



Eötvös Loránd Tudományegyetem

Informatikai Kar

Információs Rendszerek Tanszék

A semantic framework for managing smart grid consumption data

Pinczel Balázs
programtervező informatikus
MSc, 2. évfolyam

Laki Sándor
tanársegéd

Mátray Péter
tanársegéd

Budapest, 2012

A dolgozat a TÁMOP 4.2.1/B-09/1/KMR-2010-0003 számú pályázatának támogatásával készült.

Contents

1	Introduction	2
2	Smart grid overview	2
3	Data model	3
3.1	Our approach: a semantic data model	3
3.2	Semantic technology overview	5
3.2.1	Resource Description Framework	5
3.2.2	SPARQL	5
3.3	Ontology and Smart grid emulator	6
4	System architecture	7
4.1	Data source	8
4.2	Database	8
4.2.1	The SemWeb Portal	8
5	Middleware	9
5.1	Functionality	9
5.1.1	Authentication and session management	9
5.1.2	Managing consumption data	10
5.2	Web methods	11
5.2.1	The <code>login</code> method	11
5.2.2	The <code>getAppliances</code> method	12
5.2.3	The <code>getConsumption</code> methods	12
5.2.4	The <code>getShare</code> methods	13
5.2.5	The <code>getBill</code> method	13
5.3	Query optimizations	14
5.4	Special features to preserve the clients' resources	17
6	Clients	19
6.1	Web client	19
6.2	Android client	21
7	Conclusion and future work	21
8	References	23

1 Introduction

In recent years, energy providers of many countries have introduced the so-called smart meters which enable them to monitor the current, real-time energy consumption of their customers, eliminating the need for monthly readings. The consumption data from these smart energy grids is useful to both the energy providers, and their customers.

We designed and implemented a flexible semantic framework to store, handle, and present consumption data coming from a smart grid. It can be used by smart home owners to have a detailed view about the consumption of their household, thus helping them to be more conscious about their ecological footprint, and allowing them to optimize their energy usage, and reduce their costs. Data is presented to them via easy-to-read charts, or in the form of a detailed energy bill. It also contains functions to help estimating the expected energy cost in advance, using the information about the past. The system lets its users answer questions like ‘Which appliance used the least electricity today?’, ‘Which appliance was the most expensive in the past month?’, or ‘How much would I have to pay at the end of the month?’.

The framework provides means for intelligent data management for both the energy providers, and the customers. Energy providers benefit from the federating capabilities of the framework’s semantic data model, and the customers benefit from finding out new kind of information about their lives.

2 Smart grid overview

Over the last decade, humanity’s awareness grew towards the problems of energy consumption. Governments support the usage of renewable resources, and international campaigns try to persuade the people to be more conscious about their energy consumption.

Smart meters could be the source of this consciousness. But they can solve other problems, too. One of the main problems in the energy industry is that electricity cannot be stored. This can lead to two outcomes. Either we have too much, or too few electricity. If we have too much, we should sell it at a low price. If we have lack of electricity, we should buy at a high price. Either way, it’s not profitable.

A smart grid is an electric grid, which can collect live data from its components: the households, the offices, the power plants and generators, etc. With the help of smart grids, energy providers can monitor more precisely the demand for electricity. Moreover, they can influence the demand, by fine tuning the prices. This can lead to a much more balanced market. This is good for the providers, good for the customers, and good for the nature.

Although the topic of smart grids is a very dynamically growing area, there are still difficulties which has to be fought. The main difficulty is that there is no uniform way of dealing with the huge amount of data a smart grid produces. We tried to address this issue with introducing a general data model within our framework, which can be adapted to be used with the existing solutions.

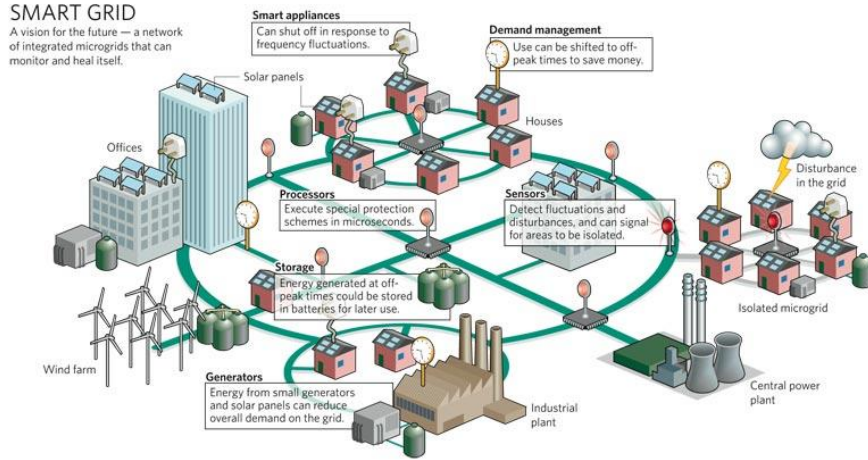


Figure 1: Overview of a smart grid

3 Data model

Choosing or designing a suitable data model is essential for software systems that are processing large amounts of data — such as our framework for smart grid data management. The decisions about the data model affect many properties of the resulting software, for example, expressiveness, efficiency, or ease of maintenance. Also, these decisions are hard to modify once they are carried out and built upon.

In the case of smart grid consumption data, we have to deal with an additional peculiarity. The de-facto standard for describing smart meter data and communication is still missing. Different smart meter manufacturers use different communication protocols and data models in their products. But usually a smart grid has to support multiple manufacturers and products, because different types of energy consumers (houses, apartments, offices, factories, etc.) may need different types of smart meters.

3.1 Our approach: a semantic data model

When we made our decision on the data model of our framework, we had to take into account the following aspects:

- **Compatibility.** The model should be able to represent consumption data coming from any current or future smart meter of any manufacturer. Conversion from the smart meters' data model to the framework's data model should be straightforward.
- **Flexibility.** It should easily be extended with new concepts, if necessary. If the model is extended, the applications using the old model should remain operational.
- **Standardization.** The data model should be built upon an already existing, well-tried paradigm, which is supported by standards, and various

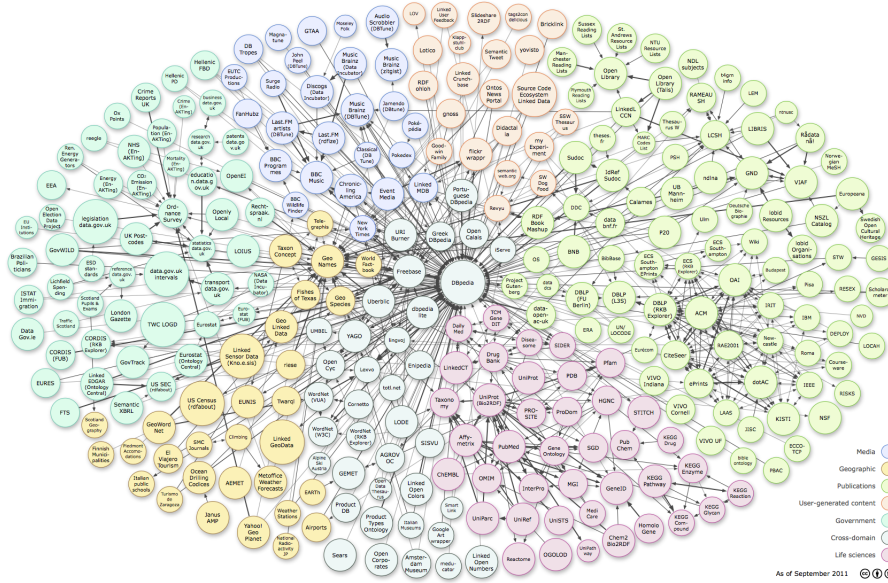


Figure 2: The Linking Open Data cloud.

tools and utilities.

Our decision was a semantic data model, which uses the Resource Description Framework (RDF) [1, 2], with a well-defined ontology. It satisfies all the mentioned requirements, and has other advantages too.

The semantic approach focuses on the meaning (the semantics) of the data, instead of the format (the syntax). This makes it capable to represent consumption data coming from any smart meter, because while the format of the data can be different for different smart meters, the meaning of the data is the same. Transforming data from arbitrary input format to the format of our data model is not hard, thanks to the simplicity of the graph-based semantic data model. Furthermore, there are tools and standards for the automatic data conversion from other common formats, for example, W3C's RDB2RDF¹, which aims to standardize the mappings from relational databases to semantic data models.

This model is flexible too. To extend the model with a new concept, all we have to do is to modify the ontology. If we want to apply the new concept to the old data as well, we could do this with the help of inference.

The semantic data model is standardized by the W3C, and it gains more and more popularity. DBpedia, LinkedGeoData, GeoNames, OpenStreetMap, Apple, Google, Yahoo and Microsoft are all among the supporters and users. It shows the popularity of the paradigm, that a high variety of topics are covered by openly available semantic data (see Figure 2).

¹ See <http://www.w3.org/2001/sw/rdb2rdf/>

3.2 Semantic technology overview

3.2.1 Resource Description Framework

The Resource Description Framework (or RDF) [1, 2] is a data model designed by the World Wide Web Consortium (W3C) for representing knowledge. Its concepts are similar to human thinking and statement making, and this makes it capable of representing knowledge focusing on the semantics instead of the syntax.

In RDF, we represent knowledge by making statements. Each statement consists of a subject, a predicate, and an object. For example, the statement “John is a person” has *John* as a subject, *is-a* as the predicate, and *person* as the object.

If our statements are represented this way, then we can easily imagine our knowledge as a graph: for each statement, we have a node labeled with the subject, a node labeled with the object, and a directed edge from the subject to the object, which is labeled with the predicate. It is possible for a node to be a subject in one statement, and an object in another, enabling the construction of rich graph structures representing complex relationships among concepts.

To represent a statement in RDF, we have to assign RDF terms to its subject, predicate and object. An RDF term is either a URI reference, an RDF literal, or a blank node. A URI reference is used to uniformly identify an entity, concept, or property, for example, `http://xmlns.com/foaf/0.1/age` represents the concept of age as a property. URI references can be used as subjects, predicates, and objects. An RDF literal is used as the actual value of a property, for example, `"25"` is the literal representing the value 25. RDF literals can only be used as objects. (In the graph model this means that they are always leaves.) To abbreviate long URI references, we can define aliases for prefixes, and use the aliases instead of the original, long prefixes. For example the short form of `http://xmlns.com/foaf/0.1/age` could be `foaf:age`.

A blank node is used when we don’t need, or we don’t want to use a uniform identifier for an entity, but we want to be able to use the entity as a subject. It is particularly useful, when a nameless entity is represented by multiple nodes, and we need a single node to group them together. For example, an address can consist of a street name, a number, a zip code, and a town. Together they identify the address, so no further identification is needed.

A tuple of an RDF subject, predicate and object is called an RDF triple, and a collection of such RDF triples is called an RDF document. In this paper, the term semantic model (or just model) is used as a synonym of RDF Document.

RDF could only be useful, if people around the world would use the same URI references for the same concepts. A mapping from concepts of the real world to URI references is called a vocabulary. If we extend a vocabulary with additional information about the relationships between the defined concepts, we get an ontology. Ontologies could be used for inference, where an existing piece of data and the rules represented in an ontology together imply a new piece of data.

3.2.2 SPARQL

SPARQL [3, 4] is the de-facto query language for the RDF data model. The queries are based on graph patterns, and the query results are determined by

matching the patterns against an RDF document. A graph pattern is basically a set of RDF triples, but where each component of a triple can be substituted with a variable.

```
1 PREFIX foaf: <http://xmlns.com/foaf/>
2
3 SELECT ?person
4 WHERE
5 {
6   ?person a foaf:Person .
7   ?person foaf:age "25" .
8 }
```

Above there is a simple SPARQL query. Its graph pattern consists of two triples. To evaluate the query, we need to find matching parts in our data graph, and find the values which matched the variables. For example, if the dataset contains the triple `http://www.example.org/John a foaf:Person` and it also contains `http://www.example.org/John foaf:age "25"` then the URI reference `http://www.example.org/John` would be in the result of the query.

SPARQL contains operators, which can be used in the graph pattern of a query. The two most important operators are the **FILTER** operator and the **UNION** operator. **FILTER** is used to evaluate a boolean expression, and filter the query results based on its value. **UNION** is used to merge the results of two independent query parts.

```
1 PREFIX foaf: <http://xmlns.com/foaf/>
2
3 SELECT ?person
4 WHERE
5 {
6   { ?person foaf:age "30" . }
7   UNION
8   { ?person foaf:age "25" . }
9 }
```

For example, the query above will find persons aged 30 or 25. The following query does the same, with the **FILTER** operator.

```
1 PREFIX foaf: <http://xmlns.com/foaf/>
2
3 SELECT ?person
4 WHERE
5 {
6   ?person foaf:age ?age .
7   FILTER(?age = "25" || ?age = "30")
8 }
```

3.3 Ontology and Smart grid emulator

The ontology for our data model is based on the smart grid ontology made by Andread Wagner and Andreas Harth.² The main components of the ontology are the following:

- households: these are the smart homes in the smart grid,

² See <http://people.aifb.kit.edu/awa/2011/smartgrid/>

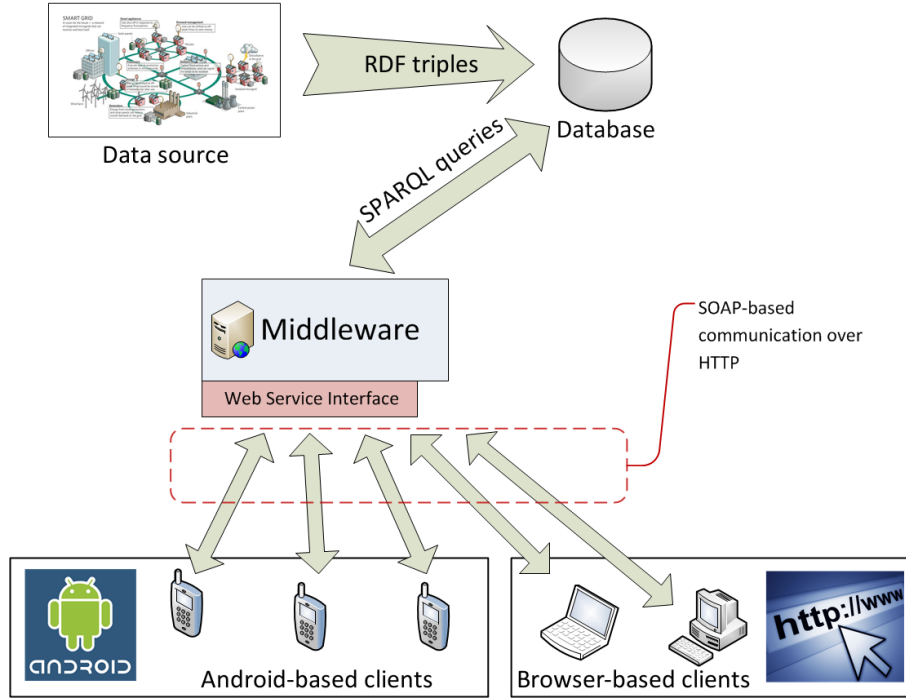


Figure 3: The framework’s components, and connections between them

- users: each user belongs to a household, and has a username,
- appliances: there are multiple electric appliances for each household,
- consumption data: for every electric appliance in every household, we know the amount of consumed electric energy over continuous time intervals, as measured by the smart meter.

The authors of the ontology made a tool for generating smart grid data which corresponds to the smart grid ontology. The generated consumption values are based on distributions taken from real measurement, so the data is realistic-looking.

4 System architecture

The framework has a classic three-tier architecture, where the data, business logic, and presentation are separated.

The data tier contains the database, in which the consumption data is stored. This data comes from an external data source, which is not part of the framework. The business logic tier contains the middleware, which serves the clients by querying the database, processing the query results, and sending the prepared content back to the client. The presentation tier contains the client applications, currently the framework has two client applications: a browser-based client (a web application), and a specialized client application for mobile devices with Android operating system. The resource-intensive operations are carried out at

the middleware, thus the clients can be thin: their only function is to provide a visually appealing interface so that users could access their consumption data with ease. This property is crucial, because it enables the clients to efficiently run on devices with limited resources, such as smart phones, or tablets.

4.1 Data source

During the establishment of the framework, we suppose an external data source which produces the consumption data. This data source is typically a data collection module of the smart grid system. As we do not have access to such a system currently, for the prototype we used the Smart Grid Emulator (described in Section 3.3), though – thanks to the benefits of the semantic paradigm – this can easily be replaced with any other data source, for example, a real smart grid with real consumption data.

4.2 Database

The consumption data for the framework is stored in a database, which supports semantic querying. The only requirement the database must meet is being able to query the data using SPARQL. In the prototype we used an Oracle 11g database, because it supports the storage of RDF data in semantic models, and with the Jena Adapter³ it is possible to query those models using SPARQL.

4.2.1 The SemWeb Portal

During the development of the framework, we used the SemWeb Portal⁴ as a utility in certain design tasks and testing. The SemWeb Portal is a web portal, which functions as a front-end for semantic databases. It is developed by the Department of Information Systems at Eötvös Loránd University. It provides an easy-to-use web interface for common tasks in semantic technology, and allows the users of a semantic database to use existing semantic utilities with their own data. The SemWeb Portal offers the following features:

- **Semantic models:** the users can create semantic models, and fill them with RDF data by uploading files in various RDF serialization formats. Alternatively, they can create so-called virtual models, with which they can merge multiple existing models into a single one, without actually materializing the union of the individual models.
- **Inference:** the users can create rulebases, fill them with rules, and use them to infer additional knowledge from the combination of the stored facts, and the provided rules.
- **Querying:** the stored data can be queried using SPARQL. The queries can be executed over multiple models or virtual models, with or without inference with the selected rulebases. The queries can be saved and reloaded, to avoid the need for typing frequently used queries.

³ See http://docs.oracle.com/cd/E11882_01/appdev.112/e11828/sem_jena.htm

⁴ The portal is currently available at <http://tomx.inf.elte.hu:7001/SemWeb-war/>

- **Cooperating with other users:** the Portal supports sharing the stored knowledge with other users. The semantic models, virtual models, rule-bases, and saved queries are private by default, but the owners can make them public, allowing every other user to see and use them. For a better control, user groups can be created, and knowledge can be shared with selected groups.
- **Visualization:** in order to facilitate the comprehension of complex data, and the construction of complex queries, the Portal offers visualization functionalities as well. In the case of data visualization, the user can see the actual graph representation of the stored knowledge. Query visualization shows a similar graph, based on the graph pattern in the query. The purpose of this feature is to aid inexperienced users with query building: it is possible to create a query using only its visual representation, without writing it in SPARQL. [5]

For the development of this framework, the SemWeb Portal is proven to be helpful in designing, debugging, testing, and benchmarking queries, by letting us perform these tasks with a few clicks instead of writing code for them.

5 Middleware

The middleware constitutes the core of the framework. It plays the role of a mediator between the storage and presentation components. However, it has to do more than just simply forwarding the data to the clients: certain complex operations are also carried out here, since other components may be less efficient at performing these tasks, for example, because of their limited resources.

This component is implemented as a SOAP web service, with the JAX-WS Java API [6], so the communication with the clients is done by calling the web methods of the web service. The connection to the database tier is configurable, thus the middleware can be used with any kind of storage system which supports SPARQL querying. The configuration can be done by implementing a Java interface, which consists of a single function for query execution. In the prototype, we have created a reference implementation which uses the Jena Adapter to connect to an Oracle 11g database, and to query the RDF data located in a semantic model.

5.1 Functionality

5.1.1 Authentication and session management

Information about a household's consumption is considered to be of personal nature, thus it is required for the users of the framework to properly authenticate themselves, before accessing their data. The authentication follows the classic model: each user has an account, which is identified by a username, and protected by a password. For the authentication the user has to provide a matching username-password pair.

As it was mentioned in Section 3.3, the data model contains user information about each household's members. This user information contains usernames, and connects each user with the corresponding household, but it does not contain passwords. Therefore, the administrators of an instance of the framework

have to provide other means to store passwords, and link them to accounts. It is important to mention that this is not a flaw in the data model, the lack of password data is deliberate: it is debated, whether the semantic paradigm is a good choice for password storage in some cases⁵, and this way the framework can remain flexible, by leaving the decision to the administrators. During authentication, the framework checks if a user exists with the provided username, and gets information about the corresponding household, but any particular way of password checking should be configured by implementing a function designated for this purpose.

After successful authentication, a client application can get the user's data by calling the web methods of the middleware's web service. Typically, a user wants to perform more than one operation while using the client application, for example, rendering different views of the data, and this results in multiple successive web method calls. However, sending account credentials for each web method call and checking it at the middleware would be inefficient, as it would consume both bandwidth and the computational resources of the middleware and the database. To avoid this, the framework uses session management. After successful authentication, a user has an active session, of which the following information is stored at the middleware: a session ID, the username, the time when the session was created, and the household of the user. (The latter is there for efficiency reasons, otherwise it would be necessary to look it up from the database for every kind of consumption data handling.) The session ID (an integer of 64 bits) is sent to the client application upon successful authentication, and is a necessary parameter to all subsequent web method calls.

5.1.2 Managing consumption data

It is the middleware's primary function to manage the various consumption data. As it was described in Section 3.3, the framework's data model contains the following attributes about consumption: for every electric appliance in every household, we know the amount of consumed electric energy over continuous time intervals, as measured by the smart meter. Although the Smart Grid Emulator (described in Section 3.3) generates consumption data over regular time intervals, the framework handles irregular intervals too, that is, the length of the time intervals need not necessarily be the same.

Most users probably find it more practical, if they can review their consumption in terms of money. For example, they would like to know how much did their dishwasher cost them in the past week. To calculate this, the middleware needs additional information about the pricing of electric energy. As it was mentioned in Section 2, smart grids enable the use of more sophisticated pricing schemes, for example, the unit price of electric energy can depend on the time of day, the day of the week, or there can be unexpected limited offers. Since the pricing can be complex, and it can be unique to an energy provider, the pricing model is not part of the framework. Instead, to calculate consumption in terms of money, the middleware uses a function to determine the unit price of electric energy at a given time interval. The pricing model can be incorporated into the framework by implementing this function. The rest of the middleware's code only uses it to get the output (the unit price) from its input (a time interval),

⁵ See <http://answers.semanticweb.com/questions/1003/store-passwords-as-rdf>

and we do not suppose anything about its internal logic. This way, the framework can be flexible about pricing too. In the prototype, we created a reference implementation for demonstration purposes, which determines the unit price based on the time of day, with an afternoon peak period, and a night off-peak period, with the rest of the day being in normal price range.

The client applications can access the data using different views, to illustrate different aspects about the user's consumption. The middleware provides the following views:

View 1: Consumption over a period of time. In this view, the user can select one or more appliances, and a period of time, and get the consumption data for the selected values. This basically means, that the user would like to see a subset of his consumption data, by filtering for only the selected appliances, and narrowing the examined time interval to the selected period. The user can choose whether he wants to see the consumption data in terms of energy, or money. He can also decide to view the total consumption (summed for all individual appliances) for the same period of time, together with the consumption of the individual appliances. This view is designed to be displayed as a line chart in the client applications, with a separate line for every selected appliance.

View 2: The appliances' share of the total consumption. The user can use this view to answer questions like 'Which appliance used the least electricity today?' or 'Which appliance was the most expensive in the past month?'. He can select a period of time, and the result is a percentage for each appliance, indicating its share of the total consumption for the selected period. The proportions can be calculated in terms of energy or money, and since the pricing is usually more complex than multiplying with a fixed unit price, the percentages can be different for the two. This view is designed to be displayed as a pie chart in the client applications.

View 3: Detailed electric bill. After selecting a period of time, the result of this view is like a detailed electric bill, where there is a cost for each appliance, so it can be seen how the total cost got summed up. When using this view, the user can set the endpoint of the period to future dates. In this case, the user will get predicted values as a result. The prediction is based on the already known data: this way the estimate will be close to the real value if the average daily consumption remains the same.

5.2 Web methods

The functions described in Section 5.1 are implemented as web methods of the middleware's web service. This section provides a brief summary of these methods.

5.2.1 The login method

The method `long login(String username, String password)` performs the authentication, as described in Section 5.1.1. The return value is the session ID, if the authentication is successful, 0, if the `username` or `password` is incorrect, and `-1`, if an error occurred during authentication, for example, a database error or network error.

5.2.2 The `getAppliances` method

The method `String getAppliances(long sessionId)` can be used to retrieve the appliances of the user, whose session has the provided `sessionId`. The return value is a multi-line text, which contains information about one appliance per line. Each line contains the long name of an appliance (which is a URI), and the short name of that appliance, separated by a semicolon. The client applications need these long names when calling a `getConsumption` method (see Section 5.2.3), because that can be used to identify an appliance in the database, while the short names are the ones that are displayed to the user. A typical return value can be seen in the box below.

```
http://people.aifb.kit.edu/awa/2011/smartgrid/dishwasher0;Dishwasher
http://people.aifb.kit.edu/awa/2011/smartgrid/light0;Light
http://people.aifb.kit.edu/awa/2011/smartgrid/fridge0;Fridge
http://people.aifb.kit.edu/awa/2011/smartgrid/oven0;Oven
http://people.aifb.kit.edu/awa/2011/smartgrid/tv0;Tv
http://people.aifb.kit.edu/awa/2011/smartgrid/cookTop0;CookTop
http://people.aifb.kit.edu/awa/2011/smartgrid/outletDesk0;OutletDesk
http://people.aifb.kit.edu/awa/2011/smartgrid/microwave0;Microwave
```

5.2.3 The `getConsumption` methods

The method `String getConsumptionInEnergy` provides consumption data, using View 1 described in Section 5.1.2. It has the following parameters:

- `long sessionId`: the session ID,
- `String from`: the start time of the selected period in `xsd:dateTime` format,
- `String to`: the end time of the selected period in `xsd:dateTime` format,
- `String[] appliances`: an array containing the long names of the selected appliances (see the `getAppliances` method in Section 5.2.2),
- `boolean sum`: `true`, if the user would like to get the total consumption too.

The return value is a multi-line text, which contains consumption data about the time intervals, which are inside the selected period. The lines hold information about an appliance's consumption during a time interval. Each line contains the midpoint of the time interval in `xsd:dateTime` format, the short name of an appliance, and the value of consumption in terms of energy (the unit is Wh). These fields are separated by a semicolon. The output lines are sorted by ascending order of time. Lines with the same time are arranged by alphabetical order of the appliance's short name. A typical return value can be seen in the box below.

```

2012-02-01T01:20:00;Dishwasher;0.0
2012-02-01T01:20:00;Total;1.3524564967543125
2012-02-01T01:20:00;WashingMachine;0.0
2012-02-01T02:20:00;Dishwasher;0.030928706367064338
2012-02-01T02:20:00;Total;2.65840376523261
2012-02-01T02:20:00;WashingMachine;0.09750984153865923
2012-02-01T03:20:00;Dishwasher;0.04477151231712637
2012-02-01T03:20:00;Total;0.9312049253871866

```

The method `getConsumptionInMoney` has exactly the same parameters as `getConsumptionInEnergy`, and returns a similarly structured result, with the consumption in terms of money.

5.2.4 The `getShare` methods

The method `String getShareInEnergy` provides consumption data, using View 2 described in Section 5.1.2. It has the following parameters:

- `long sessionId`: the session ID,
- `String from`: the start time of the selected period in `xsd:dateTime` format,
- `String to`: the end time of the selected period in `xsd:dateTime` format

The return value is a multi-line text, which contains the information about the individual appliances' share of the total consumption. Each line contains the short name of an appliance, and a percentage indicating the appliance's share, separated by a semicolon. A typical return value can be seen in the box below.

```

Dishwasher;0.01887381103790684
CookTop;0.08228143175555451
OutletDesk;0.05903480262141643
Oven;0.07780926366302021
Fridge;0.23102556615617023
Microwave;0.01872291641382586
WaterBoiler;5.0564222672487E-164
OutletBedside;0.007287844911628412
Light;0.23344719082143278
Tv;0.465856753065993
WashingMachine;0.03783129177533173

```

The method `getShareInMoney` has the same parameters as `getShareInEnergy`, and returns a similarly structured result, with the proportions calculated in terms of money.

5.2.5 The `getBill` method

The method `String getBill(long sessionId, String from, String to)` provides consumption data, using View 3 described in Section 5.1.2. The return value is a multi-line text, where each line contains the short name of an appliance, and the cost of that appliance's electric energy consumption in the selected period of time, separated by a semicolon. A typical return value can be seen in the box below.

```

Dishwasher;1.0400578923449359E-4
CookTop;4.5341903825850996E-4
OutletDesk;3.2531645180781173E-4
Oven;4.287747641837529E-4
Fridge;0.001273086622153814
Microwave;1.0317427118881301E-4
WaterBoiler;2.786385789027121E-166
OutletBedside;4.01603078641696E-5
Light;0.0012864311970271398
Tv;0.0025671444508761293
WashingMachine;2.0847264767837525E-4
Total;0.006789985540273009

```

This method also provides functionality for the estimation of future costs. For this purpose, the client has to call the method with an arbitrary future end time, up to which the estimation will be carried out. The prediction uses simple linear approximation: let t_s , t , and t_e denote the start time, current time, and end time respectively. Then, if the known cost up to the current time is c , then the estimated cost at the end time is $c_e = c \cdot \frac{t_e - t_s}{t - t_s}$.

5.3 Query optimizations

As it was mentioned in Section 3, the framework’s data model is based on the Resource Description Framework. Choosing the data model this way immediately implies that data management operations should be carried out by executing SPARQL queries. In our framework, the communication between the middleware and the database consists entirely of SPARQL queries and query results. Every data management task, and even the task of authentication requires query execution. Due this design philosophy, SPARQL queries constitute central building blocks of the system. In this subsection we focus on these queries.

SPARQL – similarly to SQL – is a declarative language, which means, that a query defines perfectly what the results should be when it’s evaluated on a given dataset, but it does not define the method of execution. Theoretically, every execution method is correct, as long as it gives the result defined by the query. This means that the query execution engine can choose from different methods, but unfortunately it does not always choose the best one, because finding the optimal (or even a good enough) execution method could be a complex problem. What makes the problem even more difficult, is that different semantic databases may behave differently on the same query, for example, if an execution method for a certain query is efficient on a native triple store, it could be inefficient using a triple store which is built upon a relational database, and vica versa⁶.

Most query optimizers work by rearranging the queries using rewriting rules. These rules guarantee, that the modified query is equivalent to the original one, meaning that they give the same result when evaluated on the same dataset. Such equivalences are introduced, summarized and proved in [7]. The idea behind the optimization is that the execution engine may find a query, which is equivalent to the original one, but is easier to evaluate. Despite the ongoing

⁶ See <http://lists.w3.org/Archives/Public/public-lod/2008Aug/0061.html> for benchmarking results which compare native triple stores with mapped relational databases

```

1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 PREFIX sg:
3   <http://people.aifb.kit.edu/awa/2011/smartgrid/schema/smartgrid#>
4
5 SELECT ?household
6 WHERE
7 {
8   ?person a foaf:Person .
9   ?person sg:actorDomain sg:CustomerDomain .
10  ?person sg:actorType sg:NonGovernmentActor .
11  ?person sg:actorNetwork ?household .
12  ?person foaf:name ?username .
13  FILTER(str(?username) = "user0")
14 }

```

Query 1: Checking the existence of a given user, and retrieving the corresponding household

```

1 PREFIX sg:
2   <http://people.aifb.kit.edu/awa/2011/smartgrid/schema/smartgrid#>
3
4 SELECT ?appliance
5 WHERE
6 {
7   ?appliance a sg:Appliance .
8   <http://people.aifb.kit.edu/awa/2011/smartgrid/household0>
9     sg:device ?appliance .
10 }

```

Query 2: Retrieving the appliances of a given household

research on SPARQL query optimization, it seems that currently deployed optimizers are still in a rudimentary state, and often fail to achieve considerable speedups compared to “brute force” query execution strategies. Due this fact, we chose to manually fine tune the queries used by the framework. This was done by benchmarking different variants of the queries, and picking the ones that proved to be the most efficient. In the following paragraphs, these queries are described and explained, along with the performed optimization steps. For each query we present its code in a separate box.

Query 1 is executed during the authentication process. From its result, we can decide whether an account exists with the given username, and (if there is such an account) we can retrieve the corresponding household. Lines 8–10 identify the smart grid users, line 11 retrieves their households, and lines 12–13 narrow the results to the single user we would like to find (in this example: a user with username “user0”).

Query 2 is used by the web method `getAppliances` (see Section 5.2). Here, line 8 contains the URI of the household in question, which we got as the result of Query 1.

Query 3 is used by the web methods `getConsumption` (see Section 5.2). This query can be quite complex if the web methods are called with many selected appliances. The reason behind this is our query optimization process: we rewrote the query from a simpler form to this more complex structure to achieve shorter running times. It is built up from parts, which are connected with `UNION`

```

1 PREFIX sg:
2   <http://people.aifb.kit.edu/awa/2011/smartgrid/schema/smartgrid#>
3 PREFIX es:
4   <http://people.aifb.kit.edu/awa/2011/smartgrid/schema/eventstream#>
5
6 SELECT ?appliance ?startTime ?endTime ?value
7 WHERE
8 {
9   {
10     ?consumptionData a sg:EnergyConsumptionData .
11     ?consumptionData es:source
12       <http://people.aifb.kit.edu/awa/2011/smartgrid/fridge0> .
13     ?consumptionData es:source ?appliance .
14     ?consumptionData es:startTime ?startTime .
15     ?consumptionData es:endTime ?endTime .
16     ?consumptionData sg:value ?value .
17   }
18   UNION
19   {
20     ?appliance a sg:Meter .
21     <http://people.aifb.kit.edu/awa/2011/smartgrid/household0>
22       sg:device ?appliance .
23     ?appliance sg:dataFlow ?informationRelation .
24     ?informationRelation sg:data ?consumptionData .
25     ?consumptionData es:startTime ?startTime .
26     ?consumptionData es:endTime ?endTime .
27     ?consumptionData sg:value ?value .
28   }
29   FILTER(str(?startTime) >= "2012-03-01T00:00:00"
30         && str(?endTime) <= "2012-03-10T00:00:00")
31 }

```

Query 3: Retrieving consumption of the given appliances, and (optionally) the total consumption, for the selected period of time

operators. Lines 9–17 are responsible for the querying of the fridge’s consumption (the fridge’s long name can be seen in line 12, and line 13 is seemingly redundant, however, without this line, we would have an empty **appliance** column in the query result). If more appliances are selected, there are similar parts for each of them. Lines 19–28 are responsible for querying the total consumption, which is done by querying the consumption of a special appliance: the smart meter of the household. Finally, lines 29–30 are narrowing the examined time interval to the selected period.

This way the query defines separate parts for each appliance, where each part uses a variable called **appliance** specifying the desired device (or the smart meter for the total consumption). Originally the query had only one part for the appliances, and the variable **appliance** took its values based on a filter expression like `FILTER(?appliance = appl1 || ?appliance = appl2 || ...)`. However, we found that this solution does not scale well with the number of appliances. When many appliances were selected, evaluating the filter expression for each candidate was slow. This phenomena is due the imperfection of query optimization methods in Jena/Oracle⁷. Instead of filtering, the current version uses pattern matching, which works faster.

Another optimization step that we apply can be seen in lines 29-30: we compare times as strings, not as date types. The same result could be achieved with a comparison like `?startTime >= xsd:dateTime("2012-03-01T00:00:00")`, but this would be slower, because upcasting a string to `xsd:dateTime` takes longer than downcasting `xsd:dateTime` to string. The results are the same with the string representation, because `xsd:dateTime`’s string serialization contains the date and time fields in decreasing order of significance.

Query 4 is used by the `getShare` and `getBill` methods. This is very similar to Query 3, the only difference is that this always consists of exactly two parts, where the first part queries the consumption of all appliances, and the second part queries the total consumption. For these methods all of this data is always needed, that’s why the complex form of Query 3 could be simplified. However, executing this query is not enough to get the results of a `getShare` or `getBill` method call: to get the summed consumption for each appliance over the selected period, we need to perform additional aggregation. Although SPARQL supports aggregates from version 1.1 (see [4]), when we tried to perform the summation using advanced queries, it turned out that performing the aggregation at the middleware would be more efficient.

5.4 Special features to preserve the clients’ resources

One of the framework’s design goals was that it should provide flawless user experience for users with mobile devices and tablets too. This was achieved partially by implementing the main features at the middleware, so that the clients could be thin. During development we realized that we could (and should) exploit further ideas in order to fully achieve this goal.

While the Android client was under development, we faced a problem with the `ksoap2-android` library⁸. If an Android client’s query resulted in large

⁷ The Jena Adapter for Oracle works by translating SPARQL queries to Oracle-specific SEM_MATCH semantic queries, and it is not clear at which point this scaling issue is introduced into the process.

⁸ A SOAP library for the Android platform, see <http://code.google.com/p/>

```

1 PREFIX sg:
2   <http://people.aifb.kit.edu/awa/2011/smartgrid/schema/smartgrid#>
3 PREFIX es:
4   <http://people.aifb.kit.edu/awa/2011/smartgrid/schema/eventstream#>
5
6 SELECT ?appliance ?startTime ?endTime ?value
7 WHERE
8 {
9   {
10    <http://people.aifb.kit.edu/awa/2011/smartgrid/household0>
11      sg:device ?appliance .
12    ?appliance a sg:Appliance .
13    ?consumptionData a sg:EnergyConsumptionData .
14    ?consumptionData es:source ?appliance .
15    ?consumptionData es:startTime ?startTime .
16    ?consumptionData es:endTime ?endTime .
17    ?consumptionData sg:value ?value .
18   }
19   UNION
20   {
21    ?appliance a sg:Meter .
22    <http://people.aifb.kit.edu/awa/2011/smartgrid/household0>
23      sg:device ?appliance .
24    ?appliance sg:dataFlow ?informationRelation .
25    ?informationRelation sg:data ?consumptionData .
26    ?consumptionData es:startTime ?startTime .
27    ?consumptionData es:endTime ?endTime .
28    ?consumptionData sg:value ?value .
29   }
30   FILTER(str(?startTime) >= "2012-03-01T00:00:00"
31          && str(?endTime) <= "2012-03-10T00:00:00")
32 }

```

Query 4: Retrieving consumption of all appliances, and the total consumption, for the selected period of time

amounts of data, the library consumed the device's memory while processing the SOAP response containing the query result. To solve this issue, we had to reduce the size of the middleware's responses, especially the result size of the `getConsumption` methods. We did this by placing three important and significant changes.

1. We switched from our original XML formatted responses to plain, semicolon-separated text responses. This reduced the size, because we could eliminate the repeating, relatively long⁹ tag names from the result.
2. We introduced short names for appliances in results. This had double advantages: we could eliminate the long, repeating prefixes of the appliance URIs, and we got readable, displayable names for the appliances.
3. The third change is the most important, and most useful in preserving resources and bandwidth, yet it is based on a simple idea. Usually the resolution of a mobile device's display is not too high, so why should we send detailed information, if the device can't display the details? For example – since a line chart consists of data points, which are connected by lines – if the screen's width is 500 pixels, then 200 data points are usually enough to get a nice view of the data. If the result size of a consumption query is above a certain threshold, we aggregate small groups of consecutive data points. For example, if we have 400 data points of hourly consumption, we can reduce it to 200 data points by summing them in groups of two, and we would get 200 data points of consumption data about two-hour long intervals. Our data remained the same, we only changed its resolution.

6 Clients

6.1 Web client

The web client is a web application built with the Google Web Toolkit¹⁰, using Google Chart Tools¹¹ to display data using interactive charts. It provides an easy-to-use, multi-language web interface for carrying out data management operations.

Consumption data is displayed using line charts. Each line corresponds to an appliance (or the total consumption), the horizontal axis represents time, and the vertical axis represent the value of the consumption in terms of energy or money. The data can be displayed using the normal view, or the cumulative view. The normal view displays the measured consumption at any given point in time, while the cumulative view displays the sum of the measurements up to any given point in time. The user can see additional information about every data point, by hovering the mouse pointer over them.

`ksoap2-android/`

⁹ Consider `<value>0</value>` as an example, where only one character is useful out of the sixteen.

¹⁰ See <https://developers.google.com/web-toolkit/>

¹¹ See <https://developers.google.com/chart/>

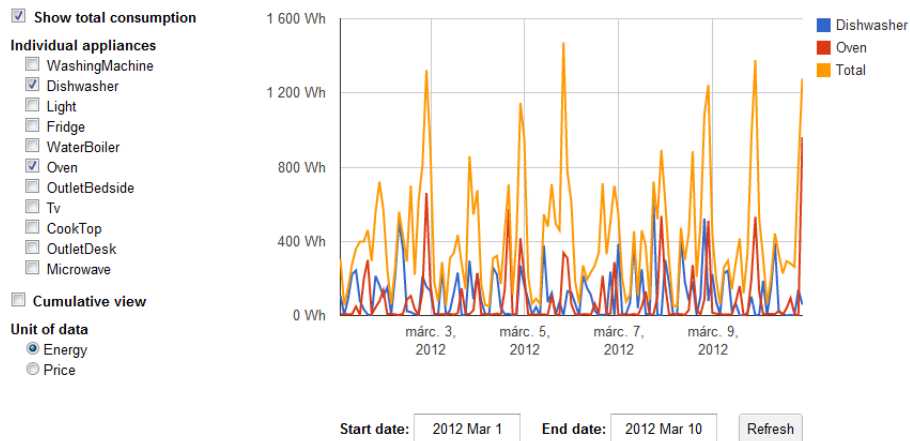


Figure 4: The web client's line chart displaying the consumption data

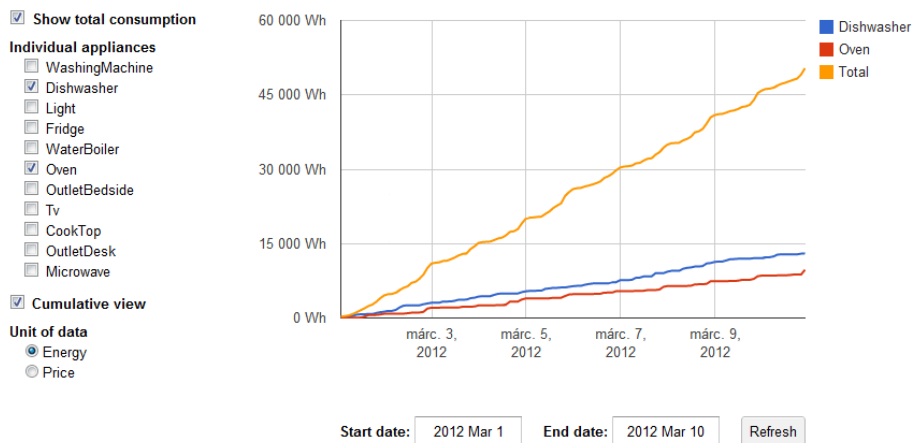


Figure 5: The web client's line chart displaying the consumption data using the cumulative view

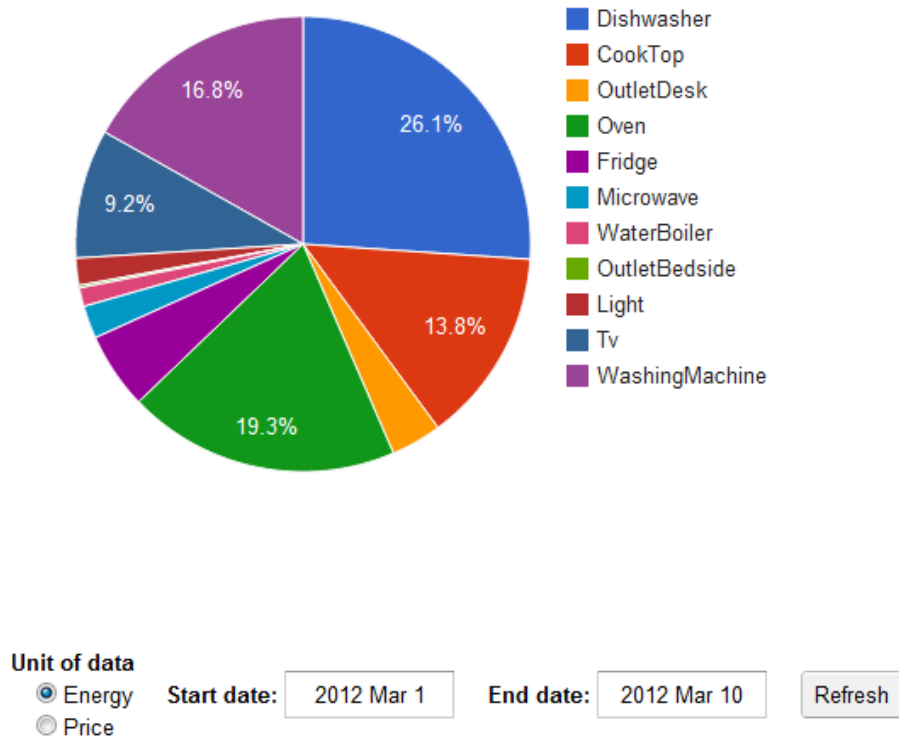


Figure 6: The web client’s pie chart displaying the appliances’ share of the total consumption

The appliances’ share of the total consumption is displayed using pie charts. Percentages and labels for the narrow slices can be seen by hovering the mouse pointer over them.

The bill can be viewed as a table, which is formatted to look like an actual bill.

6.2 Android client

The Android client is the work of Máté Pinczel. It provides the same functions as the web client, with some more additional features for comfort. For example, the application automatically detects the phone’s language, and sets its interface to be the same language (if it is supported). Since typing the account credentials every time using the on-screen keyboard could be rather cumbersome, the application stores the last successfully used username and password, and every time it loses connection, it reconnects automatically.

7 Conclusion and future work

We designed and implemented a framework which performs smart grid data management tasks efficiently. Thanks to its semantic data model, it can be

Appliance	Price
Dishwasher	149.26 Ft
CookTop	89.77 Ft
OutletDesk	21.24 Ft
Oven	110.05 Ft
Fridge	30.51 Ft
Microwave	14.13 Ft
WaterBoiler	8.80 Ft
OutletBedside	0.93 Ft
Light	9.91 Ft
Tv	50.46 Ft
WashingMachine	104.71 Ft
Total	589.77 Ft

Start date:
End date:

Figure 7: The web client displaying the bill view of the consumption data.

used as a general front-end for smart grid data centers. Its flexibility lets administrators of a data source adapt it to work with their existing systems, and customize it to suit their needs. The special optimizations allow it to work efficiently with mobile devices and tablets. Using this framework, the users can get to know their consumption patterns, and this could inspire them to act more environment-friendly, and save money at the same time.

To make it even more useful and practical, there are some tweaks and new features that could be implemented. Users would find it useful, if the system could provide advices on how to be more environment-friendly, or how to save more money. For example, it could help in scheduling repetitive energy-consuming tasks, by figuring out the time of day, when they could be performed with the least cost.

Existing features could be tweaked too. We could make better predictions by analyzing the habits of the household's inhabitants based on the consumption data. We could also take weekly or yearly trends into account when estimating future costs.

There are also interesting questions in the topic of semantic query optimization, that could be investigated further.

8 References

- [1] Frank Manola, Eric Miller, and Brian McBride. RDF Primer, February 2004. (W3C Recommendation).
- [2] Graham Klyne, Jeremy J. Carroll, and Brian McBride. Resource Description Framework (RDF): Concepts and Abstract Syntax, February 2004. (W3C Recommendation).
- [3] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF, January 2008. (W3C Recommendation).
- [4] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language, January 2012. (W3C Working Draft).
- [5] Gábor Rácz. VisualQuery: vizuális lekérdezés szerkesztő program, December 2011.
- [6] Robert Eckstein and Rajiv Mordani. Introducing JAX-WS 2.0 with the Java SE 6 Platform, September 2006.
- [7] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory, ICDT '10*, pages 4–33, New York, NY, USA, 2010. ACM.