USB Proceedings

IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society

Austria Center Vienna
Vienna, Austria
10 - 14 November, 2013

Sponsored by

The Institute of Electrical and Electronics Engineers (IEEE)
IEEE Industrial Electronics Society (IES)

Co-sponsored by

Austrian Institute of Technology (AIT), Austria
Vienna University of Technology (TU Vienna), Austria

IEEE Catalog Number: CFP13IEC-USB
ISBN: 978-1-4799-0223-1

IECON 2013 November 10–13 2013, Vienna, Austria

Conference Booklet

IECON 2013

39th Annual Conference of the IEEE Industrial Electronics Society

in conjunction with

ICELIE 2013: 7th International Conference on e-Learning in Industrial Electronics

IWIES 2013: 1st International Workshop on Intelligent Energy Systems (14.11.2013)

Austria Center, Vienna, Austria
10-13 November 2013

Sponsored by the
IEEE Industrial Electronics Society

MONDAY 11th of November

	D-349	D-351	D-355	D-358	D-441	D-445	D-446	D-447	G-5432	G-560	G-631	H-633	Hall-D	Hall-G	Hall-H	Hall-I	Hall-K	Hall-P	I-140	I-141	K-142	K-246	K-253	K-255	K-261	S-357	U-556	U-637
08:00-09:30													Opening						PC Brief									
09:30-10:00																												
10:00-12:00	TT02 12	SS68-1	SS73	TT08 1	TT06 1	SS52 1	SS27 1	SS18 1	SS17 1	TT07 1	TT07 12	TT12 1	TT01 2	TT01 13	TT01 21	TT01 40	TT01 18	TT02 1	TT00	SS16 1	TT06 12	Press Conf	SS23 1	SS01 1	TT03 1	TT02 15	TT10 1	SS06 1
12:00-13.00																												
13:00-15:00	TT01 37	SS68 2	TT01 5	TT08 2	TT06 2	TU ZSI	TU Ind 4.0	TU MCS	SS17 3	TT07 3	TT07 14	TT12 3		TT01 16	TT01 6	TT01 1	TT01 19	TT02 6	TT00	SS16 2	TT09 1	TT06 13	SS23 2	SS01 2	TT03 2	TT02 2	TT10 2	SS06 2
15:00-15:30																												
15:30-16:30	SS48 1	SS68 3	SS47 1	TT08 3	TT06 3				SS17 3	TT07 3	TT07 14	TT12 3		TT01 17	TT01 7	TT01 4	TT01 20	TT02 14	TT00	SS35 1	TT09 2	TT06 14	SS23 3	SS01 3	TT03 3	TT02 3	TT10 3	SS06 3
16:30-17:30						TC ESOC	TC SG																					
17:30-18:00	TC AT	TC BACM	TC IA	TC HM	TC CRM	TC EEIT	TC EEIT	TC MEMS	TC II	TC FA		TC ES							TC NCSA	IFC	TC RES		TC MC	TC RSIA	TC EM	TC SA	TC STD	TC PETC
18:00-18:30																												

TUESDAY 12th of November

	D-349	D-351	D-355	D-358	D-441	D-445	D-446	D-447	G-5432	G-560	G-631	H-633	Hall-D	Hall-G	Hall-H	Hall-I	Hall-K	Hall-P	I-140	I-141	K-142	K-246	K-253	K-255	K-261	S-357	U-556	U-637
08:00-10:30	SS60 1	SS71 1	SS72 1	TT08 4	TT06 4	TU-EVCIDG	TU-PHM	TU-IETCPC	SS46 1	TT07 4	TT07 15	TT12 4		IF	TT01 8	EI13	TT01 27	TT01 29	TT00	SS41 1	TT09 3	TT06 15	SS23 4	TT05 1	TT03 4	TT02 4	SS62 1	SS09 1
10:30-11:00																												
11:00-11:30	SS60 2	SS71 2	EE67	TT08 5	TT06 5									IF	TT01 9	EI13	TT01 22	TT01 30	TT00	SS13	TT09 4	TT06 16	SS23 5	TT05 2	TT03 5	TT02 5	TT1 1	SS09 2
11:30-13:00						STD PAR		EI14 SC																				
13:00-14:00																												
14:00-16:00	SS58 1	SS20 1	SS57 1	SS63 1	TT06 6	TU-FCIRS	TU-MDPEM	TU-XENPI																				
16:00-16:30									SS45 1	TT07 6	TT07 17	SS21		IF	TT01 10	EI13	TT01 23	TT01 31	SS64 1	SS10	TT09 5	TT06 17	SS28 1	TT05 3	TT03 6	TT02 13	TT11 2	SS03 1
16:30-18:30	SS58 2	SS20 2	SS53	SS63 2	TT06 7				SS45 2	TT07 7	ICELIE 1	SS42		IF	TT01 11	EI13	TT01 24	TT01 32	SS50 1	SS34 1	TT09 6	TT06 18	SS28 2	TT05 4	TT03 7	TT02 7	TT11 3	SS03 2

WEDNESDAY 13th of November

	D-349	D-351	D-355	D-358	D-441	D-445	D-446	D-447	G-5432	G-560	G-631	H-633	Hall-D	Hall-G	Hall-H	Hall-I	Hall-K	Hall-P	I-140	I-141	K-142	K-246	K-253	K-255	K-261	S-357	U-556	U-637
08:30-10:30	SS32 1	SS15 1	SS59	TT04 1	TT06 8	TU-EH	TU-SSTC	TU-NET	SS26 1	TT07 8	ICELIE 2	SS43 1		IF	TT01 12	EI13	TT01 25	TT01 33	SS02 1	SS04	TT09 7	TT06 19	SS28 3	SS19 1	TT03 8	TT02 8	SS49 1	SS44 1
10:30-11:00																												
11:00-11:30	SS32 2	SS15 2	TT00	TT04 2	TT06 9				SS26 2	TT07 9	ICELIE 3	SS43 2		IF	TT01 15	EI13	TT01 26	TT01 34	SS61		TT09 8	EI13	SS28 4	SS19 2	SS07 1	TT02 9	SS49 2	SS44 2
11:30-13:00						IES WIC																						
13:00-14:00																												
14:00-16:00	SS32 3	SS38 1	IES AT	TT04 3	TT06 10	SS01 4	SS12 1	SS37 1	SS29 1	TT0710	ICELIE 4	SS43 3		IF	TT01 14	EI13	TT01 3	TT01 35	SS14 1		TT07 18	TT06 20	SS28 5	SS19 3	SS07 2	TT02 10	SS51 1	SS69 1
16:00-16:30																												
16:30-18:30	SS32 4	SS38 2	IES Chap	TT04 4	TT06 11	SS01 5	SS30 1	SS65 1	SS29 2	TT07 11	ICELIE 5	SS43 4			IES JT	EI13	TT01 28	TT01 36	SS14 2		TT01 39	TT01 39	SS08 1	SS19 4	SS07 3	TT02 11	SS51 2	SS69 2

THURSDAY 14th of November

	D-349	D-351	D-355	D-358	D-441	D-445	D-446	D-447	G-5432	G-560	G-631	H-633	Hall-D	Hall-G	Hall-H	Hall-I	Hall-K	Hall-P	I-140	I-141	K-142	K-246	K-253	K-255	K-261	S-357	U-556	U-637
08:30-10:30				IES Prep		IES Prep				IES Prep	IWIES 1																	
10:30-11:00																												
11:00-13:00				IES Prep		IES Prep		IWIES 3		IES Prep	IWIES 2																	
13:00-14:00																												
14:00-16:00				IES Prep		IES Prep		IWIES 5		IES Prep	IWIES 4																	
16:00-16:30																												
16:30-18:30				IES Prep		IES Prep				IES Prep	IWIES 6																	

Table of Contents

Software Architecture for a Smart Grids Test Facility

IT Implementation for an Emulated Low Voltage Smart Grid

Alexander Wendt, Mario Faschang, Thomas Leber and Klaus Pollhammer
Vienna University of Technology
Gusshausstrasse 27-29 E384
A-1040 Vienna, Austria
{wendt, faschang, leber, pollhammer}@ict.tuwien.ac.at

Tobias Deutsch
Siemens AG Österreich, CT RTC NEC INN-AT
Siemensstraße 90, 1210 Wien
tobias.deutsch@siemens.com

***Abstract* — in the joint smart grids application project "Intelligent low voltage grid" of SIEMENS and Vienna University of Technology, households are emulated with transformers representing photovoltaic facilities and current sinks representing the loads. The purpose is to test new control concepts in small scale before a field trial. One example is the use of smart meter measurement for controlling a local transformer's tap changer. For the interconnection and management of the emulation devices, a software infrastructure is needed. In this paper, the basic design of the software components and their interactions are presented. It is a flexible, extendable architecture which consists of a central server and multiple specialized clients. A communication service based on remote procedure calls was written, where Google Protobuf is used for the data exchange. In two applications, the realization of the concept is demonstrated.**

Keywords—smart grids, architecture, protobuf, implementation, service, datapoint, server, network, communication

I. INTRODUCTION

In order to make the nearing energy turnaround possible, several problems in the area of energy grids and in storage of energy have to be solved [2]. There is an increasing amount of renewable energy generators, which will replace conventional energy generators. One challenge for electrical grids is to stay within the voltage band limits as more distributed energy sources like photovoltaic facilities are connected [1]. The introduction of e-mobility represents another challenge because additional high loads are expected. However, smart grid concepts can minimize the need of extension of the electric grids or at least delay it. Many concepts have been developed to counter the mentioned problems, for instance to use smart meter data to efficiently control the tap changer in the power transformer between the low and middle voltage electric grid [6]. As the effort of testing in real electric grids is very high and risky, and pure software simulations cannot replace the real hardware, an emulation platform for testing concepts on hardware in small scale is needed.

This is the goal of the "Intelligent low voltage grid" project [1]. It emulates a three phase low voltage grid, which consists of four buildings, of which two use photovoltaic power generators. The devices are emulated with variable transformers, current sinks and resistances. Three grid topologies can be realized within the system: Two separate branches, a single long branch or a ring. Among others, the following concepts are being tested in the system: Controlling of the power transformer at the interface between the low and middle voltage grid based on smart meter data, analysis of asymmetric loads and resulting high neutral wire current flow, topology recognition, co-simulation of grid and its infrastructure as isolated operation mode. In order to implement such scenarios, a communication and control platform is needed [17]. In this paper, a software architecture is presented, which successfully fulfills the needs for extendibility, flexibility and simplicity for the operation in smart grid emulations. The implementation is demonstrated in two projects.

II. RELATED WORK

The purpose of this test facility is to test smart grid concepts in small scale before trying them in real electric grids. In Austria, there are many projects operating in the area of smart grids, which aim to solve the problems of high fluctuations in the electric grids, like in the projects "DG Demonet" [6] and "Smart Grid Modellregion Salzburg" [8]. Another concept, which can be tested within this facility for reducing the grid fluctuations, is to make use of load shifting in buildings. They are realized in the projects "Building to Grid" [10] and "BED – Balancing Energy Demand" [11].

In a related project, a test facility with the same purpose as the "Intelligent low voltage grid" is built in Italy. It emulates middle and low voltage grids [9]. However, this facility is built in a larger scale. Electrical and thermal load emulators are used in the middle voltage grid. Also a variable topology with different impedances can be applied through line emulators. The low voltage grid consists of multiple generators, real and emulated loads as well as storage systems. Different communication systems are tested, especially data exchange solutions, which are compliant to the IEC 61850 standard. “Intelligent low voltage grid” models only a low voltage grid in smaller scale, which makes it cheap and mobile. A customized control system is being developed adapted for the emulation needs of the facility.

A related software architecture to the one presented in this work is Reef [12]. It is an open source smart grid platform, which relies on a service-oriented architecture. It has many similarities with the architecture presented in this work, but is more extended and with a higher granularity. The communication relies on Google Protobuf [3] and services based on the REST framework. It consists of a bus, which manages the connected applications of the following types: human-machine-interfaces, bridges to communicate with other external systems, field protocol adapters, tools for calculations and automated control. The data exchange objects are represented in measurements, commands, events and alarms, which all passes the bus. The objects of the system are connected with relationships like “owns” or “uses” through a configurable semantic model. Similarly, in the Mosaic platform, which is a pure simulation platform for smart grids, devices are also configured within a semantic model [13]. Another common communication architecture, which is being introduced in the area of Smart Grid is the OPC UA [14].

III. Concept of a General Architecture

For the design of the system, the use case of controlling a tap changer based on the measured values of the smart meters was taken as base. The given hardware and external software components provided the constraints for the software design of the *Datapoint Server* and other components.

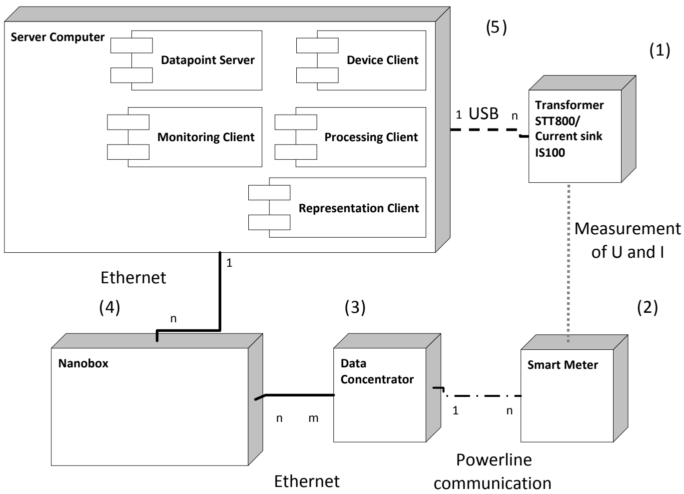

Fig. 1. System components and their communication interconnections

The system architecture depicted in Fig. 1 is derived from the following use case: the goal is to create a smart grid by using smart meter data from households for controlling a local transformer between a middle voltage grid and a low voltage grid [1]. The controlling of the grid is done in this way: In “Transformer STT800/Current Sink IS100”, three transformer devices emulate the local low/middle voltage grid transformers for each phase. The other transformers represent the photovoltaic generation in the households and the current sinks provide the loads (1). On each branch of the electric grid, there are smart meters “Smart Meter” connected, which measure the voltages and the currents (2). They are connected with a data concentrator “Data concentrator” via Power Line Communication and provide it with their measured values [6] (3). Eventually, those values are collected in a computer “Nanobox” via Ethernet (4). The “Nanobox” contains functionality for providing smart meter data as well as controlling the smart meters, e.g. with a power snapshot [7]. It is connected with the server computer “Server Computer”, which receives and processes the smart meter data. The processing results in new commands for the devices, which are sent through USB. In that way, values (U, I, P, R) can be set and read (5).

The “Server Computer” hosts a central server component, the *Datapoint Server* and several independent clients. The “Server Computer” can consist of multiple physical devices. The clients register themselves in the *Datapoint Server*, in order to be operational. The clients can be categorized into four types:

- *Device Clients*, which are gateways to the hardware devices and are used as external interfaces.
- *Management Clients*, which allows the user to configure the server.
- *Processing Clients*, which are general clients used for the execution of algorithms or actions on hardware via a device client.
- *Representation Clients*, which provide human-machine-interfaces for external representation of values and control by the user.

The clients communicate via *Datapoints*, which are managed in the *Datapoint Server*. Each *Datapoint* represents a value, which a client publishes. Other clients can subscribe datapoints, which are then pushed from the *Datapoint Server* to the subscribers. A component based approach was chosen, in order to lower system complexity as well as allowing each client to be run on other hardware and platforms. It is a centralized network architecture and it allows the usage of a push model. The push functionality is only implemented in the server, where the server notifies subscribers about updates. It can be compared to a multi agent system with one coordinator agent.

IV. Implementation

The software design will be explained in detail by describing the data model, the server, a general client and the low level communication service with help of an example.

Datapoint

The *Datapoint* consists of a logical address, a physical address and properties. The default property is the value, but other properties like time stamp or minimal and maximal values also exist. The client - which creates and writes a *Datapoint* - registers the datapoint’s physical address in the

Datapoint Server. In the *Datapoint Server*, the *Datapoint* is mapped to a predefined logical address. This mapping is equivalent to the models used in [12] and [13]. However, the difference is that much of the modeling is decentralized into the different clients instead of being kept in a single semantic model. However, semantic models are not used yet, but may be a useful extension, in order to increase flexibility. Each client manages its own subscriptions and *Datapoints*. In the *Datapoint Server* only the ownership of a *Datapoint* is managed. For subscribers, only the logical address is available. The advantage of using logical and physical addresses is that the physical address is dependent on the client creating it and if devices are exchanged, the physical address is also exchanged, but the logical address remains constant. The *Physical Datapoint* address is created in the following way:

`PH.[PhysicalAddress].[Entity].[Property]`

where the address can be built like this: "1000.USBDevice1", where "1000" is the client id and "USBDevice1" is the device name. The entity may take physical values like U (voltage) or I (current). With the hierarchical structure of the address, it is possible to use a wildcard search to receive e.g. all datapoints written by a certain client, a certain device of a certain client, a certain device type of all clients or all entities of type voltage for all devices. The *Logical Datapoint* address is put together in an equivalent way:

`S.[LogicalAddress].[Entity]`

The logical address could look like this: "Branch2.Load.Phase1". The properties are then extracted from the entity and not explicitly addressed. Here, wildcard search are also available.

Datapoint Server

The *Datapoint Server* can be seen as a database and router with extended management capability. It offers three services for clients. Clients who belong to one of the four types, mentioned in the previous chapter, offer services as well. The following service pairs are defined:

- Service-pair for device clients, where *Device Clients* registers *Datapoints*, writes values from and to hardware devices and receives new *Datapoint* values from other clients.
- Service-pair for subscribers, where *Processing Clients* and *Representation Clients* are the consumers and providers of *Datapoint* values.
- Service-pair for manager clients, where *Management Clients* can receive and set server configurations.

This architecture only allows connections of clients with the server and not the clients with other clients. Unlike service oriented architectures, there is no real service discovery or yellow pages in the system. Processing clients have to know, which *Logical Datapoints* they need to subscribe. This is set in the configuration of the clients. A future possibility could be to add semantic descriptions to the *Datapoints* and to use a match algorithm to retrieve them like in the project ORCHESTRA [4], but for a small scale system, there is no need of it at this time.

For the integration of components (server with client), the integration style "remote procedure invocation" is implemented. A remote procedure call is used as a normal method, where the *Datapoint Server* offers implemented service methods. The client implements a stub of those service methods. By executing the stub methods, the parameters are transported via Google Protobuf to the implementation, where the method is executed.

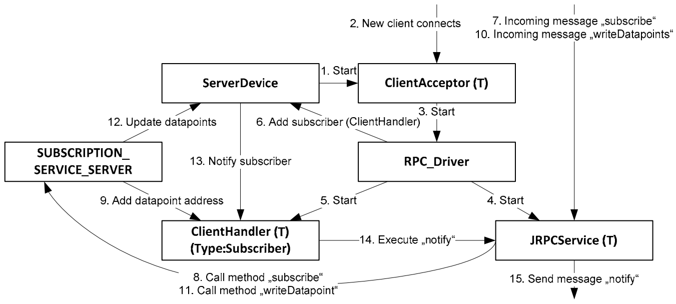

Fig. 2. Datapoint server processing structure

In Fig. 2, the server architecture is explained with an example how a subscriber is registered in the server and how it receives the subscribed *Datapoint* values. The `ServerDevice` starts a `ClientAcceptor` thread (the "T" in the figures) (1). As soon as a client connects to the socket (2) a new `RPC_Driver` is created for each new socket (3). The `RPC_Driver` is a client manager, which initializes services for the client on connect and manages the connected client in the server. The `RPC_Driver` initializes the communication service `JRPCService`, which handles all communication in the system (4). In a later chapter, the `JRPCService` is explained in detail. It initializes the correct service pair (subscription service – notify service) for the client. At the same time, a `ClientHandler` is started (5). There are three different types of `ClientHandler`, one for each service-pair. It implements the service stub of the connected client. In this example, the connected client is a subscriber and the `ClientHandler` is added to a list of subscribers in the `ServerDevice` (6). Therefore, the main role of this type of `ClientHandler` is to manage the subscriptions of each client. Then, the next step is to subscribe *Datapoints*. An incoming message "subscribe" (7) is received from the client, which contains the logical addresses of the *Datapoints*, which shall be subscribed. The `JPRCService` calls the service method "subscribe" (8) in the service `SUBSCRIPTION_SERVICE_SERVER`. This service adds the addresses to the `ClientHandler` (9). Now, a *Datapoint* is subscribed. The sending of acknowledgement messages is excluded in this explanation, in order to keep the overview. In the next step, the value of this *Datapoint* is updated. An incoming message "writeDatapoints" with new *Datapoint* values is received in the `JRPCService` (10). The corresponding method "writeDatapoint" is called in the service (11). The *Datapoint* values are updated in the `ServerDevice`, which keeps all *Datapoint* values (12).

Afterwards, all subscribers are notified (the treads are woke up) and if the *Datapoint* address matches the subscribed *Datapoint* address, the new value is sent to the subscribing client through the client service stub (14) (15). As the service `SUBSCRIPTION_SERVICE_AT_SERVER` also allows reading *Datapoints* from the server, this architecture allows both push and pull models.

Clients

In the following general client architecture is presented on which most of the *Processing Clients* rely. The architecture is described in Fig. 3 of how a client starts, subscribes a *Datapoint*, receives a new *Datapoint* value and reacts on it. The `Client` starts the `Communicator` (1), which is a common communication interface for the *Processing Clients*. The `Communicator` initializes the service for the *Datapoint Server*, the `NOTIFY_SERVICE_CLIENT` (2) and starts the `JRPCService` in an own thread (3). The service is set in the `JRPCService` (4). Then, the `Controller` is started in an own thread (5). Its purpose is to react on commands from other clients. Therefore, it is implemented as a blocking queue, in order only to change state if incoming *Datapoints* requires it. Two examples will demonstrate the functionality. The client subscribes a *Datapoint* with the commands "START" and "STOP". If the "START" command is set, the `Controller` executes some actions until the "STOP" command is given. In another case the client shall only react on the change of some subscribed *Datapoint* values and the controller always waits between the notify messages. Further, the `Controller` starts the `Manager` (6), which is the executor of all actions of the client. As soon as all components are started, the `Client` sends the *Logical Datapoint* addresses to the server, which shall be subscribed (7) (8) (9).

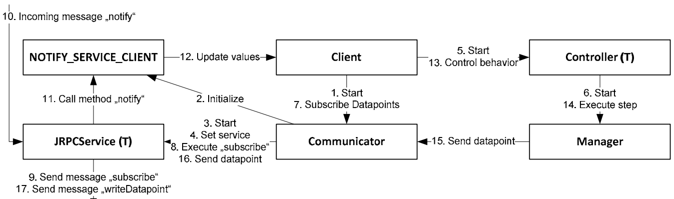

Fig. 3. General processing structure in clients

As the subscribed *Datapoint* value changes, the *Datapoint Server* sends a message "notify" with the new value (10). The method "notify" is called in the service (11). The new values are passed through the `Client` (12) to trigger some behavior of the `Controller` (13). If a defined behavior is triggered, the `Manager` is executed, which processes subscribed *Datapoints* or calculates new values to be set (14). The new values are sent to the *Datapoint Server* (15) (16) (17), where another client may be notified about these changes.

Intercomponent Communication

For this platform a TCP/IP based asynchronous RPC service based on the service concept of Google Protobuf and it uses only one socket. Google Protobuf is "a way of encoding structured data in an efficient yet extensible format" [3]. Originally, it was supposed to use ZeroMQ [5] with Protobuf, but as it was not possible to use both a publisher-subscriber and a request-response pattern on a single socket, which is realized in the `JRPCService`.

The architecture of the `JRPCService` is described in Fig. 4 and Fig. 5. It is explained with the example of the subscription service. The service-pair consists of the `SUBSCRIPTION_SERVICE_SERVER` and the `NOTIFY_SERVICE_CLIENT`. All data types, services and their methods are defined within Protobuf [3].

The `JRPCService` manages all communication and additionally allows the replacement of services. At the connection with a client, the server does not know which type of client is requesting and therefore services have to be set first, see steps 1-6 in Fig. 4. As explained before, in the server, a client manager, the `RPC_Driver` is defined. It implements a `JRPCServiceCallbacks` interface, which demands the implementation of the method "NewServiceRequest". In that way, services can be changed by putting a new "connect" request with a service identifier. If the request "connect" with the service name "SUBSCRIPTION_SERVICE_SERVER" is received from a client (1), the `RPC_driver` starts the `JRPCService` (2) in a separate thread (marked with "T" in the figure). In that way, each accepted socket (client) has an own independent `JRPCService` available. On "connect", the method "NewServiceRequest" is executed in the `RPC_driver` (3). The service `SUBSCRIPTION_SERVICE_SERVER` is started (4) and it is set in the `JRPCService` (5). Additionally, a `ClientHandler` is started, which is adapted to the connecting client (6), in this type a subscriber.

For each incoming request, an `IncomingWorkingPackage` is started in an own thread (7). It creates a predefined empty response message as a thread, the `RPC_Response_CallBack` (8). This message is passed to the service in the method call "connect" (9). At the end of the method execution in the service, the message thread is started; it builds itself with the content of the service method and independently executes the send function from the `JRPCService` (10). In case of the message "connect", only an acknowledgement is returned. The message "acknowledge" is then sent to the client via the `JRPCService` (11). For incoming messages, like messages to execute the method "subscribeDatapoints", the steps 1 and 7-10 in Fig. 4 are executed.

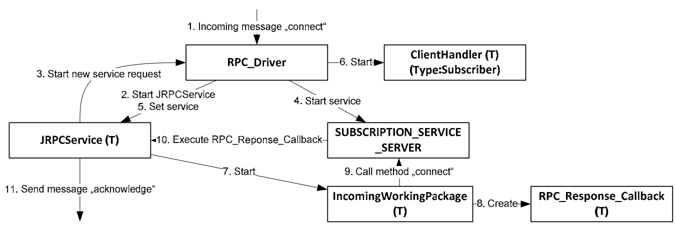

Fig. 4. JRPCService processing for incoming requests

In step 6 in Fig. 4, a ClientHandler was initialized for the client. In this example, the client handler is used to update the client with subscribed *Datapoint* values. In Fig. 5, this process is illustrated. First, a remote method is executed at the client (1), e.g. to notify a subscriber. The remote method of the stub `rpc.NOTIFY_SERVICE_CLIENT.BlockingInterface` is called (2) and the executing party waits for response during the function call. The stub-method then calls method "CallBlockingMethod" (3), which is demanded by Protobuf and implemented within the `JRPCService`. It sends the request to the client (4) and additionally creates a `BlockingQueue` (5). There, the `BlockingQueue` waits until the response is received (6) or a timeout is exceeded. The response is the trigger to wake the `BlockingQueue` (7), which is woken up (8). Finally, the response is returned (9) from the stub-function to the caller.

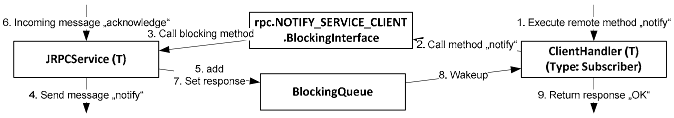

Fig. 5. JRPCService processing for outgoing requests

Security Aspects

Since Google Protobuf does not implement a client server authentication, messages within the system are unencrypted and readable for all devices within the network. However, since the communication relies on TCP/IP a SSH/VPN tunnel can be used to harden the information exchange between the substations and the server.

V. EXISTING IMPLEMENTATIONS

The general communication architecture is being realized within two projects, which are presented in the following.

Intelligent Low Voltage Grid

This project is the realization of the defined use case shown in Fig. 1. The *Datapoint Server* is implemented as described above. Most of the following clients are implemented based on the general client architecture. *Device* and *Smart Meter Clients* are the gateways for other clients to access the devices STT800 and IS100 for write and read. The *Device Client* communicates through USB and the *Smart Meter Client* through Ethernet. The *Smart Meter Client* provides two ways of reading from the smart meters: continuous independent reading of voltage values from the smart meters and power snapshot [7], where all connected smart meters are read with the same time stamp. Different to the *Device Clients* above, *Algorithm* is a *Processing Client*, which subscribes *Datapoints* from the *Device Clients*, which represent the voltage values of the loads (IS100) and the photovoltaic power generators (STT800). Based on the voltage limit violations of the *Datapoints*, the new tap position is calculated. The tap position is a discrete percentage value of a default position, e.g. tap position 1 is 230V * 1.02. Five tap positions are available. It is used to control the local transformers (STT800). The *Tap Changer* subscribes the tap position, which was provided by the *Algorithm*. Based on the given tap position, it writes new transformer values, which are set by the *Device Client*. The *Profile Client* generates the emulation values for all devices in the system. Default load and generator profiles of households are downscaled for this application. It writes the currents to the devices. 24h profiles are used, but are run through within 2 min. Consequently, the *Profile Client* also provides the system time. In order to know when to start and stop, it subscribes a *Datapoint* with the commands "START", "PAUSE" and "STOP" from the *User Interface Client*. The *User Interface Client* provides the human-machine-interface of the system. Unlike all other software, it was implemented in National Instruments LabVIEW and the corresponding communication driver (`JRPCService`) was implemented in Visual C++. Finally, the *Manager Client* is used for configuring the *Datapoint Server*, i.e. the mapping between the *Physical* and *Logical Datapoints* and to discover mapping error of devices.

Smart Heating Control

Another realization of the general communication architecture is a building automation application for controlling home heating systems. This "Smart Heating Control" system has been developed at Vienna University of Technology on base of the communication architecture described in this paper. The main goal was the development of a flexible, extensible and cheap state-of-the-art building automation system for controlling private households.

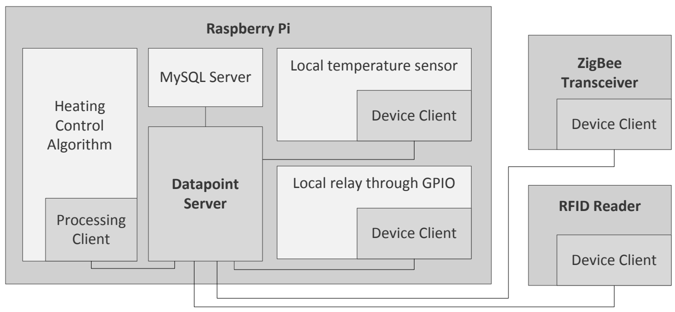

Fig. 6. Overview of home automation application relying on the datapoint communication architecture

In Fig. 6 a structural overview of the smart heating system is shown. It consists of a *Datapoint Server* and two *Device Clients*. One is used as a driver for a local I2C temperature sensor and the other is used as a driver for a mono stable relay to switch the heating system. They were implemented on a Raspberry Pi single-board-computer operating an ARM1176JZF-S processor [15]. Beside the *Datapoint Server* and the *Device Clients*, this credit-card size low cost computer hosts a *Processing Client* for the temperature control algorithm.

Additionally to the previous project, a data persistence extension in the *Datapoint Server* has been made. It uses a MySQL server to persist the historical data of the different sensors. The added persistence functionality is a crucial element for the heating control system, as its algorithm is meant to also use historical data for forecasting and decision making. Further, the *Device Clients* had to be adapted for proper system operation: The first one connected to an RFID reader unit (OMNIKEY ® 5553 Reader Board) operated by a

Device Client implementation. This unit acts a presence recognition unit, which is located at the key rack and reads RFID tags connected to the residents' keys. When detecting a newly added or removed key it updates the corresponding *Datapoint* in the *Datapoint Server*. This information is then forwarded to all subscribing clients. The second *Device Client* implementation operates a wireless 2.4 GHz ZigBee transceiver (Atmel AVR Raven [16]), which acts as a communication host towards some wireless temperature sensors.

For the implementation, temperature measurement was possible only through the wired transceiver. In further development stages, room thermostats should be connected wirelessly as well, which would provide the possibility not only to control the central house heating system, but every room individually.

VI. Results and Conclusion

A smart grid in small scale is emulated by transformers and current sinks. For the realization, a service concept, the "JRPCService" was introduced based on Google Protobuf. As the communication layer is hardly a cause of errors, it can be seen as very robust and flexible. The complexity of the *Datapoint Server* could possibly be reduced, if the usage of *Physical Datapoints* would be removed. Actually, only the clients have to know the physical addresses of their devices. For all levels above, only *Logical Datapoints* are needed. Hence, *Physical Datapoints* assure that only one client can "own" a *Datapoint*, due to the predefined mapping. In "Smart Heating Control", one drawback was the non-persistence of data in the *Datapoint Server* as no historical data can be subscribed and obtained by any client. Therefore, an extension of the *Datapoint Server* had to be made. New clients can easily be created from the common client architecture. The push model worked well for all clients except the *User Interface Client*. Here, problems with message flooding from the server emerged in the interface of the tow used programming languages. A pull model was chosen where the client has all control over the data flow. Originally, the purpose was to implement business logic within the *User Interface Client*. It showed up to be very faulty, which made it necessary to split the system functionality in many independent components, such as the *Profile Client*. In general, one of the major challenges was to create a well working system that integrated different types of hardware and software. The selected counter measure was to reduce the systems overall complexity and to use several, simple built, independent components or clients.

This software architecture was successfully implemented in two projects. Although, related software architectures provide proper frameworks, this architecture allows complete customization of the concepts needed for this application and as this test facility is still small, the need for scalability is limited. In the project "Intelligent Low Voltage Grid", the system will be extended with a building automation agent and e-mobility, where the demand for extending the platform with other types of clients rises.

VII. Acknowledgment

The development of the system in this work was supported by Siemens AG Austria.

VIII. References

[1] T. Deutsch, T. Leber, F. Kupzog, Ö. Karacan, A. Einfalt, "Versuchsanlage Intelligentes Niederspannungsnetz", in proceedings of the *8. Internationale Energiewirtschaftstagung IEWT 2013*, Austria, 2013.

[2] J. P. Barton, D. G. Infield, "Energy Storage and Its Use With Intermittent Renewable Energy". In *IEEE Transactions on Energy Conversion 19*, p. 441–448, 2004.

[3] Google Protocol Buffers, https://code.google.com/p/protobuf, accessed on April 17th 2013.

[4] U. Bügel, D. Hilbring, "Application of Semantic Services in ORCHESTRA", in the proceedings of the *International Symposium on Environmental Software Systems ISSES 2007*; Prague, Czech Republic, May 22-25, 2007.

[5] ZeroMQ, http://www.zeromq.org, accessed on April 17th 2013.

[6] A. Einfalt, F. Kupzog, H. Brunner, A. Lugmaier, "Control strategies for smart low voltage grids - the Project DG DemoNet - Smart LV Grid", in proceedings of *CIRED 2012 Workshop*: Integration of Renewables into the Distribution Grid, page 238, 2012

[7] A. Abart, D. Burnier, B. Bletterie, A. Lugmaier, A. Schenk, M. Stifter, H. Brunner, "Power snapshot analysis: a new method for analyzing low voltage grids using a smart metering system", in proceedings of *21st International Conference on Electricity Distribution*, Germany, 6-9 June 2011, Paper 1083, 2011.

[8] G. Zucker, F. Kupzog, D. Reiter, "Smart grids strategy for salzburg, austria", in proceedings of *21st International Conference on Electricity Distribution*, Germany, 6-9 June 2011, Paper 0787, 2011.

[9] S. Scalari, G. Valtorta, R. Giglioli, F. Pilo, R. Caldon, S. Massucco, C. A. Nucci, A. Testa, "An Italian facility to test distributed energy resources management for SmartGrids", in *SmartGrids for Distribution, IET-CIRED*. CIRED Seminar, 23-24 June 2008, p. 1-4, 2008

[10] A. Wendt, K. Pollhammer, "Turning Buildings into Active Participants of a Smart Grid". In proceedings of the *ComForEn 2012*, 5th September 2012 (Tagungsband ComForEn 2012), p. 74-80, Eigenverlag des Österreichischen Verband für Elektrotechnik, ISBN: 978-3-85133-072-4, Austria, 2012

[11] K. Pollhammer, A. Wendt, "BED – Balancing Energy Demand with Buildings". In proceedings of the *ComForEn 2012*, 5th September 2012 (Tagungsband ComForEn 2012), p. 110-115, Eigenverlag des Österreichischen Verband für Elektrotechnik, ISBN: 978-3-85133-072-4, Austria, 2012

[12] Green Energy Corp, "Reef Reference Documentation", http://docs.plymouthsystems.com/0.4.7/reef-docs-single/Overview.html, accessed on April 18th 2013

[13] S. Schütte, S. Scherfke, M. Sonnenschein, "mosaik - Smart Grid Simulation API". In Proceedings of S*MARTGREENS 2012 - International Conference on Smart Grids and Green IT Systems*, edited by B. Donnellan, J. P. Lopes, J. Martins, and J. Filipe, 14-24: SciTePress, 2012.

[14] S. Rohjans, D. Fensel, A. Fensel, "OPC UA goes Semantics: Integrated Communications in Smart Grids", in Proceedings of the IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA), E-ISBN: 978-1-4577-0016-3, 2011

[15] Raspberry Pi Foundation; "Raspberry Pi | An ARM GNU/Linux box for $25. Take a byte!"; http://www.raspberrypi.org, accessed on April 23th 2013

[16] Atmel Corporation; "AVR Raven"; http://www.atmel.com/tools/avrraven.aspx, accessed on April 23th 2013

[17] T. Deutsch, F. Kupzog, T. Leber, A. Einfalt, „Pilot System „Intelligent Low Voltage Grid", in proceedings of the „ CIRED 2013: International Conference on Electricity Distribution", to be published.