University of Groningen

Web and Cloud Computing

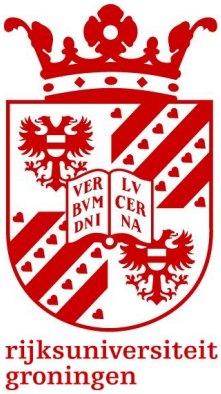
Smart Energy System

Group: 3 - Final report

Authors

Mark Soelman	s3224708
Jaap van der Vis	s2344076

October 28, 2019



Contents

1 Introduction

This document aims at giving a high-level overview of the Smart Energy System, which was developed for the Master's course of Web and Cloud Computing (class 2019-2020). The Smart Energy System that we developed is a prototype for simulating the integration of multiple households with the main power net in a smart way. In this document, we present the problem that motivated us to develop this prototype and how we came up with our solution, we give an overview of the software technologies we used to implement our solution, show a high-level graph of our architecture, briefly explain the components we have and their functionality, and, finally, we reflect on our work and draw a conclusion.

2 Problem and Presented Solution

With the growing concerns regarding climate change and global warming, the need for sustainable energy is becoming more and more important. Besides the fact that some energy providers are slowly moving from burning fossil fuels to generating green and sustainable energy, a growing interest arises under consumers to become fully self-sustainable, especially with respect to electricity. For instance, consumers are installing solar panels, heat pumps and sometimes even wind mills. Although the strive for self-sustainability moves forward, the technology is still a bit behind. There are often no clear visualization tools of a household's energy usage until the end of the month, the consumer is unable to grasp at what devices are using the most power at any point in time, and often, the consumer is unable to turn these devices on or off remotely. Furthermore, with the upcoming repeal of the "*salderingsregeling*" in the Netherlands, it will become less economically profitable for any household to generate more power than one is using.

2.1 Energy Awareness

To tackle the first point, multiple households can be connected to our system. Per household, the total energy usage, total energy production, and the financial balance is visualized in a graph, which is updated every 30 seconds. Furthermore, the battery capacity of a household is also displayed, providing the user with valuable interactive metrics. In order to give the consumer

more control over its energy usage, the power outlets (of which its device and usage are known) can be turned on or off using a simple interface, which could be useful when the consumer is off-site or when the activation status of a device is not easily toggled on or off (all devices without a power switch have to be manually removed from the power socket to turn it off).

2.2 Energy Trading

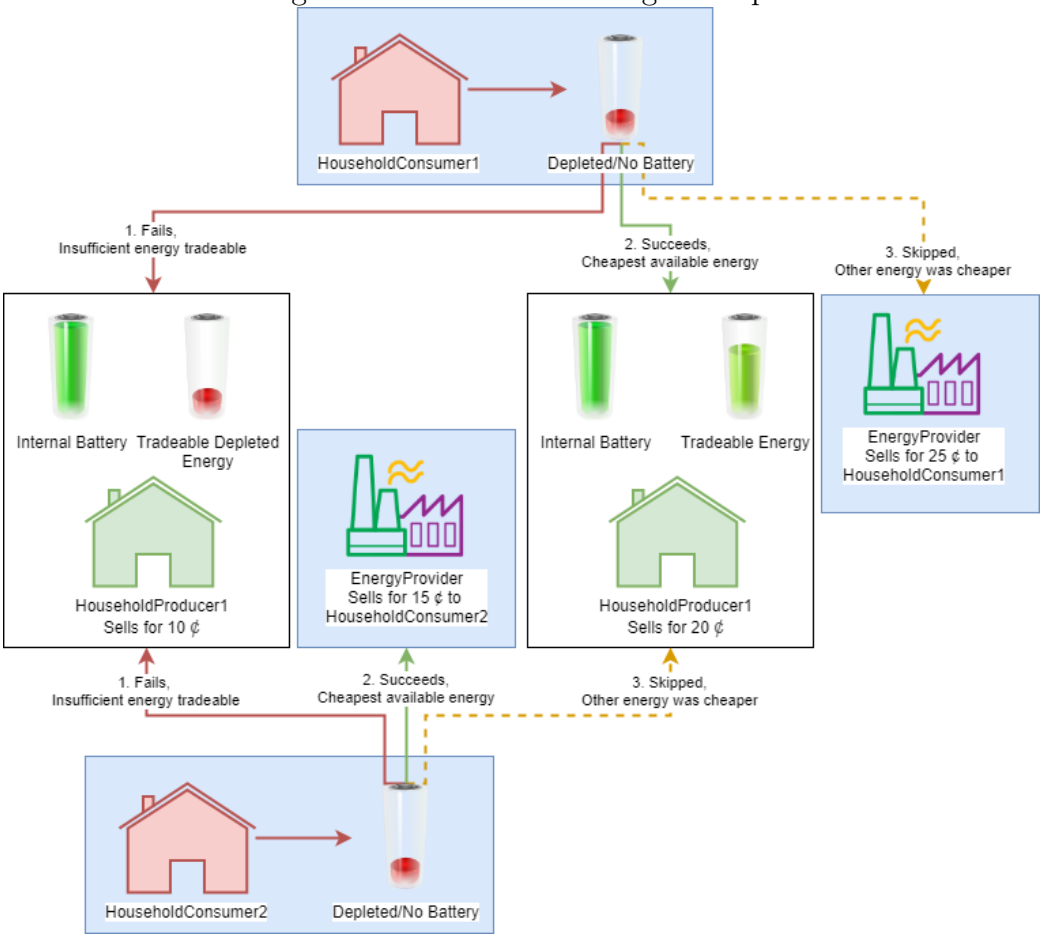
The second point of interest, regarding the impending decline of economic profitability, is tackled by introducing the notion of trading energy with other households, rather than selling power to the energy provider for an incredibly low price. Every household is able to turn trading on or off. This means that the household can decide whether or not to buy power for a lower price than the provider from other households. Secondly, the consumer also decides whether or not his/her power may be sold to other consumers, for a higher price than the energy provider. That way, a system of healthy competition is created, such that it encourages consumers to sell to other consumers for better revenue, while encouraging non-producers to buy energy from consumers as it will be cheaper. For simplicity, this could use the main power net by a small service fee for the provider.

Our implementation works as follow, assuming both *householdProducer* and *householdConsumer* have trading enabled, whereas *householdProducer* generates more energy than it uses. *householdConsumer* uses more energy than it produces. The *householdProducer* will store the excess energy it produces in its own batteries first (if any). If these batteries are full, excess is directly sold to the (low) provider's price. A fraction of this *householdProducer*'s batteries is made available for trading, say 50%, the other fraction is reserved for the household's internal usage, such that not all energy *householdProducer* generates is sold to others. The *householdProducer* can determine its own selling price.

Now, let's look at the *householdConsumer*. Consuming energy will first drain its own batteries (if any). Then, it will look for all available traders who are producing energy, sorted by lowest selling price to highest. The required energy by the *householdConsumer* is purchased from the *householdProducer* for the lowest price available on the market, slowly draining the *householdProducer*'s battery until its previously mentioned threshold (50% or so). If this *householdProducer* has no more energy to deliver, the *householdConsumer* will look for the next cheapest *householdProducer*. If

there are no more delivering *householdProducer*'s, or if their prices are more than the provider, the *householdConsumer* will simply purchase its remaining energy from its own energy provider. Our concept of trading is visualized in Figure 2.2.

Figure 1: Household Trading Concept



A limitation of this approach is that energy may only be purchased from a producer's battery. A producer's excess power (after his battery is full), can not be directly purchased by a consumer this way, and is therefore by default sold to the provider. The reason that this cannot be purchased directly, is that it would require a system similar to an exchange's buy and sell orders. Although this would also be possible, this is likely to be unstable, as power is

volatile and cannot "remain available" for a long period of time in the main power net.

3 Technologies Overview

We briefly list the technologies we used for our project, why we used this technology and for what part of our application we used the technology. There is no strict order imposed on the following list.

- **Java 8**: As this course introduces many new concepts, we decided to stick with a programming language for the web that was familiar to both of us. The reason we chose to use version 8, was that both of us had this version installed on our machines. Java 8 was used for both our front-end and back-end applications.

- **Gradle**: Many technologies and libraries already exist to ease the development of micro-architectural applications. Gradle helps with obtaining the dependencies and building the project. We also use Gradle to dockerize our JARs. Using Maven was also possible, but we found that Gradle has faster build times. Gradle is used for all of our Java applications.

- **Docker**: We use Docker because it helps us with connecting the various application instances over Docker's network, without us having to define any static IP-addresses. Docker is also required for deploying our application in the cloud using Kubernetes. Docker is used for every single application (and database) instance.

- **MongoDB Server**: NoSQL database for document-store, fully replicated over 3 nodes. We use it to store household information, which are only updated once in a while as writes are blocking. Our configuration does not have sharding because (1) it adds to much strain on our development machines, preventing us from running our application; (2) it has little to no added value to a proof-of-concept/prototype, as only small amounts of dummy data is stored; (3) it was not required by the assignment, so we decided it would be best to put more time in the application's business logic instead.

- **Apache Cassandra**: NoSQL database used for non-blocking writes, fully replicated over 3 nodes. We use Cassandra for devices and storing their historical measurements (as time series data). We have a simple replication factor of 3

- **Stomp Websockets**: We use WebSockets to stream information directly to a topic having subscribed users. The advantage of WebSockets over REST for updating graphs, is that the connection is stateful and remains open, reducing the overhead of REST calls. We use WebSockets to stream household statistics to the frontend, which are then used for updating the household graphs.

- **Project Reactor: Core**: Java Framework for fully non-blocking applications, communicating directly to the Java Functional API, and implements Futures, Streams and Schedulers under the hood. Mono is used to represent a future of a single object, whereas Flux is used for 0 to N objects. We use Project Reactor for reactive non-blocking communication with the databases.

- **RabbitMQ**: A message queue for buffering device-usage-measurements, before they are processed. TopicExchange is used for storing/processing these measurements once (i.e. compute costs and save in database), whereas FanoutExchange is used for broadcasting household activity to all listeners, which are then send to the websockets.

- **Spring Framework and Spring Boot**: The Spring Framework is an extensive Java framework for which many additional libraries exist, such as service discovery etc. Because the Spring Framework is very extensive, an opiniated version of this framework exists called Spring Boot. Essentially, all configurations have defaults. It follows the principle that 90% of the configuration can be pre-configured and doesn't need to be changed, whereas only 10% has to be filled out (overriden) by the developer. For every single application, we use the Spring Framework and Spring Boot.

- **Spring Cloud Netflix: Eureka (server+client)**: Service Registry and Service Discovery. Every 5 seconds a heartbeat is sent. After 10 seconds without heartbeat, an instance is marked down. Every 1 second, all down instances are deregistered. Used for all our Java Applications.

- **Spring Cloud Netflix: Ribbon**: Used to balance the load between the available service instances on the client-side. Client-side load balancing means that the load balancing is performed by the application itself that wants to make a request to some service, rather than having a separate load balancer in between. We use ribbon for our gateway.
- **Spring Cloud Gateway**: Entry point of our application that routes all incoming requests to the correct service. We use it for incoming requests to the frontend, as well as the API of the backend.
- **Spring Cloud Config**: Decentralized configuration store. Currently used for the gateway routing. The configurations could be moved to a Git repository instead of the folder as well.
- **Spring Data for Apache Cassandra (reactive)**: Allowing us to easily configure and use communication with cassandra in a reactive fashion, using POJOs and Annotations, such that cassandra-integration is done in a clearly structured object-oriented fashion with a less steep learning curve. We use Spring Data for Apache Cassandra for all our communication between the backend and the database.
- **Spring Data MongoDB (reactive)**: Same as above, but then for MongoDB.
- **Spring AMQP**: Same as above, but then for RabbitMQ.

4 Software Architecture

The components of our project can be seen in figure 2, the following list shortly explains what each component does:

- **Service discovery**: Every Java component (i.e. all of them, except for the databases and RabbitMQ) registers itself with the service discovery. This makes it possible for other components to find it.
- **Config server**: Contains the configuration files of the different components and serves them on their request. Is currently only used for the gateway configuration.

- **Gateway**: Communication between the components and with the client are set up through the gateway. The gateway uses the service discovery to find the components and gets its configuration via the config server.

- **Front-end**: The front-end is a simple Spring Boot server which serves our SPA statically. This gives this component very little to do, but it is the starting point for every client. The SPA is built with Vue, Vuex, Bootstrap-Vue, Vue-router, Axios, Stompjs, and ApexCharts as the primary packages.

- **RabbitMQ**: Message queue that is used as an in-between of the energy usage/production sensor data between the 'sensors' and the components handling these data points. Acts as a buffer and a central exchange for these.

- **Device Activity**: Simple component that, based on the current devices, generates a semi-random energy usage and production for each household.

- **Power Infrastructure Service**: The main back-end component, it is the only component with a direct connection with the database. API-calls are handled by this component. It also listens to the RabbitMQ queues with sensor data points and acts upon these.

  The front-end can make a websocket connection to this component, requesting updates of the energy usage/production for a household. This component will remember which session requested which household and sent updates upon receiving new data points via the message queues.

  Trading occurs within this component as well.

- **MongoDB**: The MongoDB databases hold the household information, including the trading data.

- **Cassandra**: The Cassandra databases hold the devices and the energy usage/production historical data.

A typical use case would be requesting the website for the first time. This request is made to the gateway, which uses the configuration acquired from the config server to see which service should handle the request. In this case,

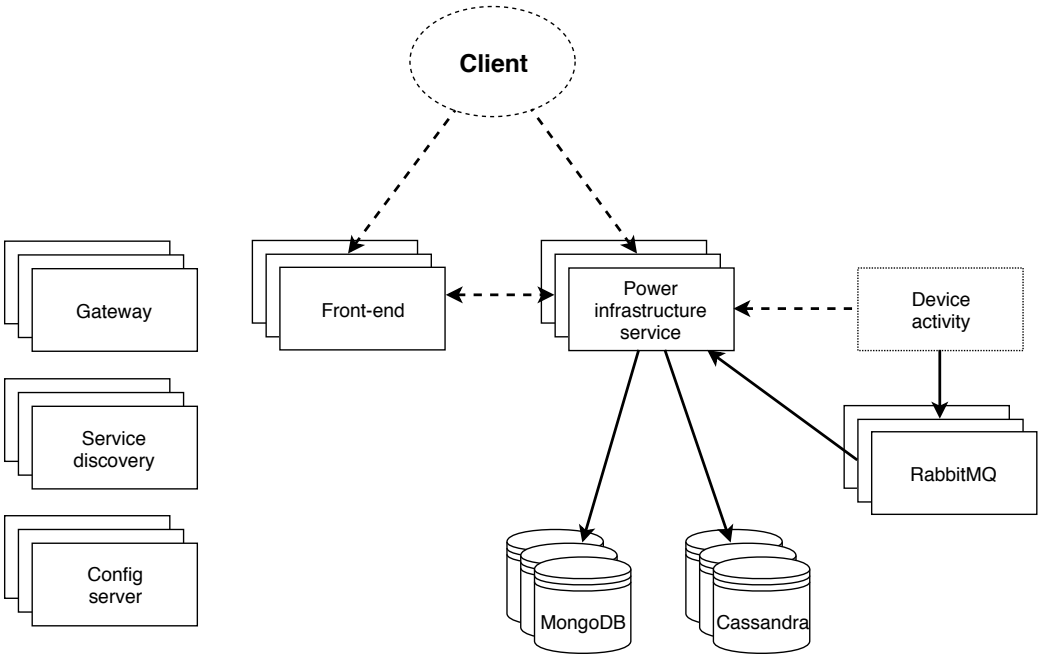


Figure 2: Components and their connections. Direct connections are shown via a non-dashed arrow, connections made through the gateway are indicated by a dashed arrow. All the components are replicated, except for the Device Activity. This component is strictly simulating the energy usage and production of the devices.

the front-end service should handle this request. The gateway then gets the instances of the front-end service from the service discovery, picks one, and routes the request to this instance. This response is then send back to the client.

5 Fault tolerance

In real life it is inevitable that some instances of the application crash every once in a while. Error handling and recovery are thus important parts of the application. We will take a look at error handling and recovery based on the failure of several levels of our application. Every module is replicated thrice (except for the Device Activity).

- **Database failure**: Failure of up to two nodes has no effect on read-

ing the data and will not be noticed by the user (unless the load is high). Full failure of a database cluster will result in error messages in the front-end for API-requests and websocket reconnects. The Power Infrastructure Service will try to reconnect to the database using a constant interval of 5 seconds. When (part of) the cluster comes online again, the Power Infrastructure Service will reconnect to the database and will be able to handle requests again.

The constant 5 second interval is chosen for testing purposes, an exponentially increasing interval would be a better choice for production-ready software.

- **Power Infrastructure Service failure**: If a node fails or network connection to a node is lost, the websocket connection with the SPA will be lost. When this happens the SPA will try to recreate this connection (with potentially another server) with a 5 seconds interval. The SPA will keep trying until a new connection has been made. When a new connection has been made, it will also subscribe to the previously subscribed queues.

  API calls will also fail when all the Power Infrastructure Services are down, resulting in error messages in the front-end. There is no automatic reloading, the user should reload the page themselves.

  The constant 5 second interval is chosen for testing purposes, an exponentially increasing interval would be a better choice for production-ready software.

- **Front-end failure**: This has no effect on the users that already have the SPA, they should not notice this at all due to browser cache. For new users this would simply result in a 'Service Unavailable' until an instance of the front-end service comes online again.

6 Requirements Check-list

In this section, we provide an overview of all the project requirements, to which we indicate whether these requirements were met, among with a small elaboration.

6.1 Minimal Project Requirements

1. [✓] **Docker**: All our application components run in Docker.
2. [✓] **Docker Life Cycle**: We can demonstrate how to create/push images, start/stop containers.
3. [✓] **SPA**: The front-end of our application is an SPA, including CSS, AJAX, events, promises using Axios, modularity and tests.
4. [✓] **Basic Fault-tolerance**: If a component is down, all other components are still functioning (to a certain extent).
5. [✓] **Database**: We have at least 1 NoSQL database (2): Cassandra and MongoDB.

6.2 Project Requirements ranging 8 to 9

1. [✓] **Admin Dashboard**: For local development, we use the dashboard from Netflix Eureka, showing the state of the current Java containers. This also shows the last 1000 registered and deregistered components. For our production environment, we can use Eureka for our Java containers, as well as the Kubernetes dashboard showing the current pods, their state and their history.
2. [✓] **Fault Tolerance**: This is satisfied, and was explained in the prev. section.
3. [✓] **Asynchronisity**: All database communication (IO) is performed using reactive libraries, to ensure that IO is non-blocking.
4. [✓] **Databases**: 2 DBs are used: MongoDB, Cassandra. Both are fault-tolerant.
5. [✓] **Orchestrator Life Cycle**: We have a basic understanding how to work with Kubernetes, and deploy our application on it.

6.3 Deals of the Year

1. [✓] **No hard-coded IP-addresses (+0.25)** Not a single IP-address is hard-coded.

2. [✓] **Using ZooKeeper/Etcd/Consul (+0.25)**: We use Eureka. By e-mail correspondence with A. Lazovik, it was decided that Eureka can also be used for this purpose (works similarly as Consul).

7 Conclusion

Through help of the Java Spring Framework we have been able to conform to most of the requirements, even though the application is far from done. The usage of Spring meant that a large part of the configuration and bootstrapping was done for us, so that we could focus on the actual application. Deploying in Kubernetes meant a lot of added configuration, though the UI of the Google Cloud Platform made this a lot easier.

Our decisions weren't always as thought out as we would have wanted, partially due to being uninformed and partially due to time constraints. Examples of these decisions are our database choices and architecture. For further projects we would think more about what database to use when and how. We would also keep our services smaller (e.g. split up the Power Infrastructure Service).

All in all this project is an interesting learning experience, one which we both enjoyed.