Lab Project

Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

Smart Heating System in Residential Areas

Cooleren Titel finden

by Michael Spiegel
and Samuel Schmid

Spring 2015

ETH student ID: 10-915-312 (Michael Spiegel)
10-919-991 (Samuel Schmid)

E-mail address: spiegelm@student.ethz.ch
schmisam@student.ethz.ch

Supervisors: Dr. Wilhelm Kleiminger
Prof. Dr. Friedemann Mattern

Date of submission: January 5, 2012

Abstract

English abstract.

Zusammenfassung

Zusammenfassung auf Deutsch.

Contents

1 Introduction

Motivate your problem and outline the contributions of the thesis. [2]

As energy consumption is growing world wide, the need for technologies to control and analyze energy consumption increases. Therefore a lot of tools exist, that show the energy consumption of a household, business or certain devices. But most of these technologies and tools are very complex for the end user, as most of the times the user is not a domain expert. This results in the need of easy to use tools that provide the end user with enough information to analyze the data but not overwhelm him with unnecessary details.

One of the areas in which easy to use tools are needed, is heating control. For this specific area we developed our project, which consists of two parts: the infrastructure and the mobile application. The infrastructure part gets the heating data and analyzes it while the mobile application provides an easy to use and well understandable visualization of the data.

With this project we want to fulfill this need and offer a easy to use mobile application that enables the user to control and analyze ... .

2 Related Work

Put related work here [2].

2.1 Prototype infrastructure in university offices

Nico Eigenmann: Prototype infrastructure in university offices [1].

technologies used: CoAP, REST wenig wissenschaftlich: Django, Android

3 Requirements Elicitation

3.1 Functional Requirements

Functional requirements are stated using use cases in the style of Martin Fowler:

> Add 2 sentences describing this style and put the links in footnotes or references.

  http://en.wikipedia.org/wiki/Use_case#Martin_Fowler
http://ontolog.cim3.net/cgi-bin/wiki.pl?UseCasesMartinFowlerSimpleTextExample

3.1.1 Register a Residence

> Add one to two sentences explaining what the user wants to achieve (i.e. user has just bought the system and wants to enable smart heating in her residence.

1. User opens the app and opens the registration screen

2. User scans RFID tag

3. System checks if RFID is not yet registered

4. System registers residence with scanned RFID

5. System shows empty home screen with no rooms

Alternative   RFID already registered

- At step 3, system fails to verify that RFID is not yet registered

- If RFID is associated to a residence

    - System shows the home screen of the registered residence

- If RFID is associated to a thermostat

  - System shows ???

4 System Overview

The system consists of two major parts. First, the infrastructure gathering and controlling temperature and organisational data. Second, the mobile application offering a user interface to view and control the temperature data and residential information. See Figure 4.1 for a graphical overview.
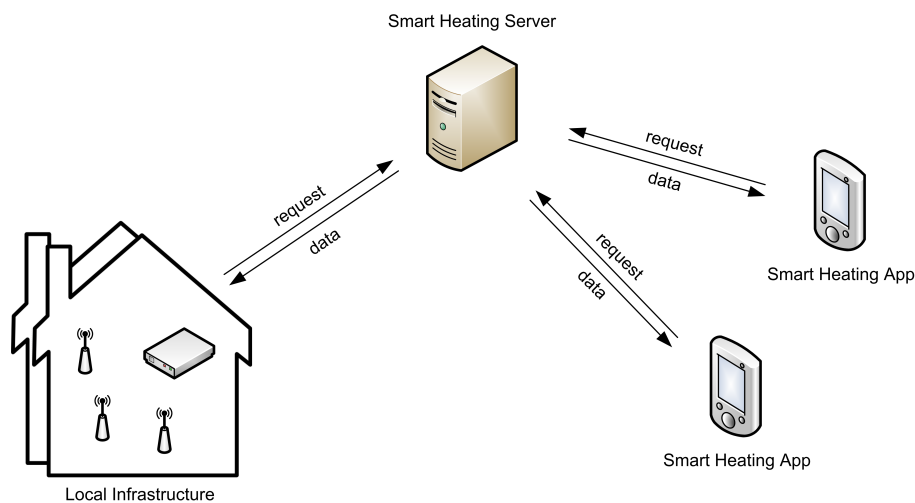
Figure 4.1: System Overview

The infrastructure consists of a local residential part and a server part. Within the residence a small computer system is installed which serves as a communication gateway to the distributed low-power thermostats. The other part is the server acting as the central entity to collect and store the accumulated sensor data and organizational data. Both parts are explained in Chapter 5.

The mobile application part . . .

SAMUEL: Mobile App text

4.1 Architecture

The smart heating system is implemented using a server-client pattern. The smart heating server provides an API which is used by the local communication gateway and the mobile application.

4.2 Models

Throughout the project there is a shared system model describing the underlying entities. The system model depicts the physical objects and places that are used for data collection or organizational purposes. The following paragraphs describe the applied models and their associated design decisions.

Residence The fundamental unit of each deployment is the residence. Each residence corresponds to an installed local communication gateway. Further details are described in Section 5.2

User Each residence can contain multiple users. A user is associated to exactly one residence and is identified by his smart phone serial number. This design decision simplifies the system design and especially the user authentication. It also prevents a user from using the same identity when accessing the system from multiple smart phones.

Room and Thermostat A residence is divided in rooms where each room can contain several thermostats. A room is a simple organizational approach to group multiple thermostats into a single unit.

Heating Table Each thermostat has an associated temperature schedule called heating table. The heating table is responsible for mapping each day and time in a week to a target temperature. The heating table is a periodic schedule repeating each week. This design decision was chosen for infrastructure simplicity as well as to reduce usability complexity.

Meta Entries Meta entries persist time depending meta information about thermostats. A meta entry consists of the received signal strength, up-time, battery level

and an associated timestamp. This data can be used to identify issues regarding the thermostat devices such as wireless connection problems or drained batteries.

5 Infrastructure

The infrastructure implements a server-client design pattern and is divided into two parts: The *server* part and the *local* part. The server is an independent unit providing a public API to its clients. The local part is a client using the provided API to retrieve configuration from the server and populate it with data gathered from the deployed residential infrastructure. Both parts will be explained in the following sections.

5.1 Server Infrastructure

The smart heating server is the central storage and communication center of this project. It persists data collected by the local deployment as also organizational information and heating schedules provided by the user via the Mobile App.

5.1.1 Design Goals

- *Modularity* for independent, interchangeable components for improved maintainability.

- *Extensibility* to easily add new resources and relationships.

- *Usability* to allow developers simple inspection of the API structure and modification of resources.

- *Testability* for good and comprehensible tests and high software quality.

5.1.2 Platform and Frameworks

The server is implemented with Python, a general-purpose, multi-paradigm programming language[1]. Python was chosen as it is suitable for developing a solid web server as well as the embedded hardware used for the communication gateway in the local deployment. See Section 5.2 for more explanation. Django is a popular, open-source web application framework[2] based on a model-view-controller (MVC) pattern facilitating the development of complex, database-driven web applications. The Django REST Framework[3] extends Django to support the design of RESTful[4] Web APIs.

5.1.3 RESTful API

The smart heating server provides a RESTful API to access a persistent storage providing the basic CRUD operations: Create, Read, Update and Delete. Representational State Transfer (REST) is a programming paradigm used for machine-to-machine communication in distributed systems. RESTful web services use HTTP as their preferred communication protocol.

REST is about resources. Each resource is identified by a uniform resource identifier (URI) and is accessible via the request methods defined in HTTP. We differentiate two major types of URIs: Resource collections and resource representations. A resource representation is a view of its resource's state and is encoded in a transferable format. This project uses the simple JavaScript Object Notation (JSON) format to represent resources. Resource collections contain lists of representations of the same type of a resource.

Design Decisions

1. The underlying model hierarchy is represented via nested URLs.

2. The resource identifier is the first field in each resource representation.

3. Within resource representations, collections are referenced via URL. Resources are referenced by including their representation.

---

[1] https://www.python.org/

[2] https://www.djangoproject.com/

[3] http://www.django-rest-framework.org/

[4] Note: REST is a programming paradigm where as RESTful is used to describe a web application implementing such a paradigm.

4. URL fields are identified by the name `url` or the suffix `_url`. Each resource representation contains its own URL in the field `url`.

See Listing 5.1 for an example of a resource representation.

```
1   {
2     "id": 2,
3     "url": "http://server/residence/04891BB9232584/room/2/",
4     "name": "Dining Room",
5     "residence": {
6       "rfid": "04891BB9232584",
7       "url": "http://server/residence/04891BB9232584/",
8       "rooms_url": "http://server/residence/04891BB9232584/room/",
9       "users_url": "http://server/residence/04891BB9232584/user/"
10    },
11    "thermostats_url": "http://server/residence/04891BB9232584/room
         /2/thermostat/"
12  }
```

Listing 5.1: Example representation of the Room resource at `http://server/residence/04891BB9232584/room/2/`. The `url` field determines the URL of the represented resource. Within the `residence` field the representation of the associated Residence resource is nested. The included Residence representation has its own `url` field. Collections of the Residence's Rooms and Users are not nested but referenced via URL to limit the response size.

Browsable API   The Django REST Framework offers the ability to dynamically generate a browsable interface accessible via web browser. It includes a formatted output of the JSON response, offers forms to add new resources and edit or delete existing resources, and displays URLs as clickable hyperlinks. This interface is an additional HTML output format enabling developers to easily interact with the API without the need of external tools. The Browsable Web API is designed for readability whereas the API renders its data as compressed JSON to reduce transmission bandwidth.

MICHAEL: "bandwith" - Can't think of the correct word..

See Figure 5.1 for an example.
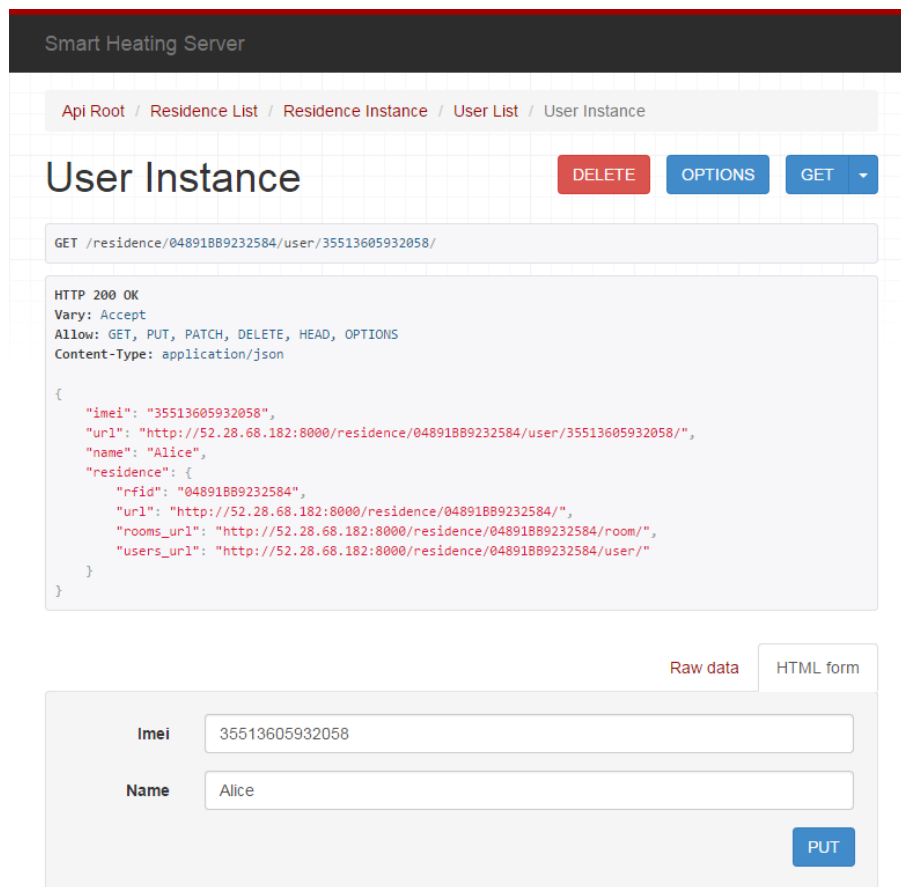
Figure 5.1: Screenshot of the Browsable Web API showing an instance of a user resource. The interface allows the developer to easily edit or delete the user. Navigating the residence or the associated room or user collections can be done by clicking on the URLs of the respective resources.

5.1.4 Program Architecture and Implementation

The server implementation is built on Django following a variation of the Model-View-Controller (MVC) architectural pattern. We assume the reader to be familiar with the common MVC pattern[5].

All the code required to setup the smart heating server is provided in the GitHub repository located at `https://github.com/spiegelm/smart-heating-server`. We highly encourage the reader to visit the project on GitHub. The main page of the repository provides a short explanation and also contains a hyperlinked image indicating the status of the latest build run on the continuous integration service as explained later in Section 5.1.5. The repository is designed to contain all code and

---

[5]We refer the interested reader to `https://en.wikipedia.org/wiki/Model-view-controller`

dependency information necessary to automatically install and run the application on a server.

The following sections describe the relevant application components in detail.

5.1.4.1 Models

Models structure the underlying data and provide operations for manipulating it. In Django models contain the business logic and are also responsible for validating user input and providing appropriate error messages. The `smart_heating.models` namespace contains all project models. Each model extends the abstract `Model` class, providing a general method used to get an objects representation for debugging purposes as well as the abstract `get_recursive_pks` method. This method is used to generate hierarchical URLs and will be further described in Section 5.1.4.2

Thermostats are organized within rooms and belong to a residence. RaspberryDevice and ThermostatDevice represent the deployed physical devices. Both are used as a global configuration of all devices that are planned to be eventually installed in a residence. This configuration associates a device's MAC address to the number of the radio-frequency identification (RFID) tag attached on the device. This is required to relate these two identifiers, as the hardware cannot read the RFID tag number from the device casing and the mobile application cannot access the internal MAC address used by the devices. Storing this mapping on the server benefits the deployment of the local devices as every device can use the same code and no configuration of the devices is required. See Figure 5.2 for a graphical representation of the used models. The most important models will be described in the following paragraphs.

Residence   is identified by the RFID tag number on the deployed local communication gateway. A residence combines users and rooms with their associated thermostats and data into a single encapsulated unit.

User   is identified by the smart phone's serial number[6] and can be registered to at most one residence.

---

[6]The International Mobile Equipment Identity (IMEI) is a 15-digit serial number associated to each GSM cell phone.
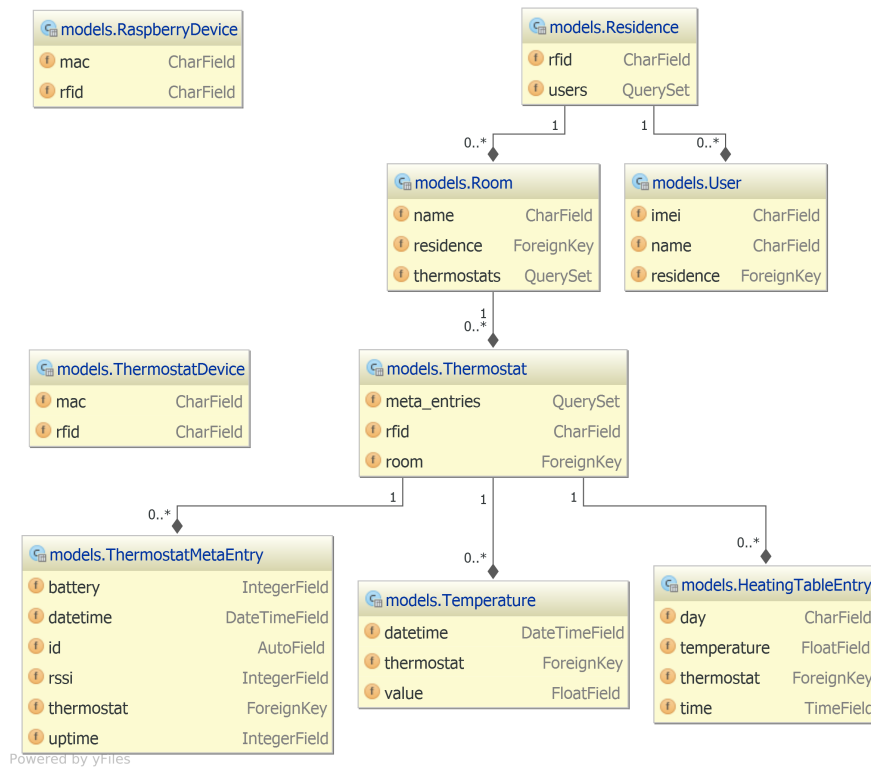
Figure 5.2: UML class diagram of the used models.

Room groups thermostats into an organizational unit. Each residence can contain multiple rooms.

Thermostat is identified by the number of the attached RFID tag. It is the parent of the models used for the historic temperature and thermostat meta data and the heating schedule.

Temperature represents a single temperature measurement perceived by a thermostat at a specific date and time.

ThermostatMetaEntry is used to combine meta data that is available by the digital thermostat such as battery level, system uptime and received signal strength indication (RSSI).

HeatingTableEntry is used to compose the heating schedule for a thermostat. An entry determines the target temperature that applies starting from the given day and time until the next heating table entry occurs.

5.1.4.2 Serializers

Serializers are responsible for translating models into a resource representation and vice versa. This project uses JSON to represent resources. Each serializer has an associated model and determines which model fields should be included into the resource representation. Additional fields can be added to the representation or individual field representations can be overwritten to offer customized output.

A commonly used customization is the replacement of the default `Hyperlinked IdentityField` with `HierarchicalHyperlinkedIdentityField`. This custom serializer field generates URLs according to the hierarchical schema applied in this project. For example the URL for a room would be `http://server.com/residence/041FB2B9232580/room/5/`. To assemble this URL the identifier of the room and its parent are required. The adapted field extends the default field and provides all identifiers of the hierarchy required to generate the URL by using the hierarchical base class `smart_heating.models.Model`.

Nested resources are implemented using nested serializers. This way existing serializers can be reused to include their resource representation into another representation. Refer to line 3 in Listing 5.2 for an example usage of a nested serializer.

```
1  class RoomSerializer ( serializers . HyperlinkedModelSerializer ):
2    url = relations . HierarchicalHyperlinkedIdentityField ( view_name ='
         room - detail ', read_only = True )
3    residence = ResidenceSerializer ( read_only = True )
4    thermostats_url = relations . HierarchicalHyperlinkedIdentityField (
         source ='thermostats ', view_name ='thermostat - list ',
5    read_only = True )
6
7    class Meta :
8      model = Room
9      fields = ('id ', 'url ', 'name ', 'residence ', 'thermostats_url ')
```

Listing 5.2: The `RoomSerializer` class as an example of the usage of the `Hierarchical HyperlinkedIdentityField` and nested serializers.

5.1.4.3 Views

A view is a class which processes requests and returns a response. Django defines the view as the place that not only defines how but also which data is presented. This slightly differs from the traditional MVC architectural pattern but will not be explained in detail here[7]. Django includes generic view classes and mixins to provide common functionality and to facilitate code reuse. The Django REST Framework further abstracts views by supplying `ViewSets`. These `ViewSets` facilitate the unified handling of different HTTP methods within a single class for each public resource.

The used `ModelViewSet` is such a class serving as a base for individual view sets representing a resource. In the very simplest case only the definition of a database query and an appropriate serializer class is required. The `ResidenceViewSet` is a good example of a fully functional resource implementation supporting all CRUD methods while only requiring three lines of code, as shown in Listing 5.3.

```
1  class ResidenceViewSet(viewsets.ModelViewSet):
2    queryset = Residence.objects.all()
3    serializer_class = ResidenceSerializer
```

Listing 5.3: The `ResidenceViewSet` class.

Pagination   is the practice of partitioning lists into smaller pieces. In this project pagination is used for expectedly large resource collections such as temperatures or meta entries. The applied pagination style depends on two parameters: `limit` and `offset`. Limit determines the page size, i.e. the maximum number of items that are displayed simultaneously. Offset determines the number of the starting item. The custom pagination class `pagination.BasePagination` implements the design decision 4 to suffix resource field names representing links.

5.1.4.4 Routes

URLs are defined explicitly in the file `smart-heating/urls.py`. For each request the responsible view is determined by matching the requested URL against a list of regular expressions. This technique allows a flexible and clean URL schema as required for our hierarchical resources.

---

[7]For more details we refer the reader to `https://docs.djangoproject.com/en/1.8/faq/general/#django-appears-to-be-a-mvc-framework-but-you-call-the-controller-the-view-and-the-view-the-template-how-come-you-don-t-use-the-standard-names`

5.1.5 Automated Testing

Automated software testing is an important part of this project part. Django and also the Django REST Framework facilitate automatic testing by providing base classes and tools helping to create and execute tests.

The tests are designed to infer the functionality and behavior of the server application. An executed test run shows at any particular time which functional requirements are fulfilled and which are not. Look at Listing 5.4 for an example. These dynamically generated test reports nicely complement traditional documentation and the tests also provide small informational code examples that are shown to work. During application development we intensively used test driven development (TDD) practices to increase productivity and achieve high software quality. Automatic software testing allows to formulate the requirements of a computer program such that they can be automatically checked already before and during application development.

```
$ python manage.py test -v 2
  [...]
  smart_heating.tests.test_api.ViewRootTestCase
    test_root_contains_residence_url ... ok
  smart_heating.tests.test_api.HeatingTableTestCase
    test_create_heating_table_entry ... ok
    test_heating_table_entries_are_ordered_by_date_and_time ... ok
    test_heating_table_entry_representation ... ok
    [...]
  smart_heating.tests.test_api.ViewUserTestCase
    test_create_user ... ok
    test_get_non_existent_imei_is_404 ... ok
    test_get_user_of_unrelated_residence_is_404 ... ok
    test_user_collection_contains_user_representations ... ok
    test_user_representation_contains_imei_and_name ... ok
    [...]
----------------------------------------------------------------------
Ran 58 tests in 0.765s
OK
```

**Listing 5.4:** Excerpt of the test output documenting the application behavior

Furthermore the usage of an Continuous Integration service like Travis CI[8] ensures that individual development branches are consistently tested and don't break the

---

[8]https://travis-ci.org/

main development line upon branch integration. Additionally this convenient services ensures the periodic execution of all tests and logging of the test results.

5.2 Local Deployment

The local deployments consists of the residential communication gateway and the deployed digital thermostats with their wireless adapters. The communication gateway collects the data read from the thermostats and sends it to the remote web server. The digital thermostats are programmable and allow us to modify their behavior by flashing custom firmware. This project uses the work of a previous master thesis as a basis to build upon [1]. The primary focus is to improve the basic functionality of the communication gateway and create an unified but loosely coupled infrastructure by using the RESTful API provided by the server. See also Figure 5.3 for an overview of the local deployment.
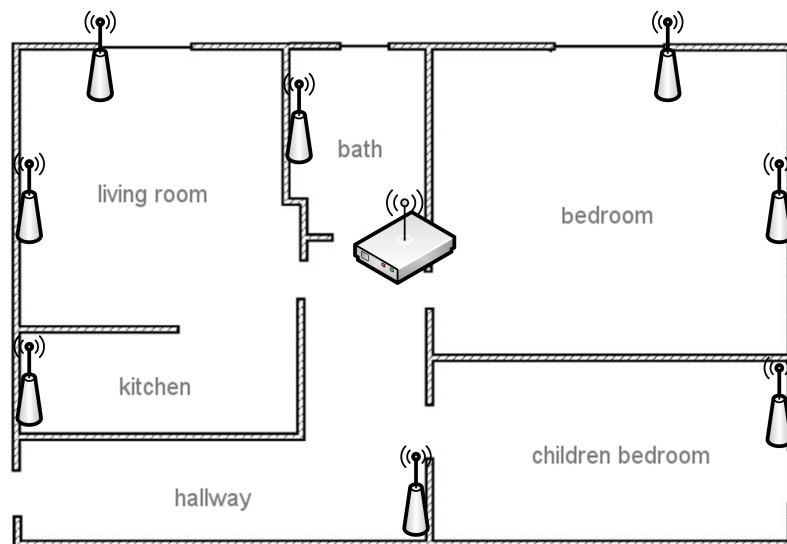
Figure 5.3: Example of a residence layout depicting a possible deployment. The local communication gateway is installed in the hallway, connected to the internet and has wireless connections to the deployed thermostats represented as antennas. Source of the original image: `http://www.haus-topplicht.de/wp-content/uploads/2013/12/planwohnung2.jpg`

5.2.1 Existing Infrastructure

This project builds upon work previously done by Nico Eigenmann as his master thesis [1]. Part of his work consisted of extending the hardware and software of

the digital thermostat HR-20[9], as depicted in Figure 5.4, to offer wireless access via 6LoWPAN[10]. Further a border router translates the 6LoWPAN messages into IPv6 packets and vice versa. This border router is connected per Universal Serial Bus (USB) port to a computer and communicates via Serial Line Internet Protocol (SLIP). The computer redirects the received packets into the connected Local Area Network (LAN) and also routes messages addressed to a sensor in the 6LoWPAN network via the border router.

Figure 5.4: Honeywell Rondostat HR-20 programmable thermostat. Source: `https://piontecsmumble.files.wordpress.com/2013/02/hr20.jpg`

5.2.2 Design Goals

Several design goals are determined in order to evaluate the system design and implementation.

- *Performance* for low computational effort and power consumption on embedded systems.

- *Reliability* for a high probability that the system operates as expected.

- *Robustness* to let the system behave reasonably in presence of failures, especially connection problems.

- *Interoperability* to cooperate with other distributed systems.

---

[9]`http://www.homexpertbyhoneywell.com/en-DE/Products/rondostat/Pages/HR-20.aspx`
[10]6LoWPAN is an acronym of IPv6 over Low power Wireless Personal Area Network

5.2.3 Software Platform and Frameworks

This sub section lists and describes the software components used on the local communication gateway.

tunslip6 is a tool to route IPv6 packets between the host and a border router via serial line. It is used to create a tunnel interface an the host system which acts as a regular network interface.

Python is a multi-paradigm programming language and is suitable for desktop as well as for embedded systems. This allows us to use a single programming language for the implemented software parts in the whole infrastructure on the local and server side. Further Python includes packages to facilitate asynchronous input and output which is required to concurrently query multiple network resources.

Constrained Application Protocol (CoAP) [3] is a application protocol created for low power devices and sensors that are heavily restricted in terms of computing power, memory size and power consumption. Unlike the Hypertext Transfer Protocol (HTTP) commonly used in desktop and mobile systems, CoAP is based on minimalistic network protocols to reduce overhead and computational requirements.

aiocoap is a third-party package for Python implementing the CoAP protocol.

requests is a library written in Python providing a simple and well designed interface to send HTTP requests. It is used to communicate with the API provided by the smart heating server.

5.2.4 Implementation

The implementation work on the local infrastructure focuses on the local computer acting as a local communication gateway, this two terms are further used interchangeably. We use the Raspberry Pi 2 Model B[11] as our local communication gateway which is responsible for two major tasks.

---

[11]`https://www.raspberrypi.org/products/raspberry-pi-2-model-b/`

First, it needs to periodically query the temperatures and other meta data from the digital thermostats and store the data locally. Additionally the heating scheduling algorithm determines the current target temperature and applies the value to the thermostats. Second, the local computer consistently synchronizes the local data storage with the remote smart heating server. The new data gathered from the thermostats is therefore sent to the server and the latest configuration such as the heating schedule is downloaded.

The local computer works as a proxy server and enables the local deployment to operate independently from the connection to the remote server. This way the last downloaded heating schedule is kept and operated until the server connection is reestablished.

All the code, binaries and instructions required to setup the Raspberry Pi 2 are provided in the Git repository located at `https://github.com/spiegelm/smart-heating-local`. We highly encourage the reader to visit the project on GitHub. The repository is designed to be self contained. This means in order to setup a local communication gateway only the hardware, a copy of the repository and an internet connection are needed. A step-by-step instruction to setup the software on a Raspberry Pi 2 is included in the `README.md` file.

5.2.4.1 Border Router Connection

Upon connection of the border router at an USB port the corresponding rule in `/etc/udev/rules.d/90-local.rules` is executed. The rule executes the file `bin/start_tunslip.sh` in the code repository to establish a tunnel to the border router using tunslip6. The border router runs a web server providing information about connected thermostat devices. The log file `/var/log/tunslip6` generated by tunslip6 shows the associated web server address:

```
Server IPv6 addresses:
 fdfd::212:7400:115e:a9e5
 fe80::212:7400:115e:a9e5
```

The address with the prefix `fdfd:` is the one we want. Requesting the determined URI `http://[fdfd::212:7400:115e:a9e5]` returns an HTML table containing all discovered devices:

```
<html ><head ><title >ContikiRPL </title ></head ><body >
Neighbors <pre >fe80 ::221:2 eff :ff00 :228b </pre >
Routes <pre >fdfd ::221:2 eff :ff00 :228b /128 (via fe80 ::221:2 eff :ff00
    :228b) 16710893s </pre >
</body ></html >
```

Each route corresponds to a CoAP URI. For example the URI `coap://[fdfd::`
`221:2eff:ff00:228b/128]/sensors/temperature` could be queried by a CoAP
client to determine the currently measured temperature.

Please note: The discovered devices are not necessarily thermostats registered to
the local communication gateway. The linked thermostats are therefore retrieved
from the smart heating server to avoid unnecessary querying of unrelated devices.

5.2.4.2 Executable Scripts

The `smart-heating-local` repository contains two python scripts in the repository
root: `thermostat_sync.py` and `server_sync.py`. Each script corresponds to one
of the two tasks described in Section 5.2.4. The first scripts is responsible for fetching
the temperature and other measurements from the thermostats to the local storage
and applying the latest heating schedule. The latter script sends the measurements
that where not yet delivered to the server and retrieves the most actual heating
schedule. Both scripts are independent from each other and use only the local
storage to share data as explained in Section 5.2.4.3 below.

Scheduling The periodic execution of the both scripts is scheduled by the *cron*
service which is provided by the deployed operating system *Raspbian* and also
by most Linux operating systems. The `crontab` program is used for defining the
schedule.

5.2.4.3 Local Storage

We combine SQLite and dbm for the local storage. *SQLite* is a lightweight relational
database engine implementing most of the SQL standard and is often used on
embedded systems. *dbm* is a family of simple database engines allowing to store
pairs of a unique key and a value. The two databases are used for different purposes.
The SQLite database persists historic measurement entries that should be held
at least until they are transferred to the server. On the other side there is the
dbm database holding configurations retrieved from the server. For example the

thermostats associated with a local communication gateway and the current heating schedule are stored using dbm as it has no relational schema and is therefore much simpler to use and maintain. Both databases are stored in files within the `data/` directory.

5.2.4.4 Application

All Python files containing application logic are located in the directory `smart_heating _local`. The following paragraphs list the files within this directory and describe their purpose.

config.py contains the `Config` class responsible for storing and retrieving the heating schedule as also the thermostat MAC addresses linked to this Raspberry Pi computer. The configuration is stored using the `shelve` Python package which itself builds on the npm database engine installed on the system.

server.py provides an interface to communicate with the smart heating server. It also determines the URI schema used by the server API.

server_models.py contains the models used by the server. The models correspond to the resources provided the server API and structure them into classes.

server_controller.py provides a layer of abstraction by packaging the most important functionality regarding server communication. It makes use of the files `server.py` and `server_models.py`. The provided methods include downloading the latest heating schedule and the thermostat configuration linked to this local communication gateway as well as storing the data in the local storage using the `Config` class.

local_models.py contains the models handling the temperature measurements and meta data entries which are stored in the local database.

thermostat_controller.py is responsible for querying the thermostats defined in the configuration and saving the data into the SQLite database. The controller also

contains the logic regarding CoAP requests and the resource schema used by the digital thermostats.

tests/ is the directory containing the tests that were created during implementation and which assisted the application development. The tests can be started using the command `nosetests -a '!local'` from the repository root. These tests are also periodically executed and logged by the Continuous Integration service Travis CI[12].

logging/ contains the central logging configuration used in the whole local project. Relevant information is logged to the file `logs/smart_heating.log`.

---

[12]`https://travis-ci.org/`

6 Mobile App

Describe your app in detail.

7 Evaluation

In this section we will evaluate our set design goals and discuss our implementation in detail. Further we will analyze our design decisions. Where applicable the functional and non-functional requirements will be validated.

7.1 Infrastructure

This section follows the same structure as used in the previous Chapter 5. The infrastructure consists of the server part and the local part. Both are evaluated in the following sections. In the end the deployed infrastructure is evaluated in a field test.

7.1.1 Server Infrastructure

7.1.1.1 Design Goals

Section 5.1.1 lists the design goals defined for this project part. Each design goal will be evaluated in a separate paragraph below.

Modularity and Extensibility   Django emphasizes reusability of components. This solid background allows us to clearly separate different concerns to achieve modularity. Further the distinct components simplify application extensions by easily adding new resources, representations, views or URLs.

Usability   In order for a developer to be able to familiarize himself with a new API it is important to provide clear documentation and a platform to use the API. The browsable API as described in Section 5.1.3 provides both. It allows a developer to interactively explore the resource structure by navigating via the

displayed URLs. The browsable API also provides easy interaction possibilities to create, read, update and delete resources. Further the API describes the supported access methods when requesting a particular resource. One of these HTTP methods named *OPTIONS* shows the accessible fields, their types and other useful meta information for interacting with the API.

Testability   The modular program structure and the chosen Django framework support the creation and automatized execution of software tests. These automatized tests are continuously executed on a hosted service and show that the tests run regularly and successfully.

7.1.1.2 Design Decisions

During the system implementation a few design decisions were chosen and will be evaluated in the following paragraphs.

Nested URL schema   The model hierarchy is represented using a nested URL schema. This schema suits the model structure in tree form and induces good readable URLs. Django is designed for flat URLs but is still flexible enough to support the design decision of implementing a nested URL schema for resources access.

Resource identifier as the first field   Within the resource representation the resource identifier is always the first field. This allows to easily recognize the identifier within resource representations and especially within nested resource representation.

Resource referencing   Resources are referenced by including their representation. This design decision improves the usability of the API, as related resources are shown in their full representation and also allows to reduce the number of required requests. In contrast collections are referenced via their URL. This is necessary to limit the representation size and avoid infinite recursions. For example in a parent-child relationship the child representation contains the representation of its parent. The parent representation itself contains the collection representation of its children. Therefore the collection representation cannot contain the full representations of its resources.

Identification of URL fields   Fields representing a URL are identified by the name `url` or the suffix `_url`. This design decision facilitates the recognition of hyperlinks which also allows an automatized navigation through the API.

7.1.2 Local Deployment

The following section will evaluate each of the design goals defined in Section 5.2.2. Please note that the reliability is evaluated separately in a field test described in Section 7.1.3.

The local infrastructure section is about the implementation part done on the residential communication gateway. Therefore the following paragraphs focus on the evaluation of software running on the Raspberry computer.

7.1.2.1 Performance

The system performance depends on two aspects. First, the software part. The programming language *Python* was chosen at it is suitable for developing on embedded hardware with possibly long-running input/output operations. Another important component is the system design of independent scripts triggered by a scheduler. Second, the embedded hardware part. The chosen hardware needs to be powerful enough to run the developed software implementation.

For evaluation of the ratio between required software performance and provided hardware performance the CPU load is monitored. We use the *sar* command from the *apt-get* package *sysstat* to analyze the average CPU utilization. Over a period of seven days of regular system runtime the average CPU load was 0.28 percent. This indicates that the hardware is powerful enough to run the designed software implementation.

7.1.2.2 Robustness

We defined robustness to be the ability of the system to behave reasonably in the presence of failures, especially connection problems. The local communication gateway is designed as two independent tasks. The first task is the retrieval and storage of the temperature data and the application of the defined schedule. The second task is the synchronization of the local data storage with the remote web server. This design splits the responsibility of the local communication system into

two simpler separated tasks with a clearly defined interface, the local data storage. In the presence of a failure only one of the two tasks is affected whereas the other task is still able fulfill its purpose.

For example in the event of a temporary internet connection disruption the system will keep operating the last downloaded heating schedule and store the temperature history. After the disruption the local data storage is synchronized and the system continues to operate on the most actual settings.

Another failure scenario would be the temporary disconnection of a wireless thermostat. In such a case the non-affected thermostats are still functional and the heating schedule of all thermostats is kept in sync with the web server. As soon as the system reestablishes the connection to the affected thermostat the system continues to work as planned. In the meantime neither the properly operating thermostats nor the synchronization of all thermostats' heating schedules is impaired.

7.1.2.3 Interoperability

We define interoperability as the system's ability to cooperate with other distributed systems. The whole infrastructure is designed to communicate according to recognized open standards and architectural styles such as *Constrained Application Protocol (CoAP)*, *Hypertext Transfer Protocol (HTTP)*, *Internet Protocol (IP)*, *JavaScript Object Notation (JSON)* and *Representational State Transfer (REST)*. This way the developed infrastructure is able to cooperate with other distributed system via the defined interfaces.

7.1.3 Field test

In order to check the reliability of the infrastructure in a real world scenario the system has been deployed in a residential home. Previous work in this area compared the defined room temperatures with the actual room temperatures to conclude on the proper work of a heating system [1]. Due to the summer season and the according temperatures this evaluation is not possible for this project. Instead we focus on the communication between the wireless thermostats, the local communication gateway and the remote server to evaluate the reliability. The main data sources for our analysis are the data storages on the local communication gateway, the database on the remote web server and the generated log files on both parts of the infrastructure. We present our analysis of two different time spans.

7.1.3.1 Long-term evaluation

In the following evaluations we analyzed the data from the time span of August 2015.

Local data storage analysis We start with the analysis of the local data storage. It is used to cache the temperature readings and other meta data read from the thermostats before it is send to the remote server. The local communication gateway queries the thermostats every 15 minutes, i.e. 96 times a day. The maximal number of temperature measurements and meta entries would therefore be 2976 for the whole month of August. The local data storage contains 2969 of these recorded temperature measurements in this date range, resulting in a coverage of 99.76 percent. The coverage of the meta data is even slightly higher. Please see Table 7.1 for the full coverage data.

Local log file analysis Each log entry has a severity level indicating the impact of a logged event on system stability and functionality. We evaluated these severity levels and their according log messages to analyze the system behavior as well as to identify potential problems.

There are almost a hundred thousand log entries which reveal a few interesting facts. The rate of successful CoAP requests varies highly depending on the queried resource. See the Table 7.2 for the full data. The requests to the operation mode and target temperature resources failed much more often than queries to other resources. Especially the PUT requests fail in more than 72 percent. Therefore the correct functioning could not be guaranteed in this time range. We suspect the low power devices to may be overwhelmed by too many closely following requests. An increase of time interval between the requests seemed to improved the request success rate and is evaluated in the next section.

Local Database entries	Maximal count	Actual count	Coverage
Temperature	2976	2969	99.76 %
Meta data	2976	2970	99.80 %

Table 7.1: Retrieved temperature measurements and meta entries in a real world deployment in August 2015.

CoAP request	Total	Success	Timeouts	Success rate
GET /debug/heartbeat	2975	2970	5	99.8 %
GET /sensors/temperature	2975	2969	6	99.8 %
GET /set/mode	2975	2970	5	99.8 %
GET /set/target	2975	1216	1758	40.9 %
PUT /set/mode	803	181	622	22.5 %
PUT /set/target	2698	741	1957	27.5 %

Table 7.2: Analyzed CoAP requests in August 2015.

Another interesting fact is the handling of temporary losses of internet connection. Due to a internet modem failure and the necessary replacement actions the residence local network was offline for about 48 hours. During this time the local gateway kept communicating with the wireless thermostats to operate the schedule and log the measured temperatures. After reestablishing the internet access all temperature measurements that have not yet been uploaded were pushed to the web server. It took about 56 seconds to upload the missing 192 measurements. During and after the whole external incident the system reacted and worked as planned.

Server analysis   The server logs all received HTTP requests into a log file. Each log entry contains the requested URL, the HTTP method and the HTTP response code. In the chosen time span there were 2969 successful POST requests each adding a single temperature measurement. This matches the number of temperature entries in the storage on the local communication gateway and in the server database and shows that all temperature measurements were successfully transmitted and persisted on the server.

7.1.3.2 Short-term evaluation

After the long-term field test a smaller test was set up to evaluate the effects of further improvements.

TODO: weitere evaluation

7.2 Mobile App

SAMUEL: Mobile App evaluation

8 Future Work

Anything that is interesting – but you did not have the time to pursue – fits into future work.

9 Conclusion

Give a summary on what you did and what the major results are.

Bibliography

[1] N. Eigenmann. Opportunistic sensing for domestic smart energy systems. Master thesis, ETH Zurich, 2012.

[2] F. Mattern, T. Staake, and M. Weiss. ICT for green: How computers can help us to conserve energy. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking (e-energy)*, pages 1–10. ACM, 2010.

[3] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014. `http://tools.ietf.org/html/rfc7252`.

A First Appendix

Use appendix for more anything that does not fit into the main document (e.g., implementation details).

Include links to all GitHub repos

Setup instructions for the local infrastructure

insert protocol