Virtual Power Plant design document

Ubbe Welling
ubbe@eng.au.dk

December 7, 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

1.3 Use cases

UC1a: Grundfoss sensor data from RabbitMQ (completed)

Trigger: RabbitMQ announces new message with Grundfoss data.
Post condition: Measurements are stored in DB
Steps:

1. Message is sent from RabbitMQ to VPP `DataProvider`

2. For each measurement in message:

    If sensor is unknown, create it and store in DB

    Store measurement (with link to sensor)

UC1b: Develco sensor data from RabbitMQ (completed)

Trigger: RabbitMQ announces new message with Develco data.
Post condition: Measurements are stored in DB

UC2: Sensor and prediction data from FTP (Priority 2)

Trigger: Periodic check connects to configured FTP
Post condition: Measurements and predictions are stored in DB, avoiding duplication of previously posted data
Steps:

1. VPP DataProvider (with FTP adapter) connects to FTP with configured credentials

2. The targeted file is downloaded

3. Relevant (new) data from file is stored as measurements and predictions, as appropriate.

UC3: Scheduled ControlAction is executed (Priority 3)

Trigger: Periodic check discovers scheduled action on ventilation system
Post condition: Ventilation system setting is changed via HTTP, and the VPP DB updated to reflect action has been carried out.

UC4: HTTP control of Develco devices (smart plugs) (Priority 10)

Extension of SmartAmm server with HTTP interface.

UC5: VPP control strategy schedules action (Priority 5)

Trigger: Control strategy periodic check (or possibly arrival of relevant measurement/prediction data)
Post condition: New control action is stored DB
Steps:

1. Read relevant data from DB

2. Control algorithm computes suitable action

3. Store new action

4. If action should be performed immediately: Notify ControllerManager

UC6: Data visualization on website (Priority 5)

Trigger: Super user wishes to view data form specific sensor
Post condition: Graph of data from sensor is displayed in browser
Steps:

1. User authenticates with username and password

2. User specifies sensor and time range

3. User is presented with data plot

UC7: Periodic status report (Priority 6)

Trigger: Status report is generated every 24 hours.
Post condition: Administrator receives email with report
Data in report:

- New sensors

- Sensors gone offline (no reports for 24 hours)

- Statistics on measurement data
    size (kilobytes)
    number of measurements

- Status on subsystems
    uptime

Possibly use Nagios for system info

UC8: alarm report (Priority 7)

Trigger: RabbitMQ has not sent data for 1 hour
Post condition: Administrator receives email with report
    Configure alert subscriber

1.4 Measurement data characteristics

Grundfoss data: 2-3GBs per day. A few thousand sensors report every 5 seconds, giving approx. 50 mio. rows per day. That is 500 rows per second!

Develco: 15MB per day, 0.5 mio. rows

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

Since the intended machine is somewhat limited, the immediate choice would be to use SQLite for the database. SQLite is however not designed for concurrent access, in that it resides in a single file which must be locked to perform writes. Supporting 5-600 writes per second (the Grundfoss dormitory) from multiple data provider threads/processes along with controller processes, monitor processes, a reporting process and a predictor process is expected to cause significant file lock contention. >500 writes per second gives <2ms per write, not counting the need to allow other processes to access the database. Since we do not expect to do disk writes in less than 2ms (and even if we did, there would be no scaling potential), the SQLite option is abandoned.

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:
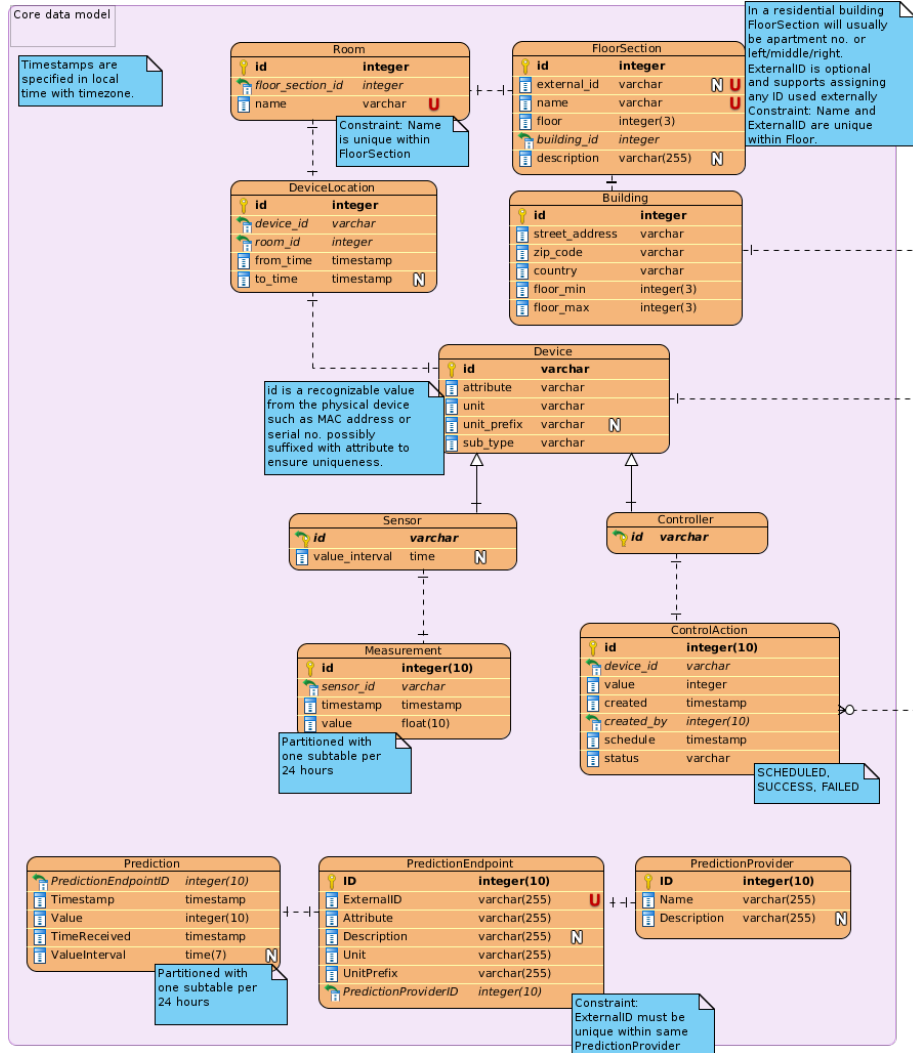
Core data model

Timestamps are specified in local time with timezone.

Room
id	integer
floor_section_id	integer
name	varchar	U

Constraint: Name is unique within FloorSection

FloorSection
id	integer
external_id	varchar	N U
name	varchar	U
floor	integer(3)
building_id	integer
description	varchar(255)	N

In a residential building FloorSection will usually be apartment no. or left/middle/right. ExternalID is optional and supports assigning any ID used externally. Constraint: Name and ExternalID are unique within Floor.

DeviceLocation
id	integer
device_id	varchar
room_id	integer
from_time	timestamp
to_time	timestamp	N

Building
id	integer
street_address	varchar
zip_code	varchar
country	varchar
floor_min	integer(3)
floor_max	integer(3)

Device
id	varchar
attribute	varchar
unit	varchar
unit_prefix	varchar	N
sub_type	varchar

id is a recognizable value from the physical device such as MAC address or serial no. possibly suffixed with attribute to ensure uniqueness.

Sensor
| id | varchar |
| value_interval | time | N |

Controller
| id | varchar |

Measurement
id	integer(10)
sensor_id	varchar
timestamp	timestamp
value	float(10)

Partitioned with one subtable per 24 hours

ControlAction
id	integer(10)
device_id	varchar
value	integer
created	timestamp
created_by	integer(10)
schedule	timestamp
status	varchar

SCHEDULED, SUCCESS, FAILED

Prediction
PredictionEndpointID	integer(10)
Timestamp	timestamp
Value	integer(10)
TimeReceived	timestamp
ValueInterval	time(7)	N

Partitioned with one subtable per 24 hours

PredictionEndpoint
ID	integer(10)
ExternalID	varchar(255)	U
Attribute	varchar(255)
Description	varchar(255)	N
Unit	varchar(255)
UnitPrefix	varchar(255)
PredictionProviderID	integer(10)

Constraint: ExternalID must be unique within same PredictionProvider

PredictionProvider
ID	integer(10)
Name	varchar(255)
Description	varchar(255)	N

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, groupings and privileges

The database schema for storing device groupings, users, user groups and their privileges has been developed to meet the requirements from section 1.2.2. This is shown below:
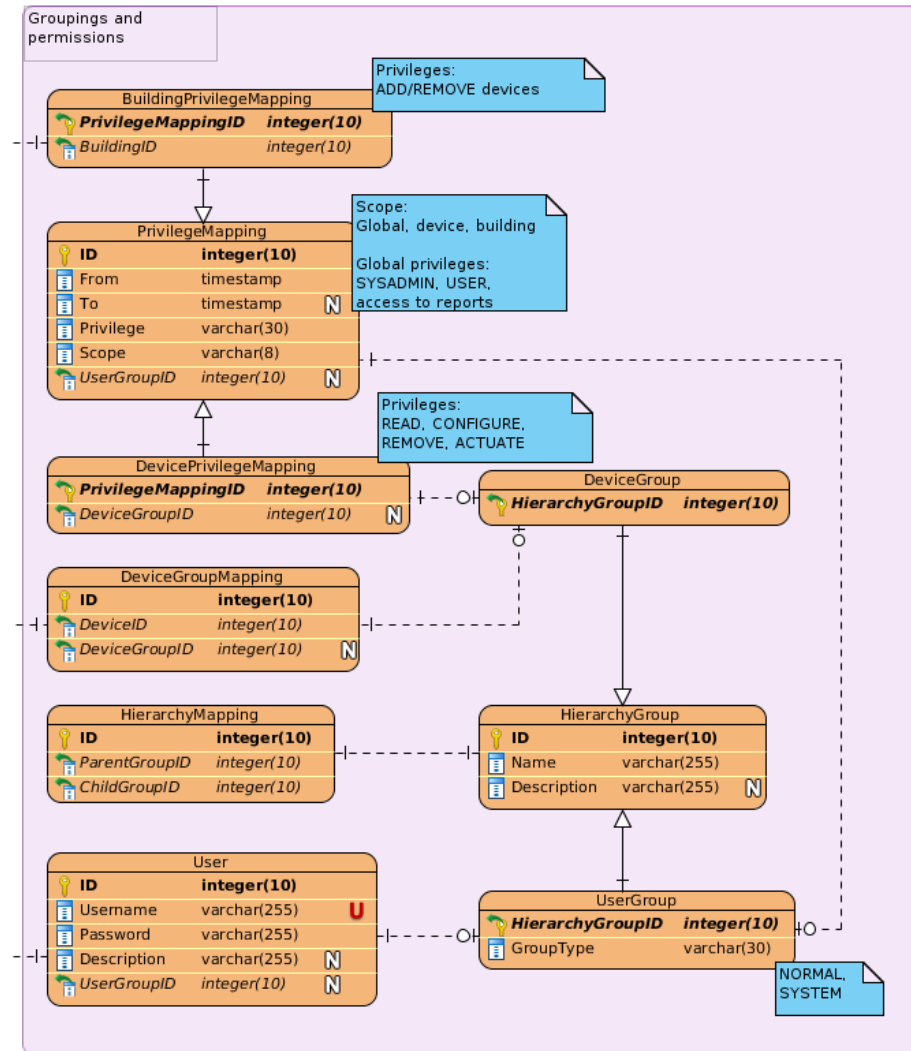
Figure 2.2: Schema for device groups, user groups and privileges

DeviceGroupMapping   Devices can be put in arbitrary logical groups which is modeled by this table.

User and UserGroup `User`s 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 `UserGroup`s.

HierarchyGroup and HierarchyMapping `DeviceGroup` and `UserGroup` are subtypes of the general `HierarchyGroup` which can be nested in parent-child relationships as modeled by `HierarchyMapping`.

PrivilegeMapping This table maps `UserGroup`s 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 `DeviceGroup` 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

structure is shown in figure 2.5 and elaborated below:
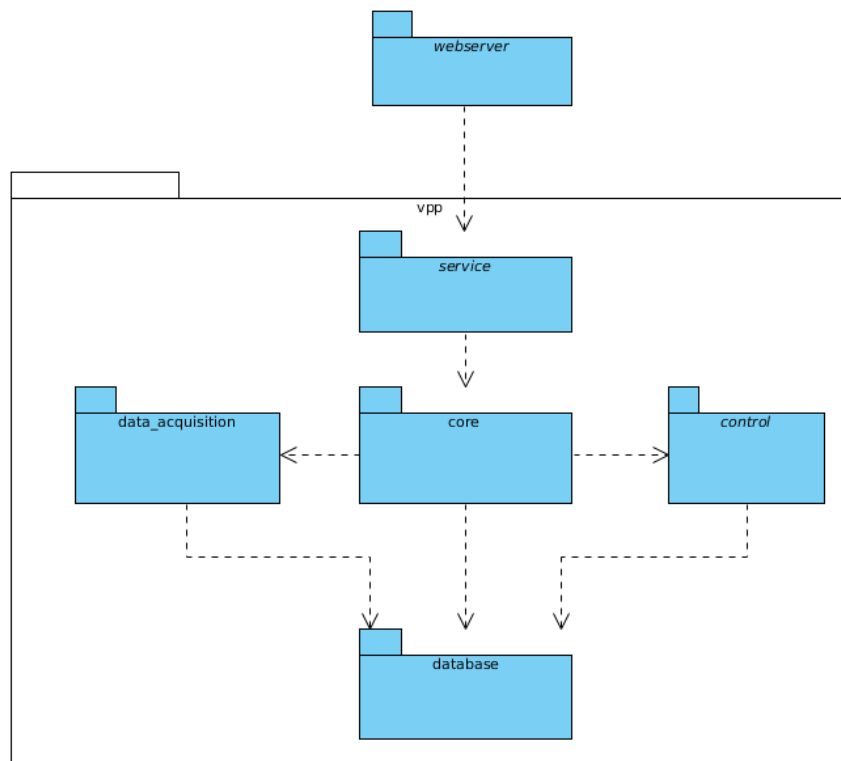
Figure 2.3: Class diagram.

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 2.2.4.

Package `vpp.core`

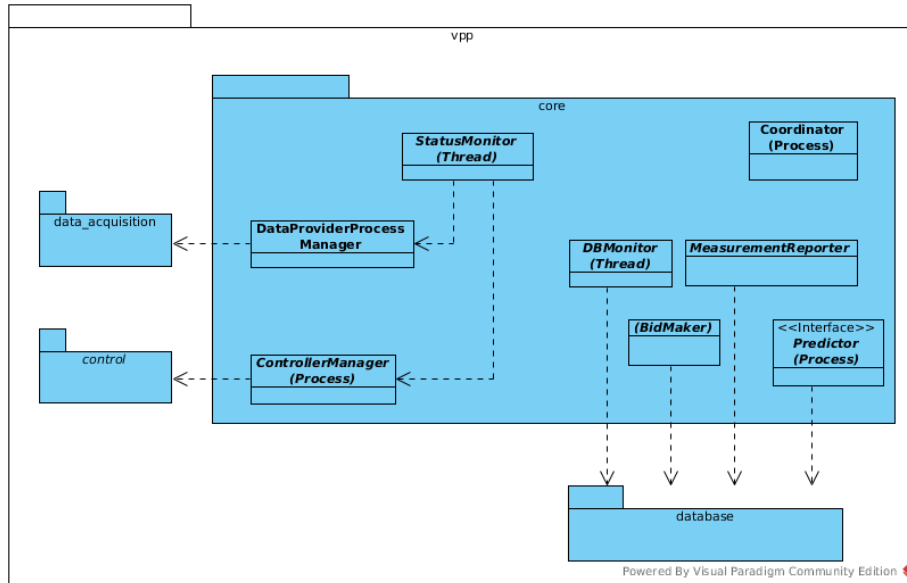This package implements the core logic of the VPP server. Classes are explained in detail below.

Figure 2.4: 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-
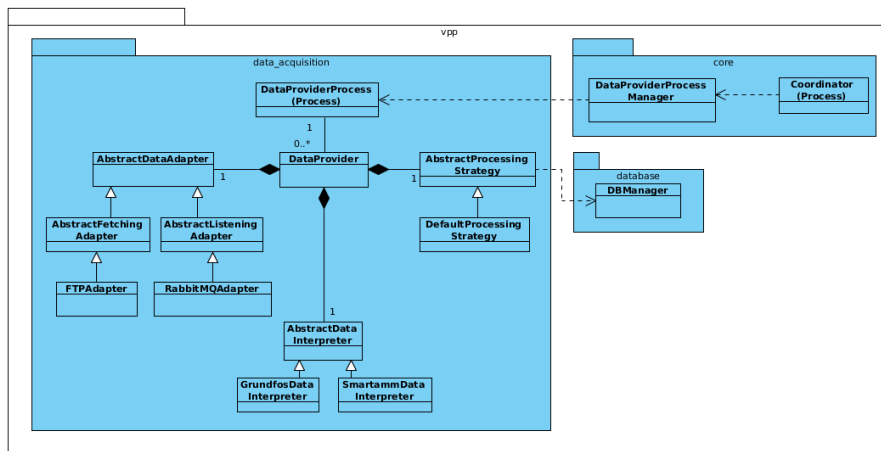
16

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 2.5: 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.

17

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:
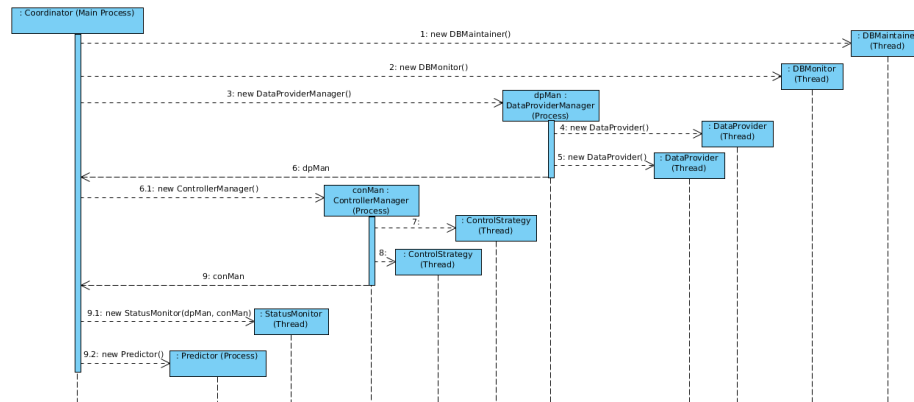

Figure 2.6: Sequence diagram of thread creation

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
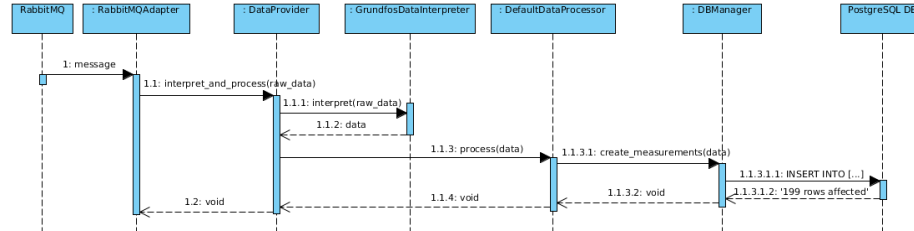below:

Figure 2.7: 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 de-
velopment effort. In the initial version, we plan to support only very basic
interaction as proof of concept.

Chapter 3

Development tasks for initial prototype

3.1 Tasks

Database maintenance

Clearing tables older than x days, transfer of data to data warehouse.

UC2: Sensor and prediction data from FTP

Task: Database schema for predictions

DB schema and access through `DBManager`.

Task: FTPAdapter

Implement an `FTPAdapter`. Store measurements and predictions in DB, avoiding duplication.

UC3: Scheduled ControlAction is executed

Task: Initial version of Controller framework

Initial `ControllerManager`, dummy `ControlStrategy, Controller`.

Task: VentilationControllerAdapter

Communicate with ventilation system using HTTP.

Task: Execute scheduled action

Read a scheduled action from DB and actuate ventilation controller.

UC5: ControlStrategy schedules action(s)

Task: Simple ControlStrategy

Simple `ControllerStrategy` which schedules ventilation system to run at lower setting at night.

3.2   Summary

The above covers acquiring measurements from RabbitMQ, predictions and measurements from FTP and scheduling actions for controlling the ventilation system.