The Virtual Energy Market

An Approach to Intelligent Energy Management Systems

Max Obermeier

OpenEMS | FENECON GmbH

Abstract

This document summarizes my work on a virtual energy market for [3]. I am not only going to describe my implementation of a simple market system and the problems and issues I ran into while developing it, but I am also going to write about how I solved them, or how they could be solved. From there on, I am going to document my rather theoretical thoughts on how such a system can work and which problems it could be able to solve.

CONTENTS

LISTINGS

1 Introduction

Today energy management systems have to meet the challenge of going beyond realtime calculation. With more and more semi-intelligent consumers being introduced into households the owner's expectations for their energy management systems rise. They are expected to minimize mains power and maximize autarky by adapting consumption so that it fits pv-production. This requires the energy management system or the intelligent consumers to know about the other consumers' state and priority. However, this would e.g. require a charging station for electric vehicles to understand, what a temperature of 20°C means to a heater, or vice versa, which would create way too much effort for developers.
This problem can be solved using a more generic interface: money. The only thing a consumer must know is, when a specific amount of energy will cost how much. This can be derived from a virtual energy market running on the energy management system. This market enables realtime balancing at the mains connections as well as long-term planning for consumers like electric vehicles.
Nevertheless, there are a couple of problems:

obligation to participate: Every component - consumers and producers - *must* be represented at the market. Otherwise, the power, which isn't represented at the market, is exerted on the mains connection and that decreases autarky. Of course devices can be grouped, but each group *must* provide a prediction on how much energy it will need or provide. Not only for the next second, but also for the next three hours, the next day, or - theoretically - the next week. This is easy, when it comes to intelligent or scheduled systems like a heater, but almost impossible for pv-production, or - even worse - non-regulated-consumption.
wrong predictions: Also, every component *should* keep the promises it made at the market. A wrong prediction in the market's realtime section equals the one-second input lag OpenEMS has today, because many devices' predictions for the near future will be based on measurements of their current consumption/production. However, this point becomes more disastrous, when big consumers delay their actions due to good pv-production forecasts and then are forced to buy all the energy from grid if the weather becomes worse. Of course this is a general problem when using forecast-data, but every wrong prediction reduces autarky and therefore costs money.
obligation to cover the whole period of time: Every component *must* provide data for the whole time, that is observed by the market system. Otherwise the other consumers would delay their actions, if an other consumer only announces load for a short time in advance, because then the energy-price would always be cheaper in the more distant future.
power limitations: Physical restrictions on the devices concerning minimum and maximum power don't allow fluid transitions, but cause big leaps in the market's power- and price-values.
monopolies: Bigger consumers like charging stations for electric vehicles have a huge impact on the virtual market's energy-price. Therefore they *must* be able to estimate the consequences of their announcements in advance, so that switching between two states is avoided. E.g. electric vehicles can only be charged with a specific minimum current. So, if the charging station is only willing to pay 15ct per kWh, but the virtual market's energy-price exceeds this price as soon as the charging station is activated, the charging station is going to switch between announcing no power and minimum power every second. Such a behavior would disable all other devices to use the market-system at this point in time.
mutual exertion of influence: A similar effect could also be caused by two different devices having similar criteria for when to consume power. If, for example two devices need 400W each, but don't want to pay more than 15ct per kWh and the energy storage system can only offer another 500W, then both devices are going to announce 400W of additional load and the price rises to 28ct (grid's price). In the next negotiation-cycle both devices announce 0W of load and afterwards they go up to 400W again.
performance: The market-implementation on the energy management system *must* be performant and versatile enough to calculate exact values for the near future, while also providing long-term forecasts. This becomes especially challenging, when you consider the three latter problems. They require many negotiation-cycles to be done until the final state everyone can agree on is found.
integration of restrictions: It the market system is integrated very deeply, it may become difficult to implement restrictions or modifications like a balancing offset in a somehow

idiomatic manner.

2 THE MARKET I CREATED

My market implementation can be found at [2]. This implementation was meant to be a basis for experiments with different algorithms and techniques and is *not* integrated into the energy management system's actual logic. Although the higher market-api isn't really usable at all, some parts of its basis were properly implemented and tested, so that they can be used in future projects. Those parts are going to be named and described later in this document.

general structure: The `LinearMarket` does split the next 24 hours into periods of fifteen minutes each. It does not dynamically adapt the periods' durations to the time being left until they reach realtime. During each period all values are constant. My `MarketSquare` interface provides methods that allow `MarketAgent`s to register themselves and add `Diagram`s of two types: `LoadDiagram`, which represents demand and `PriceDiagram`, which represents supply. While the former only has one value for power, the latter can also store a relative price. The `LinearMarket` calculates its so called market-state, which is also stored as a `PriceDiagram`. The power-value depicts the total turnover, the price-value firstly showed the average price, but this was changed to the maximum price later on. The meaning of "average" and "maximum" will be clarified soon.

data management: The whole market is integrated into OpenEMS' Cycle-System. The `MarketAgent`s are called at the `AFTER PROCESS IMAGE` event, the `LinearMarket`'s processing method at the `BEFORE CONTROLLERS` event. The data is synchronized between market-square and -agents using the pointer nature of every Java object. Thus, the whole system is ready for asynchronous writes, also, because the underlying diagram implementation's read- and write- methods are synchronized and because `LinearMarket` creates a deep-copy of all input diagrams before analyseing them. Furthermore, `MarketAgent`s are able to request a consistent deep-copy of the market's state, so that they don't have to handle this task themselves. One mistake I made was storing all the input-diagrams as local variables inside the `MarketAgent`s. Some agents like the `HistoryBasedAgent`s don't change their power-announcements based on the market-state, but only based on the measurements of their represented meter. The `HistoryBasedAgent`s e.g. only copy those measurements 24 hours ahead. So, if the `MarketAgent` is destroyed and recreated by the OSGi framework due to an error in some of the agent's dependencies all this data is lost. For that reason the market's input and output diagrams should be outsourced to an extra component, which does not rely on the `MarketAgent`s or a `MarketSquare`.

market algorithm: In `LinearMarket` the market algorithm is located in the `calculateMarketState` method. A shortened version of this method can be found in listing 1. After erasing the outdated data in the input it gets from the `MarketAgent`s and the output, the whole input-data is copied in order to avoid concurrency problems. Afterwards a new `LoadDiagram` is created, which contains the sum of all fifteen-minute-averages taken from all demand-diagrams over the next 24 hours. Then the method enters a for-loop over the next 24 hours - again in fifteen-minute steps. This is where listing 1 begins. Inside this loop, firstly, the averages over the current quarter of an hour are collected from the input diagrams and stored in a list. Afterwards an other source of supply is added to this list. This supply-source is a `ValueDecimal` with infinitely high price- and power-values. This supply basically represents an overload at the mains connection. The now completed supply-list is then ordered by ascending price. Finally this supply-list is matched with the total demand calculated in the beginning by iterating over the supply-list - from lowest to highest price - and applying the following algorithm:

- subtract the current supply's power-value from the demand
- *if* the demand is smaller $0W$, *then* set the market's output at the current time-period to a new `ValuePrice` consisting of the current supply's price-value and the total demand's power-value

The if condition in the second point will always be true at some point, because we added the infinite supply-source to the supply-list. This also means, that taking supply is *optional*, while satisfying demand is *mandatory*. This algorithm works, assuming that there always is a grid buying for a higher price, than e.g. the pv-power costs. The version explained above calculates the maximum price. As mentioned

```
// from now for 24h in 15min steps
for (long l = now; l < now + (long) (86400000 * speedFactor); l += (long) (900000 * speedFactor)
    ) {
        final long counter = l; // copy l to final variable because ,err, java !

        // sum up supply
        List<ValuePrice> supplyList = new ArrayList<ValuePrice>();
        inputCopy.forEach((agentID, diagramMap) -> {
                diagramMap.forEach((diagramID, d) -> {
                        if (d.getClass().equals(PriceDiagram.class)) {
                                PriceDiagram pd = (PriceDiagram) d;
                                supplyList.add(pd.getAvg(counter, (long) (900000 * speedFactor))
                                    );
                        }
                });
        });
        // in case mains connection is too small we add a very expensive, but infinite
        // power-source, so that the market-system never fails
        supplyList.add(new ValuePrice(Integer.MAX_VALUE, Integer.MAX_VALUE / 2));
        // sort supply sources by price
        Collections.sort(supplyList, (a, b) -> a.getPriceDouble() < b.getPriceDouble() ? -1 : a.
            getPriceDouble() == b.getPriceDouble() ? 0 : 1);
        // calculate new market state
        ValueDecimal totalDemand = demand.getAvg(counter, (long) (900000 * speedFactor));
        for (ValuePrice s : supplyList) {
                demand.setValue(counter, (long) (900000 * speedFactor), demand.getAvg(counter, (
                    long) (900000 * speedFactor)).subtract(new ValueDecimal(s.getPowerDouble()))
                    );
                // must be true at some point (at least with the virtual supply selling for
                // Double.Max_Value as price)
                if (demand.getAvg(counter, (long) (900000 * speedFactor)).getDecimalDouble() <=
                    0) {
                        output.setValue(counter, (long) (900000 * speedFactor),new ValuePrice(s.
                            getPriceDouble(), totalDemand.getDecimalDouble()));
                        break;
                }
        }
}
```

Listing 1: LinearMarket's Algorithm

before, an earlier version calculated the average price. This was done by adding up the current supply's power multiplied by the supply's price while iterating over the supply-list and dividing the resulting value by the total demand inside the if condition.

market reactivity: When implementing `MarketAgents` one question rises pretty quickly: By how much can I change my power-announcement, without making the price rise too much ? This is caused by the fact, that only very few agent's trade at the market. Therefore, a single agent can have a major impact on the market's state. This problem could be solved by altering the power-announcement just slightly in one negotiation-cycle, but this costs time and sometimes the market is required to react quickly, when e.g. the pv-agent changes its prognosis for the near future, because a cloud just started to cover the sun. Thus, I introduced a value called market reactivity, which basically describes how much power it takes to increase the price by 1€ at the given time. This value is provided by the `MarketSquare`. In the `LinearMarket` it was

calculated by storing the market in its state before the latest negotiation in a `history` variable and then calculating the power- and price-difference between the current market-state and the one stored in `history`. Then those differences were divided by each other. However, this value was very inconsistent, since price and power-changes were pretty small sometimes, which resulted in huge values for the market's reactivity. Also, often the price didn't change at all, especially, when I switched to the maximum price. This, of course, would have resulted in an exception, if I hadn't defined a constant to return instead. I altered the market reactivity's definition over time and tried to find a more consistent value, but in conclusion: I ran out of time. Still, a couple of suggestions are included in the next section.

agents: Next to the `HistoryBasedAgents` mentioned earlier, I tried to develop some other, more intelligent agents. The `GridAgent` was the only one I managed to experiment with before I ran out of time. It basically is a `PriceBasedAgent`, but handles a supply- and demand-source. Supply is always set to maximum power, which makes sense, since supply is only an offer in the `LinearMarket`'s system. Demand, in contrast, is adapted to the current price. It simply rises when the energy-price is low enough and decreases, if it is too high. This system heavily depends on the market reactivity, which is set to a constant until now. Therefore the `GridMeter`'s reactions are pretty slow to date. If the problem concerning the market reactivity can be fixed, `PriceBasedAgents` can be quite effective. However, such `PriceBasedAgents` have one problem: Many devices don't have a concrete revenue which they can base their decisions on. A charging station for electric vehicles for example doesn't know, how much it costs its owner, if he can't use his car at the time he wanted it to be fully charged. Instead the car could know, to which percentage it has to be loaded at a specific time. The missing energy can then be consumed, when the price is lowest in the given period. This approach is implemented in the `RequirementBasedAgent`.

3 Designing a Market

A market system can be designed in very different manners. This approach contains the learnings from my first attempt.

prediction character: Since most normal electric devices are not buffered via a battery, their demand/supply must be provided/taken in realtime. However, OpenEMS is limited to a delay of one second. Therefore we can ether accept this delay and its drawbacks, or we try to rely totally on predictions. The latter option provides more flexibility and in the worst case it just equals the one-second-delay. A market agent must provide a forecast on the load caused by the represented device. For the far future, this forecast doesn't need to be extremely accurate at all, only accurate enough, so that sluggish big loads can be scheduled. But when it comes to the near future, especially the next second, the forecast must be as accurate as possible, so that no power is exerted on the mains connection when it is not necessary. On the one hand this is an easy task, if the agent represents a charging station, because it knows, when it is going to be activated and deactivated. On the other hand it can be impossible, if the agent represents pv-production or non regulated consumers, but in this case the agent can use the realtime power-measurement taken from its client-device and assume the value won't change in the next second. This behavior equals the OpenEMS' current one-second-delay. Hence it is important to keep in mind, that in the virtual market system - at least formally - everything is a prediction.

units: Due to the devices' different needs regarding forecast-range, the market is going to divide up the total forecast-range into periods of very different duration. Starting with a duration of $1s$ for the next seconds and ending with multiple hours in the far future. Thus it is advantageous to use values, which do not depend on time, so that unnecessary calculations can be avoided. Therefore using power-values has a major advantage over using energy-values. Also, prices are stored as $\frac{€}{kWh}$ instead of assigning an absolute price to an energy-bundle of a specific size.

role of the grid: The role of the mains connections is a pretty central design question. If the measurements of all devices in a household and the measurements from the mains connection are summed up the resulting power always equals $0W$. Therefore, if the grid was also a trading component of the market, the market's power balance would always have to be zero. The grid's agent could try to even out the market's balance as soon as its off, but this would create lag. Ideally, this would even

be regulated by the resulting price, but since power and price are not directly proportional to each other, every component has to slowly alter its power-announcements until the price it aimed for is reached. This results in another huge lag. Hence it's better to firstly calculate the household's balance, then calculate the resulting power at the mains connection and finally calculate the resulting price using the grid's and the single other producers' price- and power-values. Afterwards those values can be broadcasted.

role of the ess: One of the energy storage system's tasks usually is to optimize self-consumption by keeping the power exerted on the mains connection at $0W$. Therefore treating the energy storage system's agent similar to the one of the grid seems beneficial, but this approach has a major disadvantage: loss of flexibility. It may be beneficial for maximizing autarky to have a power-offset at the mains connection which could be varied based on the virtual market's energy-price and/or the ESS' state of charge. Such logic should be strictly separated from the market's algorithms, just to keep the code clean and to ensure modularity, although this creates a reaction-lag of $1s$ when it comes to self-consumption-optimization. Modularity is important for example, because basically any controllable device can try to improve self-consumption. However, there should be one device, which is ultimately responsible for the last fine-tuning in matter of self-consumption optimization, which - in most cases - will be the ESS. This device should get the advantage of being allowed to react to the new market-state after the last negotiation before realtime. This is shown in pseudo-code in listing 2. With that advantage over other devices, the ESS' reaction-lag of $1s$ mentioned above is eliminated again.

which values to share: To calculate the virtual energy market's state not much information is required. For one period every agent representing a consumer only needs to declare, with which power his device is going to consume. Agents representing producers also have to provide a price or production cost.

In regards to which information the market has to broadcast, there are a couple of values to be considered:

- The household's power-balance or the power exerted on the mains connection. This information is crucial for the ESS, when it tries to improve self-consumption. Calculating the required power using only the price would require way too many negotiation-cycles, especially since the power-values have to be very accurate for this task.
- The energy-price of the most expensive producer, that is currently producing. This information is crucial for consumers, that are limited to a specific price.
- The average energy-price of the whole production (weighted by powers). This information is crucial for consumers, that need to consume a specific amount of energy in given period of time, for the lowest possible price. Also producers, which rely on some sort of storage can use this value, so that they produce energy, when the overall price is most expensive.
- The market reactivity, or some value, that enables agents to estimate approximately how strong the market's reaction will be, when they alter their power-announcement. Finding the perfect value is left for future study. However, the following values could be of interest:
  - total turnover of power
  - change in power divided by change in price compared to the last negotiation-cycle
  - price of the next higher or lower supply-source in combination with the power, that is needed, so that this source is used or so that the current source isn't used anymore
  - live-calculation of required power: If the right values are stored as local variables in the `MarketSquare` implementation the power required to raise the average price to a given aim can be calculated pretty effortlessly. Those are the required formula, if poC is the current turnover of power, prC is the current price, prN is the price of the next more expensive supply-source, poN is the total power, the next more expensive supply-source can provide and prA is the price, the

```
// normal device
public void callAtAFTER_PROCESS_IMAGE() {
    // iterate over forecast-range
    for (MarketState ms : MarketStateTimeline) {
        // calculate reaction
        myReactionTimeline.setAt(ms.getTimeStamp(), calculateMyReactionToMarketState(ms));
        sendToMarket(myReactionTimeline.getAt(ms.getTimeStamp()));
    }
    waitForMarketToFinishCalculations();
    sendToDevice(myReactionTimeline.getAt(ONE_SECOND_BEFORE_REALTIME));
}

// ess
public void callAtAFTER_PROCESS_IMAGE() {
    // iterate over forecast-range
    for (MarketState ms : MarketStateTimeline) {
        // calculate reaction
        myReactionTimeline.setAt(ms.getTimeStamp(), calculateMyReactionToMarketState(ms));
        sendToMarket(myReactionTimeline.getAt(ms.getTimeStamp()));
    }
    waitForMarketToFinishCalculations();
    // calculate reaction
    myReactionTimeline.setAt(ONE_SECOND_BEFORE_REALTIME,calculateMyReactionToMarketState(
        MarketStateTimeline.getAt(ONE_SECOND_BEFORE_REALTIME)));
    sendToDevice(myReactionTimeline.getAt(ONE_SECOND_BEFORE_REALTIME));
}
```

Listing 2: Advantage of ESS

agent aims for:

$$x = \frac{poC \cdot (prA - prC)}{poN \cdot (prN - prA)}$$

If $x \notin [0;1]$, then the input-value prA is illegal. Otherwise the power, the agent aims for poA and the resulting total turnover of power $poAT$ are defined by the following equations:

$$poA = x \cdot poN$$
$$poAT = poC + poA$$

4 Diagram Implementation

Diagrams are the basis of communication in a market system. [1] contains a Java definition and implementation. It allows to store any Object implementing the package's `Value` interface in relation to unix-time. The package is tested and is not of experimental nature. The interfaces defining the `Diagram`'s behavior are located inside the package's `api` folder. Please read the in-code documentation fore more information on functionality.

The interfaces are implemented in the `Load-` and `PriceDiagram` classes using the generic `universal.ArrayListDiagram` implementation of the `Diagram` interface and its dependencies. The `ArrayListDiagram` allows to store any type of `Value` implementation, if an implementation of the package's `ValueFactory` producing the same type of `Value` can be provided. The `ArrayListDiagram` is capable of addressing single milliseconds, while keeping read- and write-speeds high when dealing with long durations. It stores a start- and an end-node for every value at the according points in time. Those nodes are stored in an `ArrayList` using binary-search algorithms for insertion, deletion and manipulation. This ensures high performance, while keeping the interface simple and versatile.

REFERENCES

[1] Fenecon GmbH. *Diagrams*. URL: https://github.com/OpenEMS/openems/tree/develop/market/src/market/diagram (visited on 08/16/2018).
[2] Fenecon GmbH. *Market*. URL: https://github.com/OpenEMS/openems/tree/develop/market (visited on 08/13/2018).
[3] Fenecon GmbH. *Open Energy Management System*. URL: https://github.com/OpenEMS/openems (visited on 08/13/2018).