.5 specifies that for all instances of the ghost application server (cf. Section 3.1) CPU load shall be monitored and that an average over a 10-minute window shall be computed. It further computes the average on all of these 10-minute averages.

On each of the metrics (aggregated or raw) the user can then define threshold-based conditions, and further combine conditions using Boolean operators. Any condition or group of conditions can be linked to an action defining what shall happen in case this (group of) condition(s) is satisfied. Possible actions are scaling up/down/out/in. Listing 1.6 presents a scale out action to be triggered when any of the 10-minute averages is above 80% and the global average is above 60%.

## 4 Cloudiator Realisation

This section describes the approach of Cloudiator on dealing with cross-cloud deployment and adoption during runtime. It gives a more detailed description on the components which were introduced in the overview section (cf. Section 2). The entities of the home domain are also summarised in Fig. 1.

### 4.1 Registries

Cloudiator contains a built-in discovery mechanism that enables providing the data for the registries (cf. Section 2.2) in an automated way: Whenever a user registeres credentials for a cloud provider the Cloudiator discovery mechanism will fill the cloud registry for that cloud provider. In addition, it will connect this information to the specification registry with the abstract cloud properties.

As it may not be possible to retrieve all information via the providers' APIs, Cloudiator allows the manual completion of entries. In the future, we plan to rely on using meta-information provider such as CloudHarmony[5]. These provide

[5] `https://cloudharmony.com/`

```
{
 "sensors" :[
   {"name" :"CPU", "type" :"system.cpu", "interval" :"1s"}
 ],

 "metrics": [
   {"name" :"raw_cpu", "scope" :"blog.ghost.EACH", "type" :"raw", "sensor" :"CPU"},
   {"name" :"avg_cpu", "scope" :"raw_cpu.EACH", "type" :"compute",
                                              "params" :["AVG", "10min", "raw_cpu"]},
   {"name" :"avg_global", "scope" :"SINGLE", "type" :"compute",
                                              "params" :["AVG", "avg_cpu.ALL"]}
 ]
}
```

**Listing 1.5.** specification of sensors and aggregation

```
{
   "rule" :{
    "condition" :["AND",
            ["avg_cpu.ANY", "GT", "80%"]
            ["avg_global", "GT", "60%"]
        }
        "action" :["SCALE_OUT",
          {"scope" :"component",
          "target" :"ghost"}
        ]
   }
}
```

**Listing 1.6.** specification of conditions and scalability

additional information such as the actual geographical locations of the cloud provider-specific location (either region or availability zone).

## 4.2 Deployment Engine

In order to deploy component instances on a virtual machine, the Cloudiator Deployment Engine uses a three-step approach: *(i)* It determines the deployment configuration of the virtual machine. In case, the user has not defined a concrete deployment, this includes to determine the cloud provider to use; the region and availability zone; and the image. For abstract specifications (an operating system instead of an image, a geographical location instead of a cloud provider) the Deployment Engine tries to match this to exactly one deployment plan. In case, multiple possible deployment plans exist, it rejects the deployment. Sophisticated algorithms for reasoning about deployment plans can be plugged in to our platform. Alternatively, they can be realised on top of it as all information from the registries (cf. Section 2.2) is accessible through the Cloudiator API.

*(ii)* It acquires the virtual machine and installs essential Cloudiator components on it. For that purpose it accesses the cloud provider API to start a virtual machine with the deployment configuration, to assign a public IP address to this machine, and to configure network and firewall rules. The imple-

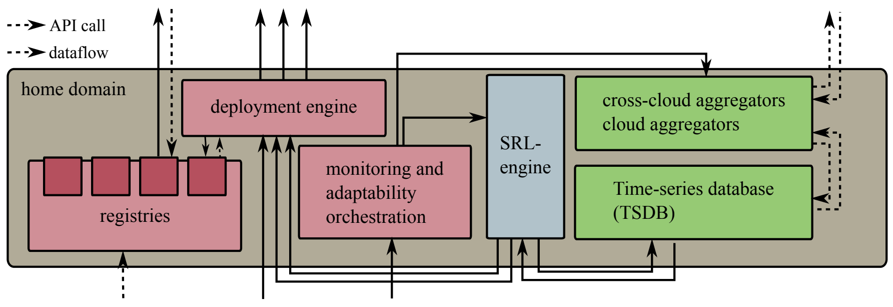

API call
dataflow
home domain
deployment engine
registries
monitoring and adaptability orchestration
SRL-engine
cross-cloud aggregators cloud aggregators
Time-series database (TSDB)

**Fig. 1.** Components of and interactions from the home domain.

mentation makes use of the Apache `jclouds` library[6] and only provides an own implementation for cloud platforms and features not supported by jclouds.

*(iii)* Once the virtual machine has been acquired and started, the Deployment Engine deploys and starts the lifecycle (cf. Section 4.3) and monitoring agents (cf. Section 4.4) as well as the infrastructure for processing the monitoring data (cf. Section 4.5). The installation of the application itself, and hence the execution of its lifecycle actions, is then handed to the lifecycle agent. Access to the scalability and monitoring API is relayed to the monitoring infrastructure.

## 4.3 Lifecycle Handling

The lifecycle agent has the primary task to read a component specification, reserve a separate space for it on the virtual machine, and to start the component by running the lifecycle handlers. For each started component instance, the lifecycle agent will run the stop detector and report a failure to the home domain in case the instance has stopped unexpectedly. The lifecycle agent also interacts with other lifecycle agents for informing them that an instance has been created and ports have been wired. Currently, the agent isolates component instances on its virtual machine only by providing an own directory to each of them. For the future, we envison applying Docker containers[7] to improve isolation.

## 4.4 Monitoring Agent

The monitoring agent produces raw monitoring data through a set of installed sensors. A user can install sensors by defining monitoring requirements or scalability rules through the CLOUDIATOR interface (cf. Section 4.5). The Deployment Engine will then forward the installation request to the Monitoring Agent of the specific virtual machine or to all virtual machines in case multiple of them are affected from a single interface access. The monitoring agent contains a set of well

[6] `http://jclouds.apache.org/`
[7] `https://www.docker.com/`

known sensors for measuring system parameters such as CPU and RAM utilisation as well as I/O rate. Each sensor can be configured to have a dedicated interval for which the data shall be collected as well as a measurement context that defines whether the monitoring shall capture the entire virtual machine or only a particular component instance on that virtual machine.

In addition, the monitoring agent offers an interface which component instances running on that virtual machine can access in order to report own component-specific metrics, e.g. the number of active users connected to this component instance. Finally, the monitoring agent forwards all monitoring data collected to the aggregation and rule processing sub-system (cf. Section 4.5). Cloudiator currently supports two different implementations of this sub-system: one based on a *time-series database (TSDB)* [8] and one based on complex event processing. It is the tenant that decides which sub-component to use.

### 4.5 Monitoring and Aggregation

As discussed throughout the document, there are two scenarios where aggregation on monitoring data is needed: *(i)* A user has requested that monitoring data be collected and aggregated such that it is available outside Cloudiator. *(ii)* Scalability rules require that data be aggregated to build higher level metrics on which rules can be applied. In both cases, we use the same chain of mechanisms to provision the requested information.

Cloudiator currently supports two approaches how to actually perform the aggregation. One uses a time-series database to temporarily buffer the values and uses the TSDB's built-in aggregation functionality. The second approach applies complex event processing techniques for computing the aggregations on the fly without explicitly buffering the intermediate values. This section sketches the architecture of both realisations after having presented the general approach.

**General Approach** Section 4.4 clarified when and how monitoring data is collected by the monitoring agent. For aggregating this data, it is necessary to *(i)* ensure that all data needed for aggregation is available, *(ii)* define where the aggregation is performed, and *(iii)* specify where to put the results.

The basic approach that we use for both sub-systems is that the monitor agent forwards the data to a collection component running on the same virtual machine. This is responsible for making the monitoring data available to the aggregation functionality including relaying the data to multiple locations if necessary. The aggregation itself is implemented by aggregation processors. The exact way how the aggregators are implemented depends on the underlying strategy. Yet, the requirements with respect to the location where the aggregation is actually executed can be considered independent from the implementation:

We distinguish between three scopes that define where the aggregator shall be run: The *host* scope considers aggregation tasks that take into account only values from a single virtual machine. In that case, the collector will forward the data to a virtual machine-local aggregator to do the aggregation. Afterwards, the aggregator will relay the resulting data to the collector again. The *cloud scope* deals with data from multiple component instances or virtual machines from within one cloud. Finally, the *cross-cloud scope* defines aggregation on data from different clouds. For that scope, aggregation happens in the home domain. We use a dedicated collector for each of the scopes.

(Aggregated) monitoring data whose collection was requested by the user is always stored in a TSDB in the home domain. It is accessible via the Cloudiator API. Improvements on aggregator locations are subject to ongoing work.

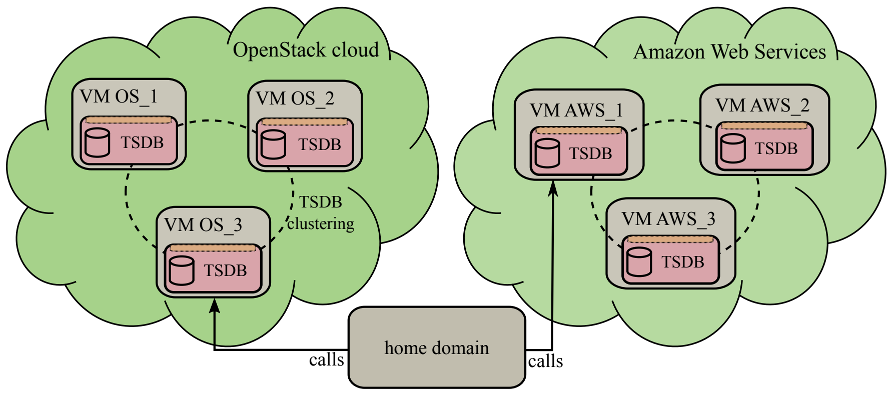


**Fig. 2.** TSDB-based architecture of Cloudiator with the local areas marked orange

**Time Series Database** In the TSDB-based approach for aggregation, we reserve a small fraction of each virtual machine (e.g. 10%) for buffering monitoring data. This strategy assumes that the amount of monitoring data increases linearly with the number of virtual machines. At the same time using a cluster avoids that the TSDB becomes a bottleneck when scaling the application. The reserved area is split into a local storage area and a shared area for replicas of data items created on other virtual machines on the same cloud. The current implementation uses KairosDB[8] to access the two storage areas.

Each data element added to KairosDB is stored in a local Couchbase[9] instance using the in-memory memcached option representing the local area and in a distributed Cassandra datastore [11] representing the shared area. Fig. 2 shows the set-up of the underlying storage systems. It is noteworthy that each cloud uses its own distributed storage. This set-up avoids that the storage suffers from large latencies and that additional costs incur for inter-cloud traffic.

With respect to aggregation, KairosDB plays the role of the collection component. Each virtual machine runs an aggregator that processes Host aggregations

[8] `https://github.com/kairosdb/kairosdb/`
[9] `http://www.couchbase.com/`

and stores the results back into KairosDB using both the local and the shared areas. Aggregations with a cloud scope are triggered from the home domain. Yet, as the aggregation takes place using the Cassandra store, only the results are sent to the home domain. The results of such a process are stored back to the cloud using any KairosDB instance and the shared area. In case this data has to be made available to the user, a further aggregator is running that pulls this data from any of the cloud's KairosDB instances and stores it in the KairosDB instance running in the home domain. Finally, aggregations in the cross-cloud scope are also run through an aggregator in the home domain, but then stored in the KairosDB in the home domain. Higher-level metrics working on this data will read from the home domains KairosDB and store results back there.

**Event Processing** The second approach is based on the usage of the event processing framework Riemann[10]. Here, no datastore is used, but monitoring data is aggregated on the fly. Data needed for processing, e.g. for computing average values, are kept in memory on the machines actually performing the aggregation. For this implementation, the collector component splits the monitoring data as follows: *(i)* Data to be processed in the host scope, is processed locally according to the aggregation function. *(ii)* Data to be processed in one of the other scopes is relayed to the home domain. *(iii)* Other data is dropped.

In contrast to the TSDB-based approach, data in a cloud scope has to be transmitted to the home domain; at the same time, the results of this computation will not be sent back to the cloud, but kept in the home domain for computing higher-level metrics. They may also be stored in the KairosDB instance, in case they have been requested from the users.

## 4.6 Enacting Auto-Scale

Cloudiator's auto-scaling capabilities (cf. Section 2) for individual components of an application instance require the aggregation of metrics and the evaluation of conditions on these metrics. The generation of monitoring data and their aggregation has be discussed in Section 4.4 and Section 4.5.

In order to evaluate the conditions on the metrics, we apply the strategy to consider conditions on metrics as binary metrics by themselves. These metrics take values in $\{0, 1\}$ and their value is computed as a function that compares the values of the source metric against the threshold of the condition. Composite conditions are computed from their source conditions.

The SRL-engine is an aggregator-like component that runs in the home domain. Its sole task is to check for all conditions that causes an SRL-related action whether the conditions are satisfied. In case it is, the SRL-engine triggers the actual actions at the Deployment Engine. In addition, it stores the fact that this action has taken place as a separate metric value in the KairosDB instance in the home domain so that it can be queried by higher-level components. Whenever

[10] `http://riemann.io/`

the application instance has changed either by manual intervention or by running a scaling action, the SRL-engine adapts the scalability-related configuration if necessary. This is for instance the case whenever the abstract rule description requires conditions to take into account all instances of a particular component.

### 4.7 Intended Use

Cloudiator can be used in two ways by the end users (cf. Section 3). First, requirements regarding the virtual machines can be specified in a cloud-independent way relying on Cloudiator's simple reasoning functionality.

The second approach is to specify in a fine-grained way which virtual machine flavours shall be used on what cloud and with what image. In this case, it is desirable to combine Cloudiator with a more powerful reasoning mechanism and even together with a modelling approach. This is the way Cloudiator is integrated in the overall system architecture of the PaaSage[11] platform that applies a CAMEL model to a multi-step reasoning process that eventually yields a deployment plan; monitoring and aggregation rules; and scalability rules. CAMEL itself is a combination of several Domain Specific Languages, including CloudML [7], Saloon [16], CERIF [9], and SRL [6, 10].

## 5 Related Work

Multi-cloud libraries such as Apache libcloud[12] and Apache jclouds aim at harmonising the different cloud APIs by providing a programming library offering a common interface. A slightly different approach is used by Apache $\delta$-cloud[13] and rOCCI[15] which offer a standardised web interface for calling multiple cloud APIs. Both approaches lay the foundation for using multiple cloud providers, but do not help to fully utilise this environment. Cloud API standards like CIMI[5] or OCCI[14], define a common interface for the provisioning and handling of cloud provider offers, but lack widespread support.

Multiple DevOps tools such as Chef, Puppet[14] and Ansible[15] use a proprietary low-level Domain Specific Language for describing the deployment of applications and also support the provisioning of virtual machines on clouds. Cloud modelling approaches and their execution engines such as CloudMF[7] and TOSCA (OpenTOSCA)[4] use a modelling language for describing the application as well as the dependencies between application components and deploy the model to the cloud environment. Yet, these tools mainly target provisioning and initial deployment and lack features for efficient runtime adoption like monitoring and scaling.

[11] http://www.paasage.eu/

[12] https://libcloud.apache.org/

[13] https://deltacloud.apache.org/

[14] http://puppetlabs.com/

[15] http://www.ansible.com/home

PaaS tools wrapping IaaS functionality such as Cloudify, Apache Stratos[16], Apache Brooklyn[17], and SeaClouds[18] additionally support basic runtime adaptation by using load-balancers and auto-scaling. However their scaling is limited to specific metrics and thresholds. The same holds true for provider specific auto scaling solutions like Amazon's CloudWatch[19].

In earlier work [3], we presented a preliminary version of CLOUDIATOR lacking monitoring and adaptation aspects.

## 6 Conclusions and Future Work

In this paper, we have introduced a system architecture for dealing with the challenges given by a cross-cloud environment. We have implemented this architecture with our CLOUDIATOR platform, that particularly addresses the demands imposed on adaptability and reconfigurability of cross-cloud applications. The first prototypical implementation of our tool is an initial step towards a platform that enables the deployment and runtime adaptation of application instances across multiple clouds.

Future work targets an evaluation of our current approach. Furthermore, in order to ease access to our platform by third-party tools, we target the implementation of adapters that map other APIs, e.g. for TOSCA [13] or CloudML [7], to the CLOUDIATOR interfaces. Regarding the specification of applications, the lifecycle we proposed for applications and application components needs to be validated and evaluated for its practicability. Regarding the deployment, the additional usage of PaaS offerings needs to be considered. Concerning the aggregation functionality, extensive performance evaluations between our two subsystems are required in order to figure out when to use which implementation. With respect to the TSDB-based implementation of the aggregation and scaling mechanism, the performance of the underlying storage backends needs to be further evaluated and our choice of a mixed storage system be validated. We also target a comparative analyses of InfluxDB[20], once a stable version has been released. We also envision using the monitoring data in order to better understand the needs of the application and heterogeneity of the cloud offerings. This leads to better fitting deployments.

**Acknowledgements** The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 317715 (PaaSage).

[16] `http://stratos.apache.org/`
[17] `https://brooklyn.incubator.apache.org/`
[18] `http://www.seaclouds-project.eu/`
[19] `http://aws.amazon.com/de/cloudformation/`
[20] `http://influxdb.com/`

## References

1. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification (WS-Agreement). Tech. rep., Open Grid Forum (March 2007)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (Feb 2009)
3. Baur, D., Wesner, S., Domaschka, J.: Towards a Model-based Execution Ware for Deploying Multi-Cloud Applications. In: Proceedings of the 2nd International Workshop on Cloud Service Brokerage September 2014 (2014)
4. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA - A runtime for TOSCA-based cloud applications. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 8274 LNCS, pp. 692–695 (2013)
5. DMTF: Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol (2013)
6. Domaschka, J., Kritikos, K., Rossini, A.: Towards a Generic Language for Scalability Rules. In: Proceedings of CSB 2014: $2^{nd}$ International Workshop on Cloud Service Brokerage (2014 (To Appear))
7. Ferry, N., Chauvel, F., Rossini, A., Morin, B., Solberg, A.: Managing multi-cloud systems with CloudMF. In: Solberg, A., Babar, M.A., Dumas, M., Cuesta, C.E. (eds.) NordiCloud 2013: 2nd Nordic Symposium on Cloud Computing and Internet Technologies. pp. 38–45. ACM (2013)
8. Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., Breivold, H.P.: Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. In: 2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014. pp. 602–609 (2014)
9. Jeffery, K., Houssos, N., Jörg, B., Asserson, A.: Research information management: the CERIF approach. IJMSO 9(1), 5–14 (2014)
10. Kritikos, K., Domaschka, J., Rossini, A.: SRL: A Scalability Rule Language for Multi-cloud Environments. In: Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on. pp. 1–9 (Dec 2014)
11. Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (Apr 2010)
12. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ Run.time to Support Dynamic Adaptation. Computer, IEEE 42(10), 44–51 (2009)
13. OASIS: Topology and Orchestration Specification for Cloud Applications Version 1.0 Committee Specification Draft 08 (2013)
14. Open Grid Forum: Open Cloud Computing Interface - Core (2011)
15. Parák, B., Šustr, Z.: Challenges in Achieving IaaS Cloud Interoperability across Multiple Cloud Management Frameworks. In: UCC 2014: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. pp. 404–411 (2014)
16. Quinton, C., Haderer, N., Rouvoy, R., Duchien, L.: Towards multi-cloud configurations using feature models and ontologies. In: Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds. pp. 21–26. MultiCloud '13, ACM, New York, NY, USA (2013)

# Design and Implementation Issues of a Secure Cloud-Based Health Data Management System

Frank Steimle[1], Matthias Wieland[1], Bernhard Mitschang[1], Sebastian Wagner[2], and Frank Leymann[2]

[1] Universität Stuttgart, Institute of Parallel and Distributed Systems, 70569 Stuttgart, Germany
firstname.lastname@ipvs.uni-stuttgart.de
http://www.ipvs.uni-stuttgart.de/

[2] Universität Stuttgart, Institute of Architecture of Application Systems, 70569 Stuttgart, Germany
firstname.lastname@iaas.uni-stuttgart.de
http://www.iaas.uni-stuttgart.de/

**Abstract.** eHealth gains more and more interest since a lot of end-user devices which support health data capturing are available. The captured data has to be managed and securely stored, in order to access it from different devices and to share it with other users such as physicians. The aim of the German-Greek research project ECHO is to support the treatment of patients, who suffer from Chronic Obstructive Pulmonary Disease (COPD), a chronic respiratory disease. Usually the patients need to be examined by their physicians on a regular basis due to their chronic condition. Since this is very time consuming and expensive, we develop an eHealth system which allows the physician to monitor patients conditions remotely, e.g., via smart phones. Therefore, a secure data processing and sharing eHealth platform is required. In this paper we introduce a health data model and a corresponding platform-architecture for the management and analysis of the data provided by the patients. Furthermore, we show how the security of the data is ensured and we explain how the platform can be hosted in a cloud-based environment using the OASIS standard TOSCA, which enables a self-contained and portable description and management of cloud-services.

**Keywords:** eHealth, mHealth, cloud data, data analysis, security

## 1 Introduction

The usage of eHealth (electronic health) technologies for improving the health care gains more and more attention in recent years. Providing health applications on mobile phones, is often referred to as mHealth (mobile health) and builds on top of eHealth infrastructures. As more and more mobile health applications running on devices like smart watches, fitness bands or smart phones get available, the need for processing and storing the produced health data increases. Here specialized systems have to be developed for eHealth solutions that provide advanced data security and consider regulations given in the health care domain.

In the ECHO project we develop an eHealth system that is used to monitor patients, who suffer from *Chronic Obstructive Pulmonary Disease* (COPD). COPD is an obstructive lung disease characterized by chronically poor airflow worsening over time. The most noticeable symptoms are shortness of breath, cough, and sputum production. Tobacco smoking and air pollution are the major causes. The purpose of the health services delivered by the ECHO-Platform is to improve quality of life and reduce exacerbations of COPD patients. The methodology applied for COPD is described as follows: The patient provides answers to predefined questions on a daily basis, such as: *Did your shortness of breath increase?* or *Did your cough increase?* The answers to these questions are stored in the ECHO-Platform and are made available to the physician. The condition of the patient is automatically analyzed and notifications are generated for the patient and the physician. Furthermore, the physician can send recommendations or treatment advises to the patient. For further imformation, please refer to [4].

To enable such an eHealth system an underlying *active Health Data Management System* (aHDMS) is needed that securely stores the health data, analyses them and provides the notifications. We derived design and implementation requirements for an aHDMS and considered them during the realization of the ECHO-Platform.

1. **Data privacy and security:** Health data are sensitive data and have to be protected well. Hence, security is a key issue in an aHDMS. The aHDMS has to secure the data with technologies like encryption, secure authentication and access control, SQL injection prevention, data input verification, and fine grained access control based on tuples (see Section 3 and Section 4).
2. **Support of various user roles:** Different roles of users have to be supported to separate patients from physicians (see Section 4).
3. **Different access channels:** The users have to be enabled to access the aHDMS with an end-device of their choice. Hence, both native applications for mobile devices and web-based applications have to be supported by the aHDMS (see Section 3). The data presentation has to be tailored for each end-device accordingly. Furthermore, the aHDMS has to support the parallel access of multiple users at the same time without side effects.
4. **Support of multiple hospitals:** In order to adapt the aHDMS to different hospitals the aHDMS has to support multi-tenancy, that is it is easy and fast to configure (see Section 5).
5. **Cloud readiness:** For a fast operational readiness of the aHDMS automatic management and deployment of the aHDMS has to be provided with cloud tooling (see Section 5).
6. **Easy development of applications:** To support the development of new applications using the aHDMS a central interface that aggregates the access to all components in a uniform API, e.g., by offering REST operations, should be provided. Furthermore, tool support is needed for providing a good API

documentation, a test bed for query execution, input verification and test sets. This is needed to ensure a good code quality with less bugs and vulnerabilities (see Section 3).

7. **Scalability:** The aHDMS has to be scalable in order to provide a good user experience even if many users access the aHDMS at the same time. In the long run, the data managed by the aHDMS is constantly growing. Hence, also scalability regarding the data size has to be ensured (see Section 3).
8. **Automatic health data analysis and active behavior:** The data provided by the patient has to be analyzed in order to generate notifications about changes or issues that could lead to exacerbations (see Section 4).
9. **Data quality:** The data quality has to be managed in the aHDMS and has to be considered in the applications. Issues in this area have been described in [16] and have to be implemented by the aHDMS (see Section 4).
10. **Extensible service-based architecture:** The definition of new complex services in the aHDMS should be supported based on orchestration of existing services like Health Services, which manages the patients' data, or Analytic Services, which analyze the patients' data (see Section 5). These orchestrations could be realized with BPEL flows.

All these issues have to be considered in an implementation of an aHDMS. The main contribution of this paper is to exemplary show how to deal with these issues by describing how our prototype–the ECHO-Platform–meets these issues. The focus of this paper is the data management and the implementation of the platform. The frontend applications to access the ECHO-Platform as end user, e.g., as patient or physician, are not described in further detail. The paper is structured as follows: In Section 2 the related work is presented. Then, we show the overall architecture of the ECHO-Platform in Section 3. Here, we show how the ten requirements introduced above are solved. Furthermore, we explain the secure API for accessing the ECHO-Platform. In Section 4 the database model and methods to secure the data are presented. Section 5 shows how to make the ECHO-Platform cloud-ready. Finally, Section 6 gives a summary of the paper.

## 2 Related Work

Several studies have shown that mobile technologies can be used to monitor patients in a cost-efficient manner. In [9] a mobile assisted home care model is developed in order to monitor and manage COPD conditions of patients at home. The system is a mobile application that allows patients to report their COPD symptoms. Furthermore, a web portal allows physicians to manage and analyze patients data and give feedback when necessary. A probabilistic model to automatically assess the risk of COPD aggravation based on collected symptoms is presented in [10]. The ECHO system has a similar goal but differs considerably in the system architecture. Our system architecture is clearly separated into a platform for health data management and the frontends for the patients and physicians. The advantage is that the frontend and backend can be separately designed, implemented, used, secured, and deployed. This provides a higher degree of flexibility. Another capability of the ECHO-Platform is to support analytic data processing combined with active behavior by means of cloud computing technologies.

There are many cloud-based systems available for storing electronic medical records in health data models which can be accessed trough the Internet. OpenMRS [15] is a web-based, open source electronic health platform. It is developed and supported by a worldwide community and based on Java. OpenMRS provides a good look and feel and is easy to use [5]. It is focused on documentation of consultations and hence only supports accounts for physicians and is not supposed to be used by patients or users themselves. The aim of the ECHO-Platform is to provide access to different user roles and for multiple purposes, where the first application domain is COPD. Casisis [1] is similar to OpenMRS–an open source, web-based, patient data management system–which focuses on management of cancer data. Nosh [8] is another open source system, web-based system and provides the look and feel of a desktop application in the web-browser. It supports accounts for patients and physicians and it supports communication between the actors, e.g., by providing an online calendar where patients can arrange their appointments with the physician. None of these open-source systems do support all issues we identified, but they could be extended since the source code is available. However, they are tightly coupled to a certain use case and provide frontends exclusively for that use case. Furthermore, they do not provide a unified API and therefore no third-party vendors can implement additional applications using their existing patient data. Last but not least the related systems do not provide support for automatic provisioning as a cloud-service in different cloud environments. The systems have to be installed manually. Only the Nosh "In The Cloud" offering provides an Amazon EC2 cloud image that can be instantiated in the Amazon Cloud but not in a local cloud environment.

Commercial cloud-based health data systems like Microsofts HealthVault [2] allow end users to create accounts to store health and fitness data for themselves and their family. This data can be shared with others (e.g., a physician). Being closed source and installed in a public cloud system the patients and physicians have no control over their health data. In contrast, the ECHO-Platform can be deployed as a cloud service in private cloud environments, e.g., in the infrastructure of a hospital. This is important for data security reasons and flexibility.

## 3 Active Health Data Management System Architecture

In the following we describe the overall architecture of the ECHO-System (see Fig. 1) first. Hereby we detail on the Health Data Management Layer (ECHO-Platform) which implements an *active Health Data Management System* (aHDMS). Then we focus on how to interface the Health Data Management Layer via RESTful API techniques.

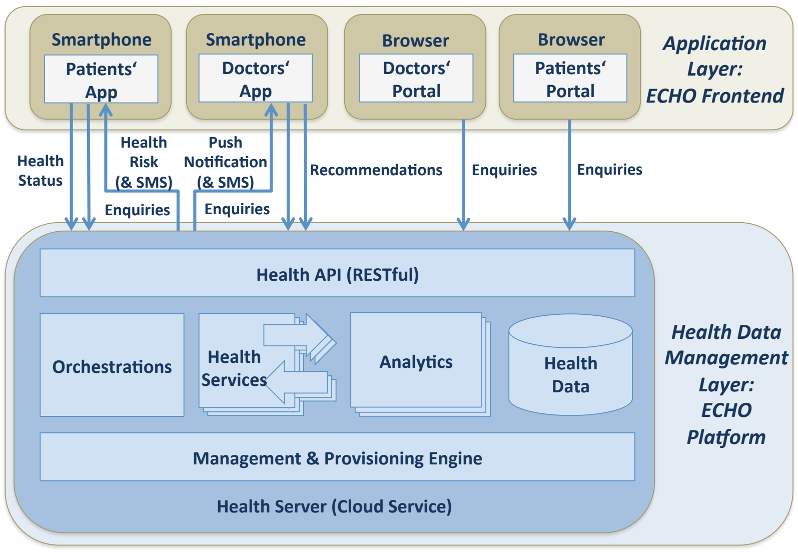


**Fig. 1.** Architecture of the ECHO-System Prototype

## 3.1 Overall Architecture

In this section, the architecture of an aHDMS that satisfies the issues described in Section 1 is presented. Regarding Issue 5 the architecture has to be separated into a cloud-based platform that stores the health data and an easy to install frontend. Our prototypical implementation of the aHDMS–the ECHO-Platform–is shown in Fig. 1 and is based on our previous work [4]. As mentioned above, it consists of two layers: an "Application Layer" and a "Health Data Management Layer".

The "Health Data Management Layer" represents the "Health Server", which consists of different components. The "Health Server" can be modeled as a TOSCA [14] cloud service. TOSCA stands for *Topology and Orchestration Specification for Cloud Applications* and is an OASIS standard that enables a self-contained and portable description of applications and their management functions. Using the OpenTOSCA platform this cloud service can be automatically deployed in any supported cloud environment (e.g., OpenStack or Amazon EC2). This is needed because of Issue 4 and 5, since this enables the automatic configuration and deployment of the "Health Server" for multiple hospitals. The "Health API" provides a unified access for the "Application Layer". This is important for Issue 5. This API follows the *Representational State Transfer* (REST) architectural style and simplifies the development of new applications by providing a HTTP based REST interface. Furthermore, the REST-style API enables the scalability of the aHDMS by using self-contained and stateless messages that can be load-balanced between different servers. Therefore, using REST for implementing the Health API supports Issues 5 and 7. “Health Data” stores both, the patients’ health data (like insurances, prescriptions, or examination results) and physicians’ recommendations. Since data quality is vital concerning health data, Issue 9 has to be followed. Furthermore, since health data are very sensitive the system has to prevent unauthorized data access and has to be encrypted. Therefore, the user roles `doctor` and `patient` are introduced. Users with the role `doctor` can only access patients data if the patient is assigned to them. Furthermore, users with the role patient can only access their own data. This supports Issue 2. “Health Services” manage patients’ data and are made available to the “Application Layer” via the “Health API”. Since the system should notify the patient and the physician if the condition of the patient declines, “Analytic Functions” are needed. These “Analytic Functions” should automatically analyze the data that is passed to the “Health Services”. Existing Analytic Functions are implemented using event-based function calls. In future work, we may integrate data mining tools for advanced analytic functions. Hence, the “Analytics Functions” support autonomic and active behavior, i.e., Issue 8. Both “Health Services” and “Analytics Functions” can be composed by “Orchestrations”. Therefore, it is easy to build new complex services just by combining existing ones. This leads to an easily extendable service-based architecture that supports Issue 10. All mentioned components are managed by the “Management and Provisioning Engine” OpenTOSCA.

The “Application Layer” includes mobile applications for patients and physicians as well as web portals which can be used to access and edit data.

### 3.2 RESTful Interfacing the Health Data Management Layer

The Health API is the interface between application layer and cloud environment. It can be used to store data and query the system. To simplify application development the Health API follows the REST architectural style. In contrast to e.g., SOAP, RESTful APIs can be easily integrated because of their unified interface. The use of REST helps solving Issue 6.

The resources used by the Health API are depicted in Fig. 2. There are so-called collection resources such as accounts or patients which pool all resources of this type. Using the collection resources one can receive all items of this collection via HTTP GET, whereas a new element can be created by using HTTP POST. An item can be retrieved using HTTP GET, updated by HTTP PUT and deleted by HTTP DELETE command.

The following resources need to be exchanged between the application layer and the health data management layer:

**Accounts:** This resource holds all account-specific information. This includes username, password, and notification settings, like notification mode and notification time. The user roles (and therewith the access rights) are also associated with an account.

**Notifications:** The notifications resource holds the history of notifications for the current user. Please note that there is no single notification resource, as no altering of notifications is intended.

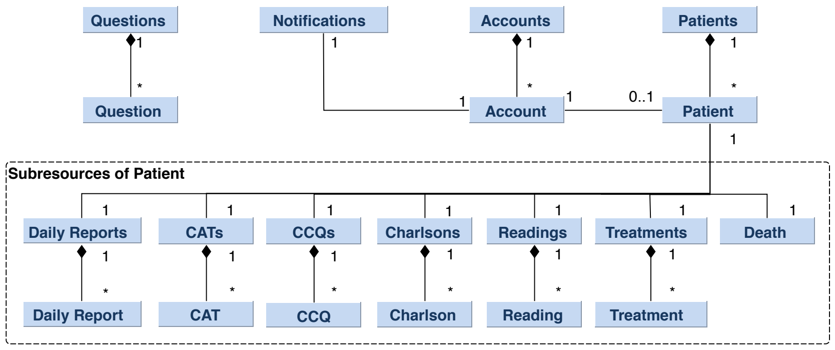


**Fig. 2.** Resources of the Health API

**Patients:** The basic patient data like name, sex, birthdate, or address and the responsible physician is stored in the patients resource.

**Subresources of a Patient-Instance:** Besides basic patient data physicians also need data like prescriptions or examination results. This data could be also saved in the patients resource, but this leads to a patients resource which gets bigger over time and this would be very traffic intensive at some point in time. Another possibility is to introduce stand-alone resources for prescriptions or examination results. But since we think of a resource as a document, which should contain only the needed information, this would also not be feasible. A physician does not want to see all the prescriptions of all patients he did in the last weeks but of a specific patient. To support this requirement, we decided to introduce subresources. This means that a collection of a subresource can only be used in the context of a specific patient. The following subresources are available:

- "Daily Reports" contains all reports a specific patient has given. A single resource contains the answers to the questionnaire and some measurements like temperature or heart rate, if the patient submitted these values.
- To treat patients physicians often need access to illness-specific data. In case of COPD physicians need access to data like the *COPD Assessment Test* ("CAT") and the *COPD clinical questionnaire* ("CCQ"). CAT and CCQ are standardized questionnaires, which describe how the patient is influenced by the disease in his daily life. They consist of several questions and the result is computed based on the answers given.
- The "Charlson" Index describes how likely it is that the patient dies because of other diseases.
- The "Readings" subresource contains all measurements a physician has determined in one examination.
- Physicians can also use the system to track all "Treatments" a specific patient has received.

– If a patient dies, his physician can save the cause of his "Death" for analytic purposes.

**Questions:** In order to easily adapt the mobile applications to a new Health Data Management Layer, the questions resource provides the daily questions which are valid for this instance.

To support Issue 6 the Health API uses Swagger[1]. Swagger is a project that aims on describing RESTful APIs with JSON objects to support easier discovery of the API through machines and humans. Swagger also enables the developer to define models of the used resources, which can be used for input validation or documentation. In combination with Swagger-UI the Swagger data can be used to interactively test and use the API.

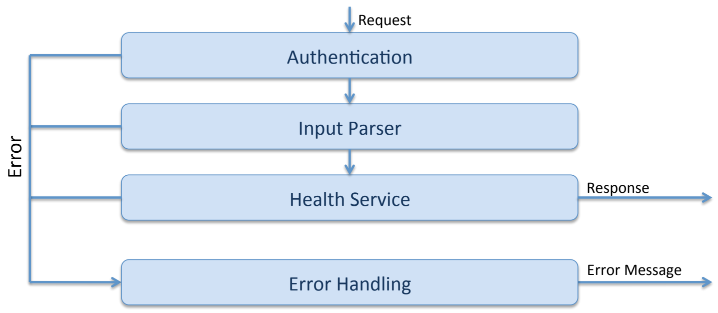


**Fig. 3.** Stack for calling a Health Service through the Health API

The Health API in the ECHO-Platform is implemented using node.js[2] with the express framework[3]. The express framework allows to dispatch incoming requests through a stack of processing functions. Fig. 3 shows the stack for calling a "Health Service" through the "Health API". In the first step the "Authentication" checks if the user can be authenticated. To use the "Health API" an access token is required, which has to be passed via the HTTP Authorization header field. It can be obtained by POSTing username and password to the token endpoint. The token endpoint checks the credentials and issues an access token and a refresh token, if the user provided valid credentials. The token itself is a *JSON Web-Token* (JWT). This means it is a Base64 representation of a digitally signed JSON Object. The token contains the account id of the user, his role and a timestamp, which describes when the token expires. If the token has expired, it can easily be renewed by sending the one-time refresh token to the token endpoint. Hence, the "Authentication" has to check if the token is still valid or if it was altered. In the next step the request body is parsed into a JSON Object by an "Input Parser". The "Health Service" eventually processes the request

[1] http://swagger.io/ [2] http://www.nodejs.org/ [3] http://expressjs.com/

and sends the response. The “Health Service” also does input validation and authorization checks using the database. If an error occurs in one of these steps, it is thrown and handled by the error handler. This interrupts the processing of the stack.

## 4 Health Data Management

In this section, we describe the Health Data Model and how the security of this data is ensured. For the Health Data Management we need a mature relational database system, like MySQL, which supports security features like views and which enforces data integrity.

### 4.1 Health Data Model

The Health Data Model is shown in Fig. 4. The key symbol stands for a primary key, red symbols represent foreign keys, and blue diamonds represent NOT NULL columns. The “accounts” table represents all users of the system. For every user the login information and the notification settings are saved. Furthermore, every user has a role (`patient, doctor`, or `admin`) which determines his access rights in the system. Accounts with the role `admin` are supervisor accounts which can e.g., transfer a patient to another physician or create a new `doctor`-account. But an `admin`-account has no access to any medical data.

Notifications are saved in the “notifications” table. A notification basically consists of a recipient and a type. Type is an integer value (in order to reduce the data volume), where e.g., 0 means *Call your physician!*. For every created patient notification there is also a notification for the physician, e.g., *Your patient X should call you!*.

In the “patients” table basic patient data like name, address, birthday, and sex are saved. Triggers make sure that new entries in the patient table can only be created when they are linked to a valid `patient`-account. They also take care that a patient can only be assigned to a valid doctor, i.e., a user having a `doctor`-account.

The remaining tables hold patient specific results. The table “cats” stores the results of the *COPD Assessment Tests* and “ccqs” the results of the *COPD Clinical Questionnaire*. These are standardized questionnaires which consist of a number of questions, which can be answered with an integer value. Using these values the total results of the questionnaires can be computed. The “charlsons” table contains the Charlsons Index for each patient. This index tracks if the patient suffers from common severe illnesses and expresses the risk that the patient dies because of these illnesses. Each illness is mapped to an integer value and the index represents the aggregation of these values. In the table “deaths” the cause of death and the date are saved. This data can be used later on for analytics. The tables “readings” and “treatments” save the examination results and the medication a physician prescribes. Readings is not fully shown in Fig. 4 in order to keep the figure neat. The daily data input is stored in “dailyreports”.

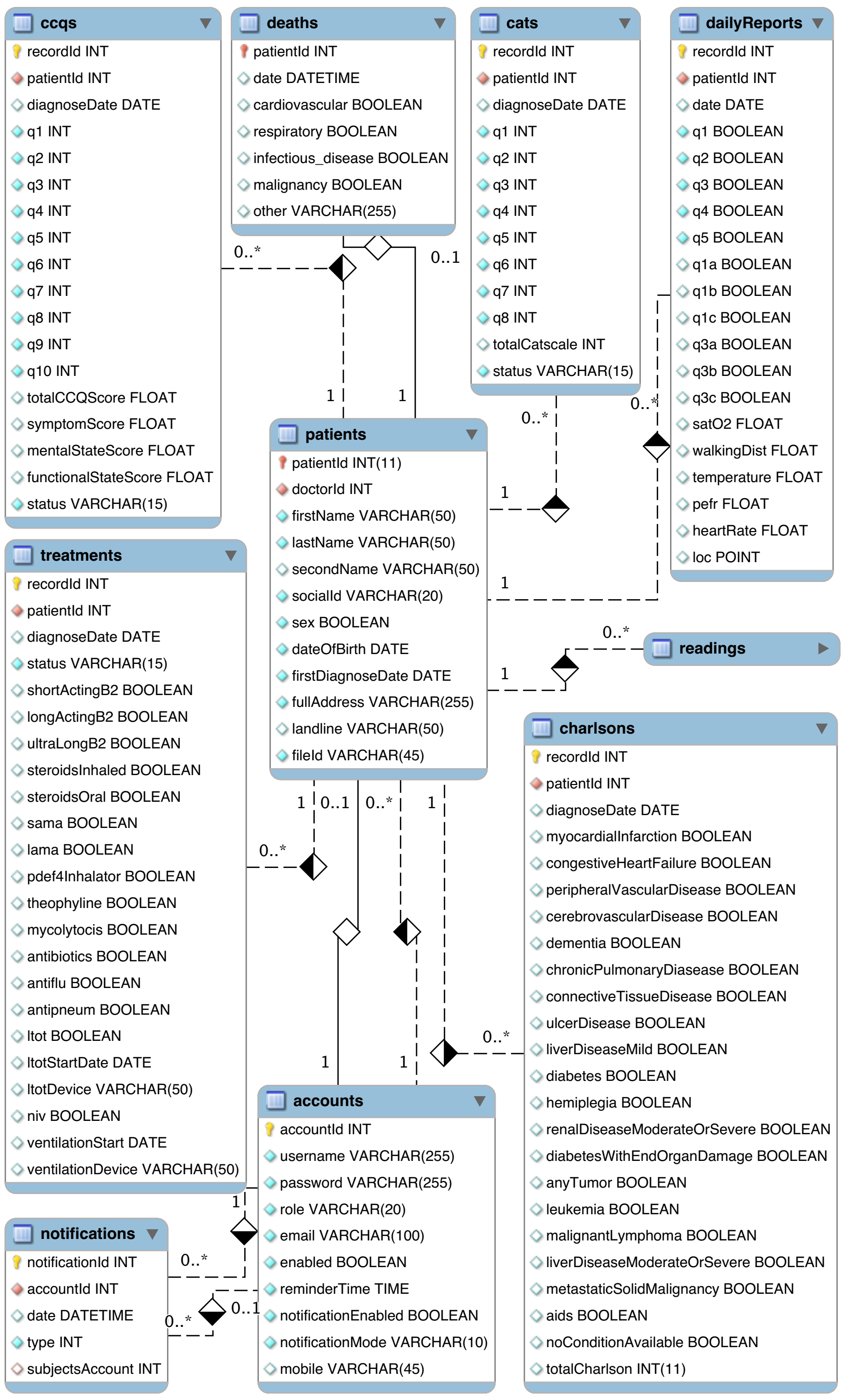


**Fig. 4.** Detailed Health Data Model (modeled with MySQL Workbench)

This table stores the answers to the questionnaire, which the patients get on their mobile phone. Furthermore, we store measures like temperature and heart rate, if the patient supplied them.

This data can be used to run simple analytics. If the health state of a patient aggravates, his physician is notified and can react appropriately. The analytics of the daily data input is rule-based. Daily input is given based on the following questions:

**Q1:** Did your shortness of breath increase?
 **a:** Can you do the daily work you did before?
 **b:** Can you support yourself (go to toilet, shower)?
 **c:** Can you walk?
**Q2:** Did your cough increase?
**Q3:** Did your sputum change?
 **a:** Is your sputum yellow?, **b:** Is it green?, **c:** Or bloody?
**Q4:** Did you have chest pain or discomfort?
**Q5:** Did you take the same medications? Or increased them?

Questions labeled with *a, b* or *c* should only be answered when the corresponding main question was answered with "yes". With these answers given and the following rules, the system can create notifications to inform the patient and his physician (Issue 8).

1. Two days in a row Q1 answered "*yes*" → Notification "*Call your physician!*"
2. Q1, Q2 and Q3 answered with "*yes*" → Notification "*Call your physician!*"
3. Q3a or Q3b answered with "*yes*" → Notification "*Call your physician!*"
4. Q3c answered with "*yes*" → Notification "*Go to the hospital!*"
5. Two days in a row Q5 "*yes*" → Notification "*Call your physician!*"
6. Questions not answered for 2 or 10 days → Notification "*Fill in your Report!*"

### 4.2 Health Data Security

To secure the Health Data, we use the access management of the database combined with the role system introduced above. Every time a new account is created, the Health Data Management Layer creates a new database user. This user can be used by the Health Services when they connect to the database to query data. Based on the role of the new account, the new database user gets access rights on tables and views. Since the used MySQL does not support user roles, the rights are granted by a stored procedure which loads the permissions from the database.

Instead of creating a set of views for every user, we use dynamic views which filter data based on the currently logged in database user. This is done by naming the database accounts according to the id assigned to the newly created account. This means if a new account gets the id *5*, the corresponding database user would also have the name *5*. With this technique one can easily create views that compare the username of the currently logged in database user with e.g., the physician's account id in the patients table to filter only those patients who are assigned to this physician. The role based technique and the usage of multiple database users support Issues 1 and 2. Indirectly, Issue 3 is supported, too, since the developers of the web/mobile applications have not to take care of filtering unwanted data.

To prevent SQL injections and since node.js doesn't support prepared statements, we use MySQL's stored procedures, which use the prepared statements provided by MySQL. Hence, we implemented a stored procedure for every available Health Service. These procedures are also used to bundle several SQL statements into one call. That is to support the asychronous node.js programming model, which would "jump" back into the Health Service after every issued statement, just to issue the next statement.

## 5 Automated Provisioning and Management of the Health Data Management Layer

The ECHO-Platform must be automatically deployable in a variety of IT environments of different hospitals (Issue 4). Typically IT environments in hospitals are IaaS-based private clouds based on virtualization technologies. Deployment in such environments is possible with our approach using OpenTOSCA. PaaS-based provisioning could be supported too. However, PaaS platforms are usually not available in hospital environments. Furthermore, management tasks, such as backup creation or workload-dependent scaling of the ECHO-Platform (Issue 5) must be also facilitated in an automatic manner. To enable automatic provisioning and management of the ECHO-Platform the TOSCA standard is used to describe the platform and their management functions in a machine-readable and self-contained (portable) manner. The TOSCA runtime environment can process this description and perform the management operations in a fully or semi-automatic way.

The main artifact of a TOSCA description is the application topology. The topology is a directed graph where the nodes represent the components required to run the application and the edges the relationships between these components. Figure 5 shows the topology of the ECHO-Platform (modeled using Vino4TOSCA [6]). There each node has a name and a type (in brackets). The relationships define dependencies between the components. The relationship "hostedOn" specifies for instance that the "Health API" and the "Health Service" are deployed on a "node.js 10.2" platform, which is, in turn, hosted on an Ubuntu server that runs on an OpenStack virtual machine. On the same Ubuntu server also a WSO2 BPEL engine is installed that provides the execution environment for the "Service Orchestration". This service orchestration is implemented as BPEL process [12] orchestrating the execution of the "Health Service", the "Analytics Service", and other external Services. The "calls" relations between this process and the two services specifies that the process executes the functions provided by the services. Accordingly, the service functions are called from the "Health API". The "Analytics Service" is hosted on a separate stack to scale it out individually if sophisticated analytics tasks have to be performed that result in heavy workload. The services access the “Health Data DB” that contains the actual health data. Note that the components in the stack can be exchanged by other components to meet the infrastructure requirements of each hospital where the aHDMS has to be provisioned on. The OpenStack virtual machines could be for instance replaced by VMware machines if the designated hospital is using VMWare instead of OpenStack.

Depending on its type, a component within the topology provides different properties and management operations. The node type “Operating System” provides for instance management operations to install software, to perform backups etc. The “Apache Tomcat” node type, on the other hand, provides operations to start or stop services deployed on it. This node type also defines a property “port” that can be set to the port number where the Tomcat listens for connections (e.g., port 8080). The implementation of management operations is realized by implementation artifacts, which can be for instance Web services or scripts. Deployment artifacts present the actual piece of software that is being managed, e.g., the Ubuntu operating system, the MySQL database or the JavaScript code of the “Health API”. The actual provisioning and management tasks are implemented by BPEL processes (Issue 10). The provisioning plan orchestrates the execution of management operations provided by the nodes to set up the complete aHDMS in an automatic manner. The plan sets up the three required virtual machines along with the operating systems. Then it installs the

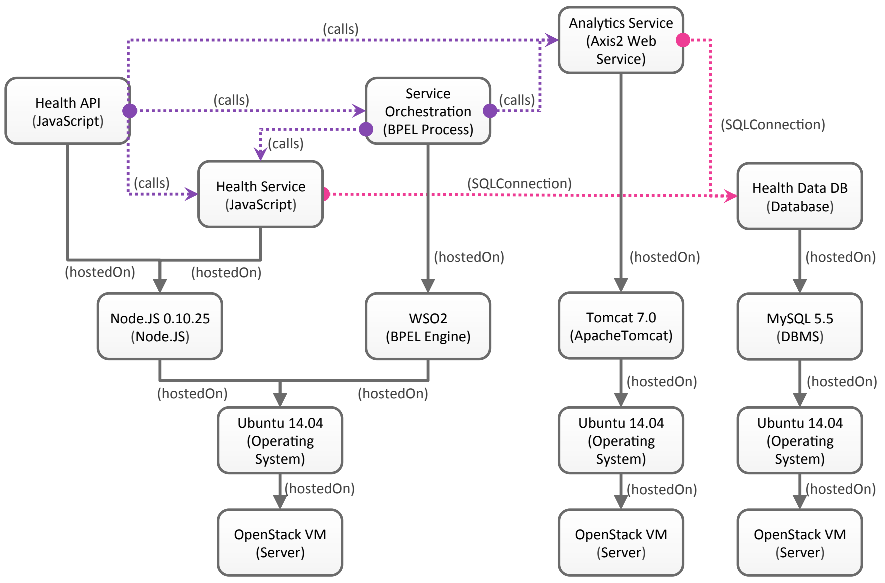


**Fig. 5.** TOSCA Topology of the ECHO-Platform

software (node.JS, WSO2, etc.) components and configures them according to the property values of the respective nodes. In the future, management plans will be added that implement watchdog and scalability functionality in order to improve the stability and resiliency of the aHDMS. The topology, the implementation and deployment artifacts (or the references to these artifacts [13]) and the plans are bundled into a *Cloud Service Archive* (CSAR).

As management environment for the aHDMS the OpenTOSCA container is used which provides the runtime environment for the implementation artifacts and plans bundled in the CSAR. It also manages configuration data such as the installation status of components, which ports are used and so on. A more detailed description about the architecture of the OpenTOSCA container can be found in [3]. The container is part of the OpenTOSCA ecosystem[4] that provides also the modeling tool Winery [11] for creating TOSCA topologies such as the one shown in Fig. 5. Winery also assists the user in bundling the topology, along with it plans and artifacts to a CSAR. Furthermore, in future work policies could be attached to the CSAR or single topology nodes. This can be used to encrypt the database using the encryption policy [17]. The service archives can be directly deployed on the OpenTOSCA container and the provisioning and management plans can be instantiated by the user with the self-service portal Vinothek [7].

## 6 Summary and Future Work

While implementing the ECHO-Platform—a secure and cloud-enabled *active Health Data Management System* (aHDMS)—we identified a set of design and implementation issues. This paper first describes these issues and furthermore explains how we solved them in our prototypical implementation. The ECHO-Platform is an open-source aHDMS that can be automatically deployed in different cloud environments, but is still under control of the authority that deployed the system, e.g., a hospital. In contrast to existing cloud-based health systems, e.g, Microsoft Health Vault [2], the control on the health data is given away to the external system.

Our health data model was designed to support data exchange from/to the HL7 (Health level 7) standard. Hence, future work deals with building adapters for this data exchange as a proof of concept. In the future, it is also planned to conduct data mining on the stored health data in order to find new knowledge about diseases and their courses and treatments in general, as well as on an individual patient-aware level. However, this can only be done with the consent of the patient and in an anonymized way. Afterwards, the new general knowledge can than be applied individually for each patient, in order to improve the health care by giving hints to the physician.

**Acknowledgment:** This research and development project is funded by the German Federal Ministry of Education and Research (BMBF) within the Framework Concept “Innovationen mit Dienstleistungen” (01XZ13023G) and is

[4] http://www.iaas.uni-stuttgart.de/OpenTOSCA/

managed by the Project Management Agency at the German Aerospace Center (PT-DLR) Unit “Work Design and Services” (PT-AD).

## References

1. Caisis: An Open Source, Web-based, Patient Data Management System to Integrate High Quality Research with Patient Care, `http://www.caisis.org/`
2. Microsoft HealthVault, `https://www.healthvault.com`
3. Binz, T., et al.: OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In: Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13). LNCS, vol. 8274, pp. 692–695. Springer Berlin Heidelberg (Dec 2013)
4. Bitsaki, M., et al.: An Integrated mHealth Solution for Enhancing Patients Health Online. In: Lackovi, I., Vasic, D. (eds.) 6th European Conference of the International Federation for Medical and Biological Engineering, IFMBE Proceedings, vol. 45, pp. 695–698. Springer International Publishing (2015)
5. Borner, T., Balatzis, G., Röhrdanz, O.: Vergleich von Gesundheitsdatenmodellen. Fachstudie Softwaretechnik: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Anwendersoftware (Sept 2014)
6. Breitenbücher, U., et al.: Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In: Proceedings of the 20th International Conference on Cooperative Information Systems (CoopIS 2012). Lecture Notes in Computer Science, Springer-Verlag (September 2012)
7. Breitenbücher, U., et al.: Vinothek – A Self-Service Portal for TOSCA. In: ZEUS 2014. CEUR Workshop Proceedings, vol. 1140, pp. 69–72. CEUR-WS.org (Mar 2014)
8. Chen, M.: NOSH (New Open Source Health) ChartingSystem, `http://www.noshchartingsystem.com/`
9. Ding, H., et al.: A Mobile-Health System to Manage Chronic Obstructive Pulmonary Disease Patients at Home. In: Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE. pp. 2178–2181 (Aug 2012)
10. van der Heijden, M., et al.: An autonomous mobile system for the management of COPD. Journal of Biomedical Informatics 46(3), 458 – 469 (2013)
11. Kopp, O., et al.: Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13). LNCS, vol. 8274, pp. 700–704. Springer Berlin Heidelberg (Dec 2013)
12. OASIS: Web Services Business Process Execution Language Version 2.0 – OASIS Standard (2007)
13. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0 (Jan 2013), `http://www.tosca-open.org`
14. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0 Committee Specification 01 (Nov 2013), `http://www.tosca-open.org`
15. OpenMRS: Open Medical Record System, `http://openmrs.org/`
16. Orfanidis, L., et al.: Data Quality Issues in Electronic Health Records: An Adaptation Framework for the Greek Health System. Health Informatics Journal 10(1), 23–36 (2004)
17. Waizenegger, T., et al.: Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing. In: Meersman, R., et al. (eds.) On the Move to Meaningful Internet Systems: OTM 2013 Conferences. Lecture Notes in Computer Science (LNCS), vol. 8185, pp. 360–376. Springer Berlin Heidelberg, Heidelberg (Sept 2013)

# A PaaSage to Multi-Site Security for Clouds

Tom Kirkham[1], Kyriakos Kritikos[2], Bartosz Kryza[3], Philippe Massonet[4], Franky Vanraes[5]

[1] Science and Technology Facilities Council (STFC), Chilton, United Kingdom
(Tom.Kirkham@stfc.ac.uk)
[2] ICS-FORTH, Crete, Greece
(kritikos.kiriakos1@gmail.com)
[3] AGH, Krakow, Poland
(bkryza@agh.edu.pl)
[4] CETIC, Gosselies, Belgium
(philippe.massonet@cetic.be)
[5] BE.WAN, Waterloo, Belgium
(Franky.VANRAES@bewan.be)

**Abstract.** Taking a model driven approach catering for automation and integration capabilities, multi-site security can be applied to Clouds which consist of various infrastructure services provided by different service providers. The proposed solution ensures attribute based access control and security policy integration across multi-cloud deployments integrating providers using common standards for identity management. Models are used as carriers of security information in terms of user attributes and security policies which are then used for the proper authentication and authorization of user requests in the developed PaaS platform. Models are also used to express domain specific security requirements which are then satisfied during the deployment planning and provisioning of applications that span multiple Cloud Providers.

## 1 Introduction

Cloud Computing architectures typically consist of provider centric implementations which offer a variety of cloud services at different levels (IaaS, PaaS & SaaS). As a result, applications deployed on the Cloud are restricted to the specific security infrastructure and metrics offered by the selected provider. This constitutes a problem for developers who might desire at some time to either port their applications to another provider or provide software services across multiple Cloud providers. In such cases, providers have to support security solutions using metrics which are calculated in different ways [1]. This metric flexibility is, however, a feature not currently offered by the major cloud providers in the market. To this end, there is a need of a platform able to support such a feature by also raising the abstraction level and having the ability to satisfy the security requirements of application developers across different clouds.

The PaaSage project has the vision to "specify once and deploy anywhere" Cloud applications. Central to the realization of this vision is the development of a platform to enable automated application deployment and adaptive execution across Cloud Providers. Novelty from a security perspective is present where we apply typically static policy enforcement technology in a dynamic way to control how Cloud providers are both selected and managed during execution.

The dynamicity is present when policies can adapt to for specific deployments taking into account quality metrics from the deployment model in relation to the computational environment in which they are hosted. These metrics relate specifically to captured and maintained security requirements across the Cloud lifecycle. The PaaSage solution spans he stage of requirement specification leading to policy formulation and then the enforcement during reasoning, execution and monitoring phases of the Cloud lifecycle.

To date no solutions exist to handle this automated evolution and management of security requirements across the whole Cloud lifecycle. The innovation described in this paper will reduce the ambiguity and gap between Cloud provider terms of service and application requirements ensuring that the process of Cloud adoption is more transparent for users. This step is significant as it will enable users of Clouds to better integrate security requirements and thus develop trust in Cloud providers. The current lack of flexibility and trust presented by Cloud Providers is a key stumbling block in the adoption of Cloud technology particularly for the Small to Medium Enterprises [2].

The rest of the paper is structured as follows. Section 2 provides background information for security metrics in clouds. Section 3 analyzes our security solution. A particular use case on which our solution has been applied is detailed in Section 4. The state-of-the-art is analyzed in Section 5. Finally, the last section concludes the paper and draws directions for further research.

## 2. Security Metrics for Multi Clouds

The European Commission recently published a list of security metrics that can be used to evaluate individual Cloud Providers [21]. From a standardization and integration perspective the documentation of these metrics constitutes a first but significant step. If the Cloud is to be fully embraced by business it is widely recognized that trust in Cloud Provision needs to improve. Transparency of data processing is significant to achieve this trust and the ability to agree standard metrics to monitor security would be a significant step.

However, security metrics by nature contain sensitive data and often are sourced from within the Cloud Providers internal systems. Releasing this data for monitoring or evaluation is a risk for the provider both in terms of security threat but also potential impact on reputation. In addition, experience in relation to Safe Harbour compliance has demonstrated that often data released by a provider is not necessarily a true reflection on reality [22].

PaaSage provides a means around this by the provision of the PaaSage Social Network. This enables the sharing of data between application developers on experiences with Cloud providers. Such data can extend to security and referring to the list mentioned in [21] we have extracted the main metrics we propose to use in our solution which are shown in Table 1.

**Table 1: Metrics Related to Security**

| |
|---|
| Availability |
| Level of uptime (Often termed "availability") |
| Percentage of successful requests |
| Percentage of timely service provisioning requests |
| Average Response Time |
| Maximum Response Time |
| User authentication and identity assurance level |
| Authentication |
| Third party authentication support |
| User authentication and identity assurance level |
| Logging parameters |
| Log access availability |
| Logs retention period |
| Certifications applicable |

The list above has been filtered to include metrics that the end user can measure and share with others on the Social Network. At first glance the metrics relate more to quality of service than security in a traditional sense. However (as mentioned), such low level security information is sensitive and unlikely to be shared by providers. Using this level of data security provision can be measured and monitored albeit in a higher level way. Thus, it is only a fraction of the metrics listed in [21]. It is likely that this list of metrics will expand with the development of specific security monitoring tools for use by application developers to gather data from Cloud Providers for sharing on a portals such as the PaaSage social network. It is also fair to assume that if enough user / market traction gets behind the need for clearer security metrics providers will react and provide them. However, this currently is not the case. We therefore have based our solution on metrics published by Cloud providers and the metrics / experiences that are likely to be shared between application developers.

## 3. Delivering Multi-Clouds

In order to put into context how we apply our security metrics to span Multi-Cloud environments, it is important to introduce the PaaSage approach to Multi-Cloud provision.

### *3.1 The PaaSage Platform*

Significantly the PaaSage platform promises the automatic deployment and provisioning of applications in multi-cloud environments through the use of model-driven techniques. It relies on a particular unification of Doman Specific Languages (DSLs) called CAMEL for the description of the respective required information aspects. CAMEL models are used at all phases of the cloud lifecycle from deployment to execution. Three main modules manage the application through the Cloud lifecycle namely the Upperware, Executionware and Metadata database (MDDB).

The Upperware module is responsible for the creation of concrete deployment plans and the generation of low-level deployment actions. These are executed by the Executionware module which deploys the respective application and monitors its performance in one or more clouds. The Executionware is also capable of performing adaptation actions when violations on metrics occur which trigger certain scalability rules captured in the applications' CAMEL model.

The MDDB module is responsible for the storage and retrieval of the CAMEL models as they evolve during the Cloud application lifecycle via their handling by the respective core PaaSage modules. Thus, the MDDB is not only exploited for indirect communication between the various PaaSage modules/components but also maintains the state of an application and feeds this knowledge directly into the PaaSage Social Network.

The PaaSage Social Networks exist for specific deployments of PaaSage. When PaaSage is deployed it also associates itself with a PaaSage Social Network. As will be explained later the Social Network can be dedicated to the instance of PaaSage or shared across instances. The common thread that ties the Social Network to the PaaSage implementation is the MDDB as this is used by the Social Network to reference and store models.

Knowledge is built upon in the social network by integrating data from past and current deployments in the MDDB with extra context and knowledge shared between users. Thus, the Social Network is not only a consumer of knowledge from the MDDB but also a primary source of added-value information for the MDDB. An example of added-value knowledge to be produced is the proposal of a certain optimized deployment model for a certain type of application which has been deemed to have the best performance in the past.

The flow of information between the different PaaSage modules is depicted in Fig. 1. As it can be seen, there are two optimisation loops involved, the runtime and design-time loop. During the runtime optimisation loop, the Profiler constructs the profile of an application and hands it over to the Reasoner in order to discover the best possible deployment plan that matches it by also satisfying all the respective user requirements. The deployment plan is then sent to the Adapter which calculates the minimum amount of low-level deployment actions that have to be executed in order to go from the previous application deployment configuration to the new, proposed one. These low-level actions are then executed by the Executionware which also monitors the application performance and performs cloud-specific adaptation actions, when SLO violation occurs, according to particular scalability rules. In case that the limits of possible scaling are reached or there are no cloud-specific ways to react to a certain situation, then the Executionware informs the Upperware in order to modify the current application configuration. At all times, when scaling actions are performed, the Adapter is notified in order to have a complete picture of the current deployment structure of the application.

According to the design-time loop, the Upperware informs the application developer when the requirements posed are over-constrained or cannot be handled any more according to the current application and cloud context. In this case, the user requirements will have to be modified in order to reach a new and optimized deployment configuration of the respective application.

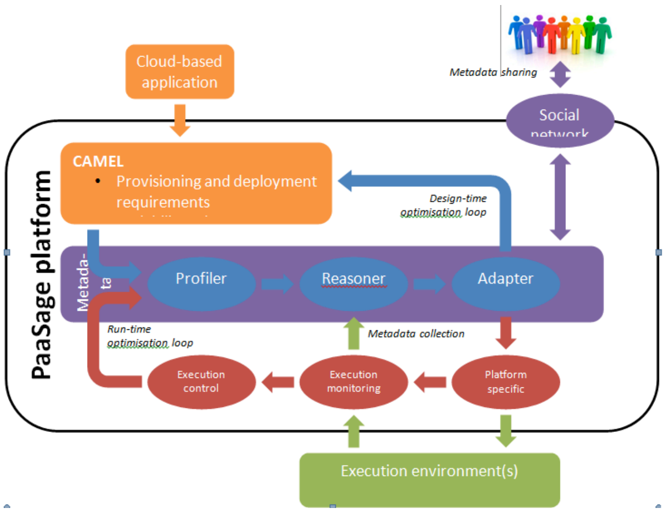


**Fig. 1 - The flow of information between the PaaSage modules**

During the PaaSage control flow, security needs are enforced at both the module/component and information/data level. At the information/data level, security requirements in the form of security SLOs involving security metrics are incarnated in the respective application CAMEL model. At the module/component level, security requirements are used to guide the reasoning phase towards the production of a security-compliant deployment plan as well as for the monitoring of the application security levels and reacting when these levels are violated by executing particular adaptation/scaling rules.

### *3.2 Architecture for Multi-Site Security*

The PaaSage platform (not an application) can itself be deployed in different ways: (a) multi-site or (b) single-site. Examples of multi-site PaaSage deployments include domains where a single Social Network exists, used to specify application deployment requirements, which incorporates data sharing across potentially multiple domains. In such global models, different PaaSage operators/providers exist within a trust network akin to a federation (using certificates signed by trusted third parties). Local/single-site deployments occur in information sensitive domains.

Typical examples here include instances of PaaSage where one Social Network is specific to the business of a large organization. In such cases the information is commercially sensitive and enough knowledge may exist and be shared among the clients of this organisation without the need to share with other installations. In this latter case, there is actually no need and meaning to have the respective PaaSage operator in any kind of federation as this operators does not share any information with the other federation members.

The security solution for PaaSage comprises four main components as illustrated in Figure 2.

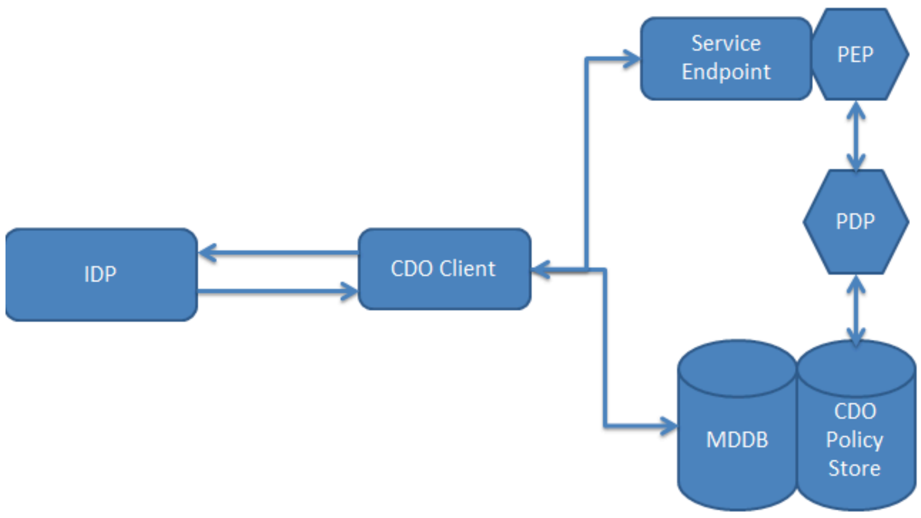


**Fig2: Main PaaSage Security Components**

As illustrated in Figure 2, (a) *Identity Provider - IdP*: this component is responsible for user authentication and for sending restrained user information (attributes) to those services/components which require subsequent user authorization, (b) *Policy Decision and Enforcement Points - PEP/PDP*s: these components are responsible for the actual user authorization by checking the role played by the user (expressed in attribute token from IdP) and the respective organisation policies expressed in XACML [16], (c) *Service Endpoint*: it is the main access point for a service/module/component offered by a single-site instance of the PaaSage platform which exploits the authorization mechanisms provided by the local PEP/PDP points to authorize the user access to these services; (d) *CDO Policy Store*: it is a private CDO Server [24] with an underlying database which is responsible for the storage of organisation models (covering authentication & authorization information aspects) specified through the organisation DSL. As XACML is a XML-based language, through the use of appropriate Eclipse tools, it is also considered and handled as a DSL whose models (i.e., policies) are stored in the CDO Repository. Organisation models describe organisation (which can be a PaaSage operator or one of its clients) information, such as contact details and types of services offered, user information including authorization data, such as credentials, as well as other important information artifacts, such as roles and allowed actions, which can then be exploited in order to express authorization information in terms of XACML policies. Thus, what we have achieved here is true integration between two security-oriented DSLs that allows us to capture and integrate both authentication and authorization information.

The security architecture mirrors existing well established approach of policy based security in distributed systems from organizations such as the Liberty Alliance [23]. The CDO store acts as the MDDB store for CAMEL linking it to the wider PaaSage community for knowledge sharing on policies (after removing sensitive information and just capturing particular policy patterns recurring in the cloud domain) for Cloud deployments. Here, our specification of policy can be made on a broad level not specific to Cloud providers and incorporating other constraints such as quality of service (i.e. if response rate dips it could be indication of security threat). This differs from typical deployments where the XACML policy is often specified and often limited to a particular domain and Cloud environment.

## 4 Use Case

The PaaSage project will be demonstrated via several use cases. In order to showcase the security concerns handled in this paper, we will use the PaaSage Enterprise Resource Planning (ERP) Use Case as an example. BE.WAN is a Belgian SME who provides ERP based solutions largely to industry in Belgium. Although it is an IT company, it has not performed constant and decisive steps towards (fully) adopting the cloud. Reflecting the findings in [2], the reasoning behind such lack of Cloud adoption often is down to lack of customization for specific customers and concerns over security of data.

However, a Cloud based solution for BE.WAN that can be customized by end users to exhibit tailored security levels is an attractive offer that instigated the company's involvement in the project. Such an offer could provide for a differentiation in the Belgium market as well as assist BE.WAN in extending its portfolio of solutions by also including differentiated security level offerings with flexible and attractive pricing. Motivated by BE.WAN customer requirements we now explain how security can be adapted for specific end users.

### *4.1 Integrating Identity from the Use Case*

In order to administer security in applications deployed across Clouds, common roles have to be defined through which policies can be constructed. Identity is the start point at which an application is modelled and within CAMEL role definition follows a standard form. The user links his / her organisations roles to particular role taxonomies across all organisations in one site and across all sites. The taxonomy covers all possible roles that are envisioned to be involved in the operation of the PaaSage platform. The roles considered include:

(a) *System Administrator*: responsible for the deployment at the local site and the proper functioning of a PaaSage platform instance/installation; it should have access to all possible types of information and services - this includes the management of authentication and authorization information.

(b) *Application Owners* (*business users*): responsible for the definition of the application high-level requirements which can then be drilled down to more concrete requirements at lower levels to guide the deployment, provisioning and monitoring of the respective application.

(c) *Developers*: responsible for the development of the application and thus for the creation of the respective application profiles and deployment models.

(d) *Software engineers*: responsible for the development of the software supporting the functionality of the application as well as the low-level requirements posed by this software (thus they can actually contribute to the refinement of requirement (e.g., specific SLOs for the software) and deployment models (specific VM requirements dictating the way the software should be deployed/hosted)).

(e) *Devops*: responsible both for the development as well as the operation of the application. This last role, as also has the responsibility of the application operation, can generate not only application profiles, deployment & requirement models but also scalability rules which can drive the adaptive operation/execution of the respective application.

To support this process simple to use graphical tools are presented via interfaces such as the Social Network. Once defined, the roles can then be mapped to particular policies specific to an organisation. Different sets of policies will be enforced depending on the security level required by the respective organization and the specific context in which it is applied. Policies are defined and simplified in PaaSage using the Create, Read, Update, Delete (CRUD) approach to data access.

Policies are defined in XACML [16] and stored privately in the CDO server linked to the Policy Decision Point (PDP). We advocate for the high-level specification of policy-based information in terms of security levels. This means that the organisation has just to choose the particular security level that suits its needs. Once this is performed, then the PaaSage platform will take care of mapping this security level to a set of security policies that need to hold. If the organisation needs to perform any kind of customization, then it can then modify one or more of the security policies that have been generated automatically.

We advocate for the consideration of the following security levels:

- *high security level*: no information sharing is performed as only the users of the organisation can have access to the information manipulated by this organisation. This means that external users either from the same PaaSage installation or a different one will not be able to see anything that is handled by the organisation at hand.
- *low security level*: information sharing is allowed for all information generated by the organisation apart from its organisation model which includes sensitive information. This means that external users will have read access to all of this information while internal (i.e., organisation) users will have both read and write access.

- *medium security level*: information sharing is allowed only for application profiles and deployment models. This means that external users will have read access to these models while the internal users will have read and write access to all types of models manipulated by the organisation (apart from the organisation model itself which should be visible and customizable only for  a certain role of the organisation). The rationale of allowing read access only to these models is that the organisation does not want to reveal important or critical information, such as which requirements lead to which deployment model and which scalability rules are used for a specific application. Thus, the sharing of basic knowledge is just allowed here.

Once roles are defined, identity in the live system can be mapped to them. According to [3], identity of a user can be classified into the following categories:

- *Anonymous* – not possible to identify user between sessions
- *Pseudonymous* – the same user identifier is used between sessions
- *Self-asserted* – user provided information, without validation
- *Socially  validated* – user information is valid through their social graph
- *Verified* – formally verified user identity (e.g., through a ID document)

In PaaSage, to support the mapping of roles across domains, a standard way of describing users and their attributes is achieved using CERIF to exchange extended metadata about user profiles.

CERIF [4] is a modelling framework for describing organizations, people, projects and other aspects related to the research domain. It is an EU recommendation for information systems related to research databases, in order to standardize research information and foster research information exchange. A selected subset of CERIF has been used to formulate the organisation DSL such that CERIF models, providing descriptions about an organisation's structure, users and resources, can be transformed into organisation ones and be integrated and stored in CDO Repository for the purpose of improved user and organization modelling. To facilitate the CERIF-to-Organisation model transformation, a specific tool has been developed which is able to take as input a specific CERIF XML-based model and transform it into an (XMI-based) organisation model one.

An example user description in CERIF for BE.WAN is presented below.

```
<cfPersName>
        <cfPersNameId>persname-id1</cfPersNameId>
    <cfFamilyNames>Fontnot</cfFamilyNames>
    <cfFirstNames>Todd</cfFirstNames>
</cfPersName>
<cfPers>
    <cfPersId>pers-id1</cfPersId>
    <cfGender>m</cfGender>
    <cfPers_Class>
      <cfClassId>CAMEL.Administrator</cfClassId>
      <cfClassSchemeId>CAMEL</cfClassSchemeId>
    </cfPers_Class>
    <cfPersName_Pers>
      <cfPersNameId>persname-id1</cfPersNameId>
    </cfPersName_Pers>
    <cfPers_EAddr>
      <cfEAddrId>ToddMFontenot@dayrep.com</cfEAddrId>
      <cfClassId>35d43364-2160-4b6c-a487-5019458321e8</cfClassId>
      <cfClassSchemeId>05cc5ff9-bc58-4743-ab59-46e5013e0039</cfClassSchemeId>
    </cfPers_EAddr>
    <cfPers_OrgUnit>
      <cfOrgUnitId>123</cfOrgUnitId>
      <cfClassId>ebd55ab0-1cfc-11e1-8bc2-0800200c9a66</cfClassId>
```

```
        <cfClassSchemeId>e9616dbd-0d38-4b7d-a6cd-3c4df1e95462</cfClassSchemeId>
        <cfStartDate>2012-06-01T00:00:00</cfStartDate>
        </cfPers_OrgUnit>
</cfPers>
…
```

**Fig 3: Example User Description in CERIF**

The user roles are extracted from the cfClass user attributes, which can reference roles either from domain specific role hierarchies defined within organizations or directly from the role ontology of PaaSage. In the former case, a mapping needs to be provided by the respective organisation to map between these roles in case multi-site deployment Such a mapping could be enforced during the CERIF-to-Organisation model transformation as an extra input to be considered.

Once this information is specified and transformed to the respective organisation model part, then a particular instance of the *User* class will be generated covering user-specific information, including its credentials to the PaaSage platform as well as to cloud providers. The latter credential information is a necessity in order to enable the platform to perform the deployment and adaptation of applications in multi-cloud environments. An instance of the *RoleAssignment* class will also be generated which will be able to map a user or user group to a certain PaaSage-specific role.

When authentication takes place the aforementioned generated user and role information is used. Currently, the default IdP implementation in PaaSage is the SimpleSAMLphp identity service [5], which supports the SAML 2.0 [6] standard, while the private CDO server is used as the data source for this IdP as it contains all the organisation models immediately generated by organisations or transformed from CERIF. The Social Network has been developed using the Open Source platform Elgg [17] which supports both SAML and OAuth identity integration. BE.WAN's customers typically support SAML 2.0 based SSO.

Authentication is based on SAML attributes which are then mapped to roles and policies regarding data access expressed in XACML. As with the wider security architecture this is done via the Service Provider for the resource being accessed. For BE.WAN the choice of SAML enables formal representations of organizational identity attributes when compared to less formal / structured standards such OAuth or OpenID. This suits business systems where attributes of users are typically mapped to an identity store such as LDAP.

## *4.2 Security Models*

Once identity management requirements are defined through the organisation DSL the next step is the creation of the end user security model. Here, the security DSL is involved. This DSL can be used to describe security requirements and capabilities at a coarse or fine-grained level related to metrics. The key concept considered is a security control which indicates at a higher level of abstraction a part of the security level offered by a particular cloud provider or required by the end-user. In this way, security requirements and capabilities can be matched as they both map in a symmetric manner to a set of security controls. Moreover, a security capability, in the way it is defined, maps to the actual security level provided by a cloud provider.

The implementation of security controls can be checked through the enforcement of more fine-grained security guarantees in the form of security service level objectives (SLOs). Such SLOs include conditions on security properties or metrics which are mainly defined through using constructs of another CAMEL DSL called SRL (Scalability Rule Language) [20]. The linkage between security controls and security SLOs is enabled through the mapping of security controls to particular security metrics and/or attributed. This of course requires the specification of a detailed security model which comprises different levels and their linkage such that we can navigate downwards or upwards and be able to associate higher-level constructs, such as security controls, to lower-level ones, such as security metrics.

Through the use of security controls and security SLOs, two main PaaSage platform functionalities can be supported: (a) the filtering of the cloud offering space according to high-level and low-level security requirements and (b) the evaluation of security SLOs which can be mapped to particular adaptation actions to be performed once the evaluation result is a violation. The latter functionality is realized through the specification of the respective scalability rule(s) through the SRL DSL which connect events or event patterns mapping to (security) SLOs to the respective adaptation actions.

The filtering based on high-level requirements relies on checking whether the required security controls are a subset from those supported by a cloud provider. On the other hand, the filtering based on low-level requirements relies on the way generally SLOs are handled by the reasoner. As a general rule of thumb, an advertisement matches a request when the solution space of the request is a subset of the solution space of the requirement. In other words, the SLO requirements should be less strict than the SLO capabilities.

### *4.3 Deployment Workflow*

Once the organizational and security models have been defined the end user is ready to deploy his / her application on the platform.

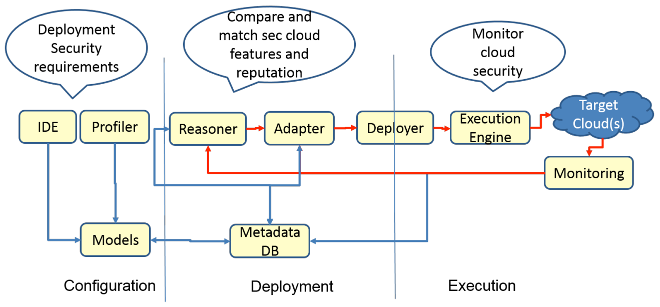


**Fig 4: Deployment Workflow**

Figure 4 shows the PaaSage deployment workflow and how the security DSL enables security to be taken into account in the different phases of the workflow. For a company like BE.WAN this provides reassurance that the requirements set in the previous models are linked to each stage of the Cloud lifecycle.

Example requirements expressed as CAMEL service level objectives (related to security) set at the profiling phase and their evolution through the lifecycle can be seen in Table2.

**Table 2: Evolution of Service Level Objectives in PaaSage Cloud Lifecycle**

| **Requirement** | **Profiling** | **Deployment** | **Execution** |
|---|---|---|---|
| Availability | ➢ 99% | Providers ABC | 99.2% |
| Level of uptime (Often termed "availability") | ➢ **99%** | Providers ABEG | 99.3% |
| Percentage of successful requests | ➢ 99% | Providers ABCEG | 99.8% |
| Percentage of timely service provisioning requests | ➢ 99% | Providers ACG | 99.0% |
| Average Response Time | 0.1ms | Providers ABCF | 0.04ms |
| Maximum Response Time | 0.3ms | Providers ABG | 0.15ms |

At profiling phase the metric can be seen to specify measurable levels of service. This is taken from the application designer requirements during the modelling phase. The interface the designer uses may not ask for specific figures but could translate inputs such as very high to specific metrics. At deployment phase the list of Cloud Providers which fulfil the profiling criteria are listed. This enables the optimal provider to be selected on the basis that they fulfil most or all of the criteria specified. This is done using a Utility Function which is a focus of work in PaaSage. In such a function the compliance with security requirements will be weighed against the need to comply with other metrics such as cost. Finally the execution metrics are the actual performance values of the infrastructure that the application is deployed upon. Any dip below the thresholds set at profiling phase causes the platform to adapt and check if alternative providers exist from the deployment phase to take over execution.

This combination of model refinement by the end user and the use of models to deploy, reason and monitor execution provides a level of reassurance and finer grained control over clouds to reassure SMEs which use BE.WAN's offerings deployed via the PaaSage platform.

## 5. Related Work

Security in Clouds links closely to previous work in the establishment of cross domain trust in Service Oriented computing environments. A prominent example of this can be seen in the work of the Liberty Alliance [9]. The PaaSage approach adopts some of the concepts of Liberty particularly around the authentication and authorization approach to resource access [10]. Further realizations of the Liberty model can be seen within implementations such as ZXID [11] and research on the TAS3 [12] project. In such projects the finer grained control of access to resources has been implemented following the concept of sticky policy [13] authorized access to resources. This again is replicated in PaaSage and built on by including the policies within CAMEL enabling them to evolve dynamically as the application passes through the Cloud lifecycle.

In terms of security across federated clouds [26] presents a model where analysis and detection of threats can be deployed across federations. In PaaSage we deploy a similar approach based on execution data against constraints captured in CAMEL. We have yet to defined detailed threat models to enabled advanced analysis of monitoring data for emerging threats. This could be advanced using VM monitoring techniques as outlined in [27]. In PaaSage enforcement is done on an instance basis and redeployment is the main method for averting threats. In terms of the model [25] outlines approaches for modelling of security constraints but is limited to the deployment phase of the Cloud lifecycle. The PaaSage security meta-model goes beyond this to support security metrics for the monitoring part. So the PaaSage security meta-model covers the whole lifecycle.

With respect to modelling as an approach to integrate Clouds the concept of models to join terms between different implementation domains is established in domains such as Cloud brokerage [14]. In such cases previous work has tended to focus on the use of model driven Clouds for application deployment in terms of resource use [15]. Data protection and general security criteria have been limited in previous model based approaches, this can be seen because they have focused on single rather than multi-cloud deployments. PaaSage is a project looking to develop a wider scope for security models across the whole Cloud lifecycle. The adoption of the PaaSage platform and the further development of the Social Network in specific domains should aid this purpose.

## 6. Future Work

Future work will focus on the following directions. First, the application of our approach in the other use cases of the PaaSage project. Second, its thorough evaluation to prove that the proposed features are indeed exhibited. Third, the creation of a repository which will be able to hold not only basic security models through which security requirements and policies can be described but also security advertisements for cloud providers that are needed for the proper matching of the end-user requirements.

It is essential to highlight that cloud providers should gain by reporting their security capabilities as this will lead to making informed cloud selection decisions, it will more clearly establish what is to be expected by the cloud users and it will increase the trust in the cloud providers. This reporting will also provide them with appropriate flexibility by being able to match different security levels to different class of clients and can finally lead to extending the existing clientele and increasing their gains. The populated repository can then be considered as an essential asset for any kind of cloud marketplace or broker which needs to consider security aspects for the purchasing and selection of the appropriate cloud services or the adaptive deployment and provisioning of multi-cloud applications. Fourth, as CDO provides a particular authentication and authorization mechanism, we will explore how we can exploit and integrate it in our current multi-site security solution architecture.

## 7. Conclusion

Using a model-driven approach a lifecycle approach to security in multi-cloud environments can be developed. Using a start point of identity management linked to security policies and application developer requirements the management of security can adapt to the context of the Cloud and application deployed upon it. This will benefit SMEs who lack the skills and knowledge to create detailed models and security specifications for application deployments onto the Cloud. By using PaaSage and its supporting services such as the Social Network the adoption of Clouds by less technical SMEs can be enabled.

## 8. References

1. S. Subashini, and V. Kavitha. "A survey on security issues in service delivery models of cloud computing." *Journal of network and computer applications* 34.1 (2011): 1-11.
2. J. Leonard "Cloud Computing No Way" Computing Magazine http://www.computing.co.uk/ctg/feature/2327162/cloud-computing-no-way-say-half-of-smes *10/05/15*
3. Kaliyah Hamlin, The Identity Spectrum, http://www.identitywoman.net/the-identity-spectrum *10/05/15*
4. CERIF Home Page, http://cordis.europa.eu/cerif/home.html *10/05/15*
5. SimpleSAMLphp Home Page, https://simplesamlphp.org *10/05/15*
6. N. Ragouzis et al., *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. OASIS Committee Draft, March 2008.
7. Cloud Control Matrix. Online: http://www.cloudsecurityalliance.org/cm.html. 2011 *10/05/15*
8. A. Pannetrat, Security-aware SLA Specification Language and Cloud Security Dependency Model. CUMULUS Deliverable D2.1, 2013
9. Alliance, Liberty. "Liberty alliance project." *Web page at http://www. projectliberty. org* (2002).
10. A. Mansour, C. Adams. "Enhancing consumer privacy in the liberty alliance identity federation and web services frameworks." *Privacy Enhancing Technologies*. Springer Berlin Heidelberg, 2006.
11. ZXID home page www.zxid.org *last accessed 10/05/15*
12. EU TAS3 project homepage www.tas3.eu *last accessed 10/05/15*
13. M. Mont, S Pearson, P Bramhall. "Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services." *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*. IEEE, 2003.
14. A. Simons et al. "Cloud Service Brokerage-2014: Towards the Multi-cloud Ecosystem." *Advances in Service-Oriented and Cloud Computing*. Springer International Publishing, 2014. 121-123.
15. A. Danilo, et al. "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds." *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. IEEE Press, 2012.

16. S. Godik et al. *OASIS eXtensible access control 2 markup language (XACML) 3*. Tech. rep. OASIS, 2002.
17. Elgg homepage https://elgg.org/ *last accessed 10/05/15*
18. N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg. (2013, June). "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems." *In Proceedings of the IEEE Sixth International Conference on Cloud Computing, CLOUD (Vol. 13, pp. 887-894).*
19. C. Quinton, D. Romero, L. Duchien. (2015). SALOON: a platform for selecting and configuring cloud environments. *Software: Practice and Experience*.
20. J. Domaschka, K. Kritikos, A. Rossini (2014). Towards a generic language for scalability rules. In *Advances in Service-Oriented and Cloud Computing* (pp. 206-220). Springer International Publishing.
21. European Commission: Cloud Service Agreements Standardization Guidelines https://ec.europa.eu/digital-agenda/en/news/cloud-service-level-agreement-standardisation-guidelines *last accessed 10/05/15*
22. D. Heisenberg. *Negotiating privacy: The European Union, the United States, and personal data protection*. Lynne Rienner Publishers, 2005.
23. Alliance, Liberty. "Liberty alliance project." *Web page at http://www.projectliberty.org last accessed 10/05/15*
24. [24] Stepper, Eike. "CDO Model Repository Overview." (2012).
25. P. Massonet, J. Luna, A. Pannetrat, R. Trapero. "Idea: Optimising Multi-Cloud Deployments with Security Controls as Constraints." In Engineering Secure Software and Systems, pp. 102-110. Springer International Publishing, 2015.
26. L. Weiliang, L. Xu, Zhenxin Zhan, Q. Zheng, S. Xu. "Federated cloud security architecture for secure and agile clouds." In High Performance Cloud Auditing and Applications, pp. 169-188. Springer New York, 2014.
27. J. Gionta, A. Azab, W. Enck, P. Ning, X. Zhang. "Dacsa: A decoupled architecture for cloud security analysis." In Proceedings of the 7th Workshop on Cyber Security Experimentation and Test. USENIX. 2014.

# An Attribute Based Access Control Model for RESTful Services

Marc Hüffmeyer and Ulf Schreier

Furtwangen University of Applied Sciences, Germany

**Abstract.** RESTful services offer communication and interaction with information systems (e.g. mobile devices, sensor networks) through well know techniques in a large scale. As the popularity of REST grows more and more the need for fine-grained access control grows in the same way. Attribute Based Access Control (ABAC) seems to be the most suitable candidate to meet the requirements of flexibility and scalability. XACML is a very generic implementation of an access control system that follows the ideas and principles of ABAC and is established as the standard mechanism. Its flexibility opens the opportunity to specify detailed security policies. But on the other hand XACML has some drawbacks regarding maintenance and performance when it comes to complex security policies. Its generic design is the reason that authorization decisions only can be computed at runtime. Long processing times for authorization requests are the consequence in environments that require fine-grained or complex security policies. Our approach to implement ABAC for RESTful services is inspired by XACML and addresses its drawbacks by taking advantage of the style of resource oriented architectures. We describe a lightweight but powerful language that enables to specify security policies and to process access requests. We also describe algorithms and techniques to compute authorization decisions for that language in a very short time even for complex security policies.

## 1 Introduction

Today's information system often handle large amounts of data and users and perform complex operations. The potentials of information systems grow more and more and so does the need for flexible and efficient access control mechanism. Traditional access control mechanisms were built to support basic security concepts. For example Access Control Lists (ACL) were designed to specify *who* may access a single resource (e.g. a network interface or a file in an operating system) while Role Based Access Control (RBAC) groups multiple subjects together under a role property, reducing the amount of rules required to describe *who* may access a resource. Having applications that support complex processes requires more fine-grained mechanisms that can handle questions like *who*, *what*, *how*, *why*, *when* or *where* and that are capable to adapt to frequent changes.

In times of social media, smart objects and the Internet of things [15] users and systems often create and share new content within applications. A substantial need to control access to this content is the consequence. In an environment where large amounts of subjects provide large amounts of data and specify multiple access rights, a flexible, high-performance access control mechanism is required. Because traditional access control mechanisms have been designed for a different purpose, efficient access control mechanisms and models must be found, that offer flexibility and guarantee high performance even in complex environments.

## 2 REST

The concept of Representational State Transfer (REST) describes an architectural style for distributed systems and services [2]. Services that follow this style are usually called *RESTful*. A RESTful service must follow four main principles.

The first concept is **resource orientation and addressability**. Each resource is addressed with an URI that identifies the resource. A good URI design is important and might be a challenge [14]. URIs can be described with the expression *scheme:authority:path:query*. An example for an URI is *http://example.org/users/1/photos?date=20150101*. The scheme is *http*, the authority is *example.org*, the path is */users/1/photos* and the query is *date=20150101*. While scheme and authority are usually unchanged in one application, the path has a big impact on the application structure and requires a good design. A proper design has a hierarchical nature forming a graph of resources and subresources. A query can be interpreted as a filter that selects a subset of resources. In the example only the photos of a specific date are requested.

Another important concept of REST is a **uniform interface** for resources. For each resource the same finite set of actions may be executed. Usually REST is associated with HTTP and the HTTP methods specify the methods of the interface. That means for each resource GET, POST, PUT, DELETE and some other HTTP methods can be applied. That offers the opportunity to use standardized clients (e.g. browsers) to perform operations on a resource. The only required client capability is the support for the uniform interface.

The differentiation between **resources and representations** is the third concept of REST. A client requests a resource and a server returns a representation of that resource. For example the client may request a single user and the server responds with an identity card of that user which represents him. The client usually has the option to specify preferred representations of a resource.

The fourth concept is that **communication is stateless** in RESTful Services. That means that the server does not hold any information about the state of the communication, but only stores the state of the resource expressed as hypermedia. The application state is handled on client side. Therefore the concept is often called *Hypermedia as the engine of application state (HATEOAS)*.

## 3 Attribute Based Access Control

Attribute Based Access Control (ABAC) is a suitable candidate to fulfill the need for flexibilty and may be the next important concept for access control mechanisms [11]. The main idea of ABAC is that any property of an entity can be used to determine access decisions. For example the location of a subject or resource may be used to determine the access decision if the access conditions depends on the question *where*. Gartner predicts that in 2020 70% of enterprises will use ABAC as the dominant mechanism to protect critical assets [16].

### 3.1 eXtensible Access Control Markup Language

The eXtensible Access Control Markup Language (XACML) is a standard that describes how to implement ABAC and is established as a de facto standard [3]. It consists of three parts: an **architecture** describes multiple components and their responsibilities in the authorization context, a declarative **policy language** can be used to write security policies in XML and a **request/response language** can be used to formulate access requests and responses. This work focusses on the policy language and the request/response language and offers an alternative access control mechanism to XACML, which enables flexible and efficient access control for RESTful Services.

There are three core elements in the structure of a XACML policy: **Rules** describe if an access request is permitted or denied. **Policies** group different rules together and **policy sets** group different policies together. Policy sets may also contain other policy sets enforcing a hierarchical composition. Each of these elements has a **target** that describes if the element can be applied to a request by defining a condition. A single access request may be applied to multiple policy sets, policies and rules. In that case those rules may have different **effects** (*permit* or *deny*) and a winning rule must be found (based on the structure of the policy). XACML uses **combining algorithms** for that purpose. An example for a combining algorithm is *PermitOverrides*. It states that an applicable rule with the effect *permit* will always win against a rule with the effect *deny*. A full list of algorithms can be found in [3].

Listing 1 shows an example of a XACML policy for a web application. That policy prohibits the deletion of a photo list of a user using a HTTP DELETE request. The root element is a policy set with the combining algorithm *FirstApplicable*. The target of this policy set is empty, what means that the policy set can be applied to any request. The policy set contains one policy, which also has a combining algorithm of *FirstApplicable*. The target of the policy specifies one attribute condition for a resource: the *URI* must match the value */users/1/photos*. The policy in turn contains one rule with the effect *deny*. The rule declares a target with one attribute condition: the action must have an attribute named *HTTP-method* with the value *DELETE*. Additionally one can see *AllOf* and *AnyOf* elements which can be used to express logical conjunctions resp. disjunctions.

## 4 Problems with XACML for RESTful Services

We identified the following problems that come up if XACML is used to secure RESTful Services: performance, maintenance and target extensions.

```
<PolicySet PolicyCombiningAlgId="first-applicable">
  <Target/>
  <Policy RuleCombiningAlgId="first-applicable">
    <Target>
      <Match MatchId="function:string-equal">
        <AttributeValue>/users/1/photos</AttributeValue>
        <AttributeDesignator AttributeId="URI"
          Category="resource" />
      </Match>
    </Target>
    <Rule Effect="Deny">
      <Target>
        <AnyOf>
          <AllOf>
            <Match MatchId="function:string-equal">
              <AttributeValue>DELETE</AttributeValue>
              <AttributeDesignator AttributeId="HTTP-method"
                Category="action" />
            </Match>
          </AllOf>
        </AnyOf>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

**Listing 1.** A XACML policy prohibiting an HTTP DELETE request

### 4.1 Performance - Efficient Policy Design

In a previous work we described how to optimize processing time for XACML requests and reduce performance issues [5]. We have derived a cost function that calculates the cost for processing an access request. Based on the cost function we described three optimizations to write performance optimized policies. The first optimization requires having only as few as possible attribute conditions in rules, policies and policy sets so that variations in processing time for access requests are minimized. A second optimization needs a combining algorithm of *FirstApplicable* in addition with a proper rule and policy ordering. This optimization decreases the average processing time. Finally, the measurements of our previous work showed that the most important goal is to have a well-structured hierarchy in the security policy. Repetitive targets in different branches must be bundled together to decrease the number of child nodes and thereby the average processing time for an access request.

RESTful Services already ship with a well-structured hierarchy build on Uniform Resource Identifiers (URI). Efficient XACML policies therefore should be built on this hierarchy instead of complex attribute conditions. Following this approach access requests can be evaluated much faster against the policy. Our measurements showed that for single requests with a growing number of rules, the average processing time increases linear to the number of rules if the optimizations are not applied. Optimized versions of XACML policies only have a logarithmic growth and our measurements showed approximately constant processing times that are related to an initialization overhead of the XACML implementation.

### 4.2 Maintenance and Target Extensions

To decrease maintenance efforts it is required to have the security policy based on a well-structure hierarchy. Tool support for XACML is very poor and more complex policies may easily consist of hundreds of thousands lines of XML data (e.g. for measurement purposes we created a security policy with about 4.500 rules which consists of approximately 260.000 lines of XML data). Adding changes or detecting faults can be very time consuming and error prone because XACML does not specify how to handle changes in the security policy. Hence, a well-structured policy helps to reduce maintenance efforts. But in an environment, that enforces frequent changes to the security policy, it becomes challenging to adopt the security policy to the requirements in an acceptable time.

Another drawback of XACML is extending attribute conditions in targets. Targets are the only option to pass attribute conditions from an upper level to a lower level. But with every target the policy gets more restrictive. In consequence attribute conditions have to be repeated multiple times if they should be extended. For example a RESTful application may have a user list as a resource and this list is addressed with the URI */users*. A single user may be addressed with the URI */users/1*. The single user is a subresource of the user list and the security policy should have a structure of the same fashion. Now it may be that the user list should be only accessible by admins but the user resource should be accessible by admins and the user itself. The admin condition cannot be used in targets because a target cannot be extended which means that the single user never could access the subresource. The only proper way to handle this is repeating the admin condition multiple times for all users. That increases the policy complexity and maintenance efforts unnecessarily.

## 5 REST Access Control Language - RestACL

Based on the outcome described in Section 4 we wanted to build an access control mechanism that forces security responsibles to build policies with an easy and comprehensive structure and with an easy and efficient interpretation. For unexperienced developers REST can be a challenge itself. Therefore it would be a great benefit to have an access control mechanism that supports REST in an intuitive way and removes the drawbacks of XACML described in the previous sections. RestACL is a policy language that is inspired by XACML and that is made for RESTful Services. Its design is chosen in a way so that performance loss is minimized and that gives architects, developers and administrators of RESTful applications an intuitive way to handle information security. RestACLs foundation is a tree build of URIs. The leaves of that tree specify access conditions. XACML in contrast is founded on a tree on conditions.

Listing 2 shows an example of a RestACL policy. Like the example for XACML, the policy prohibits the execution of an HTTP DELETE request on the photos of a user. Additionally a second rule allows the execution of an HTTP DELETE request on a subset of photos. To select a subset it is required to add a filter based on query parameters to the URI. In the example, the additional rule grants the execution of a HTTP DELETE on those resources which have the location *furtwangen* and were taken on *January 1st, 2015*. The following URI applies this filter: *http://example.org/users/1/photos?location=furtwangen&date=20150101*.

```
<policy>
  <resource uri="http://example.org">
    <resource uri="/users/1/photos">
      <action id="action1" method="DELETE">
        <rule id="rule1" effect="deny" priority="1">
          <condition/>
        </rule>
      </action>
      <filter>
        <parameter name="location" value="furtwangen" />
        <parameter name="date" value="20150101" />
        <action id="action2" method="DELETE">
          <rule id="rule2" effect="permit" priority="2">
            <condition match="equal">
              <value>192.168.0.0</value>
              <designator category="environment">network
              </designator>
            </condition>
          </rule>
        </action>
      </filter>
    </resource>
  </resource>
</policy>
```

**Listing 2.** A RestACL policy regulating HTTP DELETE access

The structure of our model is resource oriented. Since RESTful applications have a hierarchical structure based on URIs, our access control model also has a hierarchical nature. The root element is named **policy**. It must contain one or more **resource** elements. Each resource element may contain multiple **action** elements, **filter** elements or again resource elements. Within each action element one or more **rule** elements may be specified which may contain a set of **condition** elements and within **filter** elements additional action elements may be contained. Without loss of generality condition elements within a single rule element are logically conjuncted while rule elements in a single action element are logically disjuncted. More complex logical operations can be integrated at a later point of time. Each rule must have an **effect** that permits or denies access. Additionally a rule may have a **priority**. The priority can be used to determine dominant rules.

Listing 3 shows a request that can be applied to the example policy in Listing 2. The root element is named **request**. A request must contain one **resource**. Similar to the policy a resource contains one **action**. In contrast to the policy a request must contain exactly one resource and one action. Within the action element a **set** of **attributes** may be specified. Like conditions in a policy, attributes contain a value and a designator.

```
<request>
  <resource uri="http://example.org/users/1/photos">
    <action method="DELETE">
      <set>
        <attribute>
          <value>192.168.0.0</value>
          <designator category="environment">network
          </designator>
        </attribute>
      </set>
    </action>
  </resource>
</request>
```

**Listing 3.** A RestACL request

We want to show that our access control model fully supports RESTful applications and does not conflict with the four main concepts of REST.

As described in Section 3 resources are addressed using URIs and URIs consists of a scheme, an authority, a path and a query. It is obvious that our model supports different schemes and authorities. For example if different schemes or authorities are bundled within one application, one has to declare different resources within the policy element. Building the policy based on a hierarchical path is the main idea of our access control model and query parameters are supported using the filter element as described in Listing 4.

A uniform interface is part of the core concept of our model. Actions are used to specify methods that can be applied to resources. RESTful services are usually built on HTTP, but this is not a requirement of the architectural style. Our model also is capable of supporting other access methods. For example a uniform interface could be built on top of the CRUD pattern. In our model this can be implemented by using *CREATE*, *READ*, *UPDATE* and *DELETE* as methods of the action element.

Most applications want access decisions to be based on the request and not on what the response contains. For RESTful applications that means that the access decision likely is based on the requested resource rather than on the returned representation. Therefore representations are not the focus for access control in a resource oriented context. But even if they are, representations can be handled as attribute conditions in a rule. For example a resource may have two different representations and those representations have different access conditions. Then two different rules with an attribute *representation* of the category *resource* may be specified. Listing 4 shows an example how to handle different access rights for multiple representations.

```
<resource uri="/users/1/photos">
  <action id="action1" method="GET">
    <rule id="rule1" effect="permit" priority="1">
      <condition match="equal">
        <value>image/png</value>
        <designator category="resource">representation
        </designator>
      </condition>
    </rule>
    <rule id="rule2" effect="deny" priority="2">
      <condition match="equal">
        <value>video/mpeg</value>
        <designator category="resource">representation
        </designator>
      </condition>
    </rule>
  </action>
</resource>
```

**Listing 4.** Representations with different access control rules

Stateless communication is not affected with our access control model. The model does not depend on any state information of a resource.

## 5.1 Data Model

Figure 1 shows the data model of a RestACL policy. A policy has a one-to-many relation to resources. Resources have a dedicated attribute named URI and one-to-many relations to filters and actions. Resources also have a self-reference enforcing a hierarchical structure. A filter consists of various parameters that declare a name and a value describing a query. Filters also contain actions. If a filter matches the URI of the request, the rules of the filter are used to determine the access decision instead of the rules that are directly assigned to the resource. An action has a method name and contains many rules which in turn consist of multiple conditions. A single condition has a match function (which must compute to a boolean value) and either a value and a designator or two designators. A designator uses a category and a name to identify attribute values in the request. This value is then compared either to a fixed value or the value identified by the second designator. Listing 2 and Listing 3 use a designator to identify an attribute named *network* of category *environment*. This value is then compared to *192.168.0.0* using the match function of the condition. Using two designators enables comparing two values within a request. This allows for example to write rules that grant or prohibit access to subjects if they are the owner of a resource without specifying explicit values. The policy in Listing 5 uses two designators in a single condition.

In this model targets and combining algorithms are obsolete. Targets can be removed because the structure of policies is always based on resources and actions. Because both URIs and the methods of a uniform interface are unique, it is not

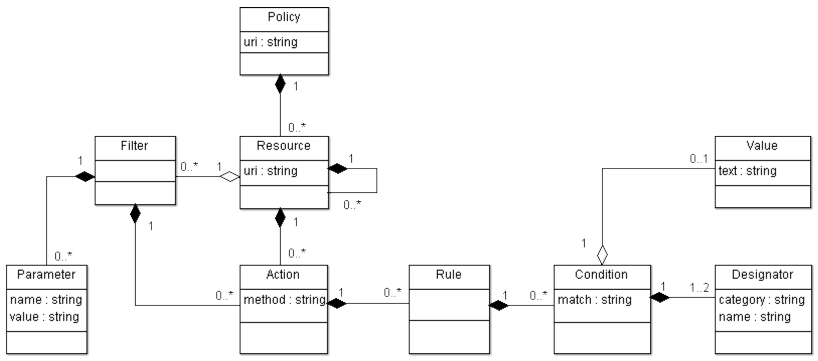


**Fig. 1.** RestACL data model

possible to have multiple matching branches. Hence, combining algorithms can be removed. To compare multiple rules within an action element we use priorities. Priorities can be interpreted as a combining algorithm of *FirstApplicable* and therefore offer the profit of performance optimization. Additionally they offer a bit more flexibility regarding maintenance.

### 5.2 References and Includes

To address the drawbacks of extending targets as described in Section 4 we enable rule references on subresource level. That means an action may refer to the action of another resource located above in the resource hierarchy. In that case all the rules of the referred action will also be applied to the referring action. All newly specified rules are logically disjuncted to the rules of the referred action. For efficient processing of access requests, the referred rules can be directly stored in the data model during the parsing phase of the access control policy.

Listing 5 shows the idea of references. A rule grants access to admins on a resource *users* using a HTTP GET request. A subresource */users/1* refers this rule using the **reference** attribute listed in an action element. Additionally a second rule is added that grants access for single users. The problem described in Section 4 can easily be addressed with the use of the *reference* attribute of the action element.

Additionally remote policies can be included using a *include* attribute. This enables splitting of large policies into multiple files and also offers the option of reusing conditions. Includes can be used in resource elements and therefore may contain other resource elements, filter elements and action elements.

```
<resource uri="/users">
  <action id="action1" method="GET">
    <rule id="rule1" effect="permit" priority="1">
      <condition match="equal">
        <value>admin</value>
        <designator category="subject">type</designator>
      </condition>
    </rule>
  </action>
  <resource uri="/1">
    <action id="action2" reference="action1" method="GET">
      <rule id="rule2" effect="permit" priority="2">
        <condition match="equal">
          <designator category="subject">id</designator>
          <designator category="resource">id</designator>
        </condition>
      </rule>
    </action>
  </resource>
</resource>
```

**Listing 5.** Rule referencing in RestACL

### 5.3 Evaluation Algorithm

Three steps are required to find a decision for an access request. First of all the requested resource must be selected in the policy. Then one has to identify the action that defines access rules for the request. Finally the rules for that action need to be evaluated. These steps must be repeated for all resources of a policy until the requested resource is found. Then the algorithm can stop the iteration.

**Definition:** We define $R(p)$ as the set of resources of a policy $p$.

```
Input: Policy $p$, Request $q$
Output: Effect $e$

$\forall r_i \in R(p)$
    $r = selectResource(r_i, q)$;
    $a = selectAction(r, q)$;
    $return\ evaluateRules(a, q)$;
```

**Algorithm 1:** findDecision

Identifying the selected resource (the requested resource) in the data model of the security policy means one has to traverse the graph of resources recursively. The algorithm must stop if the URIs of the security policy and the access request are logically equal (note that a filter may contain various query parameters in an undefined order).

**Definition:** We define $r'$ as a subresource of resource $r$ and $R'(r)$ as the set of subresources of $r$.

```
Input: Resource $r$, Request $q$
Output: Resource $s$

$if(match(uri(r),\ uri(q)))$
    $then\ return\ r;$
$\forall r'_i \in R'(r)$
    $selectResource(r'_i, q);$
```

**Algorithm 2:** selectResource

In the second step the corresponding actions must be found. That means one must compare the method of the request with methods of the actions specified for the selected resource. If they are identical, the algorithm must return the action.

**Definition:** We define $A(r)$ as the set of actions of resource $r$.

```
Input: Resource $r$, Request $q$
Output: Action $a$
$\forall a_i \in A(r)$
    $if(method(a_i)\ =\ method(q))$
        $then\ return\ a_i;$
```

**Algorithm 3:** selectAction

In the last step the rules of the selected actions must be evaluated. The algorithm must return the access decision for request $q$. Therefore one has to iterate over all rules (ordered by the priorities of the rules). For each rule one must check whether for all conditions there is an attribute in the request, so that the categories and attribute names are equal and the execution of the match function of the condition computes to true. If so the algorithm must return the effect of the rule. This is indicated in Algorithm 4. If the condition consists of two designators both of them must have an attribute that matches them and the match function is applied to the values of these attributes.

**Definition:** We define $U(a)$ as the set of rules of action $a$ and $C(u)$ as the set of conditions of rule $u$. Further we define $d(x)$ as a designator and $v(x)$ as a value of either a condition $c$ or an attribute $t$. Finally we define $T(q)$ as the set of attributes of request $q$.

```
**Input**: Action $a$, Request $q$
**Output**: Effect $e$
$\forall u \in U(a)$
    $if(\forall c \in C(u):$
        $\exists t \in T(q):$
            $category(d(c)) = category(d(t))$
            $name(d(c)) = name(d(t))$
            $match(v(c),\ v(t))$
    $then\ return\ effect(u);$
```

**Algorithm 4:** evaluateRules

A great benefit for performance is that algorithms 2, 3 and 4 are decoupled. That means traversing the graph of the security policy and evaluating access rules can be separated. This is different to XACML where the evaluation of a target conditions must be done for every node of the graph.

## 6 Tests

We performed load testing for RestACL and optimized XACML using synthetic policies. For each XACML policy we created a functionally equivalent RestACL version and compared these pairs under different load. The first pair of policies handles 10 resources and for each of the main HTTP methods we created one access rule. In the second pair we added 10 subresources to each resource and again specified one rule per HTTP method. In the third pair we added another 10 subresources to each subresource and specified one rule for each HTTP method. That means the different policies have 40, 440 and 4440 rules. We wanted to make statements regarding scalability depending on the number of rules and depending on simultaneous access requests. Therefore we f the processing time for 1, 10, 100 and 1000 simultaneous access requests. Figures 2-4 show the results of these tests.

The tests were executed on dual core system with 8 GB of working memory reserved for the tests. As XACML implementation we used Balana[1] which is an open source implementation provided by WSO2. For each test a minimum of 10 executions has been analyzed. In each request random attribute values have been used. In a first test setup we measured the processing times directly on code base without any protocol overhead. In a second setup we verify these results using Apache JMeter[2]. Therefore we set up XACML in the so called REST profile defined in version 3.0 of the XACML standard. The REST profile describes how a XACML engine can be implemented as a service itself. We used the same setup for RestACL. Load testing using Apache JMeter showed up similar results with higher average processing times related to the additional overhead due to the use of HTTP.

[1] https://github.com/wso2/balana
[2] http://jmeter.apache.org/

For optimized XACML policies the processing times increase with the number of rules contained in a policy and with the number of simultaneous access requests. A functionally equal but non-optimized version of the XACML policy required about 300ms to process one request at a time for the policy with 4440 rules. Therefore we decided to compare RestACL only with optimized XACML. Details of the difference between optimized and non-optimized XACML can be found in [5]. RestACL performs better than XACML for RESTful Services. This is related to the decoupling of graph traversion and rule evaluation. Also RestACL is more lightweight because targets and combining algorithms must not be considered during evaluation.

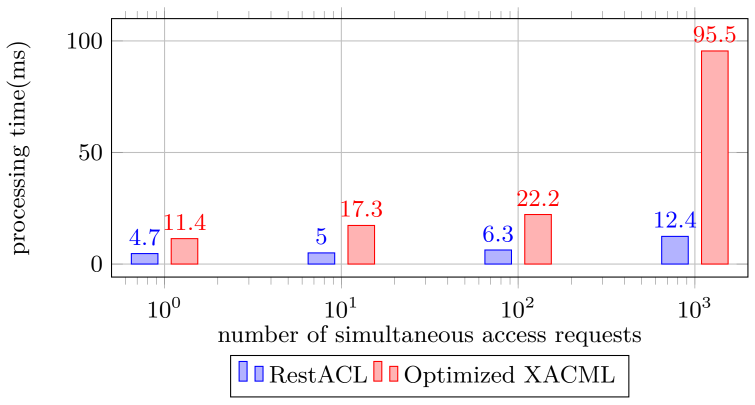


**Fig. 2.** Average processing time for a policy with 40 rules

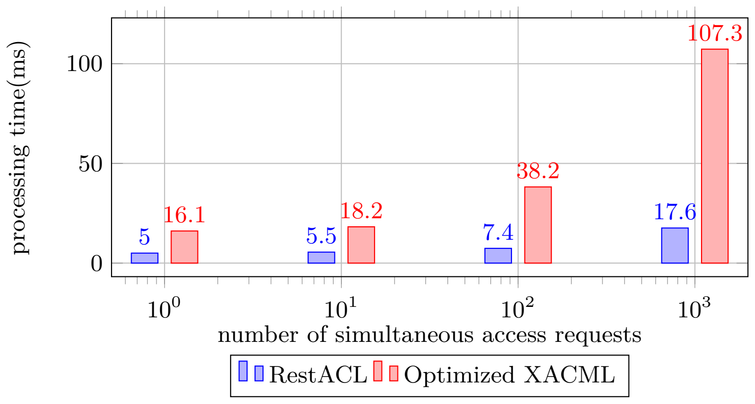


**Fig. 3.** Average processing time for a policy with 440 rules

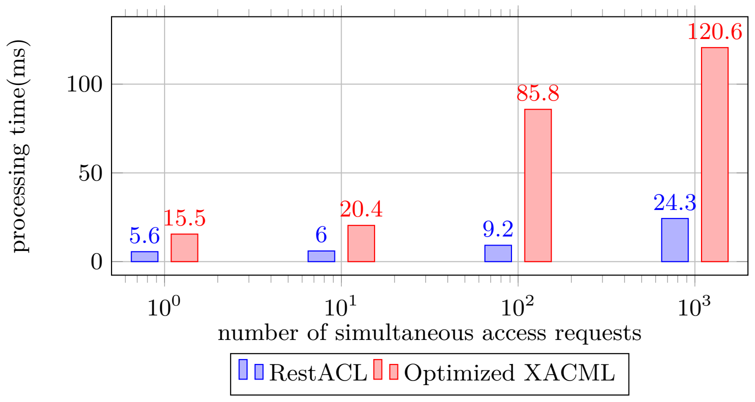


**Fig. 4.** Average processing time for a policy with 4440 rules

## 7 Related Work

Several approaches try to address performance problems with transformation and reordering techniques. A graph based approach is described in [10]. This approach uses two different trees to evaluate an access request. The first tree identifies applicable rules. The second tree holds the original structure of the security policy and determines the access decision. Another approach uses numericalization and normalization to optimize performance [6,7]. Numericalization converts every attribute to an integer value. Normalization converts every combining algorithm into FirstApplicable. In [8] processing time is optimized by reordering policy sets and policies based on statistics of past results. A similar approach also reorders policies based on cost functions but focusses on categories rather than attributes [9]. This approach also assumes that a rule always is a 4-tupel of a subject, an action, a resource and an effect. Other combinations of categories are not allowed.

Declarative authorization for RESTful services is handled in [4]. Attributes are not considered in this approach. Another approach that targets authorization for RESTful Services is described in [1]. But this work is focused on RBAC. In [17] an architecture is described to secure web services (SOAP) based on attributes. Another approach that is focused on SOAP is described in [12]. The detection of access control vulnerabilities in web applications is discussed in [13]. This work covers web applications that use RBAC as access mechanism.

## 8 Conclusions

We created a new access control model that is designed for RESTful Services and that is inspired by XACML. The model offers a lightweight mechanism to implement attribute based access control in a resource oriented environment and follows the guidelines to optimize processing time described in [5]. A similar structure of the RESTful application and the security policy is enforced, so that high performance and lower maintenance efforts can be guaranteed. Developers of RESTful Service is offered an intuitive way to implement information security based on attributes.

Test results showed that the performance of this access control model is an improvement to the performance of XACML policies in a resource oriented environment. We verified our results using two different test methods that showed similar results. Also we addressed the problem of extending access control rules in a hierarchical structure so that maintenance efforts can be reduced and security policies are less error prone.

## References

1. E. Brachmann, G. Dittmann, and K. Schubert. Simplified Authentication and Authorization for RESTful Services in Trusted Environments. *ESOCC'12 Proceedings of the First European conference on Service-Oriented and Cloud Computing*, 2012.
2. T. R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. *University of California, Irvine*, 2000.
3. Organization for the Advancement of Structured Information Standard. eXtensible Access Control Markup Language (XACML) Version 3.0. *OASIS Standard*, 2013.
4. S. Graf, V. Zholudev, L. Lewandowski, and M. Waldvogel. Hecate, Managing Authorization with RESTful XML. *WS-REST '11*, 2011.
5. M. Hueffmeyer and U. Schreier. Efficient Attribute Based Access Control for RESTful Services. *ZEUS '15*, 2015.
6. A. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: A Fast and Scalable XACML Policy Evaluation Engines. *SIGMETRICS '08*, 2008.
7. A. Liu, F. Chen, J. Hwang, and T. Xie. Designing Fast and Scalable XACML Policy Evaluation Engines. *IEEE Transactions on Compters, Vol. 60*, 2011.
8. F. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran. Adaptive Reordering and Clustering-Based Framework for Efficient XACML Policy Evaluation. *IEEE Transactions on Services Computing, Vol 4*, 2010.
9. P. Miseldine. Automated XACML Policy Reconfiguration for Evaluation Optimisation. *SESS '08*, 2008.
10. S. Ros, M. Lischka, and F. Marmol. Graph-Based XACML Evaluation. *SACMAT '12*, 2012.
11. D. Sandhu. The authorization leap from rights to attributes: maturation or chaos? *SACMAT '12*, 2012.
12. H. Shen and F. Hong. An Attribute Based Access Control Model for Web Services. *Parallel and Distributed Computing, Applications and Technologies, PDCAT '06*, 2006.
13. F. Sun, L. Xu, and Z. Su. Static Detection of Access Control Vulnerabilities in Web Applications. *SEC'11 Proceedings of the 20th USENIX conference on Security*, 2011.
14. S. Tilkov. *REST und HTTP*. dpunkt.verlag, 2009.
15. D. Uckelmann, M. Harrison, and F. Michahelles. *Architecting the Internet of Things*. Springer, 2011.
16. E. Wagner, R. andl Perkins, F. Kreizman, G. Gaehtgens, and A. Allan. *Identity and Access Management 2020*. Gartner, 2013.
17. E. Yuan and J. Tong. Attributed Based Access Control (ABAC) for Web Services. *ICWS 2005 IEEE International Conference on Web Services*, 2005.

# SitRS – A Situation Recognition Service based on Modeling and Executing Situation Templates

Pascal Hirmer[1], Matthias Wieland[1], Holger Schwarz[1], Bernhard Mitschang[1], Uwe Breitenbücher[2], and Frank Leymann[2]

[1] Universität Stuttgart, Institute of Parallel and Distributed Systems, 70569 Stuttgart, Germany
pascal.hirmer@ipvs.uni-stuttgart.de
http://www.ipvs.uni-stuttgart.de/

[2] Universität Stuttgart, Institute of Architecture of Application Systems, 70569 Stuttgart, Germany
http://www.iaas.uni-stuttgart.de/

**Abstract.** Today, the Internet of Things has evolved due to an advanced connectivity of physical objects. Furthermore, Cloud Computing gains more and more interest for the provisioning of services. In this paper, we want to further improve the integration of these two areas by providing a cloud-based situation recognition service – SitRS. This service can be used to integrate real world objects – the *things* – into the internet by deriving their situational state based on sensors. This enables context-aware applications to detect events in a smart environment. SitRS is a basic service enabling a generic and easy implementation of Smart* applications such as SmartFactorys, SmartCities, SmartHomes. This paper introduces an approach containing a method and a system architecture for the realization of such a service. The core steps of the method are: (i) registration of the sensors, (ii) modeling of the situation, and (iii) execution of the situation recognition. Furthermore, a prototypical implementation of SitRS is presented and evaluated via runtime measurements.

**Keywords:** Situation Recognition, IoT, Context, Integration, Cloud Computing, OSLC

## 1 Introduction

A major challenge for the Internet of Things (IoT) is sensor data integration and sensor data processing [10]. The sensor access should be pervasive and the integration of the sensors has to be automated. Furthermore, the sensor data have to be interpreted in order to derive situations that can be understood and processed more easily than the huge amount of low-level data, which is difficult to handle. To enable situation-awareness for the IoT, different levels of processing are needed. These levels are described in Fig. 1. Here, the first level – the data level – contains the sensor devices. On this level only the raw sensor data is available, which is very complex and difficult to process. Because

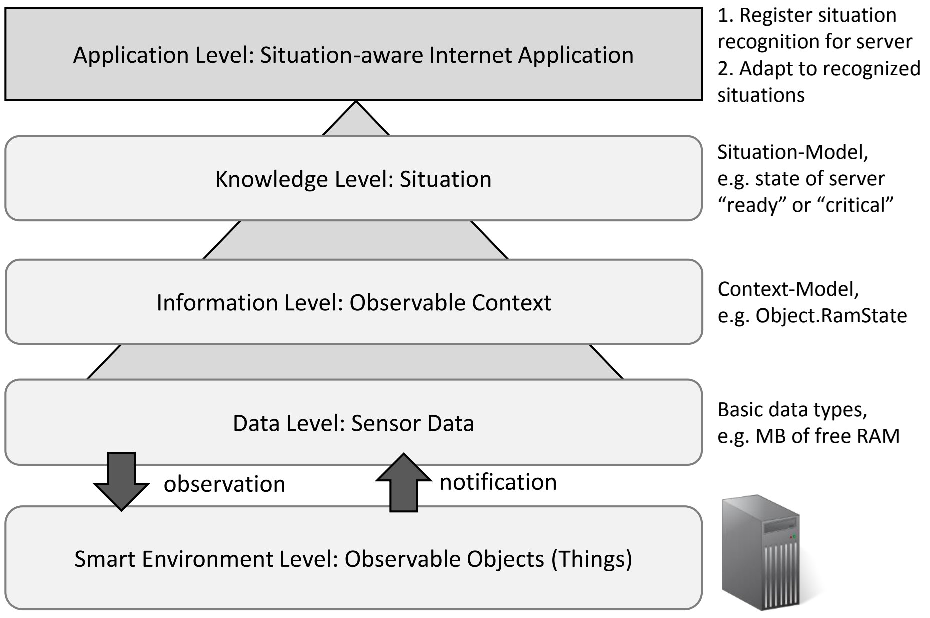

Application Level: Situation-aware Internet Application
1. Register situation recognition for server
2. Adapt to recognized situations
Knowledge Level: Situation
Situation-Model, e.g. state of server "ready" or "critical"
Information Level: Observable Context
Context-Model, e.g. Object.RamState
Data Level: Sensor Data
Basic data types, e.g. MB of free RAM
observation
notification
Smart Environment Level: Observable Objects (Things)

**Fig. 1.** Transition Levels from Data via Information to Knowledge

of that, the sensors are pushing their data to the next level – the information level. At this point, the sensor data, such as temperature or load percentage, is enhanced with information about their relations to objects, such as smart phones or computers in a smart environment. On the information level, this data i.e., the observable context, is linked to real world objects of the smart environment and becomes information about the environment. Based on this context information sensor data is aggregated and interpreted in order to derive well-understandable situations that lead to knowledge about the smart environment. This knowledge, i.e. high-level context, can be processed on a higher-level of abstraction, which simplifies building situation-aware applications.

A method and a system architecture to provide this sensor data processing for situation recognition as a service in an automatic, cloud-based manner are the main contributions of this paper. Our system architecture supports automated service deployment, a web-based front-end and loose coupling. This has many advantages like concurrent remote access, high availability and scalability in order to support multiple instances of our service as well as the integration of many distributed sensors. In addition, we provide a means to define the situations that could occur, that is, a model containing all necessary information for their recognition. This model, called *situation template* in this paper, contains the sensors being monitored as well as the conditions that have to match for a certain situation. Once the model is created, it can be used to execute a data flow that integrates the sensor information and executes comparison operations to recognize occurring situations. The result of the processing is the recognition of situations that allow applications to adapt to the smart environments observed by the sensors. The advantage of such a service is, that the applications do not have to care about the sensor access, the sensor data processing and not even about the situation recognition. Instead, the applications only query the needed knowledge or register for push notifications on occurring situations. The service cares for finding appropriate sensor devices, storing the context data for queries, providing a registration service for push notifications and finally automatically setting up the situation recognition for the needed situations. This enables smart applications to integrate real world objects into the internet by deriving their situational state based on sensors.

The remainder of this paper is structured as follows: First, *Section 2* introduces related work. After that, *Section 3* presents an architecture and method for situation recognition that copes with the mentioned issues and enables situation recognition based on sensor data. Afterwards, in *Section 4* we present our prototypical implementation of SitRS. *Section 5* evaluates the approach using runtime measurements and finally *Section 6* gives a summary of the paper and an outlook on future work.

**Motivating Scenario:** This section introduces a motivating scenario that is used throughout the paper to explain our approach. The goal of this scenario is the monitoring of sensors of several machines simultaneously and the reaction on occurring situations. For example, these machines could be web servers or cloud-based virtual machines in a data center. Using a dashboard, the currently occurring situation of all machines and, as a consequence, the state of a web server or a data center can be seen immediately. It's even possible to receive notifications in case of emerging problems. For that, we define three types of situations: (i) "Failed" indicates that the system is not available due to an occurred error, (ii) "Critical" indicates an occurring problem that could lead to a system failure (cf. example in Fig. 4) and (iii) "Running" indicates that no problem is occurring or emerging. The sensor data that is used to recognize these situations is provided by heterogeneous APIs, depending on the respective machine. A main challenge in this scenario is (i) coping with different representations of the sensor data, (ii) integrating the sensor data, (iii) computing the situation, and (iv) integrating highly heterogeneous APIs. Our solution is able to realize this scenario by representing sensor data as uniform REST resources and by integrating and analyzing them using a data flow-based integration. We will explain the following concepts based on this motivating scenario, which has also been implemented in our prototype.

## 2 Related Work

Acquiring, modeling and managing context information is a tedious and expensive task [6, 8]. As a consequence, it is beneficial to share this information between different kinds of context-aware applications. We use the definition of context given by A.K. Dey and G.D. Abowd as "any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or object" [5]. Thus, as Dey and Abowd already defined, context information can be used to identify and derive situations. Context models were introduced in previous work [6] to represent or mirror certain aspects of the real world as closely as possible thereby serving as a shared, common basis for different context-aware applications and systems. In this paper, however, we concentrate on how context and context models can be used to recognize situations. So the basic idea is to enhance an existing context model infrastructure with a situation recognition service based on so called situation templates. A situation template is an abstract, machine-readable description of a certain basic situation, which describes context information considered for being relevant for the situation and a description of how to derive the existence of a situation from these values. Situation templates were introduced before and this paper builds on the definition presented in [7]. Due to the historical development of rule-based expert systems, most context reasoning systems [11] use ontology-based and predefined rule-based approaches. Compared to our approach, most of the existing context-aware systems are supposed to cover only a limited geographical area or support only a specific use case scenario [2]. In our approach, any geographical area can be supported using a global context model and in addition any kind of situation recognition can be modeled as situation template based on the available context model. Unlike situation recognition approaches that are based on pattern recognition using e.g., machine learning [1] or on ontological reasoning [4] our approach executes the situation recognition as a data flow. Furthermore, complex event processing (CEP) [3] engines can be used for data flow execution in our approach. Hence, the only errors and uncertainties in the process result from the sensors and their sensor data readings. The data flow processing is accurate.

For the execution of the situation recognition, we use the *Pipes and Filters* pattern [9] – which is implemented in our prototype using Node-RED[3] – and build on a transformation approach presented previously in [12]. There, the concept of mapping the *Pipes and Filters* pattern to an executable representation was presented. In this paper, this concept is enhanced by a more detailed approach introducing a method for situation recognition as well as a prototypical implementation.

## 3 SitRS – Architecture and Method

The SitRS architecture, displayed in Fig. 2, consists of the situation model, the situation recognition service, and the sensors. The components of the situation recognition service can be deployed as cloud services, on a local machine or in a hybrid manner. The service is subdivided into two core components, the *situation recognition system* and the *resource management platform* and furthermore contains two repositories, one for storing the situation templates and the other for storing sensor information. In addition, it contains the following software components: the situation registration service and the sensor adapters.

[3] http://nodered.org/

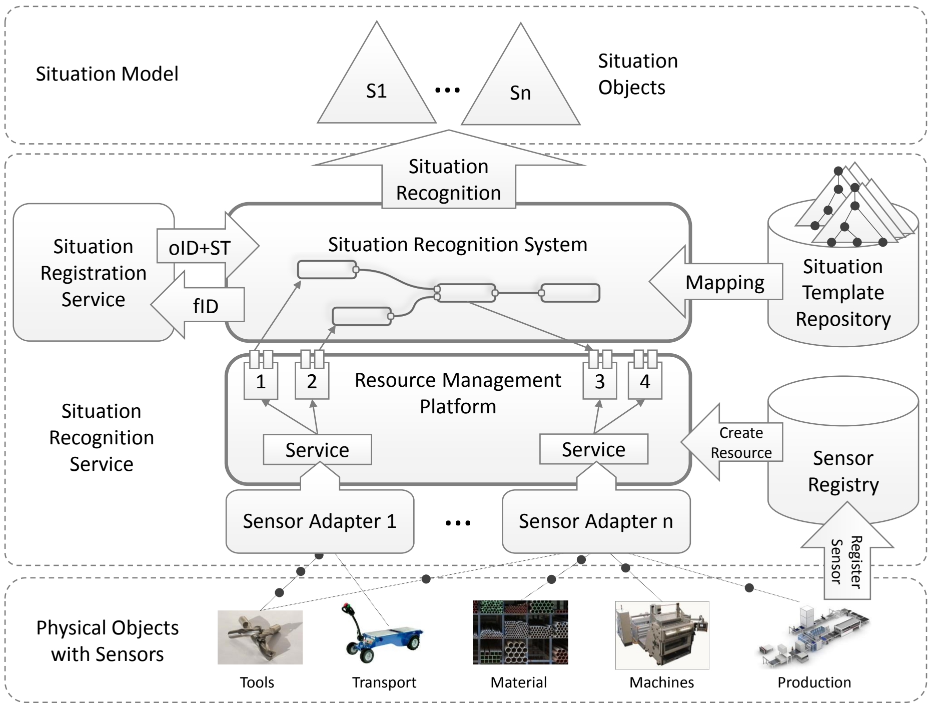


**Fig. 2.** SitRS – Architecture

The sensors at the bottom level can be registered in the Sensor Registry, which invokes the resource management platform that extracts the sensor data via the adapters and provisions them as uniform REST resources. Based on the registered sensors, a description defining the conditions for an occurring situation is modeled using so called situation templates. These situation templates are stored in the Situation Template Repository. The Situation Registration Service is used for the registration on occurring situations based on the situation templates. The situation templates are mapped onto an executable representation – we call *executable situation template* in the context of this paper – and executed in the Situation Recognition System, i.e., an execution engine. The output of this engine determines if modeled situations occurred. Note that the mapping from a situation template to an executable representation is necessary to support different execution engines, i.e., to prevent being dependent on a specific engine.

The introduced architecture is used as shown in Fig. 3. There are two kinds of actors participating in this method, the *situation recognition user* and the *situation recognition admin*. The admin has to register the sensors to be used. The situation recognition user models the situation to be recognized as situation template and processes the notifications of the situation recognition. This separation enables the usage by non-expert users regarding sensor integration and technical details. The method contains all steps needed for defining the

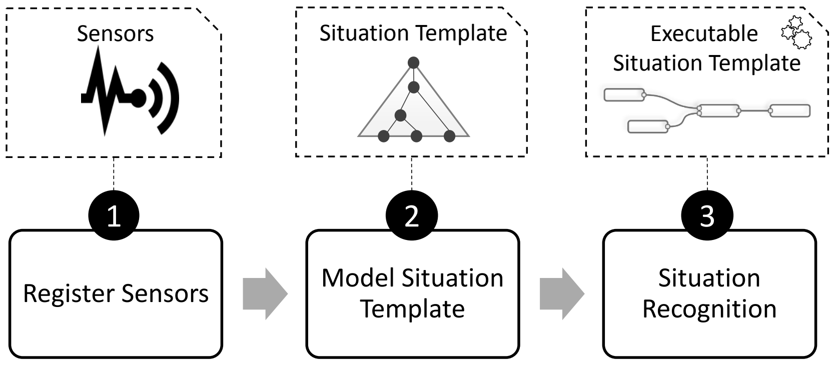
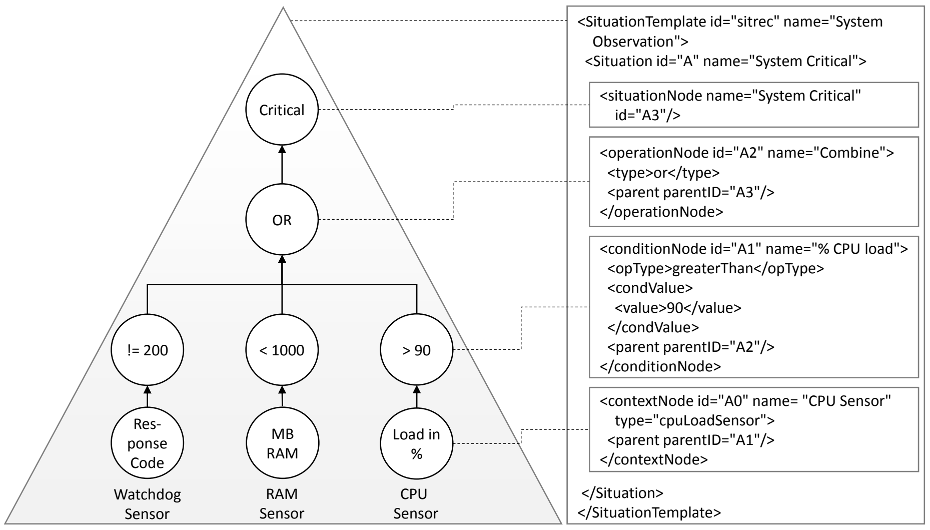


**Fig. 3.** Method for Situation Recognition

continuous recognition of a situation. Due to a design decision, only a situation for a *single* object, e.g. a web server, can be monitored by our approach. As a consequence, this method has to be re-applied for different objects. Because of that, it makes sense to create a single (cloud-based) instance of the service for each object to be monitored. Recognizing situations regarding multiple objects is part of our future work. The overall method consists of the following steps:

**Step 1 – Sensor Registration:** In the first step of the situation recognition method, the available sensors are registered in the sensor registry component (cf. Fig. 2). This registry is connected to the resource management platform, which provides the sensor's data as uniform REST resources. To register a sensor of a specific object, e.g., the heat sensor of a machine, the object's id, the type of the sensor as well as its access path have to be specified. Thereupon, an entry is created in the sensor registry containing the given information and an unique id of the registered sensor. Once a sensor is registered, an event is generated that notifies the resource management platform. Thereupon, this platform creates an adapter to connect to the sensor and provides its data through a REST resource. Note that each sensor is represented by exactly one REST resource. The URI of this resource can be requested from the sensor registry using the sensor's id and is used for the transformation of a situation template to an executable representation, which is described in Step 3.2.

**Step 2 – Situation Template Modeling:** Before we are able to recognize situations, we need a means to define them. To enable this, we build *situation templates* (ST), using Situation-Aggregation-Trees (SAT) that were defined by Zweigle et al. [13]. These SATs are directed graphs resulting in a tree structure, in which the branches are aggregated bottom-up (as shown in Fig. 4). As a consequence, all paths are joined in a single root node that represents the situation. The leaf nodes of the situation template – called *context nodes* – represent the sensors. These context nodes are connected to condition nodes for



```
<SituationTemplate id="sitrec" name="System
  Observation">
 <Situation id="A" name="System Critical">
  <situationNode name="System Critical"
     id="A3"/>
  <operationNode id="A2" name="Combine">
    <type>or</type>
    <parent parentID="A3"/>
  </operationNode>
  <conditionNode id="A1" name="% CPU load">
    <opType>greaterThan</opType>
    <condValue>
      <value>90</value>
    </condValue>
    <parent parentID="A2"/>
  </conditionNode>
  <contextNode id="A0" name= "CPU Sensor"
     type="cpuLoadSensor">
    <parent parentID="A1"/>
  </contextNode>
 </Situation>
</SituationTemplate>
```

**Fig. 4.** Example of a Situation Template modeled in XML

filtering the incoming sensor data based on a condition. The output of these condition nodes can be aggregated by operation nodes using logical operations until the root node is reached. In previous work, no machine-readable format has been properly defined for the exchange and definition of these SATs, which is important to enable automated processing. To overcome this issue, we propose a schema based on XML. Of course, other formats such as JSON could be used as well. Note that modeling XML manually is a time-consuming and also error-prone task due to the lack of an automated schema validation. To cope with this issue, we recommend using existing XML modeling tools, both graphically or textually. Due to the fact that a large variety of XML modeling tools already exist, we do not provide an additional modeling tool for situation templates.

Figure 4 shows an example of a situation template that serves the recognition of the situation "Critical" of a web server as described in the motivating scenario in Section 1. To model such a situation, firstly, the available sensors of the machine have to be modeled using context nodes that are containing the type of the sensor. These context nodes are then connected to condition nodes that compare the sensor's data with predefined values. In the shown example, (i) the CPU load percentage should be greater than 90, (ii) the available RAM should be lower than 1000 MB, and (iii) the response code of the machine should not equal 200 in order to produce the output *true*. These condition nodes are aggregated using operation nodes that represent logical operators, in this specific example the OR operation node. The root of the SAT is the situation itself, i.e., the situation occurs if the root node evaluates to *true*.

In the following, we describe the individual parts of a situation template in detail, which is defined using an XML schema definition that can be found online[4]. Each situation template has a unique identifier, a name and may contain an arbitrary number of situations. This enables the simultaneous monitoring of many different occurring situations within a single situation template. A situation describes, which conditions have to apply for its occurrence, i.e., it is defined by a directed tree. This tree contains a single root node, the situation node, which occurs once inside a situation. The situation node describes the situation to be monitored. A situation is uniquely defined by an identifier and its name. Furthermore, a situation contains an arbitrary number of context nodes, condition nodes and operation nodes. These nodes are connected using the *parent* element, which contains a reference to the parent node.

Context nodes are used to describe the sensors that provide the data and are defined with an identifier, a name and its type. Detailed information about the sensor can be requested from the registry using the type attribute of the context node as well as the identifier of the monitored object. Note that each sensor that is being modeled has to be registered in the sensor registry first (cf. Step 1). The parent nodes of context nodes are always condition nodes, as shown in Fig. 4.

Condition nodes are used to compare sensor data with values that are predefined in the situation template. Possible types of condition nodes are *greater than*, *less than*, *equals*, *not equals* and *between*. The value used for comparison can be determined in the XML element *condValue*. Furthermore, each condition node can have an arbitrary number of operation nodes as parents.

Operation nodes are used to aggregate the output of the condition nodes and are restricted to the logical operations *AND*, *OR*, *XOR* and *NOT*. That is, a situation usually occurs if more than one condition applies. However, if a situation is dependent on only one condition, no operation nodes have to be modeled. The parent of an operation node is either a single situation node or an arbitrary number of operation nodes.

To ensure reusability and concurrent access, the modeled situation templates are stored in the *situation template repository*.

**Step 3 – Situation Recognition:** The third step of our method is subdivided into several sub-steps that are shown in Fig. 5 and are described in the following.

**Step 3.1 – Situation Registration:** The situation registry serves the registration on a specific situation to be recognized. The input of the situation registry is the id of the situation template as well as the id of the object (oID) to be monitored. On successful registration, the situation registration service returns an observation flow instance id (fID) that can be used for deregistration or management purposes. Once a situation is registered, an event is generated that invokes the transformation of the situation template. This transformation receives the situation template from the situation template repository using the given id and transforms it into the executable situation template. After that, this executable situation template can be deployed and executed in the respective

[4] http://pastebin.com/TyBNPUEs

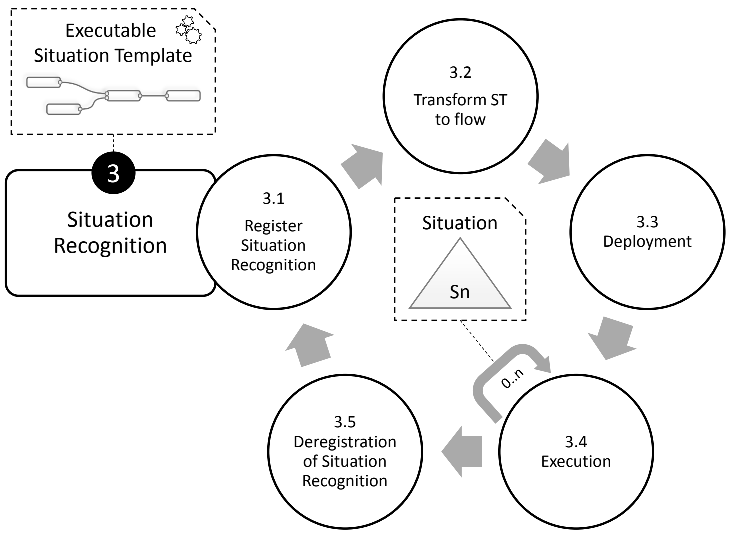

Executable Situation Template
3
Situation Recognition
3.1 Register Situation Recognition
3.2 Transform ST to flow
3.3 Deployment
Situation
Sn
0..n
3.4 Execution
3.5 Deregistration of Situation Recognition

**Fig. 5.** Detailed View of the Situation Recognition Step

runtime environment. The execution runs until a situation template or all objects relating to a situation template are deregistered in the situation registry.

**Step 3.2 – Situation Template Transformation:** The transformation of situation templates serves the creation of an executable, event- and flow-based representation that is able to recognize occurring situations based on the modeled situation template. The input of the mapping is the identifier of the object to be monitored (e.g., a web server) that was entered in the situation registry. In our prototypical implementation, e.g., the format of the executable representation is defined in JSON so it can be executed in the Node-RED environment. However, depending on the execution environment, many different formats are possible. We provide a 1-to-1 transformation from the XML-based situation template to an executable representation. That is, each element of the situation template is represented by exactly one element in the executable representation. In the first step of the transformation, we map the context nodes onto calls of REST resources that provide the latest sensor data. To do so, we receive the access information, i.e. the URL of the resource, from the sensor registry (cf. Step 1), using the object id from the mapping's input and the type of the sensor defined in the situation template. The second step of the mapping processes the condition nodes, i.e., the nodes implementing comparison operations such as *greater than*, *less than* or *equals* that compare sensor data with predefined values. These condition nodes are mapped to predefined function nodes, implementing the comparison operations, e.g., using JavaScript in Node-RED. In a similar fashion, the third step maps the condition nodes *AND*, *OR*, *XOR* or *NOT* to corresponding function nodes that implement these logical operators. In the final step, the nodes are connected using the means of the respective execution model. The result is an executable situation template that recognizes occurring situations through its execution. The time interval, in which the data flow will be executed, that is, in which a situation should be monitored, has to be predefined by the user of the solution and is used as input for the transformation. This is necessary, because the execution time interval strongly depends on the use case. For each situation, modeled in the situation template, a single flow graph is created and can be deployed and executed separately.

**Step 3.3 – Situation Template Deployment:** After its transformation, the executable situation template is deployed into the execution environment, e.g., Node-RED, CEP-Esper[5] or Odysseus[6]. As a consequence, the deployment serves as the interface to the execution engines being used. It should be flexible enough to support different engines and should also be able to handle occurring errors during the deployment. Operations supported by the deployment are *deploy*, *start situation recognition* and *stop situation recognition*. In our prototype, for example, the deployment sends a HTTP REST call to the Node-RED engine to deploy a mapped situation template. After that, the situation recognition flow is initiated by executing the *start* command. The situation recognition is active as long as the modeled situation should be monitored, i.e., until it is deregistered in the situation registry.

**Step 3.4 – Situation Template Execution:** After the deployment, the executable situation template is executed using the respective execution environment, e.g., an event-processing engine such as Node-RED. In the predetermined time interval, the sensor data is requested from the REST resources that return the latest sensor data. Thereupon, the further nodes of the situation template are processed. These are always condition nodes that were mapped onto predefined function nodes that compare the sensor data with predefined values and return a Boolean value determining whether the condition applies. After each of these condition nodes is processed, their output is concatenated using the mapped operation nodes that implement logical operations. The concatenation of the paths is processed until a single output emerges. This output is a Boolean value, determining whether a situation occurred or not. This flow is repeated in the given time interval until the situation is deregistered.

**Step 3.5 – Situation Deregistration:** The final step of the situation recognition is the deregistration of a situation template for a certain object. After the need for the recognition of a situation expires, it is deregistered in the situation registry. In case no more registrations exist for a situation, two steps are processed. First, the execution engine stops the situation recognition flow. Second, the executable situation template is undeployed from its execution environment.

[5] http://esper.codehaus.org/ [6] http://odysseus.informatik.uni-oldenburg.de/

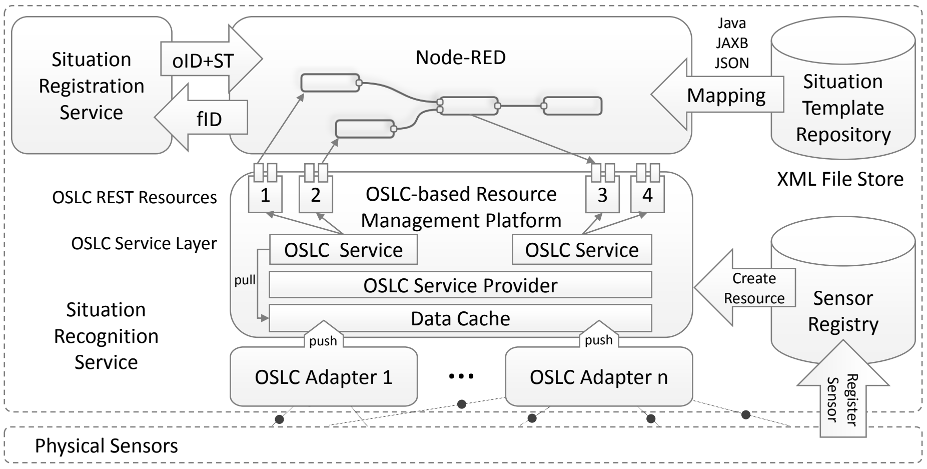


**Fig. 6.** Architecture of the SitRS Prototype

The deregistration of a situation secures that no unnecessary resources are spent for an (even temporary) unneeded situation recognition.

## 4 Prototypical Implementation

In this chapter, we describe our prototypical implementation of the introduced concepts. The overall architecture of the prototype is shown in Fig. 6. Furthermore, the prototype is available on GitHub (https://github.com/hirmerpl/SitOPT).

Firstly, we implemented a *mapping of situation templates* defined in XML to an executable representation in JSON. This mapping has been implemented as a Java library using the Java Architecture for XML Binding (JAXB), which is used to parse the situation template. Furthermore, we used the Apache Wink[7] JSON library to create a JSON model for the executable representation.

After the mapping is processed, we *deploy the executable situation template to Node-RED* using the provided HTTP REST interface. There exist many technologies that could have been used for processing the situation template such as RestFlow[8]. However, for our prototype, it is a requirement that the used engine is web-based, RESTful and offers a graphical user interface to enable an easier development as well as advantages in debugging. Because of that, we used Node-RED here, which provides a nice user interface showing a graphical representation of the executed flows, supports automatic deployment and offers a native REST support. The flow is started automatically and processes the situation recognition in predefined time intervals.

The *resource management platform provides sensor data of heterogeneous sources* to be processed by the executable situation template deployed in Node-RED. The resource management platform is currently being implemented using

[7] https://wink.apache.org/ [8] https://github.com/restflow-org/

Eclipse Lyo[9] – the Java-based implementation of the Open Services for Lifecycle Collaboration (OSLC)[10] specification –, however, it is not yet available in our prototype. Same with the sensor registry that could e.g, be realized using a web service with an underlying database to store the sensor's information.

The implementation of the resource management platform *is based on OSLC* because OSLC provides a mature specification that describes how to provide data as uniform REST services. In this paper, we use a push approach from the sensors to the resource management platform and a pull approach from the situation recognition system to the resource management platform. This mixed push/pull-approach is necessary because the situation recognition processes the sensor data independent of the sensors' reaction, i.e., sensor values have to be available at all times not only if pushed by the sensors. The resource management platform consists of (i) OSLC adapters to connect to the sensor data sources, (ii) a data cache to store intermediate data, (iii) an OSLC service provider managing OSLC services, (iv) OSLC services that create, modify or delete REST resources, and (v) the *REST resources themselves that provide data of the connected sensors* to enable uniform accessibility. The architecture of these components is displayed in Fig. 6. Note that the OSLC specification is usually used for the integration of lifecycle tools for software development. For our prototype, we designed an *OSLC-inspired* architecture that transfers the concepts of OSLC to enable the integration of sensor data sources. As a consequence, our design could slightly differ from the OSLC specification. In the following, the components of the resource management platform are described in detail:

**OSLC Adapter:** An OSLC adapter is used to connect to a sensor's API in order to extract its data. This can be realized using either a push or pull approach. In the pull approach, the adapter requests the data from the sensor, in the push approach, the data is sent directly to the OSLC adapter as soon as the sensor reacts. Note that the pull approach requires a sensor API that caches its data and provides it on request, independent of the sensor's reaction. In the motivating scenario, an adapter for each machine to be monitored has to be created by accessing the machine's sensor APIs and by extracting data, e.g., the CPU load, the currently available RAM or the CPU temperature. This data is stored into a data cache, e.g. a key-value store, to be available on request. Note that details about the binding of the sensors are part of our future work.

**OSLC Service Provider:** The OSLC service provider represents the entry point of the platform and manages the OSLC services that provide the REST resources. In our approach, we use a single service provider, managing all services.

**OSLC Services:** OSLC services are responsible for the *on-demand* creation, modification and removal of REST resources. Each service represents an object to be monitored. This object may contain an arbitrary number of sensors. For each sensor of an object, an OSLC REST resource providing the sensor data is created by these services.

**REST Resources:** The REST resources represent the interface to the user of our OSLC platform, that is, the situation recognition. The data extracted by the

[9] http://eclipse.org/lyo/ [10] http://open-services.net/

**Table 1.** Runtime Measurements of the Prototype

| Measurement | ST Transformation | ST Deployment | ST Execution |
|---|---|---|---|
| 1 | 219 ms | 141 ms | 6 ms |
| 2 | 219 ms | 126 ms | 6 ms |
| 3 | 234 ms | 125 ms | 5 ms |
| 4 | 203 ms | 141 ms | 5 ms |
| 5 | 204 ms | 140 ms | 6 ms |
| ∅ | 215,8 ms | 134,6 ms | 5,6 ms |

OSLC adapters is accessed through the data cache and is made available through a RESTful interface, providing the uniform methods GET, PUT, POST and DELETE that can be invoked using the Hypertext Transfer Protocol (HTTP). The actual implementation of the resources is defined in the corresponding OSLC service. In our prototype, only the GET and DELETE methods are relevant to either receive the sensor data or delete the resource if a sensor is deregistered.

Currently, the SitRS prototype has the following limitations: Firstly, it is not yet possible to compare sensor values with each other. It is only possible to compare sensor data with fixed values. Secondly, no concept of time exists because it is focus of this paper to recognize only current situations.

## 5 Evaluation

This section contains the evaluation of our approach by presenting runtime measurements and a load test based on the prototypical implementation.

To conduct the runtime measurements, we used an Ubuntu image hosted on Openstack[11] with 8 GB RAM and 8 Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz CPUs for our measurements. We measured the runtime of the situation template transformation, the deployment and the execution, separately. The situation template we used for these measurements monitors a remote machine, modeled as shown in the example in Fig. 4. All in all, this situation template contains 8 nodes to be mapped, deployed and executed. Table 1 shows the measurement results. These measurements are based on the transformation, deployment and execution of a single situation template. Our measurements are used as proof of concept that the introduced steps are processed in a reasonable time.

We further executed a load test to check how many situation templates can be transformed, deployed and executed in parallel inside a single runtime environment, using the same situation template as above. The results are shown in Table 2. As displayed, the runtime highly increases with increasing situations to be monitored in parallel. Our measurements show that executing two flows in parallel increases the runtime to 38 ms, when executing ten flows in parallel even to 404 ms. This means, Node-RED produces an overhead when processing multiple parallelized flows. This happens due to Node-RED's inability to process the flows in parallel using multiple threads. Furthermore, the internal execution

[11] http://www.openstack.org/

**Table 2.** Load Test of the Prototype

| # ST | Transformation ∅ | Deployment ∅ | Parallel Runtime ∅ | Sequential Runtime ∅ |
|---|---|---|---|---|
| 1 | 215,8 ms | 134,6 ms | 5,6 ms | 5,6 ms |
| 2 | 424,4 ms | 209,2 ms | 37,6 ms | 13 ms |
| 5 | 1093 ms | 350 ms | 176,4 ms | 27 ms |
| 10 | 2475 ms | 659,2 ms | 404,4 ms | 57 ms |

scheduling leads to waiting periods between the execution of nodes. However, when executing the flows sequentially, the runtime is growing approximately linearly as expected, e.g. 10 sequentially executed flows lead to a runtime of 57 ms instead of 404 ms when executed in parallel (cf. column "Sequential Runtime"). In conclusion, when using the Node-RED runtime environment, it would be necessary to implement a self-made runtime scheduler to avoid a poor runtime. As a consequence, we use the Node-RED runtime environment only for our proof-of-concept implementation. In the future, we will implement and compare further execution engines such as CEP-Esper that are suitable for highly parallel scenarios.

## 6 Summary and Outlook

In this paper, we presented an approach for a situation recognition service called SitRS. This service can be used to integrate real world objects (things) into the internet by deriving their situational state based on sensors. For that, we introduced a method for the recognition of situations based on modeling and executing situation templates. These templates represent a model to define situations by the sensor data to be used as well as the conditions for the situations. The SitRS service transforms this description into an executable situation template that can be automatically deployed and executed in a (cloud-based) execution engine. The architecture of our approach is separated into two components, the situation recognition component and the resource management platform. The situation recognition component is used to execute a data flow that reads sensor data, compares them with predefined values and uses this information to determine if a certain situation occurred. The sensor data is provided by REST. The sensor registry can be used to register new sensor data sources or deregister them if they aren't needed anymore.

As future work, we plan to integrate SitRS into a workflow system in order to realize situation-aware workflows. Furthermore, we plan to use the presented method in a different use case to enable situation recognition in advanced manufacturing (Industry 4.0) environments, i.e., we will introduce and implement an IoT scenario based on *"real"* objects such as production machines. On the technical side, we want to provide additional situation template mapping algorithms for other execution engines such as CEP-systems like Esper and data streaming systems like Odysseus. The current SitRS prototype provides the basis for further development and is available as open source implementation (https://github.com/hirmerpl/SitOPT). In addition, we plan to enable automatic sensor binding and registration based on ontologies.

**Acknowledgment:** This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Grant 610872, project SitOPT.

## References

1. Attard, J., Scerri, S., Rivera, I., Handschuh, S.: Ontology-based Situation Recognition for Context-aware Systems. In: Proceedings of the 9th International Conference on Semantic Systems (2013)
2. Brumitt, B., Meyers, B., Krumm, J., Kern, A., Shafer, S.: EasyLiving: Technologies for Intelligent Environments. In: Handheld and Ubiquitous Computing. Springer Berlin Heidelberg (2000)
3. Buchmann, A., Koldehofe, B.: Complex event processing. it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik (2009)
4. Dargie, W., Eldora, Mendez, J., Mobius, C., Rybina, K., Thost, V., Turhan, A.Y.: Situation Recognition for Service Management Systems Using OWL 2 Reasoners. In: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on (2013)
5. Dey, A.K.: Understanding and Using Context. Personal and Ubiquitous Computing (2001)
6. Großmann, M., Bauer, M., Hönle, N., Käppeler, U.P., Nicklas, D., Schwarz, T.: Efficiently Managing Context Information for Large-Scale Scenarios. In: Proc. of the Third IEEE Intl. Conf. on Pervasive Computing and Communications (2005)
7. Hussermann, K., Hubig, C., Levi, P., Leymann, F., Siemoneit, O., Wieland, M., Zweigle, O.: Understanding and Designing Situation-Aware Mobile and Ubiquitous Computing Systems. In: Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering 2010 (ICCESSE 2010 ) (2010)
8. Lange, R., Cipriani, N., Geiger, L., Großmann, M., Weinschrott, H., Brodt, A., Wieland, M., Rizou, S., Rothermel, K.: Making the World Wide Space Happen: New Challenges for the Nexus Context Platform. In: Proceedings of the 7th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom '09). Galveston, TX, USA. March 2009 (2009)
9. Meunier, R.: The pipes and filters architecture. In: Pattern languages of program design (1995)
10. Vermesan, O., Friess, P.: Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems. River Publishers (2013)
11. Wang, X., Zhang, D.Q., Gu, T., Pung, H.: Ontology based context modeling and reasoning using OWL. In: Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on (2004)
12. Wieland, M., Schwarz, H., Breitenbücher, U., Leymann, F.: Towards Situation-Aware Adaptive Workflows. In: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom) (2015)
13. Zweigle, O., Häussermann, K., Käppeler, U.P., Levi, P.: Supervised Learning Algorithm for Automatic Adaption of Situation Templates Using Uncertain Data. In: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human (2009)

# Detecting Frequently Recurring Structures in BPMN 2.0 Process Models

Marigianna Skouradaki and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart, Germany
{skouradaki,leymann}@iaas.uni-stuttgart.de

**Abstract** Reusability of process models is frequently discussed in the literature. Practices of reusability are expected to increase the performance of the designers, because they do not need to start everything from scratch, and the usage of best practices is reinforced. However, the detection of reusable parts and best practices in collections of BPMN 2.0 process models is currently only defined through the experience of experts in this field. In this work we extend an algorithm that detects the recurring structures in a collection of process models. The extended algorithm counts the number of times that a recurring structure appears in a collection of process models, and assigns the corresponding number to its semantics. Moreover, the dublicate entries are eliminated from the collection that contains the extracted recurring structures. In this way, we assert that the resulting collection contains only unique entries. We validate our methodology by applying it on a collection of BPMN 2.0 process models and analyze the results. As shown in the analysis the methodology does not only detect applied practices, but also leads to conclusions of our collection's special characteristics.

**Keywords:** BPMN 2.0, Relevant Process Fragments, RPF, process models, reusabulity, structural, similarity

## 1 Introduction

During the last years many researchers [6, 7, 12, 15, 16] have emphasized the importance of the reusability of process models. It is expected that an efficient methodology to reuse process models will contribute to a more effective engineering of process models [15]. For this reason we need to analyze the process models and decide which parts can be reused. The research field of process models similarities focuses on three different areas: 1) text semantics, 2) structural analysis and 3) behavioral analysis [5].

Large collections of process models are anonymized or modeled for documentation. Thus, they are not executable. Consequently, the approaches of text semantics (vs. anonymized models) and behavioral analysis on executable models (vs. mock-up, non-executable models) cannot be used efficiently. In these cases we need to apply the approach of structural similarities. However, structural similarities also base their functionality on text semantics and behavioral similarities [4,5,14]. For this reason we have suggested a methodology that runs (sub)graph isomorphism against a collection of process models and focuses on extracting the common recurring structures. The first approach of our methodology leads to promising results, as experiments showed that the algorithm can run with logarithmic complexity [17].

This work extends the work described in [17] with the following contributions:

1. extending the algorithm to count the recurring structures and filter the duplicate results
2. applying the algorithm on different use case scenarios
3. analyzing the exported results

This paper is structured as follows: section 2 describes the overall methodology overview of our work. Section 3 defines the problem and explains the basic concepts that frame it. Section 4 shows the implementation of the designed methodology. Section 5 discusses the results of the methodology's application. Section 6 addresses the related work in this area, and Section 7 concludes and describes our plans for future work.

## 2 Methodology Overview

The BenchFlow Project[1] aims to create the first benchmark for Business Process Model and Notation (BPMN 2.0) compliant Workflow Engines. In the scope of that project we have collected process models that reflect the diversity of application scenarios. For the construction of a representative benchmark it is needed to extract the essence of each of the scenarios and construct a set of representative process models. The extracted representative process models are called "synthetic" process models, because even though they are artificial they constitute an accurate representation of real world use cases. Synthetic process models should also be combined with appropriate Web Services, synthetic data and interacting users. To address these challenges, we develop a workload generator. In this work we focus on the methodology to synthesize representative process models while the generation of appropriate web services and interacting users is left for future work.

Figure 1 depicts the methodology for the generation of the synthetic processes, which uses the following four phases:

1. **Process Fragments Discovery**: Addresses the automatic discovery of recurring structures in a collection of process models. Our methodology is applied in a collection of BPMN 2.0 process models. The definition and implementation of this methodology are described in [17].
2. **Process Fragments Refinement**: The extracted recurring structures are stored as unstructured BPMN 2.0 code. They need to be refined as "Process Fragments" [15] in order to be stored in a process fragment library, out of which they can be managed, and retrieved.

[1] `http://www.iaas.uni-stuttgart.de/forschung/projects/benchflow.php`

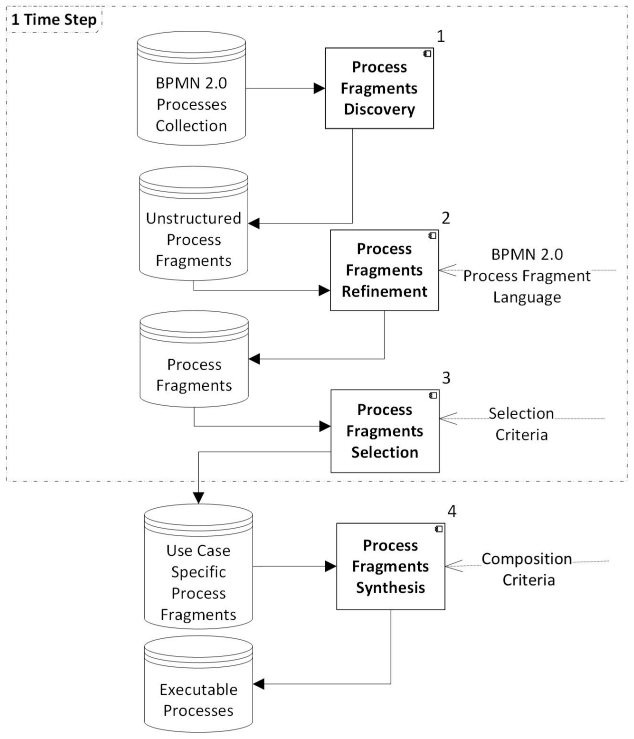

1 Time Step
BPMN 2.0 Processes Collection
1
Process Fragments Discovery
Unstructured Process Fragments
2
Process Fragments Refinement
BPMN 2.0 Process Fragment Language
Process Fragments
3
Process Fragments Selection
Selection Criteria
Use Case Specific Process Fragments
4
Process Fragments Synthesis
Composition Criteria
Executable Processes

**Figure 1.** Methodology Overview

3. **Process Fragments Selection**: All process fragments are not necessarily of equal interest for the benchmark. For example, in a benchmark scenario we are interested in high parallel fragments, while in another scenario we need to check how the Workflow Engine responds to complex control flow branching structures. This component selects fragments that satisfy benchmark related criteria. The criteria are defined by the user or from the benchmark customization. The work described in this paper focuses on this component, because it calculates the appearance rate of each recurring structure in the collection. Furthermore, it annotates the recurring structure with its appearance number. This is one of the metrics that will be used for the selection of the process fragments. Other metrics that we defined are size, structural metrics, metrics of external interaction, data handling and complexity [2,13]. The process fragments that we select from this component are stored in a separate repository.
   It is possible that phases 1-3 (Figure 1) can sometimes be omitted if the extraction criteria are compliant with the purposes of the benchmarking process.
4. **Process Fragments Synthesis**: Synthesizes the process fragments into executable processes according to the composition criteria given by the user

or the benchmark customization. For example, when the selection criteria ask for a process with control-flow nesting $\leq$ N and M external interactions, the appropriate fragments are chosen to synthesize it.

## 3 Background

### 3.1 Problem Definition

In graph theory the task of discovering similar structures is expressed as sub-graph discovery problem. Recurring sub-graph discovery is a sub-category of the general problem of subgraph isomorphism which is proven to be NP-Complete [1, 9]. However there are special cases of graphs and matching problems that are proven to be of lower complexity [20].

Figure 2 presents two process models expressed in BPMN 2.0. As seen, these models consist of nodes (in BPMN 2.0 tasks, events and gateways), directed edges (in BPMN 2.0 sequence flows), and labeling (BPMN 2.0 language semantics on events and gateways, and names on tasks). Hence, process models are a special type of directed attributed graphs. These are graphs where their vertexes or edges contain information.

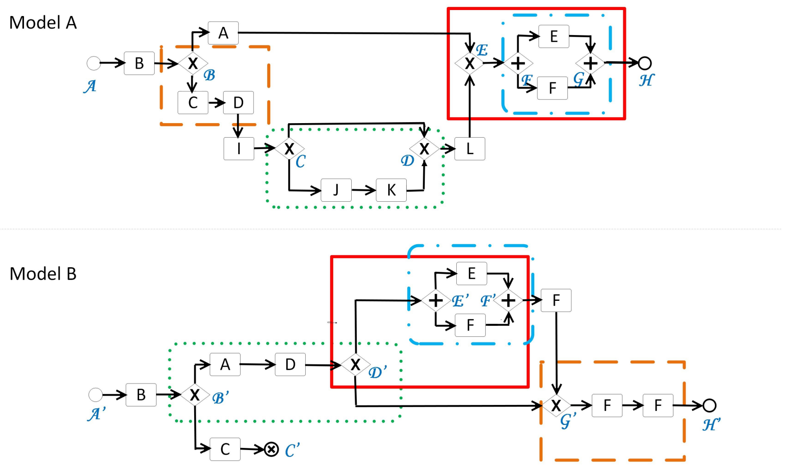


**Figure 2.** Recurring Structures in two BPMN 2.0 Process Models

Some results of the recurring structures extraction is shown in Figure 2. It shows four pairs of recurring structures in two BPMN 2.0 process models. These structures are not the complete set of reoccurring structures in these two process models. However we considered these as representative cases as they demonstrate the following attributes:

1. Structures can be nested within each other. This means that a recurring structure may be a subgraph of another, bigger recurring structure. Example structures of this attribute are marked with the red solid and the blue dash-dotted line.
2. Reoccurring structures can appear in different positions of the process model. As position we define the number of edges from the node to the model's start event This attribute applies to all represented structures.
3. Structures can be "partially" similar. This means that some of the outgoing edges of a node can lead to similar structures. Structures marked with the orange dash line and the green dotted line are some examples of this attribute.

It is also possible that a structure demonstrates more than one of the above attributes. For example the structure marked with the blue dash-dotted line is nested (Attribute 2) and appears in different positions of the diagram (Attribute 3).

We have used these attributes as a basis to develop the methodology to detect and extract recurring structures [17]. The goal of this work is to: a) detect the duplicate structures and b) to tag them with the appropriate number of appearances. This task is also reduced to a subgraph isomorphism challenge, since we have to reapply the methodology of graph isomorphism in order to decide if a structure is duplicate in the collection.

## 3.2 Basic Concepts

This section sets the theoretical work used by our methodology of fragmentation, duplicate filtering, and appearance counting. We extend the definition of "Process Fragment", initially given by [15] to fit our needs.

Before we proceed to the extension of the "Process Fragment" definition, we need to define the concept of a "Checkpoint". As "Checkpoint" we define any type of node that can be used as a starting point for our extended process fragments. The types of checkpoints can be configured by the user, and vary with respect to the process language that describes the models.

The extended definition of "Process Fragment" is called a "Relevant Process Fragment" (RPF) because its detection is dependent to its existence in at least $K$ process models. RPF satisfy the following structural requirements:

– starts with a checkpoint
– has at least $N$ nodes including the checkpoint, where $N$ is a natural number and pre-configured from the user. We use $N \geq 3$.
– contains at least one activity

For this paper we focus on the detection of RPFs when $K = 2$, while in future work the threshold of $K$ may vary.

Finally, we define as duplicate any structure that appears in $K + 1$ process models. Since in this work the threshold is set as $K = 2$, a duplicate must appear in at least 3 models before it is filtered.

## 4 Implementation

### 4.1 Algorithm: Get Recurring Structures

This section describes the methodology we developed for the discovery of the RPFs. For a better comprehension the algorithm is separated in two algorithms. Algorithm 1 starts the traversal of the model, calls Algorithm 2, and returns the discovered RPF. Algorithm 2 compares two models with each other and extracts the discovered similar structures.

Algorithm 1 takes as input one set of sequence flows (graph's edges) for each model. These sequence flows correspond to every outgoing sequence flows of each checkpoint. Figure 2 shows the scenario were the configured types of checkpoints are gateways and events. For example, see checkpoint $\mathcal{B}$ and checkpoint $\mathcal{E}'$ (cf. figure 2) . The corresponding sets given as input to the algorithm will be $Set1 = \{\{exclusive\,gateway \rightarrow Task\ A\ \},\ \{exclusive\,gateway \rightarrow Task\ C\ \}\}$ for Model A and $Set2 = \{\{parallel\,gateway \rightarrow Task\,E\ \},\ \{parallel\,gateway \rightarrow Task\,F\ \}\}$ for Model B. These sets of checkpoint sequence flows are called checkpointFlows in lines 3-4 of algorithm 2. This process is done to ensure that all possible sets of paths will be traversed by the end of the algorithm.

**Algorithm 1** Procedure that calculates the matching paths of two models

```
 1: procedure GETMATCHINGPATH()
 2:     $i \leftarrow 0$
 3:     for all $checkpointFlows\ chFlowA \in modelA$ do
 4:         for all $checkpointsFlows\ chFlowB \in modelB$ do
 5:             $tmpFragment \leftarrow \emptyset$
 6:             $tmpFragment \leftarrow$ COMPARISON($chFlowA, chFlowB, tmpFragment$)
 7:             if ISVALIDFRAGMENT($tmpFragment$) then
 8:                 ADD($collection, tmpFragment$)
 9:             end if
10:         end for
11:     end for
12: end procedure
```

For all possible pairs of checkpointFlows in the two checkpoint sets, namely for their Cartesian product, we call algorithm 2 (line 6 of algorithm 1). It takes as parameters the current instances of checkpointFlows, and an empty fragment. By iteratively calling algorithm 2 for the different checkpointFlows we manage to check all possible checkpointFlows combinations as a possible start point of an RPF. The comparison procedure returns the calculated RPF (cf. variable tmpFragment). At this point we need to validate if the returned structure comprises an RPF or not (cv. function ISVALIDFRAGMENT(tmpFragment)). The validation rules applied are the requirements described in Section 3.2. If the validation is successful, then the RPF is saved in a temporary data structure that holds all the discovered RPFs (cf. variable "collection" in algorithm 1)).

**Algorithm 2** Procedure that compares the two models

```
 1: procedure COMPARISON($sequenceFlowModelA$, $sequenceFlowModelB$,
    $tmpFragment$)
 2:     if (GETSOURCETYPE($sequenceFlowModelA$)
            equals
            GETSOURCETYPE($sequenceFlowModelB$))
            and
            (GETTARGETTYPE($sequenceFlowModelA$)
            equals
            GETTARGETTYPE($sequenceFlowModelB$)) then
 3:         $outgoingA \leftarrow$ GETOUTGOING(sequenceFlowModelA)
 4:         $outgoingB \leftarrow$ GETOUTGOING(sequenceFlowModelB)
 5:         ADD($fragment$, $sequenceFlowModelB$)
 6:         for all $outgoingA \in ModelA$ do
 7:             for all $outgoingB \in ModelB$ do
 8:                 if $outgoingB \notin tmpFragment$ then
 9:                     COMPARISON(outgoingA, outgoingB, tmpFragment)
10:                 end if
11:             end for
12:         end for
13:     else
14:         return $fragment$
15:     end if
16: end procedure
```

The functionality of algorithm 2 is to traverse and compare the models. The first step in this procedure is to check if the sequence flows that are given as parameters have sources and targets that are of the same type (e.g. task, exclusive gateway, start event etc.)(cf. functions GETSOURCETYPE() and GETTARGETTYPE()). When this condition is satisfied we say that two sequence flows match. Since we are strictly focusing on the structural similarity of the models this decision can be taken only based on the type of the sequence flow's source and target node. However, the condition could be extended if we wanted to check the similarity regarding more parameters e.g. labels of the nodes. When the sequence flows match we add the respective sequence flow to a temporary fragment structure (cf. variable tmpFragment in algorithm 2). Afterwards we get all the outgoing sequence flows (cf. function GETOUTGOING()) of the previously considered target nodes, i.e. make a step forward to the traversal, and call recursively the comparison algorithm for all pairs of outgoing sequence flows. The termination condition of the recursion is two sequence flows that do not match, or if the checked nodes do not have any outgoing sequence flows. If this condition is satisfied the algorithm returns the set of matched sequence flows until this point of execution (cf. variable tmpFragment in algorithm 2).

### 4.2 Algorithm: Find Duplicates and Count Appearance

This section presents the extensions that we developed for the algorithms of subsection 4.1. Their goal is to filter the duplicate RPFs, and count the times of their appearance. Algorithm 3 will check a set of newly discovered RPFs against the collection of stored RPFs, to find the number of appearances of an RPF in the collection. To this end, it calls algorithm 4 for their Cartesian product to check if they are isomorphic. Algorithm 4 checks if two RPFs are isomorphic to each other.

**Algorithm 3** Procedure that checks if a fragments is duplicate and increases appearance counter

```
    procedure HANDLEDUPLICATES($matches$)
        for all $match \in matches$ do
            if $collection$ equals $\emptyset$ then
 4:             ADD(collection,match)
            else if CONTAINS($collection, match$) then
                $tmpFragment \leftarrow collection[match]$
                $appearanceCounter \leftarrow tmpFragment.appearanceCounter$
 8:             $tmpFragment.appearanceCounter \leftarrow appearanceCounter + 1$
                $tmpFragment.isomorphic \leftarrow true$
            else
                ADD(collection,match)
12:         end if
        end for
    end procedure
```

More particularly, algorithm 3 takes as parameter a set of matched RPFs. This is basically the output of algorithm 1 described in subsection 4.1. As discussed the set of newly matched RPFs (cf. *matches*) is compared against the RPFs that are already stored in the *collection*. Firstly, we need to check if the *collection* is empty (cf. variable *collection*). In this case we add the first matched RPF the *collection* and we do not need apply further checks. For every other newly matched RPF (cf. variable *match*) we check if it is contained in the *collection*. The function $CONTAINS$ (cf. line 5 in algorithm 3) will internally call algorithm 4 to compare each matched RPF *match* with each RPF stored in the *collection*. If we find an isomorphic match we need to edit its corresponding isomorphic RPF that is stored in the *collection*. We mark it as isomorphic, and we increase its appearance counter by 1. If we do not find any isomorphic match then the RPF is added in the *collection* (cf. function ADD at line 11 of algorithm 3). If we apply this procedure before storing each RPF, the final *collection* will not contain any duplicates.

Algorithm 4 is called internally from the $CONTAINS$ function of algorithm 3. Its parameters are the RPFs to compare. It will return true if and only if the two RPFs are completely isomorphic. So the cases of isomorphic subsets

**Algorithm 4** Procedure that calculates if two RPFs are equal

```
1: procedure isFragmentIsomorphic($fragment1, fragment2$)
2:     $fragment1SequenceFlows \leftarrow sequenceFlows \in fragment1$
3:     $fragment2SequenceFlows \leftarrow sequenceFlows \in fragment2$
4:     if $fragment1SequenceFlows.size()$ equals
           $fragment2SequenceFlows.size()$ then
5:         $tmpFragment \leftarrow \emptyset$
6:         comparison($fragment1SequenceFlows[0]$,
            $fragment2SequenceFlows[0]$, $tmpFragment$)
7:         if $tmpFragment.size$ equals $fragment2SequenceFlows.size$ then
8:             $return\, true$
9:         end if
10:    end if
11:    $return\, false$
12: end procedure
```

are not considered in this case. We first check if the two RPFs have the same number of sequence flows. In this case we call the algorithm 2 which operates as it was described in subsection 4.1. Then we need to check if the output of algorithm 2 has the same size of sequence flows. If two RPFs are isomorphic and have the same number of sequence flows then it is sure that the RPFs are equal. If the two RPFs did not have the same number of sequence flows we know for sure they were not completely isomorphic. In this case false is returned.

## 5 Validation and Discussion

This section shows and analyzes the results of the methodology's application. For the validation we used 43 BPMN 2.0 process models that originate from the sets of BPMN 2.0 process models[2] also used in [14], and the BPMN 2.0 standard examples[3] [11]. This is a combination of artificial and real-world process models. The RPFs discovery algorithm (cf. algorithm 1) is configured to calculate the types of $Set1 = \{events, gateways\}$ as checkpoints, and we set $N = 3$ the number of nodes of each RPF.

Each model is compared with all the other models except for themselves. This leads to 903 comparisons. That results in 1544 non-filtered RPFs. The results are decreased to 259 RPFs after the filtering of duplicates. This leads to 83.22% decrease of the results. This is an important percentage of decrease because it will help us reduce the exported results and ease the analysis of the produced RPF collection.

The calculated times of appearance of the detected RPFs range from $[1, 178]$. From the RPFs that appear more than one times in the collection (54 RPFs) we calculate the median value and set it as threshold. Statistically the median

[2] `http://pi.informatik.uni-siegen.de/qudimo/bpmn/`
[3] `http://www.omg.org/spec/BPMN/20100602/`

value shows a central tendency of a set. Hence, it was considered representative as a threshold. In this case:

$$Median = Threshold = 14$$

We result in 27 RPFs with re-appearance rate above the *threshold*. Setting the *threshold* does not only eliminate significantly the results, but also gives us an insight of the most frequently used RPFs in the collection. Due to space limitations in figure 3 we present only the first 8 RPFs with the biggest appearance rate. Each RPF shown in the figure 3 has an ID number at its left side, and at its right side a number that indicates the times of appearance in the collection. As seen figure 3 most of the RPFs (1,2,3,4,5,7) comprise of simple structures.

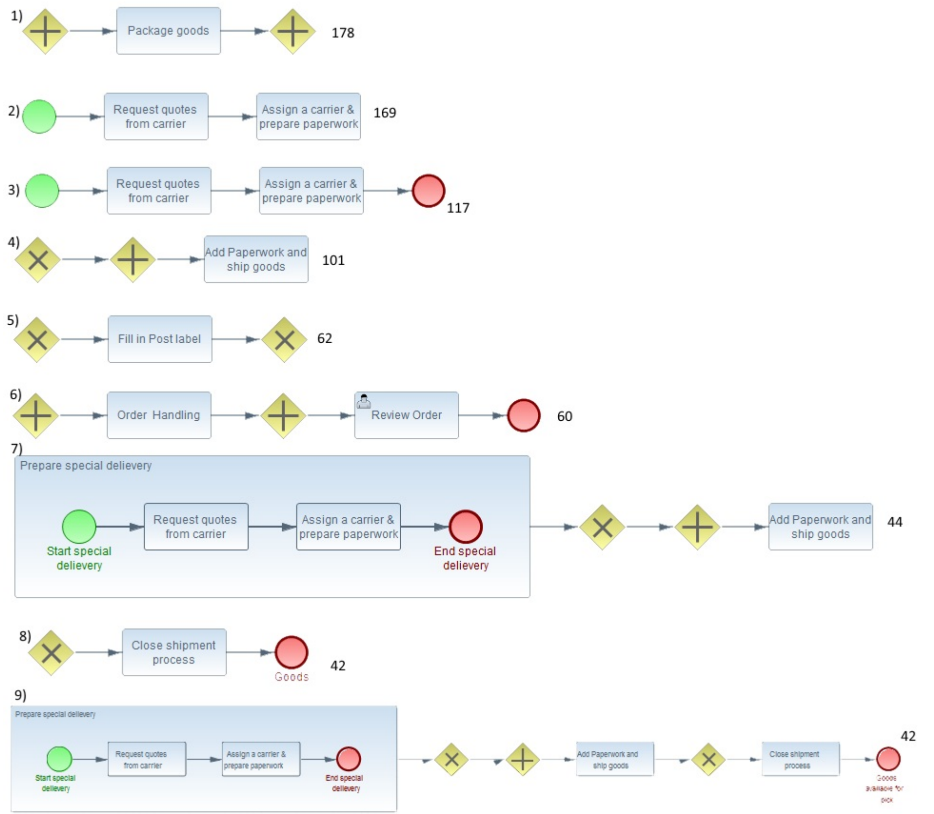

1)
Package goods
178
2)
Request quotes from carrier
Assign a carrier & prepare paperwork
169
3)
Request quotes from carrier
Assign a carrier & prepare paperwork
117
4)
Add Paperwork and ship goods
101
5)
Fill in Post label
62
6)
Order Handling
Review Order
60
7)
Prepare special delievery
Start special delievery
Request quotes from carrier
Assign a carrier & prepare paperwork
End special delievery
Add Paperwork and ship goods
44
8)
Close shipment process
Goods
42
9)
Prepare special delievery
Start special delievery
Request quotes from carrier
Assign a carrier & prepare paperwork
End special delievery
Add Paperwork and ship goods
Close shipment process
Goods available for pick
42

**Figure 3.** The first 8 RPFs with the bigger appearance rate

These structures are expected to be found in a collection of process models as they are simple combinations of parallel and/or exclusive gateways with tasks.

Hence, unless someone is interested to learn the appearance ratio of these trivial structures the number of nodes per RPF can be set to a number$>3$. Then these RPFs will not be included in the results.

By studying the resulting RPFs it is also possible to draw conclusions about usual practices in the design of process models. For example, RPF 6 reveals that parallel structures are frequently used at the end of process models. It is also interesting to notice that only one branch of the branching structures is found to be repeated each time. This indicates that the rest of the branches followed a different business logic.

General conclusions about our collection can also be drawn. For example, we observe that RPF 7 and RPF 9 are quite big and complex to have so frequent appearances. From this we can conclude many of the analyzed models were different versions of the same business process. As the real analysis will be applied on thousands of real-world process models gathered from different resources, this phenomenon is expected to be eliminated. However, we might have similar phenomena where the RPFs will reveal other details about our collection.

## 6 Related Work

Process Fragmentation is frequently discussed in the literature, with a focus to Single-Entry-Single-Exit (SESE) regions [18, 19]. These regions are then connected together, to form an hierarchy called "Refined Process Structure Tree" (RPST), which is then used to detect these sub-graphs of models that match together [8]. This approach was also examined in our work [17]. However, the division of the model to SESE regions, and then RPSTs would result in a sub-set of fragments, that do not necessarily represent the existing recurring structures in the process models. Therefore, the proposed technique was not considered compliant with our needs. In the field of structural similarities we could not find any approach that focuses on the detection of recurring structural parts in BPMN 2.0 models with a sole focus on their control-flow. A number of algorithms for process model structural matching and comparison are presented in [4, 14], but all of them have a strong focus on text semantics to detect similar mappings. As our work focuses on the structure of the process model and is independent of text or behavioral information, it was not possible to further use the aforementioned approaches. In [10] there is a focus on BPEL processes, which are tranformed to a BPEL process tree. Afterwards, tree mining algorithms are applied in order to discover recurring structures in a process. Although the further goal of this work is very similar to our goal, the different nature of BPMN 2.0 language does not allow to apply the same tree mining techniques for similar structures detection.

The Workflow Patterns Initiative[4] is an effort to provide a conceptual basis for process technology [3]. This initiative presents the aspects (control flow, data, resource, and exception handling) that need to be supported by any workflow language. As pattern they describe the minimum sequence of workflow language

[4] `http://www.workflowpatterns.com/`

elements that should be combined to represent a fundamental concept. The proposed micro-structures are not accompanied by information of real-life usage rate. Therefore, we consider this approach to complement but not replace our goals.

## 7 Conclusion and Future Work

In this work we described an extension of an algorithm defined in [17]. With this extension the algorithm automatically counts the appearances of recurring structures in a collection of process models. The ultimate motivation for this work is the development of a workload generator for benchmarking BPMN 2.0 Workflow Engines. However, the proposed methodology can be also applied to different use-case scenarios for process model re-usability.

To the best of our knowledge, this work is the first attempt to automatically detect frequently used structures in a collection of BPMN 2.0 process models. We have evaluated our approach with a collection of 43 BPMN 2.0 process models. As seen from the resulting RPFs conclusions can be made about a) frequently used structures (usual-practices) in BPMN 2.0 and b) for the collection's special characteristics.

As future work we plan to extend the algorithm for the complete set of BPMN 2.0 model elements. We will run the approach on the complete collection of real-world models, and execute a thorough analysis on the results. As a next step we will implement the first prototype of the process synthesizing methodology.

**Acknowledgments** This work is funded by the Swiss National Science Foundation and the German Research Foundation with the BenchFlow - A Benchmark for Workflow Management Systems (DACH Grant Nr. 200021E-145062/1) project.

## References

1. Basin, D.A.: A term equality problem equivalent to graph isomorphism. Information Processing Letters 51 (1994)
2. Cardoso, J.: Business process control-flow complexity: Metric, evaluation, and validation. International Journal of Web Services Research 5(2), 49–76 (2008)
3. van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (Jul 2003)
4. Dijkman, R., Dumas, M., van Dongen, B., Käärik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. Inf. Syst. 36(2), 498–516 (Apr 2011)
5. Dijkman, R., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. In: Proceedings of BPM '09. pp. 48–63. Springer-Verlag, Berlin, Heidelberg (2009)
6. Eberle, H., Leymann, F., Schleicher, D., Schumm, D., Unger, T.: Process Fragment Composition Operations. In: Proceedings of APSCC 2010. pp. 1–7. IEEE Xplore (December 2010)

7. Eberle, H., Unger, T., Leymann, F.: Process fragments. In: Meersman, R., Dillon, T., Herrero, P. (eds.) On the Move to Meaningful Internet Systems: OTM 2009, Lecture Notes in Computer Science, vol. 5870, pp. 398–405. Springer Berlin Heidelberg (2009)
8. García-Bañuelos, L.: Pattern identification and classification in the translation from bpmn to bpel. In: Meersman, R., Tari, Z. (eds.) OTM Conferences (1). Lecture Notes in Computer Science, vol. 5331, pp. 436–444. Springer (2008)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
10. Hertis, M., Juric, M.B.: An empirical analysis of business process execution language usage. IEEE Transactions on Software Engineering 40(8), 1–1 (2014)
11. Jordan, D., Evdemon, J.: Business process model and notation (BPMN) version 2.0. Object Management Group, Inc (January 2011)
12. Ma, Z.: Process fragments: enhancing reuse of process logic in BPEL process models. Ph.D. thesis, Universitt Stuttgart (2012)
13. Mendling, J.: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness. Springer (2008)
14. Pietsch, P., Wenzel, S.: Comparison of bpmn2 diagrams. In: Mendling, J., Weidlich, M. (eds.) Business Process Model and Notation, Lecture Notes in Business Information Processing, vol. 125, pp. 83–97. Springer Berlin Heidelberg (2012)
15. Schumm, D., Leymann, F., Ma, Z., Scheibler, T., Strauch, S.: Integrating Compliance into Business Processes: Process Fragments as Reusable Compliance Controls. In: Schumann/Kolbe/Breitner/Frerichs (ed.) MKWI'10, Göttingen, Germany, February 23-25, 2010. pp. 2125–2137
16. Schumm, D., et al.: Process Fragment Libraries for Easier and Faster Development of Process-based Applications. JSI 2(1), 39–55 (January 2011)
17. Skouradaki, M., Goerlach, K., Hahn, M., Leymann, F.: Application of Sub-Graph Isomorphism to Extract Reoccurring Structures from BPMN 2.0 Process Models. In: SOSE 2015; San Francisco Bay, USA, March 30 - 3, 2015. IEEE (April 2015)
18. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, B., Lin, K., Narasimhan, P. (eds.) Proceedings of ICSOC 2007. Lecture Notes in Computer Science, vol. 4749, pp. 43–55. Springer-Verlag, Berlin (2007)
19. Vanhatalo, J., Vlzer, H., Koehler, J.: The refined process structure tree. In: Dumas, M., Reichert, M., Shan, M.C. (eds.) Business Process Management, Lecture Notes in Computer Science, vol. 5240, pp. 100–115. Springer Berlin Heidelberg (2008)
20. Verma, R.M., Reyner, S.W.: An analysis of a good algorithm for the subtree problem, correlated. SIAM J. Comput. 18(5), 906–908 (Oct 1989)

# A Cooperative Game in Urban Mobility Systems

Marina Bitsaki[1], Vasilios Andrikopoulos[2], Alina Psycharaki[1]

[1]Transformation Services Laboratory, University of Crete, Heraklion, 70013, Greece
{bitsaki, psyharak}@tsl.gr
[2]Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, 70569, Germany
andrikopoulos@iaas.uni-stuttgart.de

**Abstract.** Collective Adaptive Systems (CASs) are very complex and evolve under uncertainty. Entities exchange information and coordinate their actions with each other in order to accomplish certain goals within CASs. In such an environment, entities have incomplete and uncertain knowledge about the actions of other entities. This affects their decisions that change dynamically and their interactions with each other. Game theory is the dominant approach used in decision making to model the strategic interactions of entities. In this paper we provide the description of a cooperative game that is used to model the behavior of passengers and journey planners in the process of journey selections in an Urban Mobility System (UMS). ). Our goal is to demonstrate the analytical tools and the insights into the workings of CASs that such a game theoretical approach would accrue.

**Keywords:** Cooperative game - cost allocation - urban mobility systems.

## 1 Introduction

A collective adaptive system (CAS) is a very complex system that evolves under uncertainty and is composed of a set of heterogeneous, autonomous and self-adaptive entities that exchange information and coordinate their actions with one another in order to improve the effectiveness with which they can accomplish their individual goals [1]. The incomplete and uncertain knowledge entities have for the actions of other entities, affects their decisions that change dynamically, and as result also their interactions with each other. These aspects may in principle be handled by decision theory which is used to analyze which options should be taken in the presence of uncertainty.

Game theory [2] is the dominant approach used in decision making to model the strategic interactions of entities. There are two kinds of games, called the *non-cooperative* and *cooperative games*. In non-cooperative games, individual players take actions and obtain a payoff (strategic analysis). In cooperative games, groups of players (*coalitions*) take actions and allocate the joint benefits derived from their cooperation. The main objective of a cooperative game is to determine a binding contract among all players that specifies how to divide the total generated value.

In this paper we provide the description of a cooperative game that is used to model the behavior of passengers and journey planners in the process of journey selections in an urban mobility system as well as the cost structure of a route and investigates cost allocation rules among passengers that share a common route.

Cooperative game theory [3] has a wide range of applications to cost allocation problems in which people with conflicting interests decide to work together to save costs [4], [5], [6]. In [4], it is shown that a cost allocation problem is identical to the determination of the value of a cooperative game with transferable utilities and various allocation rules (the Shapley value, the Nucleolus and others) are described. In [5], [6] reviews on the applications of transferable utility cooperative games to cost allocation problems are performed. Especially, the authors in [5] emphasize on game practical aspects rather than theoretical ones and concentrate in three specific areas: transportation, natural resources and power industry.

A few studies that apply concepts from game theory in transport analysis have been reported in the literature. Most of them use non-cooperative games to model interactions between travelers, between authorities, or between travelers and authorities with no concerns for gains through sharing [7], [8], [9]. Mode, route, or departure time choice are some of the travelers' strategies whereas time or gas consumption minimization, traffic control settings or capacity policies are some of the strategies for transportation authorities. A systematic review performed in [10] presents various transportation models and the respective games used to analyze them.

A few studies analyze the effect of collaboration in transportation networks highlighting the cost savings that result from the efficient utilization of transportation resources [11]. Samet et al [11] present an economic transportation model that uses the Aumann-Shapley prices to allocate costs among destinations in a way that each destination will pay its real part in the total transportation costs.

In this paper, we analyze a model of collaboration in a multi-modal transportation system seen as a collective adaptive system in which collaboration is performed at all levels of transportation decision-making; passengers collaborate to co-create routes dynamically according to the collective benefit they get and share costs according to the individual contribution in the total cost.

## 2 Scenario

We consider the multi-modal Urban mobility System (UMS) described in detail in [12] that supports citizens mobility providing customized transport facilities that integrate a regular bus system, a FlexiBus system and a car pooling system. The FlexiBus Management System (FBMS) described in [1] as part of the UMS scenario supports the management and operation of FlexiBuses and provides citizens' mobility services within a predefined network of pickup points. A FlexiBus combines the features of taxi and regular bus providing transportation between any two pickup points in a more convenient manner than regular buses but being less expensive than taxis. Within FBMS we can distinguish the following set of entities: Passenger, FlexiBus Driver, Route Manager, Route Planner and FlexiBus Manager. They cooperate with each other to achieve both individual and collective goals. The system must be able to manage different routes at the same time set by passengers by allowing pre-booking of pick up points. More specifically, each Passenger can request a trip to one of the predefined destinations in the system, asking to start at a certain time and from a preferred pickup point. Each Bus Driver is assigned by the FBMS a precise route to execute, including the list of passengers assigned to it, and a unique final destination. During the route realization, each FlexiBus can also accept passengers that have not already booked only if there are available seats. Bus drivers communicate with an assigned Route Manager to ask for the next pick-up point and to communicate information like passengers check-in. Different routes are created by a Route Planner that organizes them to satisfy all passenger requirements (i.e. arrival time and destination) and to optimize bus costs (i.e. shorter distance, less energy consumption, etc.). To find the set of possible routes, the Route Planner communicates with the FlexiBus Manager in order to collect necessary information (i.e. traffic, closed roads, events, etc.) and available resources (i.e. available buses), and to generate alternative routes.

In this paper, we model the interactions of the entities that are involved when a passenger request is received by UMS. The lifecycle of a request involves two phases:

- The mode of transformation phase (phase 1) in which the UMS sends a set of ordered alternatives to the passenger and the passenger selects the appropriate one according to his own preferences and utility.
- The route confirmation phase (phase 2) in which the passenger makes a request to the transportation company selected in phase 1 and the route manager of the company accepts or rejects the passenger request and sets the rules and conditions of the transfer.

Each phase is characterized by a set of entities that interact with each other and a set of actions the entities take in order to accomplish the individual goals of the request. In phase 1, the entities involved are the passenger and UMS. UMS coordinates the transportation by various modes of transportation such as regular bus, FlexiBus, taxi and car-pooling. A passenger sends a request to UMS for a specific trip specifying the origin, the destination, the departure and arrival time, and his preferences. UMS replies by sending all appropriate alternative routes specifying the estimated travel time, the estimated cost and the mode for each route. The passenger chooses the optimal route such that his expected utility function is maximized (decision point 1).

In phase 2, a specific route has a predefined origin and destination, not necessarily the same as that of the passengers. The intermediate pick-up points of the route are specified dynamically according to travel demand and may change during the trip. At any time before or during the execution of the route, the route manager calculates the estimated travel time between any two pick-up points taking into account the current passengers of the route, the itinerary and current congestion. The route manager also calculates the total current cost of the route which is shared among all passengers of the route. Each estimation of the individual cost is calculated according to the specific itinerary between the origin and destination of the respective passenger. Each individual cost is finalized and paid at the end of the individual trip for each passenger. As more passengers come to the route, the individual costs tend to decrease but the estimated travel times tend to increase.

The passenger sends a request R for a specific route to the manager of the transportation company selected in phase 1. The route manager updates the route (to include the current request) and calculates the new estimated travel time for all remaining pairs of origin and destination and the total cost of the route. According to this information and his utility function, the passenger either accepts or rejects the request (decision point 2). In case of acceptance, the route manager sends the estimated travel time and estimated individual cost and confirms the transportation. In the above procedure the route manager has to take into account the initial desired travel times defined by the passengers (in order to have small deviations) and the initial individual costs (in order to limit the final payments to the promised ones).

In this paper, we take phase 1 as given, that is, the passenger has already made his choice about the mode of transportation. In the next section we model the interactions of phase 2 as a cooperative game.

## 3 A Cooperative Game

We model the interactions among entities involved in phase 2 of the scenario introduced in Section 2 as a cooperative game aiming to highlight the collective benefits they get by sharing a common mode of transportation. We assume that the outcome of phase 1 after a request arrives at UMS (which is the route the passenger has chosen for his transit) is known and does not affect our formulation.

We define a route between an origin and a destination as a number of pick-up points assigned to a number of passengers that share a common bus in order to decrease transportation costs and benefit from cooperating. We define the cooperative game consisting of a number of players being the passengers of the route (regardless of when they entered the route) who make coalitions and create a cost (the cost of the route) that is to be allocated to them according to their utilization of the route.

We introduce a simple static case with the following assumptions:

1. The route consists of a predefined and fixed number of pick-up points.
2. The set of passengers that will use the specific route and the respective pick-up points are known prior to the beginning of the route.
3. All passengers have the same destination.
4. The number of passengers is less than bus capacity, so that there is no need for occupying another bus.

Formally, the game $G = (N, c)$ is described by the set $N = \{1, \dots, n\}$ of passengers that share the route and have the same destination (but not the same origin) and the cost function $c\colon 2^N \to \mathbb{R}$ of a route, where $c(S)$ is the joint cost of the route used by the set $S \subseteq N$ of users ($c(\emptyset) = 0$). The objective is to determine the coalition that will eventually be formed and the allocation rule for the total cost incurred for this coalition.

A required property for $c$ is sub-additivity: for every two disjoint sets of users the cost of the route if they merge is smaller than or equal to the costs of the route they would used separately. That is

$$c(S_1 \cup S_2) \leq c(S_1) + c(S_2), for\ all\ S_1, S_2 \subset N \tag{1}$$

Costs are shared in such a way such that individual users each have an incentive to cooperate. In the FBMS scenario, this corresponds to the facts that a) the cost of running one bus along a route is in principle cheaper than having two buses doing different routes between the same origin and destination points, and b) the cost of having two passengers in the bus is less than the sum of the costs of having each one of the passengers alone in the bus. In a sub-additive game: $c(N) \leq \sum_{i\epsilon N} c(\{i\})$. If this condition holds: with strict inequality then each player gains from the cooperation. If sub-additivity property does not hold, coalitions other than the grand coalition might realize.

The allocation of $c(N)$ among the users in $N$ is specified by the allocation rule $x: x(c) = (x_1, \ldots, x_n)\ s.t. \sum_{i\epsilon N} x_i = c(N)$ (such as Shapley value for fairness or core for stability). The desired properties for allocation $x$ are the following:

1. $\sum_{i\epsilon N} x_i = c(N)$: feasibility of the grand coalition (costs are reimbursed).
2. $x_i \leq c(\{i\})\ \forall i\epsilon N$: no player pays a higher price in the grand coalition than he would do independently.

An extended version of the above game is the dynamic case in which passengers may not have the same destination and may not be known from the beginning of the route. In this case, a new passenger may enter the route during its execution or a cancellation may be reported. Thus, the number of passengers sharing the route changes over time and the recalculation of the total cost and individual costs is needed (repetition of the game) each time a new request arrives at the FlexiBus system or a cancellation is reported.

In case of a new arrival, the winning coalition might not include the new passenger. The route manager is able to reject the new request if the respective passenger incurs higher costs than reported for current passengers due to a deviation from the scheduled itinerary (in this case the grand coalition is not the outcome of the game).

In case of a cancellation, the individual costs might increase since the cost of the route is currently shared among less passengers. Thus, a charging policy that takes into account the above situation has to be considered (e.g. introduce penalty fees or consider no reimbursement fees).

## 4 Conclusions

On the basis of demonstrating the collectiveness of transport systems, this paper proposes a cooperative game among passengers of a route that pay reduced individual costs by sharing transportation resources. The objective is to determine the coalition that will eventually be formed and participate in the route and the allocation rule for the total cost incurred for this coalition.

## References

1. V Andrikopoulos, A Bucchiarone, Gomez S Saez, D Karastoyanova, and C Mezzina, "Towards Modeling and Execution of Collective Adaptive Systems," in *Ninth International Workshop on Engineering Service-Oriented Applications*, Berlin, 2013.
2. Kevin Eric Leyton-Brown and Yoav Shoham, Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations, Cambridge University Press, 2009.
3. C Montet and D Serra, *Gaem Theory and Economics*, Palgrave Macmillan, Ed. New York, 2003.
4. Jean Lemaire, "An application of game theory: cost allocation," *ASTIN Bulletin*, vol. 14 (1), pp. 61-81, 1984.
5. Gloria M Fiestras-Janeiro, Ignacio Garcia-Jurado, and Manuel A Mosquera, "Cooperative Games and Cost Allocation Problems," *Top*, vol. 19, pp. 1-22, 2011.
6. S H Tijs and T S H Driessen, "Game theory and cost allocation problems," *Management Science*, vol. 32 (8), pp. 1015-1028, 1986.
7. C S Fisk, "Game theory and transportation systems modelling ," *Transport Res. B*, vol. 18B(4/5), pp. 301-313, 1984.
8. D Levinson, "Micro-foundations of congestion and pricing: a game theory perspective ," *Transport. Res. A* , vol. 39, pp. 691-704, 2005.
9. A Roumboutsos and S Kapros, "A game theory approach to urban public transport integration policy," *Transport Policy* , vol. 15, pp. 209-215, 2008.
10. Y Hollander and J N Prashker, "The applicability of non-cooperative game theory in transport analysis," *transportation* , vol. 33, pp. 481-496, 2006.
11. D Samet, Y Tauman, and I Zang, "An application of the Aumann-Shapley prices for cost allocation in transportation problems," *Mathematics of Operations Research* , vol. 9(1), pp. 25-42, 1984.
12. ALLOW Ensembles, "Deliverable 8.1 - Specification of Prototype and Architecture," 2013.

# Efficient Attribute Based Access Control

Marc Hüffmeyer

Furtwangen University of Applied Sciences, Germany

**Abstract.** Today's information systems often handle large amounts of data and users and perform complex operations. The potentials of information systems grow more and more and so does the need for flexible and efficient access control mechanisms. Traditional access control mechanisms were built to support basic security concepts. For example Access Control Lists (ACL) have been designed to specify who may access a single resource (e.g. a network interface or a file in an operating system) while Role Based Access Control (RBAC) groups multiple subjects together under a role property, reducing the amount of rules required to describe *who* may access a resource. Having applications that support complex processes requires more fine-grained mechanisms that can handle questions like *who*, *what*, *how*, *why*, *when* or *where* and that are capable to adapt to frequent changes. In times of social media, smart objects and the Internet of things users and systems often create and share new content within applications. A substantial need to control access to this content is the consequence. In an environment where large amounts of subjects provide large amounts of data and specify multiple access rights, a flexible, high-performance access control mechanism is required. Because traditional access control mechanisms have been designed for a different purpose, efficient access control mechanisms and models must be found, that offer flexibility and guarantee high performance even in complex environments.

Attribute Based Access Control (ABAC) seems to be a suitable candidate that offers the flexibility to create fine-grained access control policies [5]. The core idea of ABAC is that any property of an entity can be used to determine access decisions. This idea offers the opportunity to cover existing access control mechanism as well as extending them to build more flexible access control solutions. For example Role Based Access Control can be easily simulated using an attribute named *role* which is assigned to users and checked during the evaluation of an access request.

Currently there is only one main standard that addresses ABAC: the eXtensible Access Control Markup Language (XACML) [2]. XACML defines three parts. The first part describes an architecture that shows how access control can be built as a dedicated component of an information system instead of building it into the core of such a system. The second part describes a policy language which enables to build hierarchical access control policies based on security relevant properties of any entity in the system. Finally, the third part of XACML is a request/response language that can be used to formulate access control requests, send them to the access control component and receive the access decision.

XACML is a very powerful mechanism that allows creating fine-grained access control policies. On the other hand XACML has some drawbacks. One of these drawbacks is that, due to the design of XACML, access decision must be computed at runtime. That means that with a growing policy complexity also the computation complexity for an access decision grows and either more computation capabilities are required or the computation takes longer.

Smart algorithms, data structures and techniques are required to address this problem, so that access decisions can be found in a short time even in complex environments with large amounts of security relevant data.

The first step on a way to Efficient Attribute Based Access Control was to analyze the requirements of a dedicated environment and to find suitable techniques and guidelines so that fast access decisions can be found. Representational State Transfer (REST) is an architectural style for distributed systems and services [1]. The guidelines given for RESTful services can also be used to build access control policies in a resource oriented environment in a way that access requests can be evaluated very fast. We created guidelines how to write efficient XACML policies for RESTful services [4] and derived an Attribute Based Access Control Model for RESTful services [3].

The next step is to find data structures that enable fast access decisions in a more general environment. Therefore, the use of databases instead of in-memory data structures is targeted. Thereby the challenge is to find a suitable model that offers enough flexibility to handle access control policies of any kind on the one hand and on the other hand allows utilizing fast search algorithms and techniques.

In a second approach the precomputation of access decisions is examined. Precomputation of access decisions may allow performing only a lookup to find an access decision instead computing the decision at runtime. On the other hand an approach like that may require large computation resources to create the lookup table and to handle changes to the security policy.

## References

1. T. R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. *University of California, Irvine*, 2000.
2. Organization for the Advancement of Structured Information Standard. eXtensible Access Control Markup Language (XACML) Version 3.0. *OASIS Standard*, 2013.
3. M. Hüffmeyer and U. Schreier. An Attribute Based Access Control Model for RESTful Services. *SummerSOC '15*, 2015.
4. M. Hüffmeyer and U. Schreier. Efficient Attribute Based Access Control for RESTful Services. *ZEUS '15*, 2015.
5. D. Sandhu. The authorization leap from rights to attributes: maturation or chaos? *SACMAT '12*, 2012.

# Low Latency Cloud Data Management through Consistent Caching and Polyglot Persistence

Felix Gessert

Database and Information Systems Group
University of Hamburg
gessert@informatik.uni-hamburg.de

The ongoing increase of complexity, mobility and scale in modern applications triggered a paradigm shift towards distributed, cloud-based data management. The expanding field of NoSQL and cloud data stores encompasses a rich variety of systems that deal with non-functional requirements of these applications such as latency, throughput, availability and elastic scalability. However, two central problems remain unsolved. First, the performance of mobile and web applications is governed almost exclusively by latency. Since the recent shift to smarter clients and single-page applications, dynamic database content is mostly requested in end devices directly. This makes data requests extremely latency critical, as they block the user experience . Thus, performance cannot be solved at the database level alone but end-to-end latency has to be addressed, too. Second, the heterogeneity and complexity of different data stores make it tremendously difficult for application developers to choose an appropriate system and reason about its performance and functionality implications. The situation is frequently complicated when no one-size-fits-all solution satisfies all requirements. Until now, the overhead and required know-how to manage multiple database systems prevents many applications from employing polyglot persistence.

We introduce an integrated solution to both the latency and diversity problem as Orestes [1], a database-as-a-service middleware capable of exposing different data stores through a uniform REST interface and database-independent data model [2]. To solve the latency problem we propose the *Cache Sketch* [3]. It is the first approach to exploit the web's expiration-based caching model and its globally distributed content-delivery infrastructure which were previously considered irreconcilable with dynamic workloads. Cache Sketches guarantee rich tunable consistency ($\Delta$-atomicity [6, 7]) using Bloom filters to create compact representations of potentially stale records to shift the task of cache coherence to clients. Furthermore, the number of invalidations on caches that support them (e.g., CDNs) is minimized. With different age-control policies the Cache Sketch achieves very high cache hit ratios with arbitrarily low stale read probabilities. The *Constrained Adaptive TTL Estimator* complements the Cache Sketch by a statistical framework for inferring cache expiration dates (TTLs) that optimize the trade-off between Cache Sketch size, cache hit ratio and the number of invalidations. The *YCSB Monte-Carlo Caching Simulator* offers a generic framework for simulating the performance and consistency characteristics of any caching and replication topology. Simulations as-well-as real-world benchmarking provide empirical evidence for the efficiency of the Cache Sketch and the considerable latency reductions it achieves. To provide even stronger safety guarantees we propose *Scalable Cache-Aware Transactions* [4] that attain optimistic ACID transactions over a wide range of unmodified data stores, relying on Cache Sketches to minimize the abort rates of optimistic concurrency control.

Instead of prescribing the use of one particular data store, we propose the *Polyglot Persistence Mediator* (PPM) [5] that automates polyglot persistence based on service level agreements (SLAs) defined over functional and non-functional requirements. In a three-step process, tenants first annotate schemas with SLAs (e.g., $latency_{read} < 30ms$). In the second step, the schema annotations are recursively scored against available data stores, yielding a routing model comprised of a mapping from schema elements to data stores. In the third step, the PPM in Orestes employs the routing model to delegate requests to appropriate data stores, manages replication and collects metrics for scorings. Preliminary experimental results show drastic performance improvements for scenarios with high write throughput and complex queries.

In our ongoing work we are extending the Cache Sketch approach to query result caching. In particular we are designing a scalable stream processing system to detect required query result invalidations and Cache Sketch additions. The TTL estimator is extended to an integrated reinforcement learning process that predicts TTLs of single records as well as query results. We are also broadening the scope of the cache-aware transaction scheme by providing transparent selection of either general-purpose optimistic concurrency control or specialized transaction protocols for certain workloads like write-only transactions. For the PPM we are currently introducing active request scheduling and multi-tenant workload management, as well as improved scoring through machine learning techniques. The research results around Orestes also form the technological basis of a backend-as-a-service startup called *Baqend*.

## References

1. F. Gessert and F. Bücklers, ORESTES: ein System für horizontal skalierbaren Zugriff auf Cloud-Datenbanken, in Informatiktage, 2013.
2. F. Gessert, S. Friedrich, W. Wingerath, M. Schaarschmidt, and N. Ritter, Towards a Scalable and Unified REST API for Cloud Data Stores, Workshop Data Management in the Cloud, 2014, Bd. 232, S. 723734.
3. F. Gessert, M. Schaarschmidt, W. Wingerath, S. Friedrich, and N. Ritter, The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management, in Datenbanksysteme für Business, Technologie and Web (BTW), 2015.
4. F. Gessert, F. Bücklers, and N. Ritter, Orestes: A scalable Database-as-a-Service architecture for low latency, in CloudDB Workshop, ICDE, 2014, S. 215222.
5. M. Schaarschmidt, F. Gessert, and N. Ritter, Towards Automated Polyglot Persistence, in Datenbanksysteme für Business, Technologie and Web (BTW), 2015.
6. S. Friedrich, W. Wingerath, F. Gessert, and N. Ritter, NoSQL OLTP Benchmarking: A Survey, in Workshop Data Management in the Cloud, 2014, Bd. 232, S. 693704.
7. W. Wingerath, S. Friedrich, and F. Gessert, Who Watches the Watchmen? On the Lack of Validation in NoSQL Benchmarking, in Datenbanksysteme für Business, Technologie and Web (BTW), 2015.

# An Architecture for an Internet of Things Platform for Secure Storage and Analysis of Sensor Data

Frank Steimle

University of Stuttgart,
Institute of Parallel and Distributed Systems,
Universitätsstr. 38, 70569 Stuttgart, Germany
`frank.steimle@ipvs.uni-stuttgart.de`
`http://ipvs.uni-stuttgart.de`

## 1 Motivation

Everyday-objects which contain sensors and posses the ability to communicate build a network called the Internet of Things. Since sensors get smaller and cheaper, more and more such intelligent objects exist. Therefore, the Internet of Things (IoT) gains more and more attention. Also the variety of sensors increases. Because of this it is possible to develop IoT-Applications for various domains, e.g., health care domain or smart environment domain. Each domain has its own set of requirements to the storage and analysis of the occurring sensor data and they have to be applied to the IoT-Application. Therefore there are many different implementations of IoT-Platforms. The downside of this is that it is very difficult to access the knowledge of two different domains or to combine it.

## 2 Components of the Architecture

This work aims at creating an architecture for an IoT-Platform for secure storage and analysis of sensor data. In order to achieve this, there are several challenges to cope with.

Sensor data has to be stored using a **secure storage** system. The storage system has also to make sure that the data can only be accessed by authorized users. The platform also has to deal with data from different sensors from different vendors. Therefore the platform also needs to cope with different formats for the same data. To connect many different sensors with the platform a **sensor API** is needed. This API needs to provide an interface which supports a broad variety of sensors. Furthermore, it needs to support security mechanisms to store and query sensor data. Maybe there is also need for a registry component, where sensors can be registered. This registry could be used to recognize sensors who are allowed to send data to the platform and to store metadata, like information about the data format.

Since applications domains have different requirements to the **analysis** of the data, the platform needs to support many different analysis types, like data flow analysis, data mining, event processing, and text analysis. The analysis component has also to deal with security and privacy of the data. To customize the platform there has to be an analysis API which can be used to configure the analyses that will be executed on the data.

Finally, the whole system has to be secure and has to guarantee the security of the data.

## 3 Examples

This architecture could be applied, e.g, to the ECHO (*Enhancing Chronic patients Health Online*) project[1], which is a german-greek research project. This project aims at monitoring patients who suffer from *chronic obstructive pulmonary disease* (COPD). The patients enter data about their health on a daily basis. Every time a patient enters new data, the system analyses the data to check whether the condition of the patient worsens. Since health data can be misused in many ways, the ECHO system also needs a secure storage of sensor data. Furthermore, the analysis component could be used to improve the monitoring of the patients by enabling physicians to create individual analysis for their patients.

The architecture could also be applied to the smart environment domain, e.g, to build a monitoring and analysis system for plants. In this system the user could assign a sensor to a predefined set of information which describes a plant species. If the plant doesn't get enough water or not enough light, the system would alert the user who can water the plant or put it in a brighter place. Since the data recorded by this platform could be used to find out when the users are not at home, a secure storage is important, too. In order to deal with many different plant species, an adaptable analysis component is also needed.

Finally these two systems could be managed by one platform. This would enable physicians to monitor the air humidity of their chronic lung patients using the sensors which are part of the plant monitoring system.

[1] http://chroniconline.eu/

# Flexible Modeling and Execution of Data Integration Flows

Pascal Hirmer
Institute of Parallel and Distributed Systems, Universität Stuttgart

**Summary of the Research Problem**

For my PhD, I am working towards an approach for data flow-based, ad-hoc data integration based on cloud computing technologies. I started my work roughly one year ago. In the following, I will present the research problem and introduce a plan how to cope with the introduced issues: Today, the highly advanced connectivity of systems as well as their increasing collaboration lead to new challenges regarding data exchange and data integration. Especially the emerging topics Internet of Things (IoT), advanced manufacturing or case management are in need of a distributed, flexible processing and integration of heterogeneous data. Furthermore, the possibility to integrate data sources ad-hoc is pursued to enable high flexibility. In this context, ad-hoc means: (i) enabling an iterative and explorative trial-and-error-like data integration using different data sources without the need of creating, e.g., complex Extract-Transform-Load (ETL) processes, (ii) the flexible adding and removing of data sources with low effort (preferably automatically), and (iii) the automatic adaptation of the data integration considering the newly added or removed data sources. However, this flexibility also leads to a high complexity compared with statically defined data integration techniques such as ETL processes. Those flexible scenarios are oftentimes realized using data flow and streaming technologies based on different execution models. The use of a data flow model enables its flexible generation based on dynamic needs. That is, if a new data source is added to an existing integration, the flow model can be re-generated and re-executed in order to include it. Currently, existing solutions are tailor-made for a specific use case scenario and do not offer a generic solution. Another issue in this research area is coping with unstructured data. Most systems do not support the simultaneous processing and integration of structured and unstructured data. However, integrating unstructured data can lead to valuable information. For example, in advanced manufacturing environments, the natural language input of a worker or textually described manuals can lead to higher-level information which can e.g., be used for an automatic dissolving of occurring problems. In conclusion, the main challenges of this research area are: (i) automatic tethering of different data sources, both structured and unstructured, (ii) integrating heterogeneous data formats in an ad-hoc manner, (iii) reducing the technical expertise of the involved users, and (iv) achieving a high efficiency of the data processing.

**Proposed Plan**

I plan to address these issues in multiple steps, which are shown in Figure 1. In the first step, I concentrate on the definition and tethering of data sources to be processed by my approach. I additionally want to provide a means to automatically add data sources without human interaction. To do so, I plan to

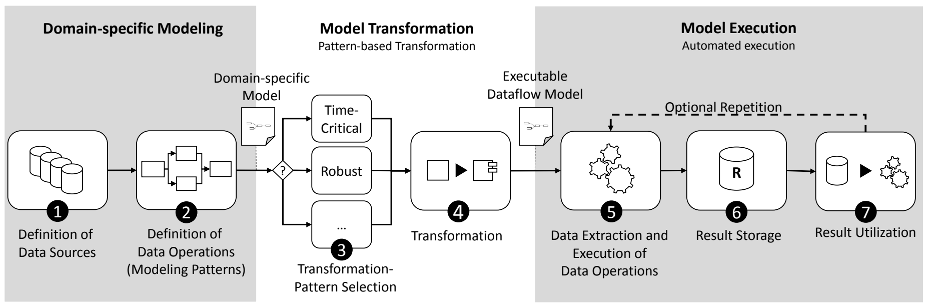

Domain-specific Modeling
Model Transformation
Pattern-based Transformation
Model Execution
Automated execution
Domain-specific Model
Executable Dataflow Model
Optional Repetition
Time-Critical
Robust
...
R
1
2
3
4
5
6
7
Definition of Data Sources
Definition of Data Operations (Modeling Patterns)
Transformation-Pattern Selection
Transformation
Data Extraction and Execution of Data Operations
Result Storage
Result Utilization

**Fig. 1.** Pattern-based Integration Approach [2]

use an ontology-based approach. In step 2, data operations are defined, i.e., a data flow is modeled that contains operations that alter or integrate the data. To ease the modeling for non-expert users, I will introduce modeling patterns. In the next step 3, transformation patterns are selected to ensure an implementation suitable for the use case scenario. For example, the *Time-Critical Pattern* ensures the best possible runtime, whereas the *Robust Pattern* ensures a robust execution. Next, step 4 transforms the domain-specific model into an executable model based on the selected transformation pattern. In step 5, this executable model is executed in a suitable engine. The result can then be stored and used for visualization, analysis or other value-adding scenarios. To enable important aspects such as scalability, availability and high performance, I create a cloud-ready approach. That is, I enable an easy provisioning in a cloud environment due to the use of web-based technologies and stateless services, exclusively.

**Progress to Date**

Until today, I worked on an overview paper of the introduced concepts that will be presented at DATA2015 [2]. Furthermore, I contributed to the Internet-of-Things (IoT) project SitOPT by implementing a prototype for a specific IoT scenario that is based on my approach. The paper containing these results will be presented at SummerSOC2015 [1]. In this paper, I introduced an approach for an automated sensor integration to derive high-level situations in a smart environment. By doing so, I achieved to integrate my concepts in a specific use case scenario as proof-of-concept. In the future, I plan to concentrate on the details of the single steps presented. In the next step, I am working on an approach for the ad-hoc, automatically processed adding and removing of data sources based on ontologies.

## References

1. Hirmer, P., et al.: SitRS - A Situation Recognition Service based on Modeling and Executing Situation Templates. In: Proceedings of the 9th Symposium and Summer School On Service-Oriented Computing (SUMMERSOC15) (2015)
2. Hirmer, P., et al.: Extended Techniques for Flexible Modeling and Execution of Data Mashups. In: Proceedings of 4th International Conference on Data Management Technologies and Applications (DATA) (2015)

# Secure Data Erasure in Untrusted Storage Systems

Tim Waizenegger

University of Stuttgart,
Institute of Parallel and Distributed Systems,
Universitätsstr. 38, 70569 Stuttgart, Germany
tim.waizenegger@ipvs.uni-stuttgart.de
http://www.ipvs.uni-stuttgart.de

**Abstract.** The secure deletion of data in untrusted storage-systems is a widely overlooked aspect of data management, although both individuals and corporations regularly face this issue. The two current solutions to secure deletion are to physically destroy storage media or overwrite data. Given the widespread use of outsourced and cloud-based storage systems, these solutions are no longer viable [1]. I evaluated the known concept of cryptographic deletion and identified the problems it causes with large storage-systems. I present a mechanism which, with the help of a key-management algorithm, enables secure deletion in large, untrusted storage-systems by means of cryptography. I measured the performance of a large object-store running a prototype implementation of this mechanism that showed promising runtime overheads. This demonstrates that cryptographic deletion is a feasible solution in large object stores, and suggests applications in other storage systems, such as databases and file systems.

## 1 Motivation

Cloud Computing is an established technology for outsourcing It resources. For individuals as well as corporations, it offers cheaper and often more advanced services than they could provide by themselves. The technology faces many challenges that prevent its widespread adoption. some of those, like network bandwidth, can be overcome with improvements to existing technologies. But security and data privacy is the major problem, which remains largely unsolved. The paradigm shift of moving from ones own, trusted infrastructure to third party cloud-offerings, creates these new security concerns which can only be addressed with new approaches instead of linear improvements.

The area of Cloud Computing security is a highly researched topic today for these reasons, but I have identified a research gap in the secure, and irrevocable deletion of data on untrusted storage-systems. We lose control over our data by trusting them to third parties like cloud-storage providers. They will usually provide some degree of confidentiality, reliability, and prevent unauthorized access to customers data. But the assured, secure deletion of data is rarely considered, neither during the use of their service, nor after. In order to offer reliability, providers keep backup copies of customers data which can remain accessible long after they deleted the data, or terminated the service. Future security breaches at these providers can make this data accessible to anyone. Furthermore, providers often keep such deleted data on purpose to enable future analysis [2].

## 2 Cryptographic Deletion

Cryptographic deletion describes the concept of deleting data by keeping it in an encrypted form, and removing any access to the encryption keys. Conceptually this follows the “divide and conquer” approach, i.e. it divides the primary problem of securely deleting data into the subproblems of encrypting the data and the subproblem of securely deleting the encryption key. The subproblem of encrypting data has been sufficiently solved by existing encryption algorithms [3]. The second subproblem of securely deleting the encryption key is an easier problem than the primary one, because the challenge is the same but with a smaller data size. With this approach, I reduce the problem of securely deleting large amounts of data to the problem of securely deleting a small encryption key.

## 3 Context

The “Secure Data Erasure in Untrusted Storage Systems” describes the core topic of my Ph.D. thesis titled “Security Aspects for Cloud Services”. In this thesis I present 1) a mechanism for formally describing security requirements towards cloud services in the context of TOSCA-based cloud-service templates, as well as 2) concepts and implementations of different security aspects that can be used with these cloud services. I gave an overview of this Ph.D. topic at the 2013 SummerSOC and presented part one, the formal definition of security requirements, as well as four initial security aspects from part two. In my presentation I will briefly introduce my framework (part one from above) and describe my current work on the security aspect for secure data erasure in more detail. I will present the theoretical background for cryptographic deletion and show, based on complexity calculations, how the existing approaches are insufficient. I will introduce my solution and present my current work on a prototypical implementation as well as first results from performance measurements.

## References

1. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the 6th Conference on USENIX Security Symposium, USENIX Association, Berkeley, CA, USA (1996)
2. Schneier, B.: File deletion (Sept 2009), https://www.schneier.com/blog/archives/2009/09/file deletion.html
3. Ferguson, N., Schneier, B.: Practical Cryptography. John Wiley & Sons, Inc., New York, NY, USA, 1 edn. (2003), p. 83 ff