# Virtual Power Plant design document

Ubbe Welling
ubbe@eng.au.dk

October 22, 2015

# Contents

# Chapter 1

# Considerations and requirements

## 1.1 Purpose

The purpose of the Virtual Power Plant (VPP) platform is to:

- receive measurements from sensors deployed in a building.
- receive current and forecast information from external sources on grid load, electricity price and more.
- process measurements and external information to make decisions on power usage.
- actuate devices deployed in a building to implement above mentioned decisions.

The above functionality is already implemented in a number of individual applications and scripts. This system should provide the same functionality in an integrated application.

## 1.2 Users and privileges

### 1.2.1 User categories

An initial listing of envisioned users and their access to the system is shown below:

**Residents**

- Actuate in own home
- See data from own home

**Janitor**

- Should be able to do anything he/she can already do before the system
- Actuate anything not in private
- Add/remove actuators

**Administrative staff (ie. housing association office staff)**

- See data on some level of aggregation. Maybe just reports?

**Aggregator**

- multiple buildings
- specific read/actuate permissions, granted by janitor/admin. staff

**System administrator**

- full access

### 1.2.2 Privileges requirements

With outset in the above user categories and tasks, the following distinct privileges have been identified:

**Global system privileges**

- Access to reports
- User access
- System admin access

**Building-specific privileges**

- Add new devices
- Remove devices

**Device-specific privileges**

- Read device status and data (measurements/actions)
- Configure device (configure parameters, disable)
- Remove device and data
- Actuate controller

# Chapter 2

# Design

The platform will consist of one main server application with an attached database.

It is intended to be deployed on a Beckhoff CX2030 PC. This machine has a dual-core Intel Core i7 1.5GHz CPU and up to 4GB RAM. The disk is a flash card. The machine is intended to reside in the same building as the devices it should interact with.

## 2.1 Database design

The database will reside in a PostgreSQL DBMS.

### 2.1.1 Schema: core

The central part of the database schema is shown in figure 2.1 and explained below:

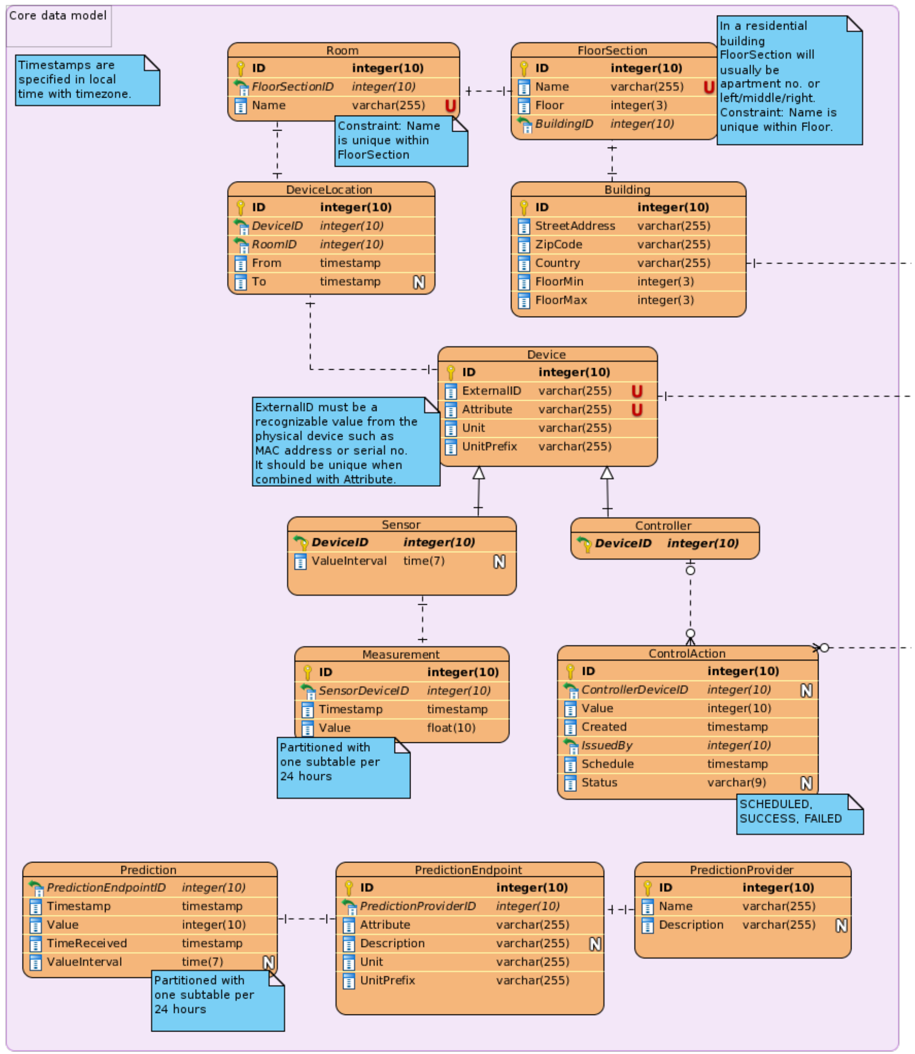


Figure 2.1: Database core schema

### Devices and measurements

**Device** Entries in `Device` correspond to physical control and sensor devices, with the modification that we store one logical device for each function of the physical device. Whether a device is a sensor or a controller is specified by its presence in table `Controller` or `Sensor`. Columns `Attribute`, `Unit` and `UnitPrefix` (such as milli) are for sensors specification of the incoming measurement values, while they for controllers specify the format of values to send to the controller when actuating it.

**Sensor** Table `Sensor` specifies an optional property `ValueInterval` which is used when measurement values are aggregated over a limited time interval (such as 15 minutes).

**Measurement** This table will contain a row for each measurement received from a physical sensor. It will simply consist of a `Value`, a `Timestamp` and a reference into `Sensor`, which enables interpretation of the value. The `Measurement` table is expected to grow very large and will therefore be partitioned into subtables that will each contain 24 hours of measurements and can be discarded on the fly according to the rolling window strategy explained in section 2.1.4.

**ControlAction** Table `ControlAction` will contain scheduled and past commands for controllers. An action is simply specified by a `Value` which can be interpreted via the reference into table `Controller` and `Device`. Column `Schedule` specifies the time for carrying out the action, and `Status` indicates if execution is still pending or has been completed. Finally, `IssuedBy` specifies which user scheduled the action. We might consider partitioning and discarding of old data in this table in the same way as for table `Measurement`.

**Building, FloorSection, Room** The physical properties of a building are modeled in these tables. A building consists of an integer range of floors. Each floor consists of `FloorSection`s which in most cases will be equivalent to apartments. The generalized term FloorSection is intended to support other types of buildings where designations such as "South wing" or other may be desired. Finally, a floor consists of named `Room`s. We do not expect to obtain device locations with a higher degree of accuracy than individual rooms.

**DeviceLocation** This table maps `Device`s to `Room`s for specified time periods, indicating that devices may be moved around.

### Predictions

Predictions of a wide range of values (power consumption, grid load, price, $CO_2$ emissions, ...) will be received from external data providers and will in addition be generated by our own application logic.

While the data stored for predictions is quite similar to those for measurements, we have chosen to store them separately because of the fundamentally different semantics.

**PredictionProvider** An entity providing predictions, such as energinet.dk or the system itself.

**PredictionEndpoint** A logical source of predictions of one type. Specifies the `Attribute`, `Unit` and `UnitPrefix` of the incoming values and provides an optional `Description`.

**Prediction** Actual prediction values. `Timestamp` indicates the time for which the value applies, while `TimeReceived` indicates when the prediction was received from the provider. This is relevant since multiple predictions for the same future point in time may be received over time. Some values actually cover an interval (for instance predicted power consumption for a given day of 24 hours), which is specified in column `ValueInterval`. This table is also expected to grow quickly, motivating the same partitioning and rolling window strategy as for table `Measurement`.

### 2.1.2 Schema: Users, groups and privileges

The database schema for storing users and privileges has been developed to meet the requirements from section 1.2.2. This is shown below:

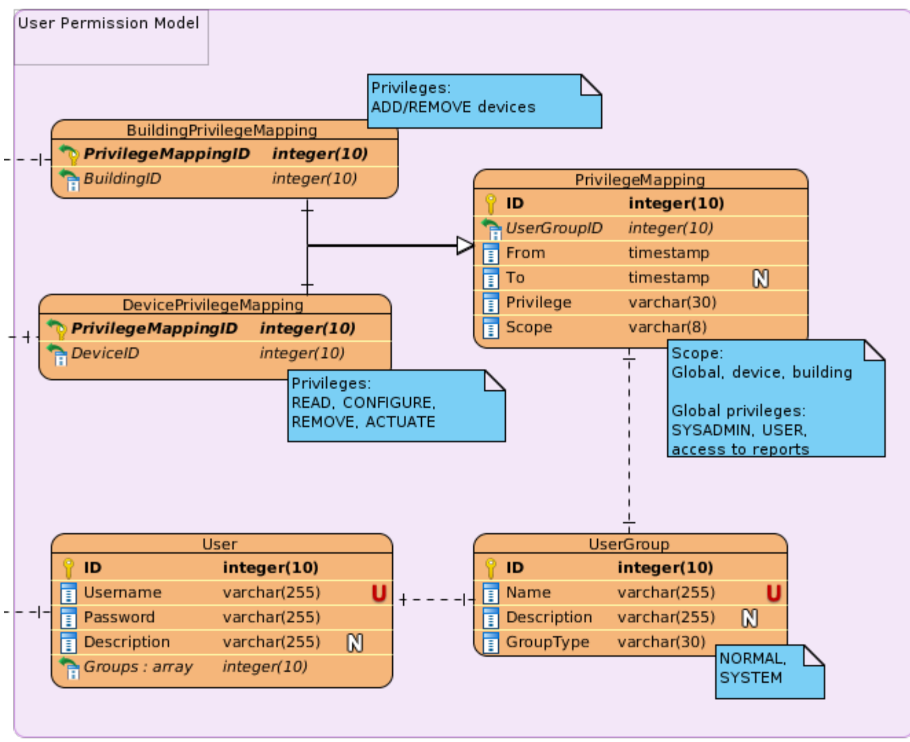


Figure 2.2: Users and privilege schema

**Users and UserGroups** `Users` have a username and a password (which will be hashed with a suitable one-way hash function) and can be member of a number of `UserGroups`.

**PrivilegeMapping** This table maps `UserGroups` to specific privileges. A mapping includes a time period which may be open-ended. In order to have privileges for specific devices and buildings as well as globally valid ones, a `Scope` must be set. In case of a device or building-specific privilege, the concerned `Device` or `Building` must be looked up in `DevicePrivilegeMapping` or `BuildingPrivilegeMapping` respectively. These tables are intended to have corresponding subtypes in the object-oriented implementation.

It will be up to application logic to interpret the semantics of the concrete privileges.

### 2.1.3 Subject to change

We can already foresee that configuration of controllers as well as other settings that must be persisted will most likely require extensions to the schema proposed above. The basic structure of the core schema should however not change.

### 2.1.4 Rolling window

Since the `Measurement` table will grow very quickly, a partitioning and data discarding scheme will be employed. The table will be partitioned in time intervald, having one subtable for every 24 hours. Furthermore, subtables older than one week will be dropped. The time limits can naturally be configured. The same scheme might be applied to tables `ControlAction` and `Prediction`. This is done in order to accommodate the VPP server on a desktop size machine with limited disk space.

### 2.1.5 Data warehouse

In order to retain data, the VPP will periodically forward data to an external database (data warehouse) that can accommodate a larger volume of data for longer periods. When forwarding data, measurements may be averaged over limited time intervals to reduce data size. The data warehouse can then be used for statistics and historical analysis. While the data warehouse schema was initially planned to be identical to the VPP rolling window DB, the presence of users, privileges and most likely various other configuration indicates that probably only the core schema as shown in figure 2.1 should be present in the data warehouse.

### 2.1.6 CIM compliance

The Common Information Model (CIM) is being taken into consideration in the design. Where it is applicable, we will aim to make our data model compliant. Units, unit prefixes, timestamps and time durations will be formatted to agree with CIM. On the other hand, CIM does not provide any guidance on how to structure for instance the building/floor/room model.

## 2.2 Application design

The application will be programmed in object-oriented Python, using Python processes to enable concurrent processing.

Using processes instead of threads is necessary to utilize both cores in the intended machine, since Python employs a *Global Interpreter Lock* which prevents threads within the same Python interpreter from executing concurrently. Using processes mitigates this as each process will run with its own interpreter.

### 2.2.1 Static structure

Key classes have been identified to form a static structure which is shown in figure 2.3 and elaborated below:

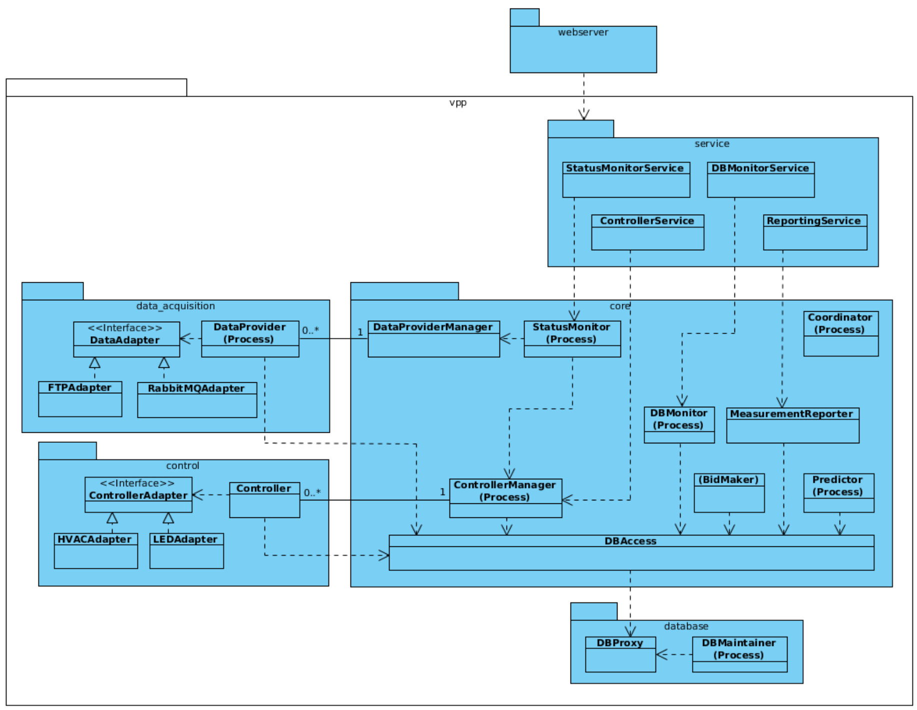


Figure 2.3: Class diagram.

The server application is roughly structured in a three-layered architecture with the `database` at lowest layer, the `core`, `control` and `data_acquisition` packages as the middle "domain" layer and the `service` package as the top layer, providing an external interface.

The `webserver` will access the `service` layer to provide GUI. It will be a separate application deployed on the same machine. See section 2.2.3.

**Package** `vpp.core`

This package implements the core logic of the VPP server. Classes are explained in detail below:

`Coordinator` is responsible for instantiating other classes and processes. For clarity, associations are not shown in the diagram.

`DataProviderManager` instantiates and keeps track of individual data providers of measurements and predictions.

`ControllerManager` instantiates and keeps track of controllers for actuating physical devices. This will run in a separate process that periodically will check for and execute scheduled actions. It is expected that one process for handling all actions will be sufficient.

`StatusMonitor` runs a process that will poll `DataProviderManager` and `ControllerManager` for status and make this information available to the service layer.

`DBMonitor` runs a process that will monitor the database status. It will give information on when the last measurements were received and similar.

`Predictor` will run a process to create predictions based the available data.

`DBAccess` provides a clean interface for retrieving and posting data from and to the database.

**Package** `vpp.data_acquisition`

This package will contain the framework for connecting to various sources of measurement and prediction data. Class `DataProvider` can be instantiated (in a separate process, or possibly just thread) to model a single source of data. Each `DataProvider` instance will employ a suitable `DataAdapter` to communicate with for instance a message queue or a SmartAmm server.

**Package** `vpp.control`

Similar to `data_acquisition`, this package will provide a framework for communicating with the various control devices. A `Controller` can be instantiated with a suitable `ControllerAdapter` to communicate with a given device.

**Package** `vpp.database`

This package contains the code that interfaces directly with the database.

`DBMaintainer` will run maintenance on the database and implement the Rolling Window strategy.

Communication with the database could be implemented simply using SQL, or we could opt for an object-relational mapper (ORM) framework such as SQLAlchemy.

### 2.2.2 Runtime processes

The runtime creation of processes within the main server application is shown below:

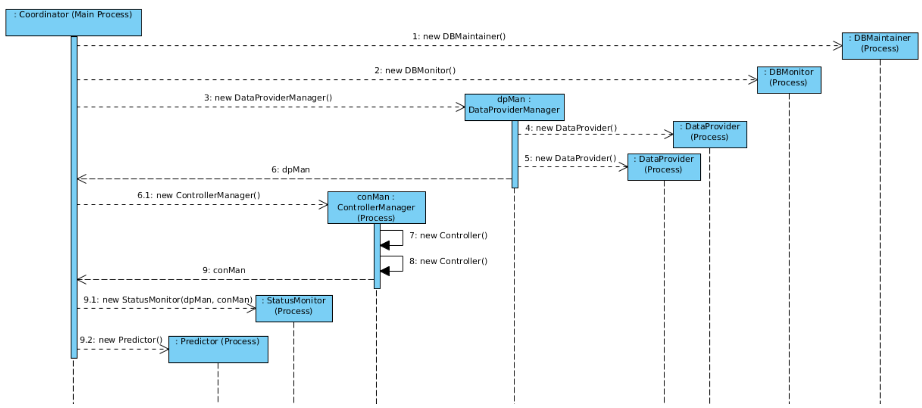


Figure 2.4: Sequence diagram of thread creation

As can be seen, at least six concurrent processes will be running in addition to any `DataProviders` configured.

#### Process communication overhead

There is of course an overhead cost involved in running processes as opposed to threads, since processes do not have shared memory which will make communication more costly performance-wise. A balanced approach could be to group processes with frequent communication as threads within the same process, thus reducing the total number of processes form the current minimum six (plus `DataProviders`) to two or three.

### 2.2.3 Web server

We intend to provide a web interface for users to access the system. This will run in a separate web server, but in the same machine. The preliminary plan is to build the webapp using Django since this supports Python.

The web interface could in itself grow to a rather large application with support for building configuration, device configuration, user administration, actuation interfaces and so on. This will require a substantial design and development effort. In the initial version, we plan to support only very basic interaction as proof of concept.

# Chapter 3

# Development tasks

Given the design described in the previous chapter, we here define initial development tasks as well as an estimate of their size. Note: These are *rough ballpark* estimates.

## 3.1 Pending design

### 3.1.1 Task: Examine Python interprocess communication

Python supports various methods of interprocess communication. Evaluate these and decide on a model to use for the VPP. The Python `multiprocessing` module looks promising. Also consider interaction if some processes are created as threads instead.

Estimate: 1-2 days.

### 3.1.2 Task: Evaluate ORM options and decide DB connectivity model

The existing solution uses plain SQL. Evaluate this against SQLAlchemy, which seems to be the Python ORM of choice.

Estimate: 1/2 day.

### 3.1.3 Task: Possibly identify use cases

It might be beneficial to identify a few key use cases / scenarios to determine which functionality is crucial and what can wait.

Estimate: 1/2 day.

## 3.2 Initial prototype

### 3.2.1 Task: Initial database

Get database up and running with core schema (possibly excluding prediction tables).

Estimate: 2-3 days?

### 3.2.2 Task: Python DB access

Implement `DBProxy/DBAccess` (according to decision from 3.1.2).

Estimate: 2-3 days?

Note: If we use an ORM, this and the previous task *might* be solved quite quickly...

### 3.2.3 Task: Initial version of DataProvider framework

Implement an initial `DataProviderManager` and a prototype `DataAdapter`, for instance for the RabbitMQ. Store measurement in DB.

Estimate: 1 week?

### 3.2.4 Task: Initial version of Controller framework

Initial `ControllerManager` and a prototype `ControllerAdapter`. Read a scheduled action from DB and actuate controller.

Estimate: 1 week?

### 3.2.5 Task: Initial web server

Get web server up and running with Django.

Estimate: 1 day?

### 3.2.6 Task: Data throughput to web server

Have the webserver display basic information on devices. Requires the `StatusMonitorService`.

Estimate: 1-2 days

### 3.2.7 Task: Actuation through web server

Click a button in the web interface to actuate a device. Requires the `ControllerService`.

Estimate: 1-2 days

## 3.3 Summary

All of the above estimates sum up to a rather unreliable figure of approx. 4 man-weeks.