A Lightweight Distributed Software Agent for Automatic Demand—Supply Calculation in Smart Grids

Eric MSP Veith[1], Bernd Steinbach[2], and Johannes Windeln[3]

[1,3]Institute of Computer Science
Wilhelm Büchner Hochschule
Pfungstadt, Germany
e-mail: eric.veith@wb-fernstudium.de

[1,2]Institute of Computer Science
Freiberg University of Mining and Technology
Freiberg, Germany
e-mail: veith@informatik.tu-freiberg.de

***Abstract*—The number of renewable energy sources participating in the world-wide energy mix is increasing. However, they come with different characteristics than the traditional power sources. Energy generation happens on a smaller scale and is more distributed, often because the location of such a power generator cannot be freely chosen. Also, some sources like wind or solar power depend on the weather, which is not controllable. This poses more difficult challenges on every grid management. We propose a distributed, self-adjusting agent-based solution for smart grids. Based on a lightweight protocol, this distributed software will dynamically and pro-actively calculate supply and demand within the smart grid.**

***Keywords*—smart grid; messaging; protocol description; agent design; renewable energy sources.**

I. Introduction

Two major parts contribute to the success of a distributed agent software. First, the software itself, which must work and act correctly. Second, a proper method of communication must exist between any two agents. This does not only include the information interchange itself in terms of encoding, but also the correct behavior when sending or upon reception of a message.

Therefore, the ground work for any distributed software is the communication between the instances that are formed by deploying the software. In [1], we have outlined a protocol that focuses on the problem at hand: A distributed, i.e., non-centralized, supply-demand calculation.

This completely distributed supply-demand calculation is the primary goal of the architecture we propose in this article. In his article "integration is key to smart grid management" [2], J. Roncero shows how different technologies are integrated in the rather abstract smart grid concept. Including the customer via smart metering is typically considered one of the cornerstones of the smart grid. However, the increasing number of renewable energy sources with either a lower power output than a traditional power plant or a not even completely controllable output (e.g., a wind farm) will also introduce more control logic at the producer side.

Considering a country such as Germany, an already high number of 3841 wind farms [3] are controlled from only a few control centers, which oversee a part of the transmission net. Figure 1 shows control centers in Germany. Including smaller, also distributed energy-generation appliances along with photovoltaic and other renewable energy sources puts an increasing management strain on these control centers since along with the number of small generators the data volume also increases.

Figure 1. Control Centers in the power transmission system

Distributing control logic along with distributed energy generation is often proposed as a solution to this problem. Several architectures exist, such as the one described by Lu and Chen [4]. In these designs, the concept of microgrids often plays an important role. Aggregating small distributed energy generator along with consumers in a microgrid that acts as one unit to the rest of the grid is considered necessary [5]. This still views energy generators as singular blocks within the grid that yield a more or less constant behavior or can even work in island mode, completely disconnected from the rest of the power grid. While this accommodates the "central

TABLE I. Share of renewable energy sources in Germany's energy mix in 2012 [7]

Type	Power Produced [GWh]	Percentage of Total Production [%]
Water	21,200	3.6
Wind (on- and offshore)	46,000	7.7
Photovoltaic	28,000	4.7
Biogas	20,500	3.4
Geothermal Energy	25.4	0.004

control"-approach, it is also an argument for a less flexible management of the power grid.

We propose an architecture that enables every consumer and producer node in the smart grid to communicate. The primary goal is to create a self-organized smart grid allowing for greater flexibility and a more efficient usage of dynamic power sources such as wind power or photovoltaic while reducing the information load for central control facilities.

II. Motivation

The number of renewable energy sources increases steadily. For the European Union, a goal of 20%, 30% and 50% for the years 2020, 2030, and 2050, respectively has been fixed [6]. Since wind turbines and photovoltaic panels are relatively easy to set up compared to other renewable energy sources, they already contribute the biggest part of power generated from renewable energy sources (see Table I for Germany). However, they are dependent on a source of energy that is not controllable by mankind, i.e., the weather.

Having a wind farm permanently connected to the power grid means that it feeds in a greatly variable amount of energy, as can be seen in Figure 2. This contrasts with the demand of a stable power supply. In order to integrate renewable energy sources more tightly, the grid needs to accommodate this dynamic energy generation characteristic. Also, wind farms are raised at positions providing strong and steady wind currents, which is typically not ideal regarding the spread of the power grid. This and smaller, local power generators lead to a distributed generation, a paradigm shift considering traditional energy generation.

One approach is to aggregate distributed generators and nearby consumers into microgrids [4] or to compensate variations in power generation by using buffers, i.e., batteries. In [8], Vasirani *et al.* use electric vehicles as buffers that form a virtual power plant (VPP) together with the wind farm.

These approaches still assume homogeneous behavior as central attribute of the power grid and its connected devices and thus favor the traditional, simplified "base load"-view. If, however, both consumers and producers act together in a grid-wide planing phase of a short period's load profile, we assume that a more dynamic profile will emerge that allows a more efficient inclusion of renewable power sources.

In the past, the initial approach has been to place smart meters at the customers' side in order to exercise indirect control of their energy consumption. Providing dynamic rates and information feedback to the customer should contribute to

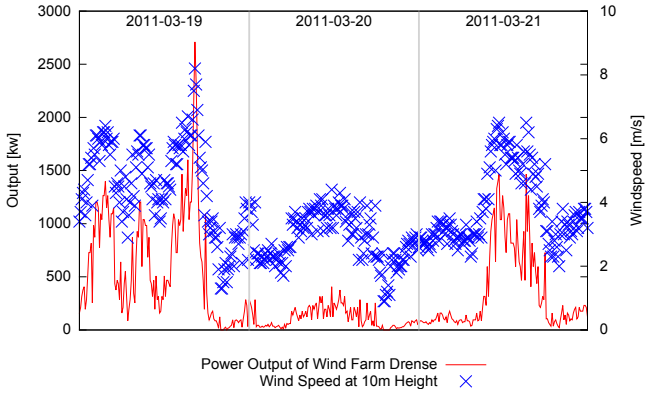

Figure 2. Modelled Output of Wind Farm *Drense* in the County of Brandenburg, Germany, and the corresponding Wind Speed during the same time

a more efficient use of energy [9]. Field studies have offered mixed results, with some even suggesting that few consumers change their behavior [10], [11]. One can also argue that electricity should be a "when you need it" resource instead of something that must be conserved.

Still, industrial consumers, i.e., factories, could accommodate to changes in the availability of energy. Also, the behavior of a large or a group of customers such as a neighborhood can yield valuable information. If a wind farm also becomes a smart node within the power grid, the additional information can be used to plan power supply more effectively.

However, this requires to introduce forecasting since we cannot control a wind farm or photovoltaic panels the same way as an operator is able to control a traditional power plant. Specifically, there is no possibility to increase power output when there is no wind blowing or sun shining. With an increasing numbers of electricity producers based on renewable energy sources, we rely more and more on a power source completely outside of our control.

Planning beforehand will therefore help to integrate these renewable energy sources better since the smart grid will be able to match a partly controllable power generation with customer behavior beforehand. This also includes a grid-wide, distributed calculation of energy storage.

To this end, we propose a protocol that defines the ground rules for such a distributed system to work. The protocol's information fields are defined by the necessity to follow these rules instead of the wish to query information. This provides us with a lightweight core allowing software agents in a smart grid to create short-lived contracts on the fly. Thus, all agents participating in this protocol act pro-actively; a consumer node is no longer a database that is queried from remote. Instead, it signals an increase or decrease in power consumption beforehand. This, in turn, triggers the mentioned supply-demand calculation.

The reminder of this article is structured as follows. In Section III we survey related work, especially paying attention to technologies useful to reaching our goal. Section IV outlines the design considerations shaping the actual protocol. The

then following Section V specifies the basic rules every node implementing our protocol must follow in order to properly do so. We then detail the actual message types in Section VI. In order to test the rules we define as inherent part of the protocol, we propose a modular agent architecture in Section VII. Although it is only a high-level view of the architecture, it identifies the important parts of an agent implementing our protocol and serves as a implementation-agnostic test case notation that we introduce in Section VIII along with a test driver proposition for an actual implementation. Selected test cases are then outlined in Section IX. We discuss all presented parts in Section X before concluding and outlining planned future work in Section XI.

III. Related Work

Several protocol proposals have been designated usable or even specifically developed for use in a Smart Grid scenario.

The *Scalable and Secure Transport Protocol* (SSTP) [12] by Kim *et al.* uses the existing IP-networking [13] infrastructure. It addresses security and durability against attacks as well as resource usage. The latter is especially important in the case of sensor hardware, where SCTP [14] alone is already too heavyweight and IPSec/TLS for encryption add to this burden. SSTP is also state- and connectionless for the same reason.

Although the authors of SSTP designate it as a "protocol for Smart Grid data collection", they do not explicitly specify data structures specific to devices or device groups. SSTP resides in layer 4 of the ISO/OSI stack just as SCTP or TCP [15] do.

The Open Smart Grid Protocol (OSGP) [16] offers a bit-by-bit protocol design that also includes transmission security. It allows remote querying of devices, mostly smart meters, which offer a virtual table-based interface. The OSGP does not utilize existing communication infrastructures such as IP networks.

The ISO/IEC 61850 standard for substation automation has also been successfully used in a smart grid scenario by Zhabelova and Vyatkin [17], where a substation shows self-healing capabilities due to a multi-agent approach that allows to detect errors and work around them.

The multi-agent approach has been chosen for several problems in the context of the smart grid. Pipattanasomporn *et al.* design and implement a multi-agent system for microgrids in [18]. In their design, the agents perform different tasks based on their roles, thereby breaking the complex problem of centralized management of a microgrid into smaller problems. Their system uses the IP. A similar approach is chosen by Oliviera *et. al* in MASGriP [19]. The decision-finding process among agents in the latter case is based on market competition.

In fact, agents competing in a market scenario is often chosen as a way to motivate an agent's behavior. The market will, so the premise, be the basis for demands and supply, and market competition will allow agents to have a scale for evaluating offers in order to select the "best" one. In fact, a price forms a simple yet effective "fitness value" for a goal-oriented behavior of agents. Hommelberg *et al.* go as far as to call automatic markets an "indispensable feature of smart power grids" [20]. For reasons we discuss in Section X, we advise against this.

Many distributed agent approaches are based on the Contract Net Protocol proposed by Smith [21]. The semantics, however, differ in the understanding of how work packages should be handled. For the contract net, a work item *can* be awarded to another node; however, each node is free to offer its services as it deems fit. This can not prevail in the power grid, where shortages *must* be handled by each agent. Considering any problem as essential for each agent is essential for all nodes to survive. Also, we do not propose any pre-selection of nodes since all agents should be able to participate in the global solution-finding process equally.

It is also important to note that many multi-agent approaches that are—directly or indirectly—based on the original Contract Net Protocol have the notion of one atomar work item that can be awarded. In a smart grid, situations will arise where a node cannot fulfill the whole contract, but only a part of it. Breaking the work package into smaller sub-packages, however, is the responsibility of the offering node. In the smart grid, this would lead to an increased amount of negotiation for the "right" work package size.

This detail excludes most service discovery protocols per se, as they are not designed offer "half a printer".

IV. Protocol Design Criteria

In striving to be as simple as possible, our protocol uses already existing technologies. This has let us to choose the ISO/OSI stack model as basis for our design, where it can be placed on the application layer (layer 7) of the stack model. This design choice allows us to draw upon the strengths of already existing infrastructure used for transport via the Internet Protocol (IP). Utilizing IPv6 [13], we gain an address space large enough for our needs.

Choosing the ISO/OSI protocol stack model also helps to integrate other technologies. We can choose between TCP/IP with IPsec for security, or SSTP, which has been specifically outlined in Section III for this purpose.

Integrating hardware in our system is possible as long as it can be attached to an IP network. Thus, we do not need to accommodate to vendor specifics and have access to a wide range of hardware through layer 2–3 protocols. For example, remote locations can use GSM or UMTS links [22] to exchange information with other nodes within our distributed system.

Nodes within the grid exchange data via *Connections*. We explicitly introduce the connection concept since it is a virtual concept not directly given by IP networks unless utilizing TCP or another connection-oriented transport protocol is explicitly chosen. Since we have already offered SSTP as possible transport protocol, we introduce the connection concept.

Here, a connection means that two nodes are known to each other; as long as message boundaries can be preserved and a loss reduction algorithm is in place, the choice of the transport protocol is up to the implementor. SSTP is therefore suitable as layer 4 protocol for our own. Connections must

be established only between two adjacent nodes and are bi-directional communication channels between exactly these two nodes. Concepts such as multicast must be realized on top of this. There is no explicit connection between two distant nodes, i.e., there is no end-to-end connection concept that crosses several hops such as TCP offers on top of IP nodes.

A connection serves three purposes. First, it identifies the two endpoints. Second, by establishing a (largely virtual) network of nodes and connections, this protocol creates a communications structure that resembles the actual power grid, recreating it on top of any other networking structure, such as an IP-based wide-area network (WAN). This way, the power grid and the telecommunications infrastructure do not have to match in their layout. The layout recreation algorithm must be implemented by the actual connection facilities, which, e.g., map to an IP network. Third, since one connection always concerns exactly two nodes, it allows us to set individual connection parameters such as compression that do not interfere with other data links to other nodes.

Having those virtual connections represent the actual physical power supply line also enables us to model "dumb" cables, which have no other properties than a maximum capacity and a line loss. Taking these attributes into account, the actual power transfer becomes part of the protocol. Smart power supply lines that are equipped with, e.g., metering devices, become nodes of their own. The simple power line–connection unit then evolves into a connection–power line–connection building block, which also adheres to the protocol semantics described in the following section.

Messages can travel further than the node–connection–node boundary. To enable nodes to answer to requests that do not originate from their immediate neighbors, each node must be uniquely identifiable. The *Sender ID* of a node must be unique at any given time. It is an opaque bit array of arbitrary length and must not contain any additionally information about the node itself or anything else. Generating an Universally-Unique Identifier (UUID) [13] whenever the node's software boots is one way to get such an identifier.

Each message must contain an unique identifier (*ID*). This is important since messages fall into two distinct categories: requests and answers. A request is sent actively by a node because of an event that lies outside the protocol reaction semantics, such as a changed forecast. Answers are reactions that occur because of the protocol semantics as described below. Since any reaction pertains to an original action, it needs to identify this action, which is the reason for the unique identifier of each message. Reactions must carry a new, unique identifier, too, since they are messages of their own.

Identifying individual messages is also important in order to identify duplicates. All messages within our design must be idempotent, i.e., they must yield the same result every time they are received. For example, a request for energy from one node must always lead to an increase in energy production by exactly the amount requested; if duplicates were not identified as such, twice or even many times the amount requested could be fed into the grid. Complex grid structures will eventually lead to duplicates, and so it is essential to identify those.

The type of the message must be denoted by a *Message Type* field. The mapping is outlined in Table II. These numbers are simple integer values with no coded meaning whatsoever. We do not distinguish between message classes or priorities here: The goal of the protocol is to remain simple, and we believe that the message types outlined here suffice in reaching the primary goal of the protocol, i.e., energy supply-demand mediation.

A message must also contain a *Timestamp Sent* field denoting the time and day when the message was initially sent as an Unix Timestamp (see [23] for the definition of the Unix Timestamp).

To prevent messages from circulating endlessly, a Time-To-Live (*TTL*) field is introduced. This TTL has the same semantics as the IP TTL [24] field: It starts at a number greater than 0. Whenever a message is forwarded or sent, the TTL is decremented by 1. If the TTL reaches 0, the packet must not be forwarded or otherwise sent but must be discarded. Messages with a TTL value of 0 may be processed.

The TTL field exists in addition to the IP TTL mechanic: First, because there is no vertical integration of our protocol with the lower-layer protocols. Even though it might seem unusual, other protocols can be used instead of IPv6 that do not offer a TTL field in the same way IP does. Second, our protocol creates an overlay network where a hop from one node to another translates to any number of hops in the IP network, which is even variable depending on different paths chosen by IP-level routers. Thus, we need a separate TTL field in order to preserve the semantics of this protocol.

Additionally, an *Hop Count* is introduced. The Hop Count is the reverse of the TTL: It starts with 0 and must be incremented upon sending a message. It allows to measure the distance between two nodes in the form of hops.

A message must carry an *Is Response* flag to distinguish original requests from responses. If the *Is Response* flag is set, the ID of the original message is contained in the *Reply To* field. If *Is Response* is not set, the *Reply To* field must not be evaluated; however, if a response is indicated, *Reply To* must contain a value that must be evaluated by the receiving system.

An answer must also contain the original message's *Timestamp Sent* field (in addition to its own), and the *Timestamp Received* denoting the time when the original message was received.

To summarize, each message must contain at least the fields of the following enumeration. In parentheses, we give the identifier used in the actual implementation.

1) message ID (`ID`)
2) message type (`type`), see Table II above
3) original sender ID (`sender`)
4) timestamp sent (`sent`)
5) TTL (`TTL`)
6) hop count (`hops`)
7) is response (`isResponse`)

TABLE II. Message Types

Value	Type
0	Null Message
1	Echo Request
2	Echo Reply
3	Online Notification
4	Offline Notification
5	Demand Notification
6	Offer Notification
7	Offer Accepted Notification
8	Offer Acceptance Acknowledgement
9	Offer Withdrawal Notification

The message type defines what additional values a message carries; these message types are described in Section VI. The message type itself is a simple integer value field with type-to-number mapping shown in Table II.

If *Is Response* is true, the following fields must be added:

1) *Reply To* (i.e., original message ID) (`replyTo`)
2) *Timestamp Received* (`received`)

V. Common Protocol Semantics

The following rules must be applied to each message, regardless of their type.

First, a message must not be ignored ("no-ignores" rule). This might seem trivial and obvious, but it is actually an essential rule: If a request for electricity or for electricity consumption would be silently ignored, other nodes would not be able to react since they probably would not even receive the message. This, in turn, would again lead to a "dumb grid" behavior. For this reason, we do not propose a mechanism for resending messages. Here, we differ from the behavior implemented in the internet. The IP routing's best effort approach is dictated by scarce buffer space. If a router's buffer is full, a packet is simply discarded; the original sender usually is not noticed about that. This behavior implicitly leads to a conservation of the node itself. However, the primary goal of a smart grid agent is to preserve the grid and not the agent itself. If an agent fails due to overload, traditional grid protection mechanism will still be available.

All messages except the *Null Message*, the *Echo Request Message* and the *Echo Reply Message* must be forwarded, partially answered and forwarded, or answered. This is the "match-or-forward" rule. It becomes important with requests and offers and it is further specified in Subsections VI-F and VI-G.

Forwarding denotes the general process of receiving a message and resending it. The message may be modified in this process, for example, the requested energy level must be lowered when a node can fulfill a portion of the request (see below).

When forwarding, message must be sent to all connected nodes except to the node from which the original message was received. If the message is an answer, the node may limit the number of outgoing connections to those via which it can reach the addressee. This prevents message amount amplification: Would the receiver also send the message on the connection on which it was originally received, it would be useless since the original sender already knows about its offer or request. It would thus only lead to additional processing and unnecessary use of bandwidth ("forwarding" rule).

Each node must keep a cache of recently received messages. If a message is received again, it must not be answered or forwarded ("no-duplicates" rule). This cache should also be used to forward answers to requests recorded in the cache only on the originally receiving connection.

If a two nodes are connected via more than one connection, only one may be used to send or forward a message. If the sending node can determine the best connection in order to reach its partner, it should choose it. However, what constitutes this "best connection" is hard to define. Many properties a connection between two nodes may have can be attributed to the lower layers of the ISO/OSI stack and, therefore, should not be known to the agent software. If, however, there were two distinct connections between two agents, with one encrypting traffic and the other one featuring low latency, these properties are not comparable in an automatic and quantified way.

Additionally, if an administrator wanted to achieve rendundancy in order to implement a resistance against failures, he can (and should) resort to proven algorithms on the ISO/OSI layers 2, 3, and 4.

We therefore advise that the administrator establishes only one connection between two nodes. As noted above, the Connection concept serves to create an overlay network and to define two connected endpoints (i.e., the agents). If there still remains a wish for redundant connections, a connection priority should be applied, and a lower-priority connection should only be used when the connection with a higher priority is disconnected.

VI. Message Types

These nine available message types constitute the minimum set that is required for the distributed supply-demand calculation. Except for the *Null*, the *Echo Request* and *Echo Reply* messages, all directly contribute to this task.

Any node must be able to signal its online or offline state. This is not only important for scheduled maintenance, but also allows a node to make itself known it its other endpoints.

During normal operation, two main cases must obviously be handled: First, a demand for energy, and second, a possible over-production that is advertised. The latter reason especially applies to renewable energy sources. A wind farm, for example, would signal an increasing power output due to increasing wind speeds.

For an power request, the corresponding *Demand Notification* must be used. It travels through the grid until either its TTL reaches 0, or it is received and answered by another node using a *Offer Notification*. In this case, the *Is Answer* flag is true. However, as stated above, an over-production can also occur, in which case the node will send an *Offer Notification* without having received a message indicating

```
{
    ID: "...eb9e90335495",
    type: 0,
    sender: "...e4d9b83d2bb7",
    TTL: 42,
    sent: 1367846889,
    hops: 23,
    isResponse: false
}
```

Figure 3. An example for a null message, encoded as JSON. The UUID strings have been shorted for clarity.

demand beforehand. The *Is Answer* flag is consequently set to `false` in this case.

What follows in both cases is the actual calculation. The requesting node—whether it requests power or requests the usage of power does not matter—receives offers or demand notifications and now has to decide which offers it takes on. Since there is no limit in how many messages related to the original request may arrive, this explicit "contract-making" needs to take place. This is the reason for the *Offer Accepted Notification*.

An explicit acknowledgment is also introduced in form of the *Offer Acceptance Acknowledgment*. The same offer could have been made to different nodes, requiring the offering node to withdraw all other offers as soon as one node takes it. This is possible using the *Offer Withdrawal Notification*.

In the following subsections, we will describe these message types in more detail, including the additional information fields they introduce along with the corresponding concepts.

A. Null Message

The *Null* message is the simplest message available in the protocol. It contains no additional information besides the basic protocol fields each message carries.

Null messages can be used as a form of heartbeat information. This is especially useful on weak links, for example for a remote wind farm, which might only have a mobile phone (GSM) connection. It thus can be sent at regular intervals to keep the line open.

A Null message in JSON representation is shown as an example in Figure 3. Please note that the message is formatted to be easy to read. In an actual transmission, the JSON string would be compressed, with unnecessary whitespace characters removed.

B. Echo Request Message

An *Echo Request* can be sent on any connection to see if the endpoint is still alive and reachable. It must be answered. An Echo Request must not be an answer, and it also must not contain any additional information.

C. Echo Reply Message

An *Echo Reply* is the answer to an *Echo Request*. It must always be an answer and thus cannot be sent independently. This message type also does not contain any additional information; the proposed common fields (*Timestamp Sent* and *Timestamp Received*) are sufficient for Round-Trip-Time measurements.

D. Online Notification

Using this message, a node in the grid can notify its neighbors that it is going online or will be online at a certain point in the future.

To actually be able to carry the second kind of information, i.e., going online at a certain point in the future, this message contains two additional fields: *Valid From* (`validFrom`) and *Valid Until* (`validUntil`). A message using validity dates must use the *Valid From* field and may optionally make use of the *Valid Until* field.

This concept of validity dates is used by other message types, too. It denotes a timespan between the time indicated by *Valid From* and *Valid To*, both inclusive. Both fields are Unix timestamps like, e.g., the *Timestamp Sent*. Whenever a node wants to indicate that a message is valid immediately, it places the current time and date in the *Valid From* field. A "valid until further notice" semantic can be achieved by omitting the *Valid Until* field entirely.

Any protocol implementor, however, must take care to adjust his implementation whenever the Unix Timestamp data type changes. As the time of writing, a Unix Timestamp of 64 bit width is typically used in modern operating systems, which provides enough seconds since midnight 1.01.1970 (UTC) for the whole lifetime of this protocol. Previously, the `time_t` C type was specified as having 32 bits, which meant that an overflow would happen on 19.01.2038, the so-called "year 2038 problem".

Note that the Unix Timestamp also allows for negative values to represent times before 1.01.1970. Although this would not be a necessary feature in the terms of this protocol, we advise against choosing an unsigned type as it would introduce the need for additional programing quirks for implementors.

An *Online Notification* may be forwarded, but can also be discarded. This type of message is important for all directly connected nodes, because it has influence on the wires connecting the originating and its neighbor nodes. Any change in power levels, however, will be communicated using demand/supply messages, which will be described later.

E. Offline Notification

The *Offline Notification* is the counterpart of the aforementioned *Online Notification*. It notifies the neighboring nodes that the originating node will be offline (i.e., possibly disconnected from the grid), utilizing the same *Valid From* and *Valid To* Timestamp fields. For all purposes of the protocol, especially for complying with the "match-or-forward" rule, the Connection to the node originally sending the Offline Notification must be considered as inactive.

Unlike the *Online Notification*, this type of message must be forwarded. It provides additional information to the energy supply/demand solving algorithms of other nodes, which get a chance to re-calculate their supply plans. It is assumed that a demand or supply message that reaches the node sending

the *Offline Notification* means that the *Offline Notification* will also be received by the original sender of the demand/supply message since the Hop Count is the same both ways.

However, since the *Offline Notification* message does not contain a field supplying the change in the grid energy level when the shutdown happens, an additional supply/demand message must be sent if the node has influence on the grid's energy level.

F. Demand Notification

A *Demand Notification* message indicates the need for energy of a particular node. It carries the *Valid From* and *Valid To* fields.

Primarily, it carries the quantified demand for energy in watts in the *Power* (`power`) field. Fractions of watts are not supported, i.e., the lowest amount that can be requested is 1 W. The field must not be 0, as this would make the message itself superfluous. This field must not carry negative values; those would mean an offer, which has its own message type.

This message type additionally features the *Answer Until* (`answerUntil`) field, which holds the date and time the requester can, at the latest, meaningfully incorporate an answer into its internal planning phase. This bit of information accommodates a wide range of different constraints that apply to a demand/supply calculation such as the time it takes to spin up turbines or scale down production in case a request is not answered as desired. This field must be present, and it must contain a time that is before the one contained in the *Valid From* field. At the time and date *Answer Until* indicates, the internal process of solution finding may begin; definite answer must not be sent until this time has arrived.

An answer arriving after the given date must not be considered by the requester. If a successful solution to the original request can not be found by offers from other nodes alone, the passing of this date and time indicates the need for an internal solution, for example, the deactivation of some turbines within a wind farm.

Demand Notification messages must be forwarded if they cannot be (completely) fulfilled. Each node must try to react to a demand message, i.e., try to match it and supply the energy requested. This is called the "match-or-forward" rule as described above. If it cannot fulfill the demand, it must at least forward it under the semantics outlined in Section IV.

If the node can supply the requested amount of energy completely, it must notify the requester using an *Offer Notification* message. It must not forward the original *Demand Notification* then.

If, however, the demand can only be partially fulfilled, the node must send an *Offer Notification* indicating the amount of energy that can be offered. It must then subtract this value from the original value indicated in the request and forward the thusly modified message. It must not change the message's ID or the message's sender ID ("same-ID" rule). The partial matching described in this paragraph is depicted in Figure 4.

A *Demand Notification* message must not be an answer.

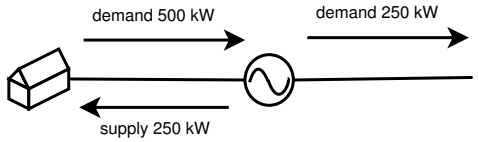

Figure 4. A *Demand Notification* having the "match-or-forward" rule applied

```
{
    ID: "deadbeef",
    type: 6,
    sender: "2d60a262",
    TTL: 42,
    sent: 1367846889,
    hops: 23,
    isResponse: false,
    validFrom: 1367846889,
    validUntil: null,
    answerUntil: 1367846289,
    power: 500000,
    cost: 12
}
```

Figure 5. An example for an *Offer Notification* message that is sent as a request to consume power in order to accommodate to an over-supply of energy. Note the `isResponse` field, which is set to `false` to express this circumstance. UUID strings are shortened for clarity.

G. Offer Notification Message

This type of message indicates an offer to the grid. It carries the fields *Valid From*, *Valid Until*, and *Answer Until* as they are described in Subsection VI-D and the amount of energy offered in the field *Power*. This number is an unsigned integer and is expressed in units of watts with no fractions possible.

Additionally, the offer includes a field *Cost*, which carries the cost of this offer in cents per kilowatt hour (ct/kWh). This allows for implementing cost-based policies, such as accepting energy only if it is cheap.

An *Offer Notification* may be an answer. If so, it is an answer to a previous *Demand Notification*, as described in the above subsection. A node receiving multiple offers must prefer offers of lower hop count over those with higher hop count. This favors micro-grids and reflects the actual flow of energy.

However, *Offer Notification* messages may also be sent as a request. This is the case whenever the agent estimates that it will output more power than it currently does. Consider, for example a wind park, which is dependent on the weather. If the agent's forecasting module predicts an increased wind speed in an hour and therefore an increased energy output, it may send an *Offer Notification* instead of pitching or stalling the wind turbines. This could allow a factories to increase its demand by powering up machines. Figure 5 shows an example of such a message.

Just like a *Demand Notification*, such an original offer must be matched by nodes in the grid. The difference between an original offer and one that is an answer to a request is the

value of the *Is Answer* field: If set to `false`, a node must try to match the offer.

For matching and forwarding, the same mechanics as for the *Offer Notification* message type applies, especially if it can only be partly fulfilled.

H. Offer Accepted Notification

Whenever a request for energy is made and the offers have been received, there may arise a situation when more energy is offered by all nodes than originally requested. For example, if a wind park, a solar park and a traditional power plant send *Offer Notification* messages after a request has been sent, the sum of energy offered is likely to exceed the original amount requested.

For this reason, a node must indicate which offer it accepts. Otherwise, all offers would be fulfilled, leading to an oversupply of energy in the grid, which would be fatal.

As soon as the node finishes its demand/supply calculation, it must send *Offer Accepted Notification* messages to all nodes that were offering energy. In the body of the message, it must list the IDs of those nodes whose offer it takes, using the *Accepted Offers* field (`acceptedOffers`). All other nodes will notice that their ID is missing from the notification and thus not actually deliver the energy they offered.

An *Offer Accepted Notification* must be an answer. It must also be sent by the node that is taking on an original offer (as indicated above). In that case, the *Offer Accepted Notification* must be addressed to the offering node only, while the original offer must be forwarded if it cannot be completely fulfilled as described in Subsection VI-G.

I. Offer Acceptance Acknowledgement

After an offering node has received an *Offer Accepted Notification*, it must reply with an *Offer Acceptance Acknowledgement* to indicate that the offer is still valid. This message type must always be an answer.

J. Offer Withdrawal Notification

If a node has offered a certain amount of energy, be it as an answer or as an original offer, and it can no longer stand up to the offer, it must withdraw it. This type of message is always an answer, carrying the ID of the original offer (in case of an original offer that was withdrawn) or the ID of the original request in the *Reply To* field.

If a node can still offer energy, but the amount has changed, the original offer must be withdrawn using this message type, and the new amount must be separately announced.

VII. Implementing the Distributed Agent

The protocol itself defines the ground rules of a distributed supply/demand calculation. In order to actually test it, however, an implementation adhering to the rules is necessary. Our own implementation consists of several modules, each contributing to a part of the agent's behaviour. We will use the route of an incoming message as common theme for describing all parts of our design, although, of course, information can flow in any direction. We will come to other reasons for action later.

Each agent has exactly one *Messaging Module* that is its interface to the rest of the world. It maintains connections to other agents, and is responsible for receiving and correctly sending messages.

The Messaging Module contains the Duplicate Message Cache. Each message received is first checked against this cache; only if it is not yet stored in this cache will it reach the other modules. Otherwise, it will be silently discarded. It is important to keep this filter in mind during the following paragraphs as every notion of a message will mean an unique sending from another agent.

This message cache is regularly cleared of old messages. For our tests, we have chosen a message retention period of 15 minutes. An implementor can, of course, choose another value. However, he must keep in mind that the Duplicate Message Cache with its retention period is vital to preserve the idempotence of all messages. A period that is too short will harm the grid. The only reason to lower this period is to preserve memory.

Instead of simply "throwing more hardware at the problem" and setting a higher, but fixed time period for message retention, a semi-dynamic cleanup should be implemented. The Messaging Module can infer from Offer Acceptance Acknowledged messages that a particular planning phase has ended and clean up its cache accordingly. Of course, a maximum retention period should still exists as a fall back.

When sending messages, the Messaging Module also takes care to use the correct agent connection, which particularly includes the application of the "forwarding" rule.

An incoming, unique message is then routed to the *Governor*. It has two main tasks. First, during startup, it initializes all other modules, including the Messaging Module, monitors them throughout the lifetime of the agent instance and is finally responsible for resource deallocation on shutdown. Second, it contains the business logic that allows it to act on incoming messages, machine sensor readings or forecasts. The latter one is, obviously, essential to the actual agent behaviour.

Upon receiving a new message, the Governor creates a `Requirement` class instance. Requirements are the building blocks the agent uses internally for bookkeeping and the actual demand/supply calculation. Therefore, it contains two attributes: The actual power delta and the immediately associated messages. The power delta is a deviation from a balanced state of the grid, i.e., it is a relative value. Since it can attain both positive and negative values, the `Requirement` class allows us to treat both an anticipated excess in energy offered as well as an anticipated demand uniformly. Consequently, Demand Notifications and Offer Notifications constitute two different Requirement instances that are matched against each other in the agent's demand/supply calculation.

The requirement class can store power deltas as IEEE 754 [25] *binary32* single precision floating point variables. Variances as they typically occur when using this data type and prompt developers to use `x + 1.0 == 1.0` when

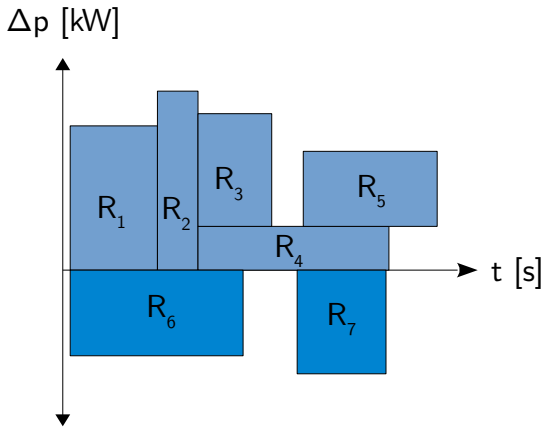

Figure 6. A simplified illustration of the timeline concept

checking whether `x` is 0 or not are small enough as to not harm the grid. However, an implementor should incorporate proper safeguards against over- and underflows.

This is the main reason for the introduction of the classes in the `Winzent::Unit` namespace, such as `KiloWatt`. History has shown that simple floating point or integer variables can lead to a mixing of units in calculations with possibly horrible results, such as the loss of the Mars Climate Orbiter [26].

This calculation is done in the *Supply/Demand Module*. Requirements are inserted into a timeline, which can be viewed as a graph representing the power variances over time. For the module, $\Delta p = 0$ means a balanced grid, but not a power level of 0.0—hence the delta. It might be noteworthy that the Supply/Demand Module at no point has any knowledge of an absolute energy level, but only needs relative levels in order to work.

Figure 6 shows a very simplified, symbolic illustration of the timeline. Requirements form blocks that are inserted into the timeline. The module tries to find the optimal solution within the search room of all blocks. Positive and negative deltas cancel each other until the grid is, from the perspective of the agent, balanced again. From an electric engineer's point of view, a total balance of $\Delta p = 0$ will hardly ever be achieved or even be desirable; we retain this formula for the sake of simplicity but point out that, when deploying, a shift will have to be taken into account.

This search is triggered by the `answerUntil` field of the Offer Notification and Demand Notification messages. Typically, the module has no simple 1-to-1 matching of offers and demands, but can choose from a number of blocks, including the possibility of the agent to adjust itself. For example, a wind farm signalling an offer pro-actively originating from a forecasted increase of wind speed can either try to literally collect Demand Notification messages until the excess energy is used completely, or it can throttle itself and pitch or stall turbines.

This exemplary case also introduces another module within the agent: The *Forecast Module*. Incoming requirements must be matched—or, at least, the agent must try—, and thus the basic question is: "Can we scale up (or down) in order to accommodate the new situation?" The Forecast Module therefore contains the logic of the node's ability to change its production or consumption.

How this is done, depends on the actual node. A traditional power plant will start its own planning phase in order to spin up or down turbines, whereas a wind farm will try to forecast weather conditions. Such a local forecast could be done using Artificial Neural Nets, which have already been proposed and successfully used for weather forecasting, for example in [27], or even in connection to load forecasting [28]. An incorporation of weather forecasting in our agent is still future work as we detail in Section XI.

As we previously noted, the Supply/Demand Module does not have knowledge about absolute numbers. In an ideal world, this is not a problem as the agent's foremost goal is to provide a stable power supply. However, constraints limit the solution space. Such a constraint is hardware-based, e.g., in the form of transformers, which have a limited capacity. All solutions are therefore first checked against the output of the *Constraints Module*. This module also allows administrator interaction, which gives us the possibility to set policies, for example, to not accept a power offering exceeding a certain cost.

As stated, this largely forms the way the agent behaves upon incoming messages. However, this is obviously not the only source of activity for a node—that initial request has to come from somewhere. Within the agent, the Forecast Module continuously creates new projections for the node. Once this forecast has a variance that exceeds a certain limit, it creates a Requirement of its own. The Governor then prompts a new demand/supply calculation, which will, in many cases, yield a deficit, i.e., no solution to the current situation. This, in turn, prompts the Governor to create a request of its own, i.e., a Demand Notification or Offer Notification message, which is sent to other agents.

Although this might seem an obvious course of action, it is not negligible. The continuous forecasting and adjustment of forecast constitutes the very source of our agent's pro-active behavior. Thus, it is not a finite state machine at its heart, but a long-running, stateful software agent.

All parts described live in the `Winzent::Agent` namespace as depicted in Figure 7.

VIII. Testing the Protocol

A. Notation

To ensure that our protocol implementation, or, in fact, any implementation of our protocol adheres to the rules defined in the previous sections, we have created a test suite. This test suite consists of two parts:

1) A written definition for test cases, initial situation and expected results
2) A software implementation of the unit tests

The latter is tied to the implementation that is being tested. We therefore provide the definition of our test cases in order to document how we assert that the behavioral rules defined

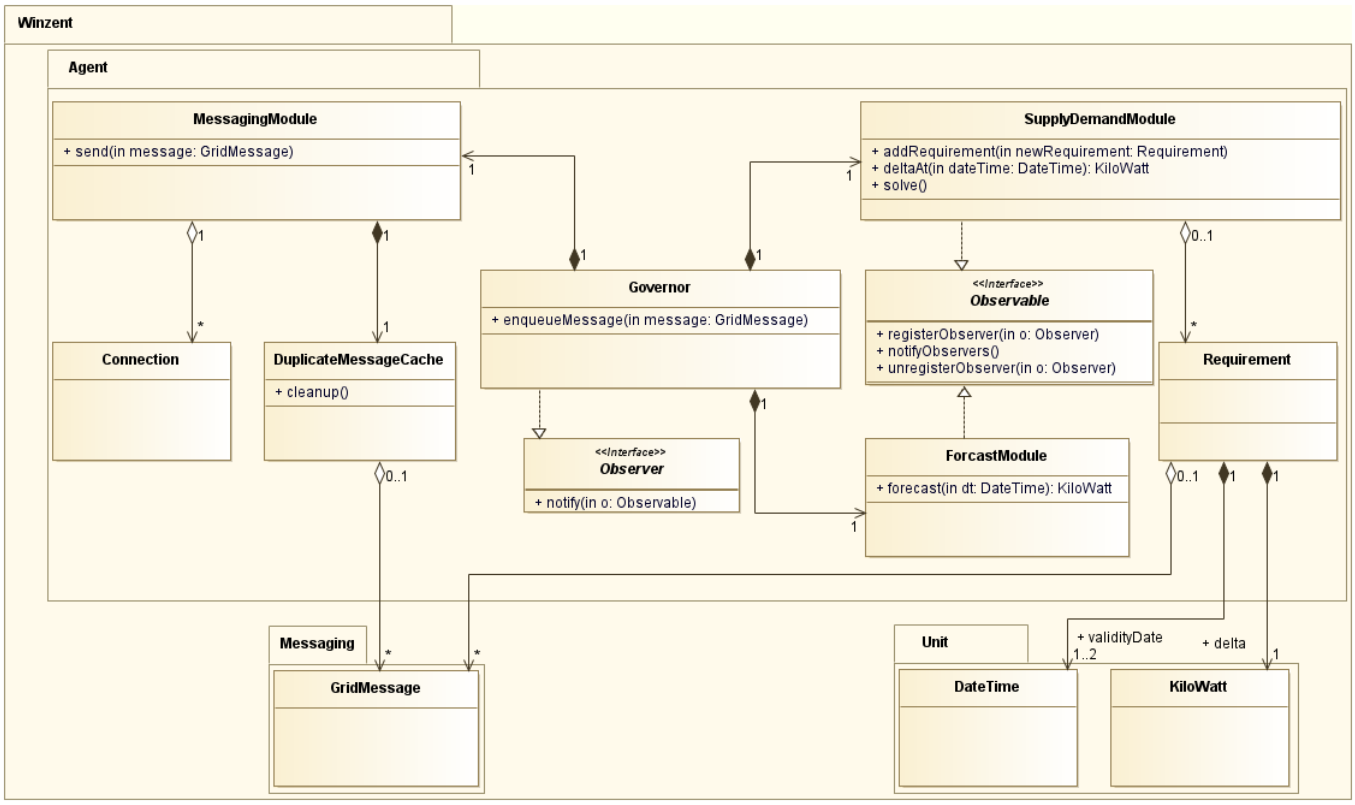

Figure 7. The Winzent Agent design

through the protocol itself ensure that the system as a whole works correctly if every agent adheres to the rules.

In our definition, we denote agents with upper case letters, starting from A. Connection between agents are written down as tuples. Since all connections are bi-directional, we simply order the letters by the alphabet. For example, (A, C) defines a data link from agent *A* to agent *C* and, at the same time, a link from *C* to *A*.

We further define the inner state of any agent also by the following quadruplet:

$$Agent = (Messaging, Forecast, Demand/Supply, Constraints)$$

This can be shorted to (M, F, D, C) for brevity. A module of a specific agent has the agent's letter as subscript, e.g., F_M denotes the Messaging Module of agent *F*. A subscript x denotes a do-not-care, i.e., "any node" or "any value".

Each module, finally, also has a state. Initial as well as final module states are sets of items specific for the particular module. All noted sets are interpreted as subsets of the actual state. For initialization, this allows for local or implementation-specific extra values, while for the final state, it defines the way a successful or failed test run is determined. The final state set must be a real subset of the actual state set of an agent: $Required \subset Actual$. This is especially necessary for the Messaging Module, where a correctly working agent implementation can send Echo Request messages at any time, which would lead to test case failures without this definition.

The state of the Messaging Module is defined by a list of messages; we simply note the JSON text representation as it is already quite easy to read. Message fields that do not matter for the final result are omitted; in this regard, it follows the real subset rule already employed for the module states.

The `id` field is never originally considered for subset matching since it is opaque and implementation-specific to begin with. However, we use it on a meta level to identify individual messages in our notation. This way, we can track messages on their way. The same technique is applied for senders and receivers, where the actual ID of any agent is similarly opaque. For example, `{ answerTo: "m1", sender: "A" }` would correctly describe a message that is an answer to another message identified as m_1 in our notation coming from agent *A*. In reality, not only would a full message travel across the line, but also would it contain ID strings like `4c23a34fab0`.

In order to keep definitions clean, we refer to individual messages with lower-case m letter with number subscript, $m_i | i = 1, 2, 3, \cdots, n$. The combination of these specifics allows us a more efficient notation: We can identify the original message through the m_i notation and refer to it using the `id` field in our notation while leaving out all fields that do not change. Also we do only note those messages that were received since they are already stored in the duplicate message cache. Otherwise, duplicate messages on forwarding

would clutter the notation too much.

Please note that an ASCII-based JSON text does not allow subscripts and thus m_1 becomes `m1`, but otherwise remains the same.

The Forecast Module is defined by a list of forecasts notes as tuples in the form of $(Timestamp, Power)$. Alternatively, for the initial state, an alternate form containing only the forecast is also used; in this case, these values are emitted when queried.

The Demand/Supply Module features a similar tuple form, i.e., $(Timestamp, Delta)$.

Constraints typically do not change over time. Aside from the fact that the Constraints Module is also defined as a possibly empty list of single constraints, they must be understandable by a human. For example, the maximum power that can be forwarded by a node could be written as $C_x = ((P_{max} = 1000kW))$.

In cases where the state of a module does not change from the initial setup, we simply note the upper case letter in the final state definition. If the state is of no interest, we use the subscript x notation, meaning "any value". E.g., $A = (\cdots, F_x, \cdots)$ would mean that, for the results, any state of the Forecast Module is allowed for Agent *A*.

B. The Test Driver

Although our notation abstracts the actual tests from the details of an agent implementation, it leaves an implementor with the task of carefully reading a written definition and creating actual unit tests out of it. This will again create tests that only check whether an implementation works with a set of tests specific to exactly this implementation. The only advantage the written definition provides here is that it forms a common ground to agree upon when it comes to the scenarios that deserve testing.

Also, most implementation-specific unit test suites will be exactly that: Code written in an imperative style. To that end, a developer has translated the set form of the original definition in source code. If two implementations yield different results, the unit test code will have to be re-translated, at least implicitly in the developer's mind, to the set definition in order to find out where things go wrong. This is obviously an error-prone process.

For this reason, we have defined a test driver that reads a JSON representation of the notation we introduced in the previous paragraph and sets up a test bed for the agents. The adapter an implementor has to create comes in the form of interfaces or abstract classes.

The test driver is initialized using the `Manager` class, which sets up the necessary environment. It creates `TestCase` objects, with one object representing one distinct test case. This class reads and parses the JSON notation of the test case itself and is responsible for setting up the test, running it and cleaning up afterwards. Success or failure is indicated by the return value of the `run()` method, which is a simple Boolean value indicating success on true or failure on false.

During setup, agents are created and initialized according to the initial state description. This is the responsibility of the `AgentFactory` class. This factory, along with the `Agent` class instances it creates, is an abstract class: The concrete factory as well as the concrete agent must be implemented by the vendor wishing to test his product. Along with the interfaces representing the modules, i.e., `MessagingModule`, `ForecastModule`, `DemandSupplyModule`, and `ConstraintsModule`, this forms the API an implementor must use when attaching his own agent code.

Although this API carries the spirit of the design we propose in this article, it can be understood as nothing more than a mere wrapper; the concrete classes implementing the module interfaces can be shallow wrapper classes.

All these module interfaces simply provide a getter and a setter for a set of objects. The setter is used during initialization to establish the initial state, while the getter allows us to retrieve the final state. The `Set` class finally implements two set primitives: `equals()` and `isSubsetOf()`. These two are necessary for testing the success of a test case. Since the final state consists of sub sets of the actual state, the success or failure of any test case boils down to a number of subset checks. If, and only if, all succeed, the test case itself succeeds.

The test driver architecture itself does not need many classes in order to provide the necessary API. Figure 8 shows an overview in form of an UML class diagram.

In order to work, this architecture needs test case definitions in a computer-parsable format. We have again chosen JSON for this for the same reasons we use it for the message format: It is easy to read and write for a human and likewise easy to parse for a computer. Also, reliable parsers exist, i.e., it is no obscure, exotic format.

A test case is a JSON object consisting of three root attributes: a list of agents, a list of connections, a list of initial states, and a list of final states. Basically, it is a transcription of the formal definition in JSON that adjusts the written notation to the idiosyncrasies of JSON's syntax.

The list of agents, `agents`, simply introduces the agent IDs in much the same way the formal notation does. The same is true for the connection list, `connections`, that contains tuples in the form of JSON arrays with two elements.

The last two attributes, `initialStates` and `finalStates`, contain the state sets of the agents. All agents are listed here along with their modules: `messagingModule`, `forecastModule`, `demandSupplyModule`, and `constraintsModule`. Each of them contains a list with items as defined in our formal representation. As variables with subscripts do not exist in JSON, references and do-not-cares, e.g., F_x or M_A are strings without subscripts in the same way as we refer to messages: m_1 becomes `"m1"`, and subsequently M_A becomes `"MA"`, D_x is written as `"Dx"`.

Figure 9 has an exemplary test case definition where two agents, *A* and *B* exist. For the test case to succeed, *A* is required to send an Echo Request message to *B*, which the latter one

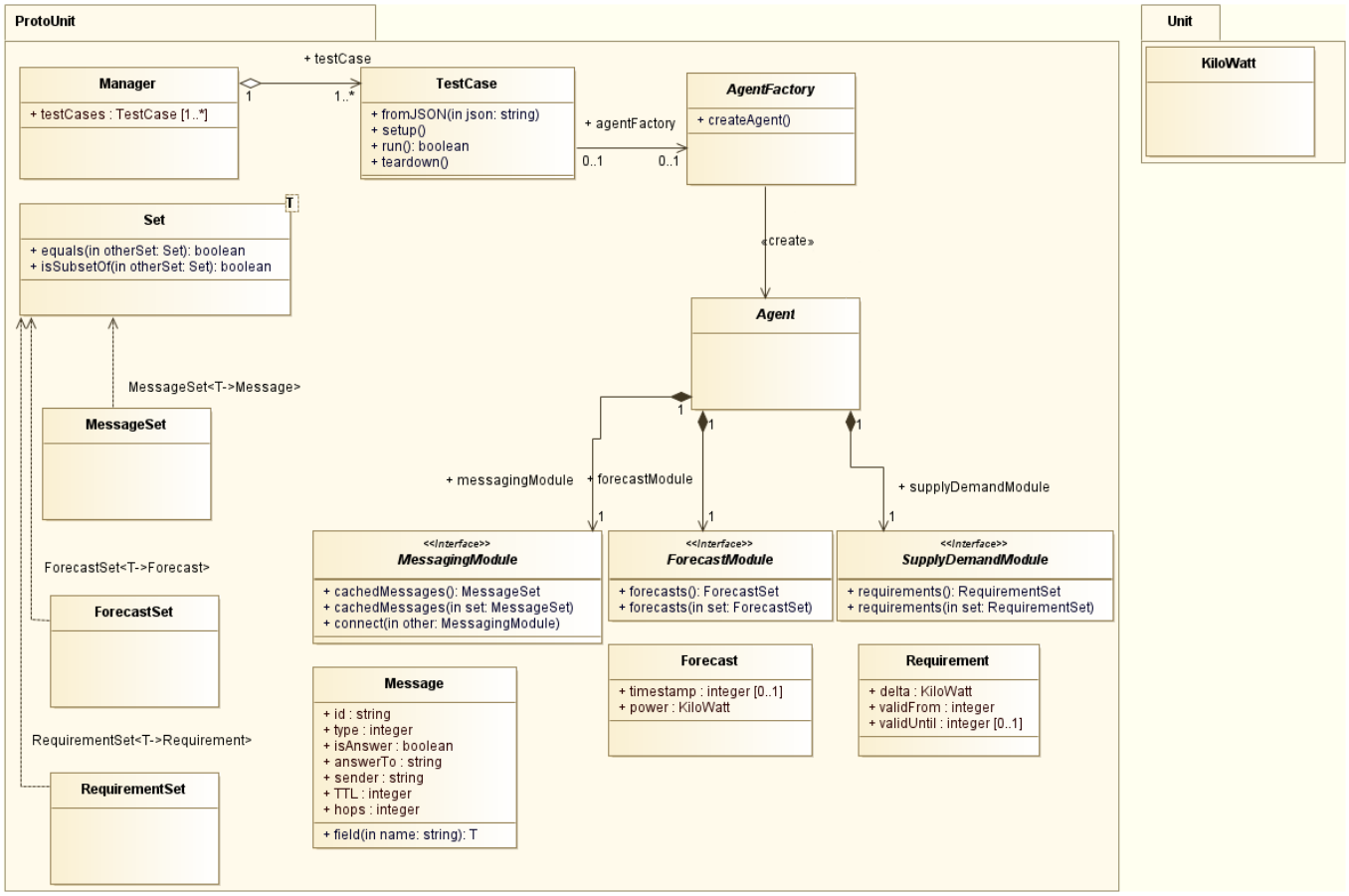

Figure 8. The Test Driver implementing the Test Case notation

answers with an Echo Reply message.

IX. Selected Test Cases of the Protocol Test Suite

This chapter illustrates how we test the semantics of the protocol and the correct functioning of our implementation using selected test cases. We refrain from publishing the full source code of each test case for the sake of readability. Also, this does not constitute a complete suite of test cases. Its intent is to illustrate not only the application of the implementation-agnostic description, but also to show how nodes that correctly implement the protocol behave.

Therefore, we illustrate the important parts using the test case notation introduced in the previous Section VIII.

A. Test Case: "Forward" Rule

Three agents are connected in a linear fashion, i.e., *A*—*B*—*C*. The correct notation of this layout is:

Agents (A, B, C)
Connections $((A, B), (B, C))$

A sends a Demand Notification message for $500kW$, which *B* and *C* cannot answer. The test case ends when *C* receives the message that *B* has to forward. This test case proves the correct working of the "forwarding" rule.

Thus, for the initial state, we need only to define the Demand/Supply Module of all three agents, all other state tuples remain empty. The final state then has to show that a message has travelled to both nodes *B* and *C*, but no answer has been transmitted. Agent *A*'s Demand/Supply tuples therefore remains the same, while the Messaging Module tuple of *A* and *B* each have to list one message.

The complete source code must contain a definition of this message. Both its *Type* and *Power* fields have to be defined in order to indicate that *A* and *B* have received a Demand Notification message. We thus note:

Initial State

$$A = ((), (0, -500000), (), ())$$
$$B = ((), (0), (), ())$$
$$C = ((), (0), (), ())$$

Final State

$$A = ((), F_x, (0, -500000), C)$$
$$B = ((m_1), F, D, C)$$
$$C = ((m_1), F, D, C)$$

Together with the definition of the message m_1, this test case completely defines initial and final state of the simulated

```
{
    agents: ["A", "B"],
    connections: [ ["A", "B"] ],
    initialStates: {
        A: {
            messagingModule: [],
            forecastModule: [],
            demandSupplyModule: [],
            constaintsModule: []
        }, B: {
            messagingModule: [],
            forecastModule: [],
            demandSupplyModule: [],
            constaintsModule: []
        }
    },
    finalStates: {
        A: {
            messagingModule: [{
                    type: 2,
                    hops: 1,
                    sender: "B"
                }],
            forecastModule: [],
            demandSupplyModule: [],
            constaintsModule: []
        },         B: {
            messagingModule: [{
                    type: 1,
                    hops: 1,
                    sender: "A"
                }],
            forecastModule: [],
            demandSupplyModule: [],
            constaintsModule: []
        }
    }
}
```

Figure 9. A Test Case Definition in JSON notation for a "ping" example: *A* sends an Echo Request message to *B* that the latter one answers.

system concerning the transmission of messages and the agent's reactions to it.

B. Test Case: TTL

The setup is similar to the previous test case, however, an additional node *D* with connection (C, D) is introduced. The message TTL of *A*'s Demand Notification is 2. Thus, *C* may not forward the message on the connection (C, D) and the final state of $M_D = ()$ must exist.

Also, the notation of messages needs to change. *A* and *B* have now received messages which have changed regarding the value of the *TTL* field. Thus, we note the TTL field with the changed value for both messages and therefore need to assign different indexes to them:

Final State

$$A = ((), F_x, (0, -500000), C)$$
$$B = ((m_1), F, D, C)$$
$$C = ((m_2), F, D, C)$$
$$D = (M, F, D, C)$$

Even though we need to distinguish the two messages in the test case definition by writing m_1 and m_2, we can still show that the message carries the same ID. We do so by indicating the *ID* field of m_1 and setting it to the identifier m_2. Thus, the message can be traced while still showing that a part of it has changed its value.

C. Test Case: Simple Demand and Supply

This test case will, again, use the topology of the first test case. But now, agent *C* is able to completely answer *A*'s request for energy.

This seems to be the simplest of all test cases, but, in fact, more messages than for the previous ones are required. Here, we need to explicitly confirm the offer using an Offer Accepted Notification.

This test case serves to check an implementation for all defined rules of behavior. It applies the "forwarding" rule and will modify a message's *TTL* field in the same way as the other two test cases, combined.

Additionally, it shows the application of the "match-or-forward" rule. The agent *C* is required to answer the Demand Notification, which can be checked using *B*'s message cache. We discuss this imperative in Section X. Also, the correct formation of the implicit contract can be monitored.

The creator of the test case therefore needs to track a number of messages: First, the forwarding of the initial Demand Notification with modified TTL Values. Second, the Offer Notification message which travels back to the requester and must be recorded in both *B*'s and *A*'s Message Module's duplicate request cache. In the same way, the Offer Accepted Notification is sent by *A* and reaches *C* via *B*, being recorded in the same way as the previous two messages. Finally, an Offer Acceptance Acknowledgement is sent by *C* to *A*.

With the reception of the final message, the Message Module of *A* and *C* must contain two received messages, while that of *B* holds copies of all four. The reception of the Offer Acceptance Acknowledgement also marks the formation of the contract between *A* and *C*. Thus, the Supply/Demand Module of the requester must appear as balanced in the final state definition.

D. Test Case: Circular Demand/Supply with Partial Offers

This test case offers a more complex topology featuring one requester and three suppliers. Since the setup is harder to imagine than the previous ones, instead of a written prose description or the formal connections notation, it is depicted in Figure 10.

This test case serves to test the protocol-implementing nodes' behavior on a more complex topology. It also uses

partial offers, which increases the amount of messages that need to be sent in order to arrive at a working contract.

Initially, the requesting node *A* sends a Demand Notification message for the same amount used in the previous two test cases. However, now three other agents answer with an offer.

First, the agents *C* and *F* will offer one half of the requested power, while agent *H* is able to supply the complete power needed. This tests not only the "forwarding" rule, but also "match-or-forward". The agents *C* and *F* not only have to send their offer, but also forward a modified version of the initial request. This forwarded version can be found in *G*'s Messaging Module's cache.

G and *H* also serve to test the correct implementation of the "no-duplicates" rule. *G* will, via *E* and *D*, receive two modified Demand Notification messages. However, it may forward only one of the two. This means that the cache of *H* may contain exactly one Demand Notification message which can easily be identified using its *ID* field as the one originating from *A*. The transmission of the Demand Notification message therefore stops at *G* and also *B*, effectively preventing an endless circulation.

This original requester will finally have three Offer Notifications received. Since we advise preferring messages with a lower hop count, it will accept the two partial offers. This Offer Acceptance Notification must be recorded by all three offering agents. This is vitally important since all parties have now knowledge of whether they need to provide the advertised power or not.

Applying the same rules, the final Offer Acceptance Acknowledgement messages will then travel through the network back to *A*.

Please note that it is not necessary to list all transmitted messages in order to show that the actual demand/supply scenario is solved. In fact, the Message Module tuple of *A*'s final state alone suffices to show that three agents have sent their offers. Listing its Demand/Supply Module definition serves to verify that it has accepted the two partial offers.

However, if all described messages are recorded in the test case definition, we can also show that an endless message circulation is prevented by the duplicate message cache. Therefore, this additional information asserts an important part of the behavioral rules of this protocol and are included.

X. Discussion

A. Comparison with SIP and RSVP

Our protocol seems to share some features with already deployed, well-known protocols such as the Session Initiation Protocol (SIP) [29], or the Resource Reservation Protocol (RSVP) [30]. SIP is especially designed to to be usable on exiting protocols in Layers 1–6, thus being not vertically integrated similar to our protocol. However, there are several differences that justify the creation of this Layer 7 protocol.

Traditional SIP relies on proxy servers. Here, individuals register in order to be locatable. The proxy server typically serves a domain and makes up the domain part of a SIP URI, e.g., `sip:bob@biloxi.com`. These proxy servers create a layer of indirection in performing their duty. In our approach, there could theoretically be any number of proxy servers from 1 to *n*, with *n* being the number of nodes in the grid. These proxy servers create points of failures while also partially obscuring the direct mesh that is used for routing the offers and demand messages.

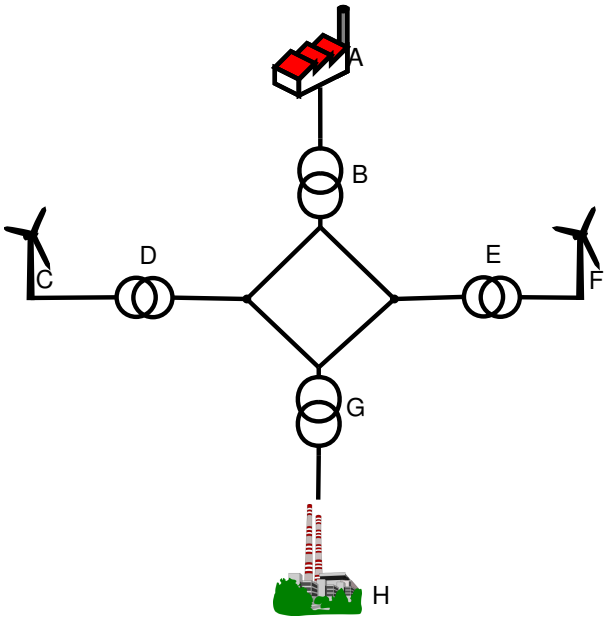

Figure 10. Topology of a circular demand/supply test case

Using a Peer-to-Peer architecture (P2P) seems to be the next logical step in order to rely on already existing architectures. *SOSIMPLE* by Bryan, Lowerkamp, and Jennings [31] is a "serverless, standards-based, P2P SIP communication system". The routing of requests still requires an a-priori knowledge of the (potential) location of the callee in the grid: "Node A is not responsible for that Resource-ID, so it sends a SIP 302 Moved Temporarily reply, including the node it thinks is closest [...] in the headers [...]."

Both SIP and RSVP are based on the premise that the initiating client knows its counterpart, i.e., the caller knows its callee as well as the video-requesting client knows which server offers the desired video. In our case, however, there is no a-priori knowledge about potential contract partners. Each node has the same chance to match an offer, and therefore, our protocol relies on and uses the meshed architecture of the power grid that is re-created in the communication network.

In that regard it also becomes apparent that there is no explicit session that is created and maintained by the protocol. Sessions require a setup, potentially keep-alive and a teardown. However, our protocol does not need the explicit notion of sessions that are maintained. The power grid itself provides the "session" since nodes act and react based on the state of the power grid itself.

It can also be noted that we do not establish an end-to-end connection in the protocol. It is the basis for a negotiation, but the actual connection—if one would call it so—is done in the power grid itself by initiating the flow of energy based on the

negotiation that took place earlier and was facilitated by the protocol.

In summary, intermediate nodes are not required to keep track of contracts made in order to reserve a specific capacity once the flow of power has been initiated. We assume that a node that requests a certain amount of power for a certain amount of time does so truthfully, i.e., that it actually consumes or delivers the power. In contrast, traffic on an Internet channel such as one created by RSVP does not necessarily flow continuously.

B. Message Transmission Volume

The test cases show clearly that there is room for improvement regarding efficiency: A lot of messages travel the information network in order to establish a contract. Many of those are redundant and are caused by the "forward" rule that, in its current, simple form, resembles a routing by flooding algorithm [32].

In fact, in our testcases no actual routing in the sense of selecting a single path is performed at all. For large-scale deployment, this behavior must change into a better version that reduces the number of messages transmitted. Ideally, a node would transmit the message only on sensible connections instead of all but the receiving ones. The intial broadcasting of requests is necessary in order to reach all potential contract partners as there is no central registry of potential suppliers and consumers. However, when those have made their offers, the multicast character of messages can transform into an unicast nature, thus using the existing communication facilities in a more efficient way.

We believe that we can use the already existing Duplicate Message Cache in order to identify a minimal set of outgoing connections for answers. Using this local piece of knowledge, the routing can be optimized for replies. This assumes that no cache expiry timer on a node along the path the answer takes has yet led to the removal of the necessary piece of information.

In comparison to other protocols in the smart grid area, the raw data volume of our proposal seems rather high. The example Offer Notification message in Figure 5 compromises 170 Bytes if all unnecessary line breaks and whitespace characters are removed. If transmitted via standard TCP/IPv6, protocol headers add another 32 Bytes (TCP) and 40 Bytes (IPv6) for a grand total of $170 + 32 + 40 = 242$ Bytes. Including the TCP three-way handshake and the connection termination increases the number to 762 Bytes for a single message. Adding in IPsec raises the grand total even more.

Of course, our Connection concept does not force a permanent setup and teardown of a TCP connection for every message. Using SSTP instead of a solution based on IPsec and TCP will also reduce protocol overhead. And since all agent Connections are end-to-end connections, two nodes can employ compression, e.g., using simple GZIP [33].

Currently, we compute the overall data volume for a contracting process using the following formulae. All variables are summarized in Table III.

First, the transmission volume of a singular message m on an established connection c can be calculated by:

$$v_u(m,c) = s(m)f(m,c) + V_c \quad \text{Bytes}$$

The message's size is given by calculating $s(m)$, i.e., the size of the message. $f(m,c)$ denotes a dynamic cost factor of the connection like compression. The constant V denotes the constant costs of the connection c, such as TCP/IP headers that get added to each transmission.

Forwarding a request generates costs on all connections of a node n except the receiving one c_0, i.e.,

$$v_m(n,m) = \sum_{i=1}^{|C_N|-1} v_u(M,c_i) \quad \text{Bytes}$$

A request is forwarded at most TTL hops, i.e., the request's initial TTL limits the number of hops it can be forwarded. Through the duplicate request cache we ensure that the message will not pass any node twice. Therefore, the maximum cost of transmitting a request message m from an initial node n_0 is given as

$$v_r(m,n_0) \leq \sum_{i=0}^{TTL_M} v_m(n_i,m) \quad \text{Bytes}$$

Since answers can be transmitted in an unicast fashion thanks to the duplicate message cache, the volume of an answer equals the transmission volume of any message, with c_0 being a connection on which the initial request was received. Choosing the "right" connection is the responsibility of the node; using the first receiving one will typically suffice.

Thus, each answer produces at least

$$v_a(m_r,n_r,m_a) = \sum_{i=0}^{h_{m_r}} v_u(m_a,c_0) \quad \text{Bytes}$$

The total volume of bytes each request generates is therefore the sum of the volume a request produces plus the sum of all answers that are sent by other nodes. If the Duplicate Message Cache is not used as a means to optimize the path an answer takes, v_a equals v_r on all nodes. This effect is apparent in the test cases we showed in Section IX.

Although optimization techniques such as using the Duplicate Message Cache where possible are employed, our protocol uses a substantially higher volume in comparison with bit-by-bit defined protocols such as OSGP. However, we approach a different problem. OSGP or the IEC protocols access highly integrated devices with low data rate links in order to query sensor, usage and billing information. However, the protocol we propose acts on the level of a whole neighborhood, a wind farm, factory or traditional power plant in order to enable an efficient, on-the-fly demand/supply calculation.

C. Choice of Offers

In cases where several solutions to the demand/supply calculation emerge, we do not enforce any priorities. We do, however, recommend to prefer messages with a lower Hop

TABLE III. Variables and functions used in the message volume calculation

c	An established connection
C_N	Set of all established connections on node n: $\{c_0, c_1, \ldots, c_n\}$
c_i	ith established connection on a node
n	A node
m	A message
m_r	A request message
m_a	An answer message
h_m	Hop count of message m
TTL_m	TTL of message m
$s(m)$	The size of a message m, in bytes
$f(m, c)$	Dynamic factor by which the message m will be modified when transmitted via connection c
V_c	Static additional costs of a connection
$v_u(m, c)$	Total volume to transmit message m on connection c
$v_m(m, n)$	Total volume to forward a message m using multicast on node n

Count, even if this leads to a solution compromised of more and "smaller" building blocks. In fact, the last test case we describe in this article explicitly checks that a decision is made on exactly this basis.

A lower Hop Count means that a node nearby has sent the message. Our network architecture practically constitutes an overlay network over existing IP-based ones and effectively re-models the existing power grid. As a result, the power that flows on grounds of a message with lower Hop Count bridges less meters of the grid than that flowing due to a message with higher Hop Count. This leads to a lower (transmission) grid load and smaller overall line loesses since it automatically prefers micro grids. Additionally, the Constraints Module can be used to implement cost-based policies.

However, a focus on costs should be avoided. A pure "cents per kilowatt hour" metric can be influenced to an amount that diminishes or even destroys its value as an objective criterion. For example, government subsidies can lead to huge distortions. A study by the German Federal Environmental Agency [34] shows that subsidies of coal and nuclear power plants greatly influence the price per kilowatt hour.

It is obvious that this does not yield to the technically best solution, or a solution that is the best from a grid-wide supply point of view. Instead, a price-based fitness metric as acting basis can even lead agents to hold back offers because the price is too low. However, we see the preservation of the power grid itself as the highest priority, whereas a profit margin is an optimization problem that arises once more than one solution can be considered.

But in comparison, it is clear that the Hop Count metric is very abstract. Power may travel many kilometers with just one hop, depending on the grid layout. Other metrics may be better applicable. Takeru Inoue *et al.* use actual physical metrics in their article [35], which will be a better application for a decision-making algorithm in the future.

Both the *Demand Notification* and *Offer Notification* message types include timestamps. Especially the *Answer Until* field is noteworthy, because it takes part in timing the start of a demand/supply calculation on a node. Any node can set a meaningful time considering its own characteristics that it knows about, like hardware constraints that require a certain time buffer in order to employ a fall-back solution, or to have a search room populated enough to arrive at a meaningful result in a local demand/supply calculation.

It is tempting to include information network constraints in the time buffer the *Answer Until* field provides. However, we advise against it for two reasons:

First, a node does not have knowledge about the latency to other nodes when it broadcasts its initial request. When answering another node directly, e.g., using a *Offer Notification*, it would have to measure the latency beforehand to include a meaningful value. In internet communication, this latency would reside in the area of milliseconds, which are not included in the Unix Timestamp that makes up the `answerUntil` field.

Second, even when packets are routed in an extremely inefficient way [36], a high delay means values of $< 300ms$, while even fast gas turbine power plants react in a matter of minutes ("Boosting times of a few minutes including synchronization to the grid are possible", translated from [37]).

XI. Conclusion and Future Work

In this article, we have defined a lightweight protocol based on behavioral rules that enable a distributed supply/demand calculation for smart grids. It allows nodes to act pro-actively based on their local energy situation and to propagate a future demand or over-supply of power. This, in turn, initiates a distributed, automatic search for a solution to this problem. This way, renewable energy sources can be used more efficiently since consumers can make use of an increased supply or scale down dynamically based on the local knowledge of individual nodes.

We have also proposed an architecture for an agent implementing this protocol and the rules related to it. This architecture serves as basis for our test cases, which are not just unit tests tied to a specific software, but also provide a written-down definition of a successful execution of a test case.

However, the way we define tests today is based on a textual representation. While this is good for parsing and to create a definite collection of implementation-independent test cases, it is clear that all topologies that are not extremely simple are best visualized. That is why we also created a graphical simulation environment, which we will publish in a separate paper.

Currently, we employ a very simple routing algorithm for requests that resembles a classical "routing by flooding" approach, as noted in Section X. We plan to employ all agents to be more aware of their immediate neighbors in order to relay requests more efficiently without using separate registry servers.

In this article, we have not touched the problem of how connections are initially created, but have simply assumed that they already exist. Connections can be a tool to negotiate individual data link parameters such as encryption for low data

rate links and can even be used to identify potentially malevolent nodes by implementing a credibility-based algorithm in the same manner as it is already done in current peer-to-peer networks. However, the actual algorithm how nodes could automatically connect to their immediate neighbours, negotiate parameters and shield themselves against attacks is still future work.

XII. Acknowledgments

This article has been created as part of a cooperative doctorate program between the TU Bergakademie Freiberg and Wilhelm Büchner Hochschule, Pfungstadt.

References

[1] E. M. Veith, B. Steinbach, and J. Windeln, "A lightweight messaging protocol for Smart Grids," in *EMERGING 2013, The Fifth International Conference on Emerging Network Intelligence*. IARIA XPS Press, 2013, pp. 6–12.

[2] J. R. Roncero, "Integration is key to smart grid management," *CIRED Seminar 2008 SmartGrids for Distribution*, no. 9, pp. 25–25, 2008, retrieved 2013-02-11. [Online]. Available: http://link.aip.org/link/IEESEM/v2008/i12380/p25/s1&Agg=doi

[3] M. Pierrot, "Wind energy data for germany - country windfarms," retrieved 2013-12-10. [Online]. Available: http://www.thewindpower.net/country_windfarms_en_2_germany.php

[4] M. Z. Lu and C. L. P. Chen, "The design of multi-agent based distributed energy system," *2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009), Vols 1-9*, pp. 2001–2006, 2009.

[5] B. Lasseter, "Role of distributed generation in eeinforcing the critical electric power infrastructure," pp. 146–149, 2001.

[6] European Parliament, Council, "Directive 2009/28/ec of the european parliament and of the council of 23 april 2009 on the promotion of the use of energy from renewable sources and amending and subsequently repealing directives 2001/77/ec and 2003/30/ec (text with eea relevance)," Official Journal of the European Union, 04 2009, date of effect: 25/06/2009; Entry into force Date pub. + 20 See Art 28.

[7] Bundesministerium für Umwelt, Naturschutz und Reaktorsicherheit, "Datenreihen zur Entwicklung der erneuerbaren Energien in Deutschland," p. 41, February 2013.

[8] M. Vasirani, R. Kota, R. L. G. Cavalcante, S. Ossowski, and N. R. Jennings, "An agent-based approach to virtual power plants of wind power generators and electric vehicles," *IEEE Transactions on Smart Grid*, vol. 4, no. 3, pp. 1314–1322, 2013.

[9] A. Faruqui, S. Sergici, and A. Sharif, "The impact of informational feedback on energy consumption—a survey of the experimental evidence," *Energy*, vol. 35, no. 4, pp. 1598–1608, 2010.

[10] P. Merrion, "Pilot test of comed's smart grid shows few consumers power down to save money," *Crain's Chicago Business*, May 2011.

[11] B. M. Buchholz, V. Bühner, U. Berninger, B. Fenn, and Z. A. Styczynski, "Intelligentes lastmanagement — erfahrungen aus der praxis," in *VDE-Kongress 2012*. VDE VERLAG GmbH, 2012.

[12] Y.-J. Kim, V. Kolesnikov, H. Kim, and M. Thottan, "SSTP: a scalable and secure transport protocol for smart grid data collection," in *IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 2011, pp. 161–166.

[13] P. Leach, M. Mealling, and R. Salz, "An universally unique identifier (UUID) URN namespace," July 2005, retrieved 2013-05-25. [Online]. Available: http://tools.ietf.org/html/rfc4122

[14] S. Kent and K. Seo, "Security architecture for the internet protocol," IETF RFC 4301. [Online]. Available: http://www.ietf.org/rfc/rfc4301.txt

[15] R. Stewart, "Stream control transmission protocol," RFC 4960, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt [Retrieved 2013-05-13]

[16] European Telecommunications Standards Institute, "Open smart grid protocol," European Telecommunications Standards Institute, Tech. Rep., 2012.

[17] G. Zhabelova and V. Vyatkin, "Multi-agent smart grid automation architecture based on iec 61850/61499 intelligent logical nodes," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 5, pp. 2351–2362, 2011, retrieved 2013-04-03. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6018303

[18] M. Pipattanasomporn, H. Feroze, and S. Rahman, "Multi-agent systems in a distributed smart grid: design and implementation," *2009 IEEE/PES Power Systems Conference and Exposition*, pp. 1–8, March 2009, retrieved 2013-06-01. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4840087

[19] P. Oliveira, T. Pinto, H. Morais, and Z. Vale, "MASGriP — a multi-agent smart grid simulation platform," in *IEEE Power and Energy Society General Meeting*. IEEE, 2012, pp. 1–8.

[20] M. Hommelberg, C. Warmer, I. Kamphuis, J. Kok, and G. Schaeffer, "Distributed control concepts using multi-agent technology and automatic markets: An indispensable feature of smart power grids," in *IEEE Power Engineering Society General Meeting*. IEEE, 2007, pp. 1–7.

[21] R. G. Smith, "The contract net protocol: high-level communication and control in a distributed problem solver," in *IEEE Transactions on Computers*, vol. C, no. 12. IEEE, December 1980, pp. 1104–1113.

[22] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke, "Smart grid technologies: communication technologies and standards," *IEEE Transactions on Industrial informatics*, vol. 7, no. 4, pp. 529–539, 2011.

[23] K. Thompson and D. M. Ritchie, *UNIX programmer's manual*. Bell Telephone Laboratories, 1975.

[24] S. Deering and R. Hinden, "Internet protocol," 1998, retrieved 2013-05-14. [Online]. Available: http://tools.ietf.org/html/rfc2460

[25] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[26] E. Euler, "The failures of the mars climate orbiter and mars polar lander—a perspective from the people involved," in *Guidance and Control*, 2001, pp. 635–655.

[27] I. Maqsood, M. Khan, and A. Abraham, "An ensemble of neural networks for weather forecasting," *Neural Computing and Applications*, vol. 13, no. 2, pp. 112–122, May 2004. [Online]. Available: http://link.springer.com/10.1007/s00521-004-0413-4

[28] S.-T. Chen, D. C. Yu, and A. R. Moghaddamjo, "Weather sensitive short-term load forecasting using nonfully connected artificial neural network," *IEEE Transactions on Power Systems*, vol. 7, no. 3, pp. 1098–1105, 1992.

[29] J. Rosenberg, H. Schulzrinne, and G. Camarillo, "SIP: Session initiation protocol," *Vasa*, pp. 1–269, 2002, retrieved: 2014-05-10. [Online]. Available: http://www.hjp.at/doc/rfc/rfc3261.html

[30] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, *Request for Comments*.

[31] D. A. Bryan, B. B. Lowekamp, and C. Jennings, "SOSIMPLE: A serverless, standards-based, p2p SIP communication system," *First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA'05)*, pp. 42–49, 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1652335

[32] M. Abolhasan, T. Wysocki, and E. Dutkiewicz, "A review of routing protocols for mobile ad hoc networks," *Ad hoc networks*, vol. 2, no. 1, pp. 1–22, 2004.

[33] P. Deutsch, "GZIP file format specification version 4.3," IETF RFC 1952, 1996, retrived 2013-12-08. [Online]. Available: http://www.ietf.org/rfc/rfc1952.txt

[34] A. Schrode, A. Burger, F. Eckermann, H. Berg, and K. Thiele, "Environmentally harmful subsidies in Germany," Federal Environment Agency, Germany, Dessau-Roßlau, Tech. Rep., 2011.

[35] T. Inoue, K. Takano, T. Watanabe, J. Kawahara, R. Yoshinaka, A. Kishimoto, K. Tsuda, S.-i. Minato, and Y. Hayashi, "Distribution loss minimization with guaranteed error bound," *IEEE Transactions on Smart Grid*, vol. 5, no. 1, pp. 102–111, January 2014, retrieved 2014-01-15. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6693788

[36] Renesys, "The new thread: targeted traffic redirection," November 2013, retrieved 2014-01-01. [Online]. Available: http://www.renesys.com/2013/11/mitm-internet-hijacking/

[37] K. Heuck, K.-D. Dettmann, and D. Schulz, *Elektrische Energieversorgung*. Wiesbaden: Vieweg + Teubner, 2010.