mosaik – Architecture Whitepaper

Stefan Scherfke, Steffen Schütte
(stefan.scherfke | steffen.schuette)@offis.de
OFFIS, Eschwerweg 2, 26121 Oldenburg, Germany
Version: October 2012

Abstract—This paper describes the architecture of the mosaik Smart Grid simulation platform at a very low level and the API that a simulator has to implement in order to be used with mosaik. This paper targets the developers and scientists that want to work with mosaik and integrate their simulators with it or anyone who is interested in an architecture for managing simulator processes in a distributed way.

1 Introduction

The Smart Grid, as it is envisioned, relies on the use of information and communication technologies (ICT) for managing a large number of active components (controllable consumers and generators as well as power grid equipment) and sensors for keeping demand and generation of electricity at an equilibrium and for keeping all these different resources, including the grid assets, within their operational limits. Due to the distributed nature of the different resources, their heterogeneity as well as their sheer number, this is a challenging task. Control strategies/paradigms for this complex and new task still need to be developed and in particular evaluated and tested with respect to the requirements stated above. In order to yield sound and scientifically reliable results, simulations have to rely on valid and (ideally) established models. As a consequence, a lot of effort is put into the modeling and validation of both single system components such as photovoltaics or wind energy converters and composite sub-systems, e. g., entire low or medium voltage power grids. Therefore, it is desirable to reuse existing models in new projects and simulation studies as much as possible. If the existing models are implemented using different technological platforms, for example because each model uses a platform that is ideal for the specific problem (e. g., load flow estimation) or because models are provided by different project partners, simulation composition ("the capability to select and assemble simulation components in various combinations into simulation systems" [9]) is an interesting approach. As no Smart Grid specific

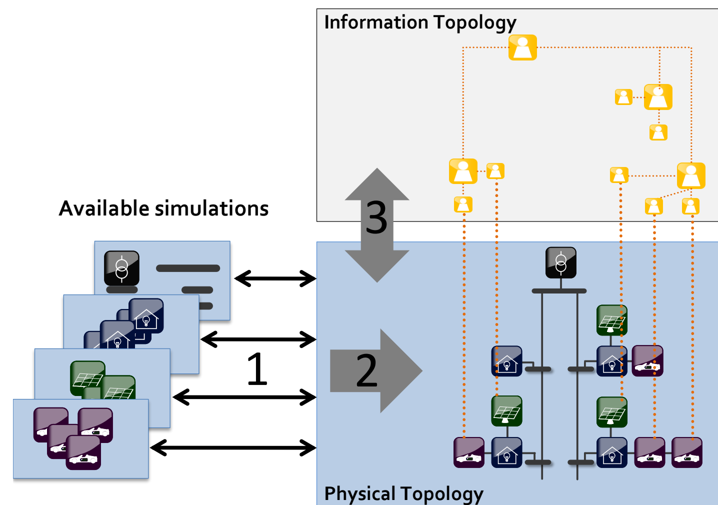

Figure 1: Identified problem areas

approach purpose-built for simulation composition could be found in literature, we initiated the *mosaik* project [13] [12] in the middle of 2010.

However, a number of problems arise when reusing existing simulation models to compose new Smart Grid scenarios. Figure 1 shows the problem areas that mosaik aims to solve. First, the available simulators[1] are usually not designed to be reused (1). Therefore they do not offer any interface that is appropriate for interacting with the executed simulation. Second, one has to find a way to compose the different simulation models in a flexible way such that different scenarios can be composed and simulated (2). The major challenge here is to find a formalism that allows to capture Smart Grid scenarios even if they involve many hundred resources without having to write thousand lines of code. And finally (3), the composed simulation has to allow the interaction with control strategies (in our research, we focus on the integration of multi-agent based control strategies, but other approaches are supported, nonetheless). Here, the major challenges include the integration of a standardized API for communication with the agents such that different strategies and simulation models can be interchanged seamlessly and the integration of the agents with the simulation time, as most multi-agent platforms are not made to work with simulated environments [1].

Inspired by the M&S architecture proposed by Zeigler et al. [16, p. 496], the mosaik platform is based on six layers as shown in figure 2.

[1]Definition: A simulator is a program that executes simulation models. The process of executing a simulation model (manually or by using a computer program) is called simulation.

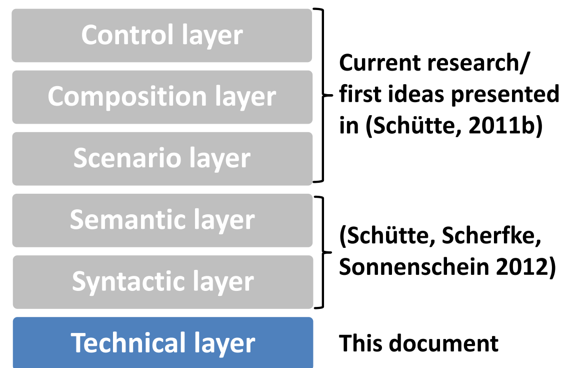

Figure 2: Layers of the mosaik concept

The technical layer provides a mechanism to find, start and control the available simulators at runtime and is within the focus of this paper. The syntactical layer offers a generic simulator API to make simulators interoperable with mosaik. On the semantic layer, the simulator properties and the models that it can execute are semantically described in a formal way. These simulator descriptions are then used on the scenario level to formally describe Smart Grid scenarios. A *scenario* defines a certain composition and can also be used hierarchically in other scenarios. Finally, the composition layer performs the actual composition of simulators based on the formal scenario and simulator descriptions, and the control layer allows to interact with the simulated entities at runtime. The syntactic and semantic layer have been described in [14] and the details of the scenario and composition layer will be published soon.

Figure 3 shows the rough architecture of mosaik. It is split into a server side part which performs the actual composition and simulation as well as the logging of all data that is produced during a simulation run. On the client side, the user will be offered appropriate means to manage the simulation study.

First, the user specifies the scenario that is to be simulated in a formal way. This includes the parameterization of the available simulation models and the definition of connection rules between the entities that are part of the models. For example, the user specifies a rule that defines that each node of the simulated power grid shall be connected to one simulated consumer. Or that every third node shall have 1 to 3 electric vehicles. At server-side the scenario definition and thus the connection rules are evaluated, the required simulators are initialized and the simulation is executed. All data generated by the simulators will be logged for later analysis (i.e. benchmarking of potential control strategies).

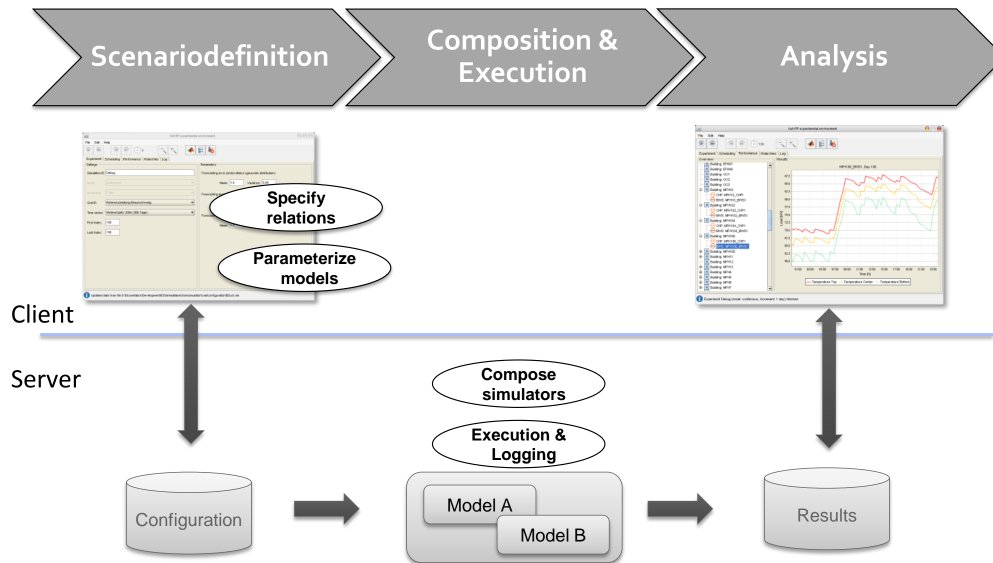

Figure 3: High-level architecture of mosaik

Regardless of the chosen approach for simulation composition, the different simulators that provide the simulation models have to be started and managed at runtime. Such a management framework is outside the scope of the available simulation interoperability standards. "HLA, for example, does not specify any tools to design or deploy a federation [simulator]" [2, p. 218]. Therefore we decided to develop an own solution to tackle this issue.

2 Requirements

The main purpose of the mosaik server package is to compose a simulation based on a given scenario and to start and control a number of external simulators to execute that simulation.

To allow the usage of proprietary simulators that may only run in dedicated virtual machines or on servers of external partners, mosaik should not only be able to start simulators on the local system but also on remote machines. It should also be able to connect to simulators that are already running (e.g. in debug mode on a developer's machine).

In order for mosaik to start and control simulators on foreign machines, there needs to be a service running on each machine that connects to mosaik's main server. That service needs to search for or detect the simulators available on that system. It shouldn't matter in which programming language a simulator

is implemented. The list of these simulators needs then to be sent to mosaik's main server. The service must also be able to start and control the simulators. Distributing the execution of the simulators over multiple machines also helps increasing the scalability of the whole framework. Finally, there should be a way to prioritize theses services. This is, for example, useful if a simulator is available on a machine used by mosaik and, for debugging purposes, also on a developer's local machine.

Mosaik's central server component needs to be able to manage the service instances on the remote platforms (including one for the local machine). It should also be able to execute multiple simulations in parallel. However, the storage of data generated by the simulations should be handled by a single, dedicated process. This process should be the only one performing disk I/O to prevent bottlenecks due to multiple processes trying to write on the same disk.

It should be possible to connect one or more (graphical) clients to the mosaik's central server component to create new simulations and monitor errors and their progress. The client should also be able to retrieve recorded simulation data to analyze or export it.

To sum up, the following major requirements (with respect to the platform manager described in this paper) can be defined:

- Starting simulator processes on different machines
- Connect to running simulator processes on different machines (i.e. for connecting to a simulator running on a developer machine in debug mode)
- Detect available simulators on the different machines
- Proiorize the simulators on different machines (preferences)
- A Single logging process for writing simulator I/O data
- A single server should support multiple clients that perform individual simulations (multi-client capability)
- Clients shall be able to retrieve the data logged by the logging process

3 Guiding Principles

Beside the functional requirements briefly described in the last section, there are also some non-functional requirements that guide mosaik's development.

Mosaik will be a long-term project and people need to work with it on several levels:

- *Core developers* will maintain and improve mosaik's server components that compose and execute the simulation.
- *Simulation developers* will implement mosaik's simulation API for their simulations.
- *Scientists* will need to setup mosaik and the required simulations on one or more servers.
- *Domain experts* will need to create new scenarios and execute them on the mosaik server via a GUI.
- And finally, they need to develop and implement control strategies for multi-agent systems that control the simulated entities via the offered API.

Therefore, one of the most important non-functional requirements is a good and exhaustive documentation as well as easily readable code. Performance is currently not a priority, but since mosaik basically just controls other simulators, it should scale quite well. Since a scenario might contain multiple variants and might be executed several times, it might make sense to also distribute its execution over multiple processes. This would allow mosaik to use a lot of CPU cores, a lot of RAM and to be relatively easy distributed over multiple machines.

We chose Python 3 to implement mosaik. Python is an open and platform-independent dynamic programming language. One of its key distinguishing features is its "very clear and readable syntax" [11]. Next to Python's comprehensive standard library, there are a lot of Open Source scientific libraries available. Koepke [5] points further reasons why Python is very well suited for scientific applications. Milano [7, pp. 39] discusses its applicability and performance for power system analysis.

Mercurial is used for source code version control. As a decentral version control system, it's more flexible and also faster than central revision systems. Compared to Git, it has a strong focus on simplicity [8] while Git is somewhat harder to learn.

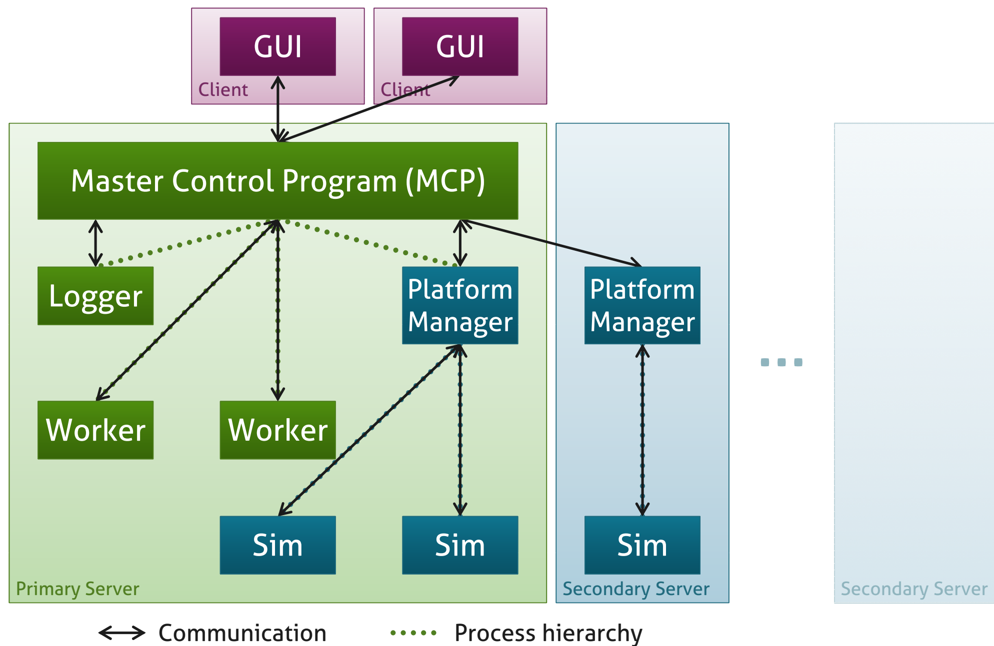

Figure 4: The overall architecture of mosaik. In addition to a primary server mosaik runs on, there can be additional secondary servers for simulators. Clients can connect to mosaik to add scenarios and simulate them.

4 Overall Architecture

To utilize the large amount of CPU cores and RAM modern servers provide, mosaik is designed as a ZeroMQ-based distributed system. ZeroMQ (also ØMQ or ZMQ) is "a socket library that acts as a concurrency framework" [3]. Piël [10] writes: "ZeroMQ is a messaging library, which allows you to design a complex communication system without much effort." ZeroMQ does that by providing socket types for various communications patterns (e. g., Request/Reply, Publish/Subscribe or pipelines) but also allows a complex routing of messages via various processes. It is also very fast and implementations exists for various languages, including C/C++, Python, Java and .NET [4]. Mosaik uses ZeroMQ's Python bindings which are called *PyZMQ*[2].

The simulators mosaik uses can be distributed over several servers, but there needs to be one primary server which manages all subprocesses and to which clients can connect. We call that central process the *Master Control Program*

[2] http://zeromq.github.com/pyzmq/

(MCP). However, the MCP does not control the simulators directly (and could not start simulations on remote machines at all). Starting and controlling simulators is handle by the *Platform Managers (PMs)*. The MCP starts one on the local machine. PMs on remote machines need to be started manually and connect themselves to the MCP. The composition and execution of a simulation is done by a *Worker* process. Worker processes are started by the MCP each time the user starts a simulation. They automatically shut down when the simulation has finished. Storing simulation data and log messages is done by a single *Log Manager (LM)* process to avoid concurrent disk access by multiple processes. One ore more (graphical) clients *(GUI)* can connect to the MCP in order to upload new scenarios, start simulations or retrieve simulation data. Figure 4 depicts the process hierarchy (dotted lines) and which processes communicate with each other (arrows).

The following sections describe the architecture of the processes in more detail.

5 Process Architecture

All mosaik processes comprise three layers as shown in figure 5. The lowest layer contains the event loop and one or more PyZMQ streams (which are basically wrapped ZMQ sockets so that they can be used with PyZMQ's event loop). On top of that, there is a message handler for each stream. The message handler deserializes incoming messages, performs message routing and sends outgoing messages based on the results of the actual application logic. The application logic is located in the third layer and is completely ZMQ-agnostic.

5.1 Master Control Programm

The MCP is—socket-wise—the most complex of mosaik's processes (see figure 6). It has distinct sockets for communicating with GUIs, PMs, Workers and the Log Manager to separate message handling code and to make it harder for (evil) clients to pretend to be a PM or another internal process. This also allows for a fine granular configuration of the server's firewall. For example, the two ports that GUIs connect to could be opened for a wider range of IP-addresses and the two ports for PMs only for a few selected machines, while the remaining ports for the Workers and the Log Manager are completely blocked and only available from the local machine.

Communication with (graphical) clients, like adding or deleting scenarios, is

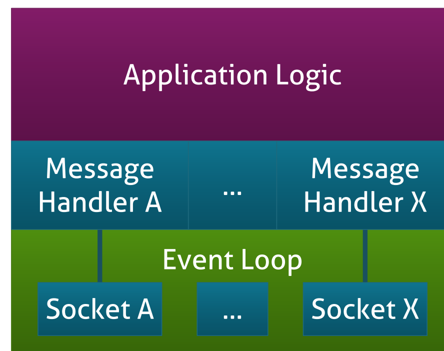

Figure 5: Mosaik's processes comprise a simple three-layer architecture. The two bottom layers are PyZMQ-specific and perform message retrieval and handling, while the application logic in the third layer is completely PyZMQ-agnostic.

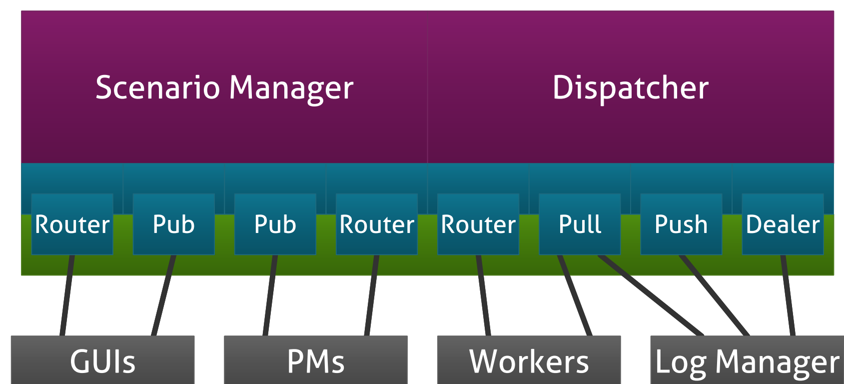

Figure 6: The MCP manages the scenarios uploaded by the users and dispatches commands between the Workers and PMs (and their simulators).

done via a *router* socket. The router socket can asynchronously receive requests from clients and remembers which client to sent the reply to, once it is available. A *pub* socket is used to broadcast the progress of a simulation or the list of available scenarios to all currently connected clients.

The scenarios (or *simulation studies*, as they are called in mosaik's scenario specification formalism) are managed by the *ScenarioManager*. It also starts the worker processes when a client requests to starts a simulation. A simulation study can contain multiple scenario variants (e. g., multiple simulation time periods like summer and winter). Each of them can be simulated multiple times to account for elements of probability in stochastic simulations and to determine the variation and probability distribution of the simulated scenario [6]. The execution of one repetition of a scenario variant is called *experiment*, so there are $|variants| \times |repetitions|$ experiments for each simulation. One worker process is started for each experiment.

Another *pub* socket regularly sends a heart-beating signal to connected PMs. PMs that receive that signal should respond with a list of their available simulators and their priority, so that the MCP knows which simulators are globally available and which PM has the precedence over others if several offer the same simulators. Via a *router* socket, the MCP forwards messages (e. g., commands to start a certain simulator) from a Worker to the proper PM.

The communication with Workers also happens mainly via a *router* socket. A Dispatcher handles the routing of messages between Workers and PMs. Simulation data and log messages are received via a *pull* socket since no reply is necessary in this case. The MCP forwards log messages and simulation data via a *push* socket to the Log Manager.

The Log Manager itself also pushes its log messages to the MCP. This may seem unnecessary, but the MCP broadcasts all log messages to connected GUIs and thus needs to receive the messages from the LM. Clients' requests for simulation data are sent via a *dealer* socket to the Log Manager.

5.2 Platform Manger

The PM is the link between simulators and mosaik's MCP. An instance of it must be running on each machine on which a simulator should be started. The PM's Sim Manager (see figure 7) starts and stops simulators and dispatches messages between them and the MCP.

The PM receives the MCP's heart-beating signals via a *sub* socket and replies via a *dealer* socket with its priority and a list of its locally available simulators.

If it starts simulators for a worker, it keeps an internal mapping of the

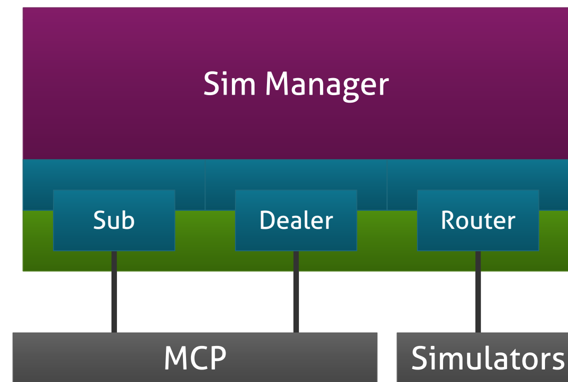

Figure 7: The Platform Manager manages the simulators available on its machine. It can start/stop them and forwards commands sent from a Worker via the MCP to running simulators.

Worker's UUIDs[3] and the UUIDs of the simulators associated with that Worker which is required to correctly route messages between them.

Commands for the simulators are also received via the *dealer* socket and forwarded to the correct simulator via a *router* socket. The simulator's reply takes the same way in the opposite direction. When a Worker signals a PM that its simulation it done, the PM sends stop messages to all simulators associated with that Worker to allow them a clean shutdown. Simulators that don't stop cleanly after a few seconds are being killed.

5.3 Worker

Workers (figure 8) execute an experiment. They parse the scenario configuration, compose the simulation based on the given variant of the simulation study defined in the scenario configuration and coordinate the simulators that are needed for the simulation.

Once the setup and composition of an experiment is done, the Worker sends a list of required simulators to the MCP via its *dealer* socket. The MCP determines the responsible PMs and sends start requests to them. The PMs signal the MCP when their simulators are started. When all PMs are done, the MCP signals the Worker that it can start the simulation.

A Worker can send a command to multiple Simulators at the same time, so that the simulators can execute them in parallel. Their replies are sent back to

[3] ZeroMQ uses UUIDs to unambiguously identify remote peers of a socket and to allow the routing of messages via multiple nodes.

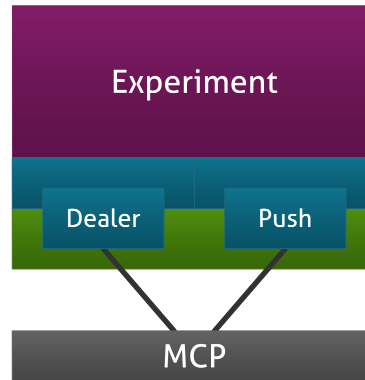

Figure 8: The Worker performs the scenario composition and manages the execution of the simulation.

the worker, one at a time, when a simulator is done executing the command. Depending on e. g., the execution schedule for the simulators, the Worker may or may not sent further commands to (other) simulators when it receives a reply from a simulator. That is because the experiment engine doesn't have an active event-loop and only gets triggered by the PyZMQ event-loop when it receives a new message. That is no problem, because the experiment engine always sends all currently possible commands at the same time and would have to wait for a reply anyways.

During the execution of the experiment, the worker regularly pushes simulation data and information on the progress of the simulation to the MCP via its *push* socket.

When the simulation is done, the Worker sends stop messages to all simulators and shuts itself down.

5.4 Log Manager

The Log Manager (figure 9) serves as a central database for simulation data and log messages to minimize concurrent disk I/O.

It receives data via a *pull* socket and stores it in a database backend. Queries from a GUI (which are routed via the MCP) are handled with a *dealer* socket. Log messages that are produced by the Log Manager itself are pushed via a *push* socket to the MCP so that it can forward them to the clients.

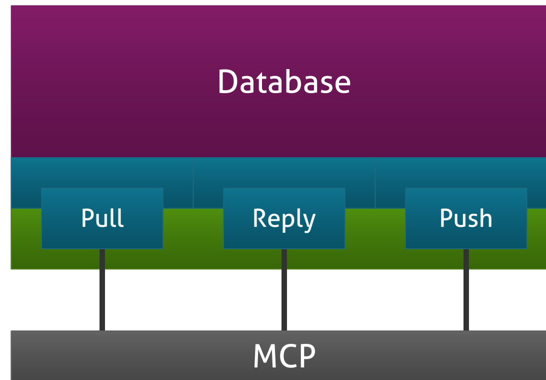

Figure 9: The Log Manager serves as a central database for simulation data and log messages.

6 API for Simulators

In order to be used with mosaik, a simulator needs to provide a self-description file as explained in [14] and it needs to implement mosaik's API for simulators. There is a low- and a high-level API. The former uses ZeroMQ as transport layer for messages and JSON to encode the messages. It is available for all languages that ZeroMQ supports. The latter offers a base class that encapsulates the low-level API and a method to start the simulator. The high-level API is currently available for Python, but we are planning to implement one at least in C and Java as well. Figure 10 depicts the differences between the two API levels.

6.1 The Low-Level API

Simulators communicate with a PM via a *request (REQ)* socket. That means that they send requests for a new command to a PM and the PM replies with a new command to execute. This is a little bit counter-intuitive, since the simulator is actually a "server" and would thus normally use a *reply (REP)* socket and answer requests (commands) from the PM with the return value of that command.

There are two reasons for this behavior. First, it would be unpractical to let the simulator bind itself to a port. There may be several instances of that simulator running in parallel, so which port should it bind to? And how would the PM know that port? So the PM binds a *router* socket to a known port which all simulators can connect to. However, the problem with this is, that the PM

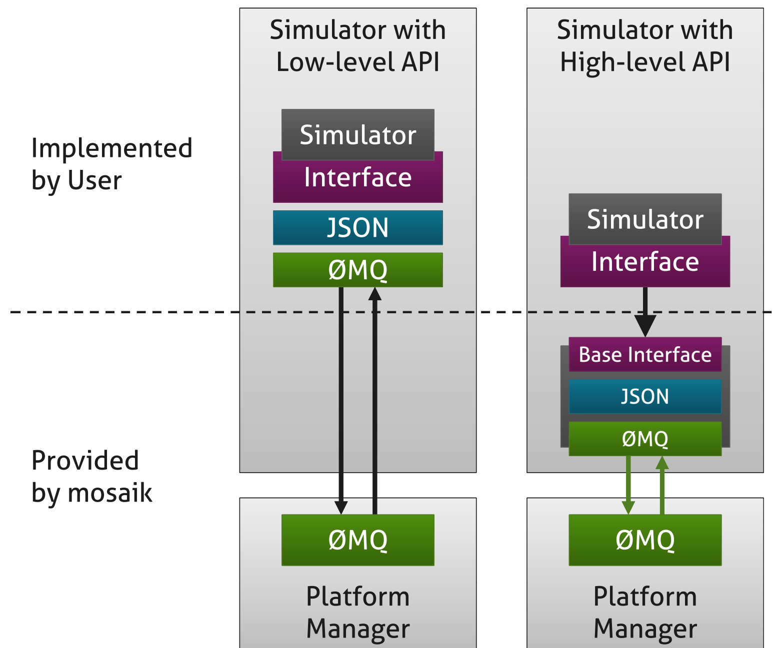

Figure 10: Mosaik offers two API levels. The low-level API allows (or requires) you to directly work with a ZeroMQ socket and manually de-/encode messages with JSON. The higher level API offers a base class that you can simply inherit from, but is available only for a few platforms.

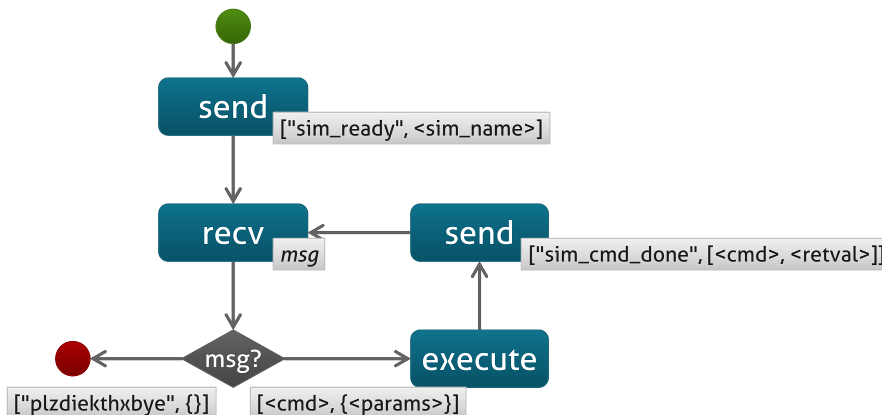

Figure 11: The *sim_ready* message tells that the PM the simulator named *<sim_name>* is ready to receive commands. *<sim_name>* must be the same name as in the simulator self-description. As long as the simulator receives no stop message, it executes a *<cmd>* and sends back its result.

cannot poll a simulator if it is ready, since a *router* socket works like an extended *reply* socket and thus only answers requests and needs an address to which to send the reply to. To solve that problem, the simulator sends a ready-message with its name in order to signal the PM that it is ready to accept commands and to let the PM know its socket UUID.

After the ready message has been sent, there may be an arbitrary number of cycles where the PM sends a command and the simulator replies with its results. The final command the PM sends is a stop message after which the simulator should shut itself down. Figure 11 depicts the communication between the PM and a simulator from the simulators point of view.

Note, that in contrast to the XML/RPC protocol, all communication with a simulator is completely asynchronous. That means that the PM can trigger multiple simulators in parallel and collect their results once they are available.

As figure 11 suggests, there are multiple commands the simulator can receive, each of which must be answered with an appropriate *sim_cmd_done* message:

cmd: "init"

The *init* message is the first command that the PM sends and it is sent only once. It is used to configure the simulator and to setup the model instances.

params: "step_size": <int>, "sim_params": <object>, "model_config": <list of tuples>

The *step_size* parameter defines how many minutes (in simulation time) the simulator needs to simulate on each *step* command (see below). The *sim_params* parameter is a JSON object (or *dict* in Python terminology) with various additional parameters which were defined in the simulators self-description. Finally, the *model_config* parameter contains four-tuples *(cfg_id, model_name, num_instances, params)* which configure a number of instances for a certain model that the simulator provides.

retval: [[<cfg_id>, <model_instances>], ...]

The return value of the *init* command is a list of tuples. Each tuple contains a *cfg_id* and a list of model instances [instance_entities +], with *instant_entities* = [e_tuple+] and *e_tuple* = (e_eid, e_type). The (Entiy-ID, Entity-Type) tuples represent the instances (entities) of a model.

cmd: "get_relations"

This is a request to retrieve the relation between the entities of the simulator.

params:
This message has no parameters.

retval: [[<eid1>, <eid2>], ...]

The return value is a list with a tuple for each relation, where the tuple contains the IDs of two related entities.

cmd: "get_static_data"

This message is a request to retrieve all static attributes for all entities. Static attributes are attributes that don't change during the execution of the simulation.

params:
This message has no parameters.

retval: [[<eid>, {"<attr>": <val>, ...}], ...]

The return value is a list of tuples, where each tuple consists of an entity ID and the static data for that entity.

cmd: "get_data"

This is a request to get the values for a number of attributes of an entity within a model. This methods returns the current values for all attributes in *attributes* for all *etype* typed entities of the model *model_name*.

params: "model_name": <string> , "etype": <string>, "attributes": [<string>, ...]

model_name and *etype* are the names of a model and an entity type within that model. *attributes* is a list of attribute names of that entity type.

retval: [["<eid>", {"<attr>": <val>,..., ...]]|

The return value of that message is a list with a tuple for each entity. Each tuple consists of the entity ID and a data object.

cmd: "set_data"

Sets the attribute values for a list of entities.

params: `"data": [["<eid>", {"<attr>": <val>, ...}], ...]`

The value of *data* is a list with a tuple for each entity. Each tuple contains the entity's ID and an object with the data to set.

retval: `null`

cmd: `"step"`

Advances the simulation by *step_size* steps (as defined in the *init* method) and returns the current simulation time.

params:

The *step* command has no parameters.

retval: `<int>`

The current simulation time should be returned.

6.2 The High-Level API

Currently, a high-level mosaik API is only available for Python, so the remainder fill focus on this implementation. However, APIs for other Languages (e. g. Java) will expose the same functionality in a similar way.

The high-level API automatically creates the required socket, connects to the PM and starts a simple event loop that sends and receives messages and (de)serializes their contents. Each *cmd* is mapped to a method with the same name and the contents of the *params* object (which will be a plain `dict` after the deserialization) are passed as *keyword arguments* (named parameters). For example, the message

```
["init", {"step_size": 15, "sim_params": {},
          "model_conf": []}]
```

will result in a call

```
init(step_size=15, sim_params={}, model_conf=[]).
```

The return value of that function is directly used for the *retval* placeholder in the simulator's reply to the PM.

You only need to implement an interface with the according methods that makes the appropriate calls to your simulator. The API therefore offers a base class, that you simply can inherit from. There is also a method that you can call from your `main()` to start the event loop. Furthermore, you can specify additional command line arguments that your simulator may require. The following listing shows how the API can be used:

```
from mosaik_api import Simulation, start_simulation


class ExampleSim(Simulation):
    sim_name = 'ExampleSimulation'

    def configure(self, args):
        # Here you could handle additional commandline arguments

    def init(self, step_size, sim_params, model_conf):
        # Initialize the simulator and create all entities
        # and return the entity IDs

    # Implement the remaining methods (step, get_data, ...)


if __name__ == '__main__':
    import sys

    description = 'A simple example simulation for mosaik.'
    extra_options = [
        (('-e', '--example'), {
            'help': 'This is just an example parameter',
            'default': True,
        }),
    ]

    sys.exit(start_simulation(ExampleSim(), description, extra_options))
```

7 Conclusion

This paper explained the architecture of mosaik's server side and the API that simulators need to implement in order to work with mosaik. Since the implementation is work in progress, the API might still change in the future due to new requirements. However, we already implemented a first prototype during the *GridSurfer* project [15] that integrated simulators for electric vehicles, PV and households as well as a static load flow analysis.

The new and overhauled architecture worked out quite well so far and the processes that are already implemented are well tested and documented. We are also planning to provide implementations of the high-level API for additional languages like C or Java.

This paper may be updated to adjust to changes of the API or the implemen-

tation of mosaik.

References

[1] Jan D. Gehrke, Arne Schuldt, and Sven Werner. Designing a Simulation Middleware for FIPA Multiagent Systems. In *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, pages 109–113. IEEE, December 2008. ISBN 978-0-7695-3496-1. doi: 10.1109/WIIAT.2008.202.

[2] G. Hemingway, H. Neema, H. Nine, J. Sztipanovits, and G. Karsai. Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. *Simulation*, 88(2):217–232, March 2011. doi: 10.1177/0037549711401950.

[3] iMatix Corporation. Ømq – the intelligent transport layer, 2012. URL http://www.zeromq.org/.

[4] iMatix Corporation. Ømq language bindings, 2012. URL http://www.zeromq.org/bindings:_start.

[5] Hoyt Koepke. 10 reasons python rocks for research (and a few reasons it doesn't), 2010. URL http://www.stat.washington.edu/~hoytak/blog/whypython.html.

[6] MathWorks. MathWorks Deutschland – Simulating Models – MATLAB, 2012. URL http://www.mathworks.de/help/simbio/ug/simulating-models.html.

[7] Federico Milano. *Power System Modelling and Scripting*. Springer, London, 1st edition, 2010.

[8] Bryan O'Sullivan. Mercurial: The definitive guide, 2009. URL http://hgbook.red-bean.com/read/how-did-we-get-here.html.

[9] Mikel D. Petty and Eric W. Weisel. A formal basis for a theory of semantic composability. In *Proceedings of the Spring 2003 Simulation Interoperability Workshop, Orlando, FL, April*, 2003.

[10] Nicolas Piël. Zeromq an introduction, 2010. URL http://nichol.as/zeromq-an-introduction.

[11] Python Software Foundation. About python, 2011. URL http://www.python.org/about/.

[12] Steffen Schütte. Composition of simulations for the analysis of smart grid scenarios. In *Energieinformatik 2011*, pages 53–64. Prof. Dr. Dr. h.c. H.-Jürgen Appelrath, Clemens von Dinther, Lilia Filipova-Neumann, Astrid Nieße, Prof. Dr. Michael Sonnenschein and Christof Weinhardt, 2011.

[13] Steffen Schütte, Stefan Scherfke, and Martin Tröschel. Mosaik: A framework for modular simulation of active components in smart grids. In *1st International Workshop on Smart Grid Modeling and Simulation (SGMS)*, pages 55–60. IEEE, 2011.

[14] Steffen Schütte, Stefan Scherfke, and Michael Sonnenschein. mosaik – smart grid simulation api. In Brian Donnellan, João Peças Lopes, João Martins, and Joaquim Filipe, editors, *Proceedings of SMARTGREENS 2012 - International Conference on Smart Grids and Green IT Systems*. SciTePress, 2012.

[15] Martin Tröschel, Stefan Scherfke, Steffen Schütte, Astrid Nieße, and Michael Sonnenschein. Using electric vehicle charging strategies to maximize pv-integration in the low voltage grid. In *6. Internationale Konferenz und Ausstellung zur Speicherung Erneuerbarer Energien (IRES 2011)*, 2011.

[16] Bernhard P. Zeigler, Tag G. Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, New York, 2nd edition, 2000.