Virtual Power Plant design document

Ubbe Welling
ubbe@eng.au.dk

May 30, 2016

Contents

Chapter 1

Overview

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.

Chapter 2

Design

2.1 Database design

The database will reside in a PostgreSQL DBMS which is open source and should run well even on a smaller machine.

2.1.1 Schema: core

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

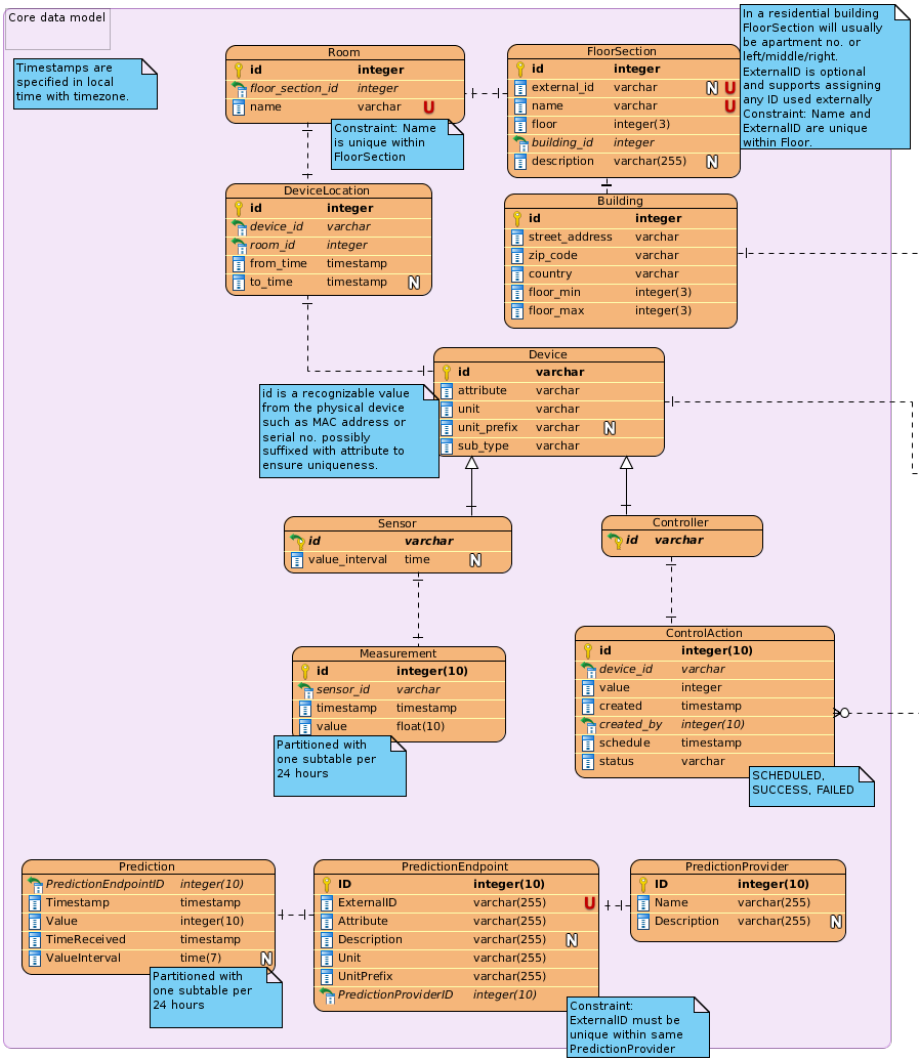

Figure 2.1: Database core schema

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 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.

2.1.3 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.

2.1.4 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

structure is shown in figure 2.5 and elaborated below:

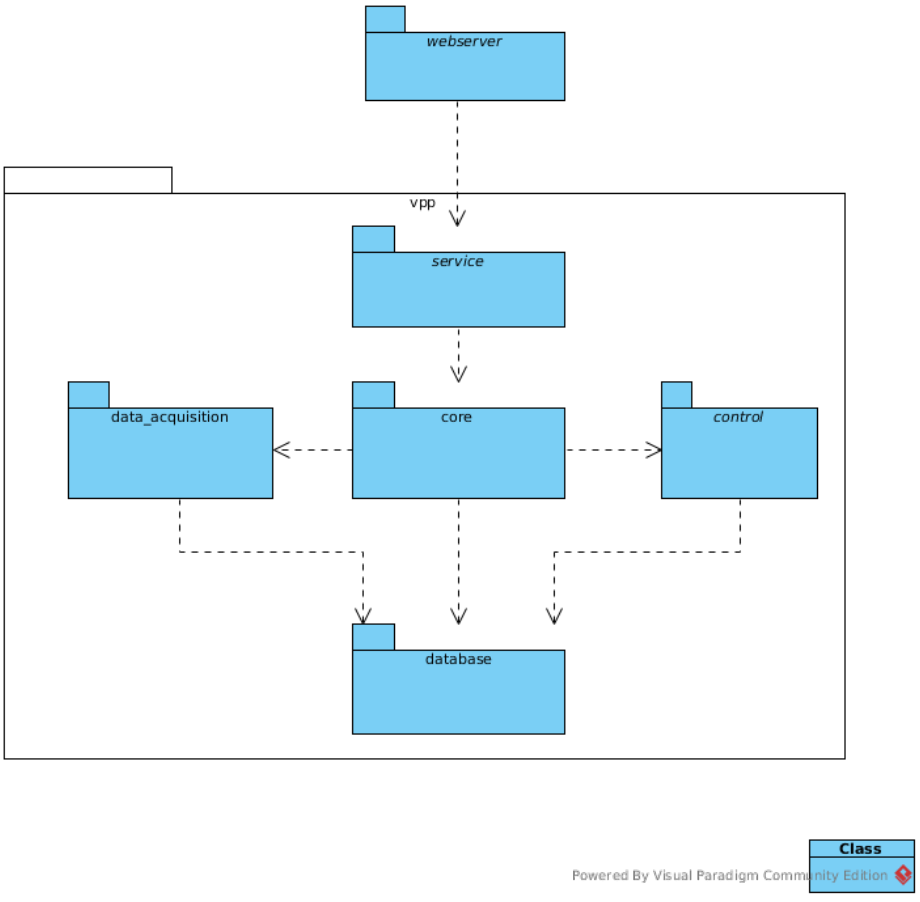

Figure 2.2: Class diagram.

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 2.2.4.

Package `vpp.core`

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

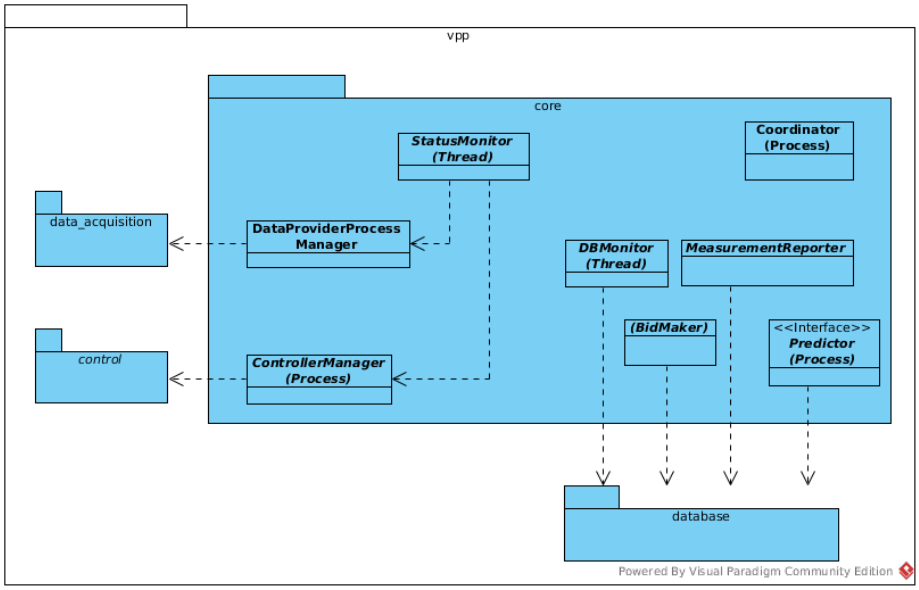

Figure 2.3: Core

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

`DataProviderProcessManager` launches the process which instantiates and manages the individual data providers that will supply 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 ser-

vice 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`

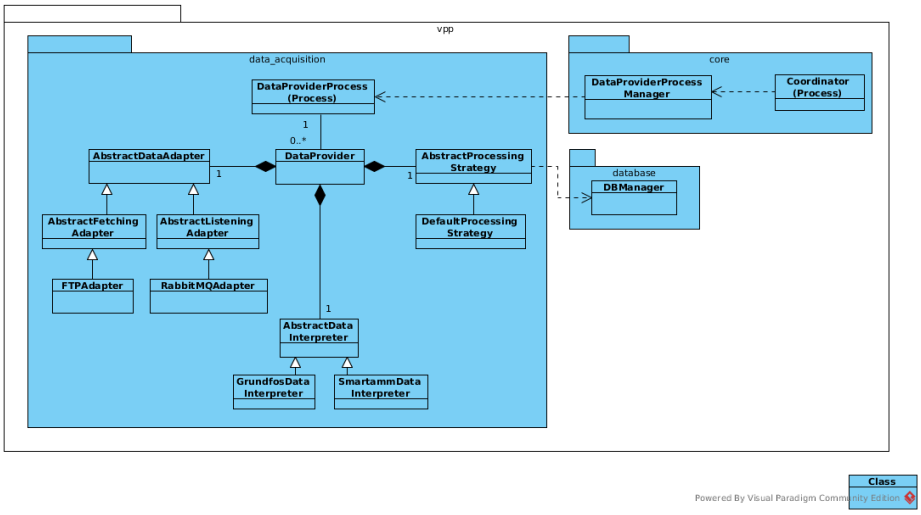

Figure 2.4: Data acquisition

This package contains the framework for connecting to various sources of measurement and prediction data. Class `DataProvider` can be instantiated in a separate thread to model a single source of data. Each `DataProvider` instance employs a suitable `DataAdapter` to communicate with for instance a message queue or a SmartAmm server.

A distinction is made between listening adapters (`AbstractListeningAdapter`), which will be activated by external events, and fetching adapters (`AbstractFetchingAdapter`) which employ an internal timer to fetch data periodically.

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:

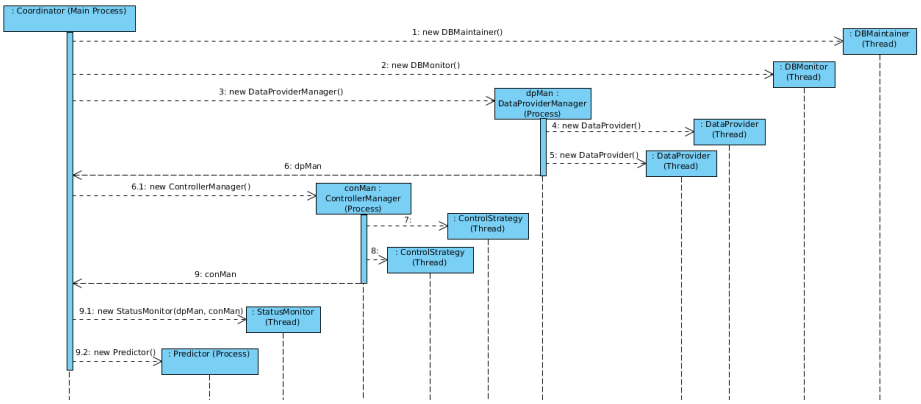

Figure 2.5: Sequence diagram of thread creation

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 Data acquisition

The execution flow when receiving measurements from a RabbitMQ is shown below:

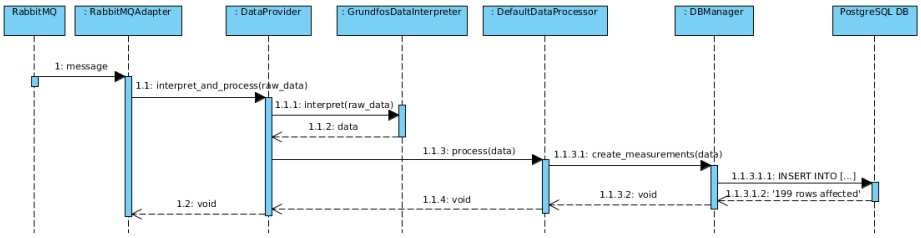

Figure 2.6: Sequence diagram of data acquisition

2.2.4 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

How to run

3.1 Configuration

3.2 Scripts