Virtual Power Plant server technical documentation

Ubbe Welling
ubbe@eng.au.dk

June 27, 2016

Contents

Chapter 1

Overview

This report provides technical documentation of the VPP software located in `https://github.com/comsyslab/vpp`. To use the software, read chapters 1, 2 and 3. To modify/extend the software, read chapters 4 and 5 as well.

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.

In the current state, only the two first bullets (i.e. data collection) have been implemented. In the design document located in `https://github.com/comsyslab/vpp/blob/master/documentation/design_doc/main.pdf`, the original ideas and visions for further extensions to the system are documented. Note that the technical information in the design document is not up-to-date. The present document is intended to provide a comprehensive description of the software in its current state (May 31, 2016).

1.2 Functionality

The system can receive measurement and prediction data from data providers via FTP, SFTP and RabbitMQ. There is thus no direct connection to physical sensors.

Received data is stored in a local database which employs a *Rolling Window* scheme: Data is put into subtables according to timestamp (one subtable for e.g. every 24 hours), and when a subtables ages beyond the configured time limit (e.g. 5 days), the contents of it is transferred to the external data warehouse database and the table is deleted. This enables deployment of the VPP software on machines with limited disk size.

Chapter 2

Configuration

2.1 Main configuration file

The main configuration file is `vpp/vpp_server/resources/config.ini`. In the repository, a `config.ini.default` is provided. It is shown and explained below:

```
[DB]
user=[FILL IN]
password=[FILL IN]
host=[external address of localhost]
database=vpp
measurement_partition_period_hours=24
rolling_window_length_days=7

[DB-DW]
enabled=False
user=[FILL IN]
password=[FILL IN]
host=[host.sub.domain.dk]
database=vpp_dw

[LOG]
level=DEBUG
```

2.1.1 Section `[DB]`

Section `[DB]` specifies how to connect to the Rolling Window DB that is intended to run on the same machine as the VPP application. Note that it is perfectly possible to run the database on a different machine if desired.

`host` This should be the external address (hostname or IP) of the machine running the database - usually the same as is running the application. While writing `localhost` is sufficient for the application to access the DB, this address is also used by the data warehouse to extract data for export, and therefore it must be the machine's actual network address. Port `5432` (the PostgreSQL default) is assumed. Note that this port must be accessible from the data warehouse machine, as the data export SQL is sent to and executed in the data warehouse DB, connecting back to the main DB.

`database` The name of the database to connect to.

`measurement_partition_period_hours` Incoming measurements and predictions will be stored in subtables covering time intervals of the number of hours specified here. Should be a divisor of 24 (e.g 24, 12, 6). Behavior with other values has not been tested. The partitioning happens with respect to UTC, not the local time of the server.

`rolling_window_length_days` The number of days to retain subtables with measurements and predictions before exporting the data to data warehouse and deleting the tables. The current day is not counted, meaning that a value of 1 will cause data received during June 10th to be exported sometime during June 12th, since the last data of June 10th will at that time be at least 24 hours old. Behavior with value 0 has not been tested.

2.1.2 Section `[DB-DW]`

`enabled` If `true`, data export to data warehouse and subsequent local deletion will be performed. If `false`, data will be retained in the local database forever. In this case, configuration of user, password, host and database name may be omitted, and the `rolling_window_length_days` value above is irrelevant. Values are not case sensitive.

2.1.3 Section `[LOG]`

`level` Specifies the level of logging that will be output in `vpp_server/logs/console.log`. Valid values (case sensitive): `DEBUG, INFO, WARNING, ERROR, FATAL, CRITICAL` (the last two are equal) and integer values. `DEBUG` is level 10. Setting value 9 will add messages on files skipped when fetching from FTP servers. The log level can be changed while the application is running and should take effect within 5 seconds.

No file rotation is performed, which can cause log file to grow very large if the log level is set to `INFO` or lower and a high volume of messages is received. It is recommended to set level `WARNING`, in which case any problems will be reported, but nothing else.

Note that the log file is cleared on application startup.

2.2 Data provider: measurements from FTP

The retrieval of data is configured by placing a `.ini` file for each data source in folder `vpp_server/resources/data_providers`.

File `ftp_energinet_online.ini`:

```
[data_provider]
adapter = vpp.data_acquisition.adapter.ftp_adapter.FTPAdapter
interpreter = vpp.data_acquisition.interpreter.energinet_online_interpreter.
processor = vpp.data_acquisition.processing_strategy.DefaultMeasurementProce
id_prefix = energinet

[fetch]
interval = 600
adapter_date_strategy = vpp.data_acquisition.adapter.adapter_date_strategy.D
last_fetch_adapter = 2016-05-27T00:00:00
fetch_again_when_date_equal = True
last_fetch_interpreter = 2016-05-27T14:55:00

[ftp]
host = 194.239.2.256
username = ftp000123
password = ...
port = 21
remote_dir = Onlinedata
file_pattern = ([0-9]{4})([0-1][0-9])([0-3][0-9])_onlinedata\.txt
encoding = iso-8859-1

[averaging]
enable = True
id_patterns = energinet_1; energinet_2; energinet_3; energinet_%%
intervals = 900; 1800; 3600; 7200
```

2.2.1 Section `[data_provider]`

This section configures the protocol for obtaining data, and how they should be interpreted and stored. The adapters, interpreters and processors (listed below) can be combined in any way, but the interpreter obviously must be matched to the data fetched/received by the data provider. It also would be pointless to combine a PredictionProcessingStrategy with a data source and interpreter that provides measurements.

`adapter` This specifies which data adapter to use for retrieving data from the provider. Available adapters:

`vpp.data_acquisition.adapter.ftp_adapter.FTPAdapter`
`[...].ftp_adapter.SFTPAdapter`

[...].rabbit_mq_adapter.RabbitMQAdapter

The adapter will handle communication with the provider and extract a raw text body and pass it to the data interpreter.

`interpreter` This specifies which data interpreter to use for parsing the received raw text messages and transforming them to an internal data format that is then passed to the processor responsible for storing measurements (or predictions) in the database. Available interpreters:

vpp.data_acquisition.interpreter.energinet_online_interpreter.EnerginetOnlineInterpreter
[...].energinet_co2_interpreter.EnerginetCO2Interpreter
[...].nordpoolspot_interpreter.NordpoolspotInterpreter
[...].thor_interpreter.ThorInterpreter
[...].nrgi_abs_interpreter.NrgiAbsInterpreter
[...].nrgi_delta_interpreter.NrgiDeltaInterpreter
[...].grundfos_data_interpreter.GrundfosDataInterpreter
[...].smartamm_data_interpreter.SmartAmmDataInterpreter

`processor` Responsible for storing measurements or predictions in the database. Available processors:

vpp.data_acquisition.processing_strategy.DefaultMeasurementProcessingStrategy
vpp.data_acquisition.processing_strategy.DefaultPredictionProcessingStrategy

`id_prefix` Among the data received will be sensors (logical sources of measurements) and endpoints (logical sources of predictions). The id_prefix specifies a prefix for the database ID of the sensors and endpoints which makes it easier to distinguish them and their measurements/predictions in the database.

2.2.2 Section [fetch]

Since the FTPAdapter must actively fetch data, this section is required.

`interval` In seconds. Every time this interval passes, the FTPAdapter will connect and fetch data which is then processed and stored. The time to next fetch is measured from launch, not finish, meaning that the interval will be maintained precisely as long as the fetch and processing completes before the next launch.

`adapter_date_strategy` Strategy for deciding whether a file should be fetched, based on the date extracted from the file name. Available strategies:

vpp.data_acquisition.adapter.adapter_date_strategy.DefaultAdapterFileDateStrategy
vpp.data_acquisition.adapter.adapter_date_strategy.CO2FileDateStrategy

The `DefaultAdapterFileDateStrategy` will fetch a file if the date is later (or equal to, depending on the value of `fetch_again_when_date_equal`) than the value of `last_fetch_adapter`.

CO2FileDateStrategy will fetch if the date extracted from the filename is in the future compared to the time of execution. This is useful for the CO2 predictions from Energinet.

fetch_again_when_date_equal Specifies whether files with timestamp identical to the value of last_fetch_adapter should be fetched. Setting this to True will enable repeated retrieval of the same file(s), which can be useful if file contents are updated.

last_fetch_adapter Updated by the data adapter when fetching files to the latest timestamp extracted from the fetched filenames. If configuring a new data provider, set this to a value close to the current time to avoid fetching many old files.

last_fetch_interpreter Updated by the data interpreter to the timestamp of the latest processed measurement/prediction. This is relevant in the case the same file is continuously updated with new data.

2.2.3 Section [ftp]

host, username, password and port should be self-explanatory.

remote_dir is the subdirectory to access on the FTP server. Use forward slashes.

file_pattern This should be a Python regex containing up to four groups which will match year, month, day and hour (in that order). Only year is required - the others are optional. Only files matching this pattern will be considered for fetching. The extracted timestamp is compared against the value of last_fetch_adapter to determine if the file should be fetched.

encoding Specifies the encoding of the files to be fetched. Tested values are utf-8, ascii and iso-8859-1.

2.2.4 Section [averaging]

enable If True or true, data transferred to data warehouse will be averaged over the intervals specified below in order to reduce data size in data warehouse. If not, entries will be bulk copied as-is.

id_patterns List of patterns of sensor/endpoint IDs to perform averaging for. Patterns are used in a PostgreSQL LIKE comparison. The wildcard % must be written twice due to Python's string parsing. Thus, to match all sensors starting with prefix energinet, write energinet%%. Patterns should be separated with semicolon ;. Every pattern should have a corresponding interval in the `intervals` list below. The patterns are processed in order, so it makes sense to add a catch-all pattern at the end of the list.

`intervals` Semicolon-separated list of time intervals in seconds, each interval corresponding to the pattern with the same position in the list above.

2.3 Data provider: measurements from RabbitMQ

File `rabbitmq_ecosense_smartamm.ini`:

```
[data_provider]
adapter =
vpp.data_acquisition.adapter.rabbit_mq_adapter.RabbitMQAdapter
interpreter =
vpp.data_acquisition.interpreter.smartamm_data_interpreter.SmartAmmDataInter
processor =
vpp.data_acquisition.processing_strategy.DefaultMeasurementProcessingStrateg
id_prefix = smartamm

[exchange]
username = user123
password = 12345
serverAddressList = ecosensemq02.cs.au.dk
port = 5672
sslConnection = False
exchangeName = ecosense-exchange
exchangeDurable = True
exchangeType = topic

[queue]
name = vpp-listener-queue-new
routingKeys = smartamm.#
durable = False
exclusive = False
autoDelete = True

[ssl_options]
cacerts = /etc/rabbitmq/ssl/tesca/cacert.pem
certfile = /etc/rabbitmq/ssl/client/cert.pem
keyfile = /etc/rabbitmq/ssl/client/key.pem
cert_reqs = verify_peer
fail_if_no_peer_cert = True

[averaging]
enable = False
id_patterns = smartamm%%RMSCurrent%%; smartamm%%ActivePower%%;
intervals = 60; 120
```

2.3.1 Section `[exchange]`

`serverAddressList` The RabbitMQ exchange server to connect to. The "`List`" is misleading. Only a single host should be specified.

`sslConnection` If `True`, the `Pika` library will enable SSL when connecting. Not tested.

For the remaining configuration, look at `https://www.rabbitmq.com/getstarted.html` to understand how RabbitMQ works.

2.3.2 Section `[queue]`

Specifies the RabbitMQ queue to connect to in the exchange with various parameters. Again: Look at `https://www.rabbitmq.com/getstarted.html` to understand how RabbitMQ works.

2.3.3 Section `[ssl_options]`

This section is not parsed, regardless of `sslConnection` seeting above.

Chapter 3

How to run

The various scripts are placed in `vpp_server/run` and explained below. All scripts exist in a Linux (`.sh`) and Windows (`.bat`) version.

3.1 Scripts

`clear_db.sh` Use with care. Connects to the database specified in the `[DB]` section of `config.ini` and drops and recreates the entire database schema *without prompting for confirmation.* This can be used to initialize a new database.

`clear_db_dw.sh` Use with care. Identical to `clear_db.sh` above, but connects to the database specified in the `[DB-DW]` section of `config.ini`. Note that this may contain data accumulated over a long time.

`run_tests.sh` Executes all unit tests and outputs results to console.

`start_server.sh` Launches the VPP server.
Logging is output in `vpp_server/logs/console.log`.

`stop_server.sh` Stops a running server. This is implemented in a very primitive fashion: The script creates a file named `stop` which the running server scans for every 5 seconds.

Chapter 4

Database design

The database should reside in a PostgreSQL DBMS. The schema is shown in figure 4.1 and explained in the following:

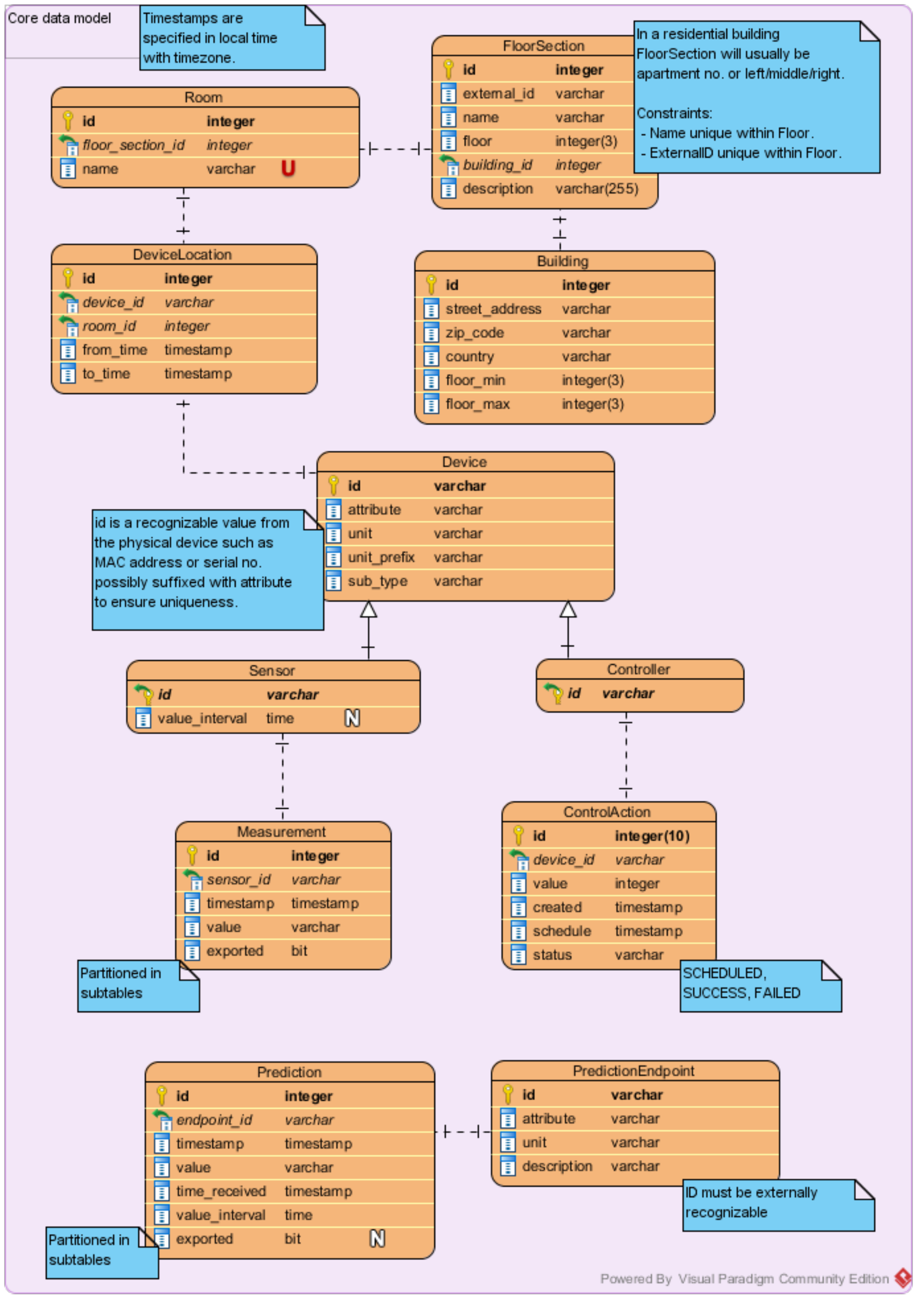

Figure 4.1: Database schema

4.1 Tables

4.1.1 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 Contains a row for each measurement received from a sensor. It simply consists of a `value`, a `timestamp` and a `sensor_id`, which enables interpretation of the value. Column `exported` indicates whether a row has been exported to data warehouse or not.

The `Measurement` table is partitioned into subtables for each e.g. 24 hours. Querying the `Measurement` table will automatically retrieve data from all its subtables, while insertion must happen directly to the correct subtable. Each subtable has a constraint on the `timestamp`, preventing measurements from being entered in the wrong subtable.

ControlAction Table `ControlAction` is intended to 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. Partitioning this table in the same way as table `Measurement` could be considered. In the current implementation of the VPP software, the `Controller` and `ControlAction` tables will always be empty.

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 specific time periods, indicating that devices may be moved around over time.

4.1.2 Predictions

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.

PredictionEndpoint A logical source of predictions of one type. Specifies the `attribute` and `unit` of the incoming values and provides an optional `description`.

Prediction Actual prediction values. `timestamp` indicates the time for which the value applies, while `time_received` 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 `value_interval`. This table is partitioned in same way as table `Measurement`.

4.2 CIM compliance

Units, unit prefixes, timestamps and time durations are formatted to agree with CIM.

Chapter 5

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.

5.0.1 Static structure

structure is shown in figure 5.3 and elaborated below:

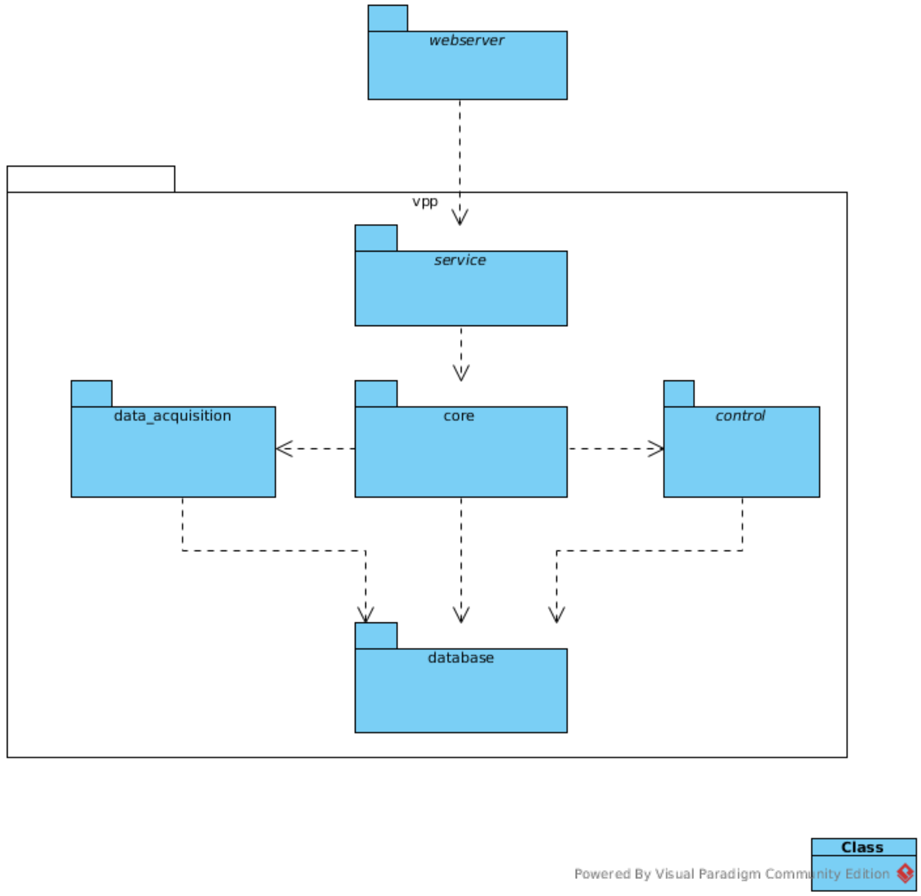

Figure 5.1: Class diagram.

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 5.0.4.

Package `vpp.core`

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

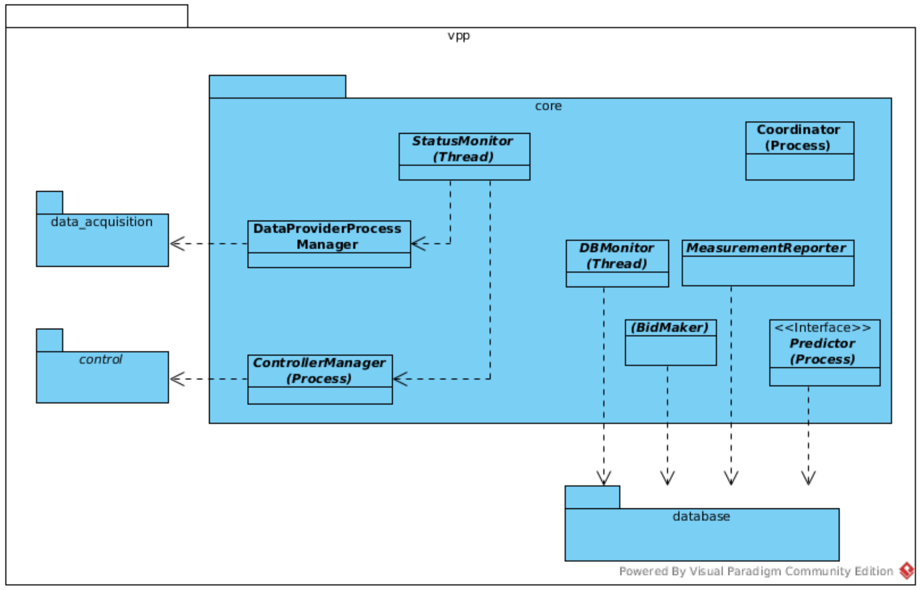

Figure 5.2: Core

`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 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

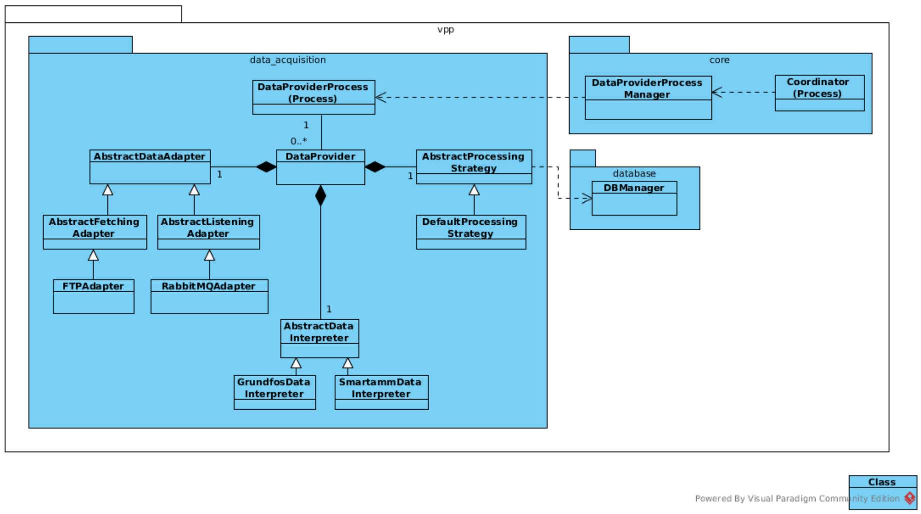

Figure 5.3: Data acquisition

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.

5.0.2 Runtime processes

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

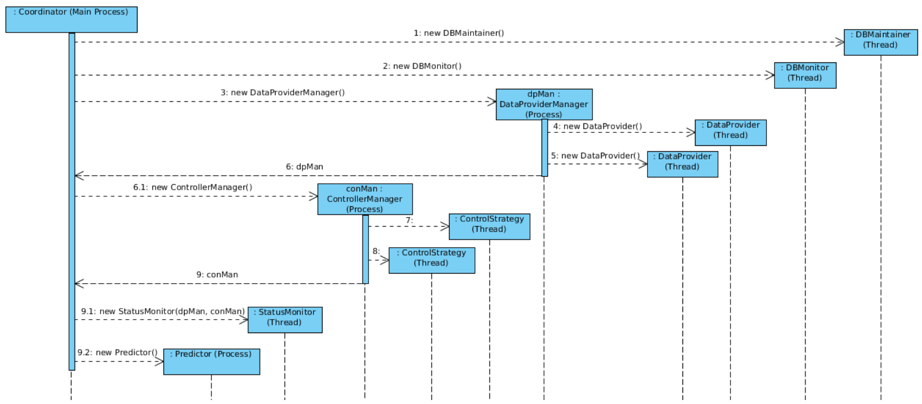

Figure 5.4: Sequence diagram of thread creation

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.

5.0.3 Data acquisition

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

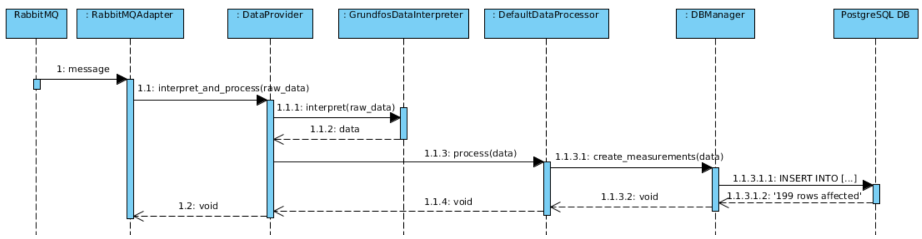

Figure 5.5: Sequence diagram of data acquisition

5.0.4 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.