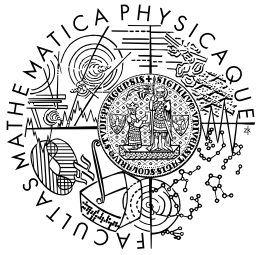


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Andrej Hraško

Use of legacy and wireless protocols in modern IoT

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Informatics
Study branch:	Software systems

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Use of legacy and wireless protocols in modern IoT

Author: Bc. Andrej Hraško

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: The main goals of this work is implementation of an extension into Inthouse system which will allow remote control of widespread Modbus-enabled devices and creation of an in-house-made simple controller controllable via Bluetooth using simplified JSON RPC protocol and addition of its control support into Inthouse. The next step is to summarize each implementation focusing on its technical complexity and a business potential. This work uncovers interesting details from the implementation and explain causes why such implementation was used in a commercial environment.

Keywords: Inthouse IoT InternetOfThings Modbus Bluetooth Cloud

I thank my thesis supervisor, Mr. doc. RNDr. Tomáš Bureš, Ph.D., for providing help with content of this work and for his friendly and optimistic attitude. Next, I thank Mr. Matej Snoha, my long-time colleague and friend, for all the success we have achieved in Inthouse. I thank Ms. Ing. Martina Mitrová for grammar advice and support. Last but not least, I'd like to thank all great people from Siemens for providing advice and feedback and, of course, I thank my dear parents for their neverending love and support.

Contents

1. Introduction

Currently, there is high demand for smart devices and ability to control appliances as easily as possible. For that reason, the concept of Internet of Things (IoT), has become widespread. IoT includes a very wide range of use cases and devices related to them, ranging from remote sensors to autonomous vehicles. All these things need to be connected to Internet to do their job. The main focus will be placed upon the part of the IoT world that is applied in building automation and industrial devices.

Looking back into history of computers, many communication protocols have been made. This work will consider a particular protocol which belongs amongst the oldest ones and is widely used until these days - Modbus protocol [Org12], as well as modern ones - JSON/HTTP, JSON RPC, while using transmission technologies such as serial line over twisted pair and serial line over Bluetooth, and communication techniques such as JSON communication with cloud or JSON communication through Bluetooth.

The main goals of this work is implementation of an extension into Inthouse [sro18c][sro15] system which will allow remote control of widespread Modbus-enabled devices and creation of an in-house-made simple controller controllable via Bluetooth using simplified JSON RPC protocol and addition of its control support into Inthouse. The next step is to summarize each implementation focusing on its technical complexity and a business potential.

Inthouse is a proprietary solution built for controlling smart devices and buildings providing user-friendly interface for end users and powerful configuration software for technicians and smart-solution integrators, such as manufacturers of smart boilers, air-conditioners and numerous other types of smart appliances. Inthouse is a system and brand owned and developed by a Czech company Inthouse Systems s.r.o., which is a small company belonging to two owners - one of which is the author of this thesis. Inthouse Systems s.r.o. is a contracted partner of Siemens Czech Republic and, within this partnership, Inthouse is primarily considered as a remote control system for programmable logic controllers (PLC) Siemens Climatix.

To ensure the source code privacy in accordance with its copyrights and to not let the know-how out, only necessary parts of the source code are to be published. Each part of the Inthouse solution which is considered as a part of this work is noticeably labeled. Parts of the solution which are not labeled may or may not be part of this work.

2. Background

At the beginning, it is useful to discuss the foundations that we were building the work on.

2.1 Inthouse

Inthouse is a proprietary solution currently owned by Inthouse Systems s.r.o.. The very beginning of the project dates back to 2013 when five students of the Charles University in Prague formed a team to fulfill their study obligations. There was an opportunity to develop a simple mobile application for controlling Siemens Climatix PLCs. The first published version of Inthouse App was developed under supervision of Daniel Toropila by team consisting of Matej Snoha, Andrej Hraško (author of this thesis), Jakub Kúdela, Petr Fejfar and Ondřej Staněk, who defended the exceptional result.

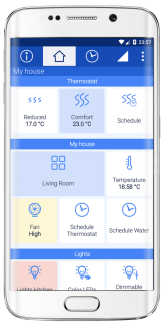
There was a close cooperation with Siemens Czech Republic since the beginning. After fulfillment of the school obligation, three of the members left and sold their business shares to the rest of the team - Matej Snoha and Andrej Hraško. With this small team remaining, Inthouse had been developed for next year and a half until the official contractual partnership between Inthouse Systems s.r.o. and Siemens Czech Republic has been settled. In fact, the development continues until present with the same team, which was only temporarily extended by part-time help of Michal Rosa.

The core of the business model of Inthouse is provision of lifetime licenses for each PLC. There have been hundreds of licenses provided up to this day. Siemens Czech Republic is an exclusive retailer of Inthouse licenses for Siemens's PLCs.

Inthouse licenses are provided to manufacturers of intelligent devices on B2B market and these devices are provided on B2C market to final customers. The most significant portion of B2B market is made of manufacturers of intelligent boilers, air-conditioning or other building technologies.

Inthouse solution consists of a few main parts which will be further discussed and elaborated on here. Inthouse App, Inthouse Studio and Inthouse Cloud are Java-based applications and each of them uses the same core of the system. This fact is very important since it ensures and enables fast development and always compatible cooperation while runtime. The approximate size of the solution currently amounts up to 100 thousands lines of code.

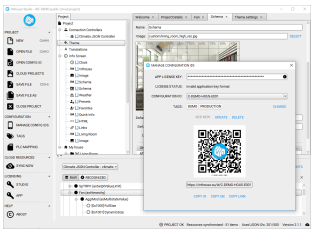
2.1.1 Inthouse App



Inthouse App is a mobile application part which is used by end customers for controlling their intelligent devices. Customers can control all kinds of data points inside of their PLCs using nice, simple and intuitive UI. In comparison to widely used SCADA systems, most of the user interface is automatically generated, therefore there is no problem with size of graphics or scaling when using different mobile devices. There is one universal Inthouse App available for download and all customizations are implemented by downloading the configuration into the App. Configuration can contain color scheme for the whole application, images, vector graphics, widgets and buttons and their customizations like order, name, limits, colors and many more UI elements. Configuration also specifies the way of communication between App and PLC for every UI element. For this reason App can communicate with more than one PLC at a time and every UI element can show data from different PLC. The App is translated to multiple languages.

In this work, we have added a support to Inthouse App for reading and writing simple and complex values from App to supported Modbus device trough Inthouse Cloud and Inthouse Cloud Access Point.

2.1.2 Inthouse Studio



Inthouse Studio is a desktop tool used for creating Inthouse App configurations in an intuitive way. It is intended for use by manufacturers of smart devices and integrators who need to make their devices remotely controllable. They can do this by using Inthouse Studio in the process where they select what data points from inside of PLC they want to control by the App, choose their color scheme and upload their graphics such as images, logos or icons and create the order in which all of the UI elements are displayed. At the end of the process of creation, the configuration is uploaded to Inthouse Cloud where it is stored for the Inthouse App users. Every printable text string in a Configuration can be translated to multiple languages by its creator. An example of translated item name is a button in Inthouse App, label of which may be the word "Temperature" in any of the translated languages, and whose icon and color depends on a real temperature value displayed in place which was chosen in the creation process.

In this work we have added a support to Inthouse Studio for creating UI elements which communicate with Modbus devices using Inthouse Cloud and Inthouse Cloud Access Point. Main change done by this work consists of the UI for creation of communication mapping used for communication between UI elements and their respective PLC's data points.

2.1.3 Inthouse Cloud

Inthouse Cloud is a distributed on-line back-end necessary for correct operation of many Inthouse services.

The most important service for this work is a snapshot of the values of data points of connected Modbus devices stored with all of their historical values. Inthouse Cloud Access Point connected to Modbus device scans this device and sends updated values of data points to Inthouse Cloud. Inthouse App can reach the snapshot of these values and/or be notified when they changed instantly without reaching Modbus device directly.

2.2 Inthouse Cloud Access Point

We brand and customize imported hardware, flash firmware, make a nice packaging and ship it to customers. We will not publish exact model name, however we point out some of its parameters:

- Powered by 580Mhz MIPS SoC
- 300Mbps 2,4GHz Wi-Fi (internal or external antenna available)
- 64MB RAM
- 32MB ROM
- 2x 100Mbps Ethernet ports
- 1x High-speed USB 2.0
- 5V Micro-USB power input
- Small, light, easy to use
- OpenWrt pre-installed
- internal I/O ports reserved for function extending
- prepared for custom branding

2.3 Modbus

Modbus is a communication protocol created in 1979 by Modicon (now Schneider Electric). Modbus standard is defined on layers 1,2 and 7 of the ISO/OSI model.

At the highest layer (7, application layer), Modbus defines PDU (Protocol Data Unit) which is a simple data structure consisting of Function part and Data part. This data structure is independent of the underlying communication layers.

In this work, we will focus on a serial binary version of Modbus protocol (there also exists a network version called Modbus TCP/IP and serial ASCII version of Modbus). The serial binary version is called Modbus RTU. Modbus RTU is defined on layer 2 (Data Link) of the ISO/OSI model and defines master-slave topology of devices. There can be only one master device and up to 245 slave devices on a single bus. Only a master device is eligible to send requests to slaves and slaves must respond. There is also timing defined on this layer. This means that data frame bits must travel at some chosen baud-rate and silent moment lengths between frames and individual bytes are strictly defined.

As a layer 1 of the ISO/OSI model, physical layer, Modbus allows usage of different physical interfaces such as RS232 or RS485. In our project, we use RS232 which is a 2-wire interface connected as bus with parallel topology.

2.3.1 Application layer

On application layer, Modbus defines type of messages and data structures and the way they can be used. Data structures are called *Modbus registers* and are categorized based on the following names and types:

- Discrete Input - one read-only bit
- Coil - one readable/writable bit
- Input Register - 16-bit read-only (word) register
- Holding Register - 16-bit readable/writable (word) register

Each register, which is published by some device via Modbus, has its own type and number (e.g. register H32 is the 32nd 16-bit register which can be overwritten). Requests that the master device can send are of two types: read value and write value requests. Write requests are allowed only for writable registers. A single request can represent only one type of the function (read or write) and one type of the register. It is not allowed to mix functions or register types on one request/response. One request/response can be applied to multiple registers in a range. For instance, this means that instead of reading only a single register at a time, one can read a range of registers of the same type in a one request/response.

2.3.2 Data Link layer

This layer defines data frames (see figure 2.1) and timing of the communication. Maximum defined size of a data frame (where the single request/response can travel in) is 256 bytes. This leaves us 252 bytes for the data, as you can see in the figure 2.1. Therefore, the standard defines that we can fit at most 2000 1-bit register values or 125 16-bit register values into a single response. Timing defined in Modbus standard requires that there can be a pause for maximum of 1.5 char during a single frame transmission and at least 3.5 chars pause in between two frames.

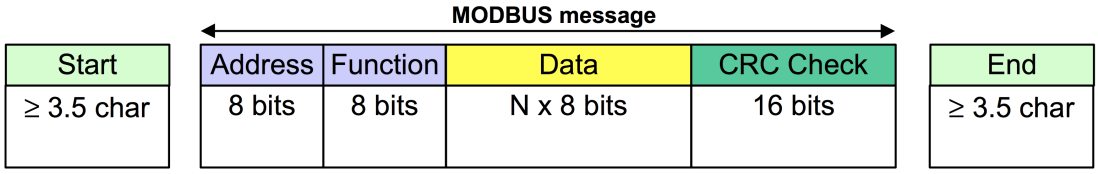


Figure 2.1: Modbus RTU frame scheme (source: [Org06])

2.4 Siemens Climatix

Siemens Climatix [Sie18] is a programmable logic controller developed and manufactured by Siemens. Multiple versions and multiple models of Climatix exist in practice. The features and qualities that differentiate them are the following:

- availability of Ethernet port
- availability of built-in display
- availability of proprietary extension port
- number and types of I/Os (digital, analog, relay)
- size of RAM and CPU speed

Every model of Climatix supports Modbus protocol over serial line (Modbus RTU, see 2.3). Climatix devices without an Ethernet port are much cheaper, yet unable to communicate via Internet easily. This brings us an opportunity to create a product which brings these controllers online and also opens us a door to the world of control of Modbus devices which are frequently used worldwide. Climatix devices which have an Ethernet port provide an interface that allows their control through HTTP REST and JSON protocol.

Climatix maps more complex data types to multiple Modbus registers. This means, for example, that boolean can be bound to a single 16-bit register (and have values 0 or 1) and 4-byte IEEE float can be bound to 32 1-bit registers (represented in IEEE binary standard). The most extreme option is that 40 chars long string can be mapped to 320 1-bit registers.

In past, Climatix devices were mainly used as controllers for production lines, however nowadays they tend to grow in the building automation market. That is the reason why the end-user simple GUI is highly required.

2.5 OpenWRT

OpenWRT [Pro18] is an open-source distribution of Linux operating system designed for embedded devices such as routers, NAS, etc. It provides complex build options, software packaging system with large number of packages, framework for compilation of own executables and many other features.

2.6 Facenika

Facenika [sro18a] is a small Prague-based company manufacturing design shelves and lights. Their shelves are made of special resin which is very resilient and can be manufactured in many color versions, patterns and shapes. To achieve better visual design and impression, Facenika uses built-in RGB light controllable by using four touch buttons or via Bluetooth. We have started cooperation in 2016 and since then we have implemented new Facenika controller firmware and added support for Facenika into Inthouse System.

3. State of Art

There are many solutions allowing the control of wide range of appliances. We focused on PLCs manufactured by Siemens and Facenika design shelves. So far there is no direct rival providing this particular usage.

Modbus protocol is a platform independent serial communication protocol, but in case we need a friendly user interface, the setup begins to be highly platform dependent. We will discuss this dependency later. Let's have a look at the other solutions which general purpose is the same - the control of the appliances.

We will describe these solutions only briefly because most of them we didn't try in real life and we didn't purchased any of them.

3.1 SmartHMI

This solution is one of the newest ones, providing similar functionality than Inthouse App and Inthouse Studio. Main difference is that SmartHMI doesn't provide native client, but everything is displayed on-line using HTML5 technology in WEB browser. This closes door for being think native client for this solution, however using this technology is easier to maintain and update.

Figure 3.1: SmartHMI solution illustration [Gmb18]

3.2 Blynk

Blynk provides a high quality user interface for control of small IoT devices. They also provide own paid cloud for a remote communication and libraries for communication with it. Blynk also provides highly customized paid forks of their application.

Figure 3.2: Blynk solution illustration [Inc18]

3.3 TapHome

TapHome is a Slovak-based company producing software and hardware for home automation. Their main market are intelligent family houses and flats. Whole TapHome solution is rather end-user focused. System is highly configurable but doesn't provide any complex tools for integrators and any design customization.

Figure 3.3: TapHome solution illustration [Tap18]

3.4 Fibaro

Fibaro is a manufacturer of their proprietary home automation solution. We took a lot of inspiration from this company trough the years of implementation of Inthouse. Their user interface doesn't provide any design customizations but it's very complex.

Figure 3.4: Fibaro solution illustration [Fib18]

3.5 Nest

Nest a smart thermostat from Google. It provides only one main feature - regulate a temperature in a room and it provides this feature very well with a great design and user interface.

Figure 3.5: Nest solution illustration [Goo18b]

4. Requirements and analysis

Licenses for usage of Inthouse App with Siemens controllers are resold by Siemens. End users of Inthouse App can never buy an Inthouse App stand-alone license for Siemens controllers without buying a product which contains any supported controller manufactured by Siemens. The complete chain of business with licenses for Siemens controllers looks as follows: Inthouse Systems → Siemens → integrator → distributor → end-user.

In case Inthouse is used with other than Siemens devices, there will also be a different way of distribution and revenue flow available.

Outside of business with Siemens, Inthouse cooperates with manufacturer of design shelves, Facenika. The Facenika shelves are made of special resin and have embedded illumination. This is controlled by Arduino motherboard running Inthouse firmware and mounted on custom-made extension board.

4.1 Stakeholders

There are several stakeholders considered who affects an Inthouse system development.

4.1.1 Inthouse Systems

Inthouse Systems s.r.o. is a small software company, which owns all rights for the Inthouse solution. Due to a small size of the company, selection of proper technologies is crucial. Every dead end or wrong decision in a development leads to a significant time and money loss.

Numerous individual clients bring their own requests. Implementation of these requests naturally increases the price of the solution. To retain competitiveness, many features are provided as a systematic solutions which unburden a need of individual approach need to be implemented. Therefore two main requirements for Inthouse given by Inthouse Systems can be outlined:

- fast and cheap development
- elimination of individual requests by providing systematic solutions

4.1.2 Facenika

Facenika is a small Prague-based brand of design shelves. Inthouse Systems has invested in Facenika in form of implementing firmware for Facenika's Arduino controller, implementing support for controlling Facenika from Inthouse App and developing Facenika's extension board. As a reward for the investment, revenue based on a number of sold items is expected. Since Facenika controller (parts of it) is the first controller ever made in-house by Inthouse Systems, these are the requirements for Inthouse defined by Facenika:

- easy hardware setup thanks to extension board
- convenient usability using 4 hardware buttons

- USB charging
- support for hardware wall-switch
- support for motion sensor
- firmware reusable for similar custom-made devices
- remote control via Bluetooth

4.1.3 Siemens

Siemens is a leading manufacturer of PLCs (Programmable Logic Controllers) and it primarily considers itself as a hardware manufacturer. Since software development often requires high team flexibility and also needs relatively high start-up investments, it's more efficient for Siemens to allow a smaller company do the less industrial software development job. The following are the main requirements for Inthouse requested by Siemens:

- increase in attractiveness of their hardware product (PLCs)
- generate a revenue by license resale
- minimize initial investments (as low as possible)
- software should be useful for end-users, for technicians and for integrators
- availability of systematic licensing model
- high stability and reachability of the system

The current contract with Siemens requires that all licenses for usage of Inthouse App with Siemens controllers must be resold by Siemens. This contract also doesn't differentiate used communication type or protocol.

4.1.4 Integrators

Integrators are manufacturers of devices which integrate Siemens controllers into their products. A significant part of integrators manufacture devices for building technologies market, e.g. heat pumps, boilers, air conditioning, ventilation. All of these integrators are independent companies possessing their own designs and business strategies. As a result, there are certain main requirements for Inthouse that the integrators desire:

- increase in attractiveness of their hardware product
- industry level of user interface
- definition of their own application structure
- definition of their own style, design and color scheme
- availability of stand-alone customized application
- short time to market
- possibility to service multiple devices at once

4.1.5 Distributors

Distributors handle distribution of the products manufactured by the integrators to final customers - end users. Since their main task is distribution and installation of the product directly in a location of a customer and subsequent provision of service for this product, these points summarize the most significant expectations of distributors towards Inthouse:

- ease of the first setup, as well as low time and financial cost of it
- possibility to service multiple devices at once
- easy license distribution system

4.1.6 End users

End users who control their devices using Inthouse App are the final part of the business chain. Users buy their product from their distributors, who also usually provide an initial setup. These are main requirements for Inthouse given by end users:

- ease of use
- easy setup of new mobile devices
- connection through cloud
- a beautiful production-grade user interface
- multilingual support

4.2 Use cases

Inthouse is a complex system because there are many use cases which it fulfills. Here we mention main of them.

4.2.1 Remote control

This use case is one of the most basic ones and it is why Inthouse project event started. Remote control of Climatix enabled-device by end users is one of the final products. Remote control should be intuitive, nice-looking and reliable.

4.2.2 Remote PLC diagnostics and service

Main stakeholder in this use-case is an integrator. Integrators often have hundreds or even thousands of devices sold and they sometimes require remote service access. They also ask for statistics of controller's internal values so they can predict/diagnose the device's malfunction.

4.2.3 Configuration creation

Inthouse App is an universal platform for all the integrators. Integrators have their own branding, design and have their own controller firmware. Each integrator receives Inthouse Studio tool where they can create their own customized Inthouse Configuration.

4.2.4 Remote App configuration updates

At the moment of change of integrator's branding or controller firmware, it's required to update the Configuration their customers have in their Inthouse App. Therefore this must be done remotely.

4.2.5 First setup help

As we already said, many integrators manufacture many devices per year and so they require helper tools for faster first setup, like batch push of setup parameters to controllers at the factory and at the installation point and also to all existing clients over cloud.

4.3 Software requirements

We have already analyzed requirements from the points of view of each stakeholder. Here we will analyze them from the aspect of their functionality.

4.3.1 Functional requirements

Inthouse is a complex system and there are many requirements on it. Here we list some of the most crucial ones:

- control of the Siemens Climatix devices over HTTP and Modbus
- control of the Siemens Climatix outside of its local network
- control of the Facenika devices over Bluetooth
- generate Modbus mapping for Siemens Climatix
- create and update Inthouse Configurations

4.3.2 Non-functional requirements

- MVC design pattern
- client-server structure
- high reachability of the server environment
- distributed server environment (cloud)
- scalable storage

- low memory and network expenses - compressed files and communication
- high stability and malfunction recovery ability
- malfunction notifications and reasons
- logging - traceability of the functionality

4.3.3 UI requirements

- easy navigation
- simple and intuitive interface
- always responsive
- consistent UI elements
- feedback mechanism
- purposeful layout

5. Overall architecture

In this chapter we will show the whole architecture from different points of view. In each section we will define what part of the overall work was created as part of this thesis.

5.1 System architecture

In this section we will discuss the main parts of the system from the user's point of view. These building blocks are put together to create a complex system.

The most visible part from the view of the end-user is the Inthouse App. Inthouse App may be considered as a thick client which can control end-user's devices also while being off-line (without Internet connection but with local network access). However, to fully achieve more complex functionality it still needs other parts of the whole system. In the figure 5.1 we can see a brief system scheme with all the main parts and the communication paths.

In this work we focused on implementing some of the new features and communication paths and we will describe them in detail in chapter 6. Main implemented parts in this work are support for Modbus-enabled devices in App, Studio and Cloud, firmware for Inthouse Cloud Access Point and firmware and support for remote control of the Facenika design shelfs.

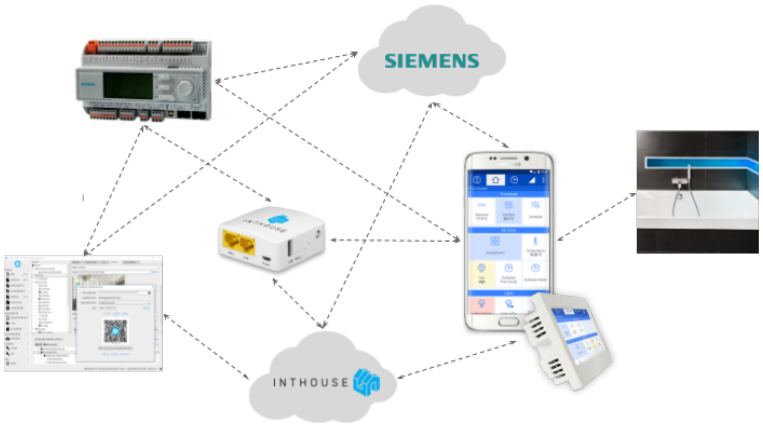


Figure 5.1: Overall system scheme

5.1.1 Inthouse Cloud Access Point

As a hardware for this device we used a carefully selected WiFi router from a chinese manufacturer (2.2) which offers nice and compact OEM design, and sufficient I/Os and power. This device's Linux support enables us to use the OpenWRT operating system. We modified OpenWRT's default build setup (6.1.1) to match all our needs and wrote an application for reading/writing Modbus data over RS232 serial cable connected over USB-to-RS232 adapter (see 6.1.3) and application for

Figure 5.2: Inthouse Cloud Access Point

uploading this data to Inthouse Cloud (see 6.1.4). Every Inthouse Cloud Access point has it's own so called RouterID in the format R-xxxx-xxxx-xxxx which is burnt into the OS image. Secure and stable connection to Inthouse Cloud is required and therefore is made by OpenVPN [Ope18] with a custom configuration (see 6.1.2). Inthouse Cloud Access Point also supports remote connection to it's USB port (used for remote administration) using USB-over-Ethernet forwarding technology. This piece of technology is nicely labeled with the Inthouse logo on the top.

Software part of this section is part of this work and will be discussed in depth in 6.1.

5.1.2 Inthouse App

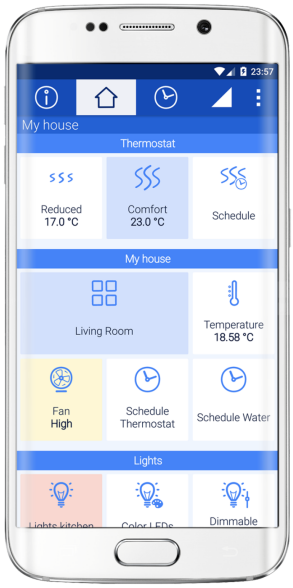


Figure 5.3: Inthouse App

Inthouse App (5.3) is a complex thick mobile client for control of PLCs. Reason for its thickness is the capability to control the PLC while being off-line (in time of Internet connection problems or when Internet is not available in general). Internet connection is needed only while setup and updates and for control over the Inthouse Cloud. In the setup sequence, user is prompted to enter a configuration ID which was provided by an integrator, then the App downloads the

configuration and necessary resources accordingly and uploads PLC's identification (usually MAC address) to the Inthouse Cloud to ensure authorization and licensing. When the setup is complete, system may stay without Internet connection (applies only when using PLCs with Ethernet port or Bluetooth). Control of Modbus-enabled devices is done always over Inthouse Cloud. Therefore, in this case, Inthouse Cloud Access Point and Inthouse App must be on-line to work properly. Inthouse App is currently an Android-only application which can be downloaded from Google Play Store [sro18b].

In this work we focused on implementation of a support for controlling:

- multiple controllers simultaneously
- Modbus-capable devices with push updates of values from the Inthouse Cloud
- Facenika devices

This led to making a massive changes in all parts of the Inthouse system and we will discuss them more in depth 6.3.

Each published Inthouse App binary is optimized and obfuscated using ProGuard tool [Gua18]. This tool provides sufficient know-how safety against decompilation attacks and also shrinks classes and parts of classes which are not used so the resulting binary is smaller.

5.1.3 Inthouse Cloud

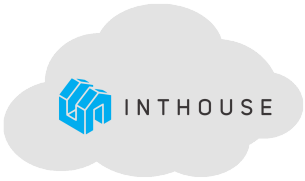


Figure 5.4: Inthouse Cloud

Inthouse Cloud is the connection hub and storage center for the whole system. As a storage platform we use distributed No-SQL database RethinkDB [Fou18]. Inthouse Cloud runs on multiple servers connected between each other using IPsec. In case of connection problems, writes are supported while more than a half of servers are reachable between each other and reads are supported when at least one server is alive (if the application specifically requests this). Inthouse components connecting to Inthouse Cloud choose the server randomly from the multiple DNS records, but the set of DNS records can be adjusted according to geographic location of the client. In time of communication between Inthouse App and PLC trough Inthouse Cloud (e.g. controlling Modbus-enabled device trough Inthouse Cloud Access Point), Inthouse App can be connected to a different Inthouse Cloud server then Inthouse Cloud Access Point is connected to and the communication works flawlessly.

Inthouse Cloud specific code is not maintained by the author and is not part of this work.

5.1.4 Inthouse Studio

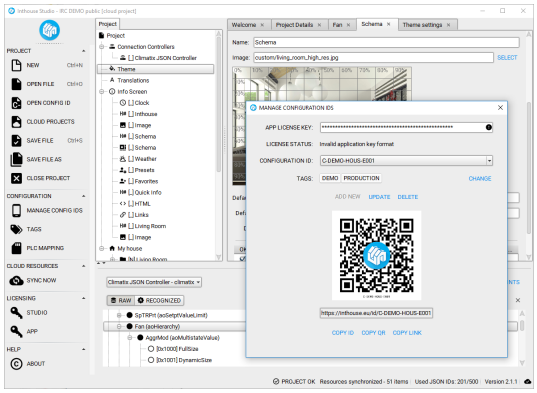


Figure 5.5: Inthouse Studio

Inthouse Studio is a complex tool for creating Inthouse App configurations, managing projects and Inthouse licenses and provides many logistical and other supporting services. It is delivered as Windows and OS X application with installer. Inthouse Studio is mainly used by integrators. Each integrator has their own license key for Studio usage. Projects and resources are synchronized between multiple instances trough Inthouse Cloud.

In this work we focused on:

- controlling multiple controllers feature
- implementing Modbus-enabled support, mainly Modbus mapping and a convenient way of creating it

Similarly as in Inthouse App, published Inthouse Studio binary is optimized and obfuscated using ProGuard tool.

5.1.5 Facenika

Figure 5.6: Facenika

Facenika is a small brand of design shelves and lights. Facenika decided to use Arduino as their control unit and RGB led strips as light sources. Arduino controller is connected to extension board and there is a possibility to extend this board with another extension board. In this setup integrator can easily add a support for PIR (Passive Infrared) sensor for a motion detection and external wall switch. Every Facenika shelf has built-in four touch buttons (infrared-based).

In this work we focused on implementing firmware for Facenika's Arduino controller which supports:

- IR-based touch buttons
- wall switch
- PIR sensor
- remote control over Bluetooth
- timers
- color effects and other bonus features

Arduino doesn't offer a lot computational resources, therefore firmware must be highly effective, single-threaded and nonblocking while running any actions.

5.2 Software architecture

The entire Inthouse system contains more than 100 thousand lines of code. In this section we will briefly discuss main parts and features of the system from the software point of view. This information is usually not important or hidden from the user's point of view. In each block we will describe which part of the code was the part of this work.

5.2.1 Software blocks

There are several blocks of software in the whole system. To achieve high compatibility and low expenses for updates, part of the source code is shared between multiple parts of the system.

Inthouse Studio and Inthouse App binaries which are used by customers and end-users are optimized and obfuscated by ProGuard. As a benefit of this, decompilation which may results in stealing of a source code and know-how of Inthouse is less likely to happen.

5.2.1.1 Shared

This block of code contains all the structures and algorithms which are platform independent, such as:

- data structures for all the data of individually created Inthouse Configurations

- the core for storing runtime contexts (runtime data objects and their states, widgets, translations, media resources, mappings)
- the core for updating context (pollers, communication structures and algorithms)
- connection administration algorithms
- algorithms for a work with different data types (Siemens Climatix special data types, Modbus data types)
- weather forecast framework
- etc.

Important part of the Shared block are serializable classes which are stored in Inthouse Cloud and transfered to and from the different parts of the system. We call these serializable parts Configurations (see 6.2.1). These Configurations serve to achieve a fully customized appearances for each setup and are created in Inthouse Studio by manufacturers of devices. These manufacturers requires fully customized user interface and fully customized communication communication setup. The data of each Configuration contains information about:

- color theme
- layout definition
  - structure of groups and contained items
  - types of items (widget, button)
  - specific settings for each item (icons, sizes, colors, limits, dimension types - units)
  - types of surrounding data types for each item (binary, string, enum, etc...)
- custom translations (translated item names, displayed values)
- communication parameters (general, item-specific)
- resources (icons, pictures)

When the Configuration is created, manufacturer publishes (exports) it to Inthouse Cloud. Every published Configuration has it's ID so called ConfigID in format C-xxxx-xxxx-xxxx Shared is written in pure Java 8 syntax and for HTTP connections uses the OkHttp library.

In this work we focused on:

- implementing support for Modbus mappings in Configurations
- support for controlling multiple controllers in one Configuration
- implementing algorithms for work with all of the Modbus data types

5.2.1.2 Cloud

It's written in JavaEE and sharing Shared software block with another Inthouse parts (see 6.2). Inthouse Cloud uses Jetty application server to run and RethinkDB as a database where it stores all the Configurations, projects, licenses, customer info and all other persistent data. RethinkDB is distributed No-SQL database which synchronizes all the data over the whole Cloud. To connect to the database the RethinkDB Java driver is used. As we mentioned before, Inthouse Cloud runs on multiple server instances. Every instance is connected with all the other instances using IPSec, therefore every server is reachable from any other server. For a storage over network between servers we use GlusterFS set up in highly redundant mode where we store necessary configuration files and backups of the database synchronized. Java application runtime instances don't communicate between each other directly. For any communication RethinkDB is used, where the communication is made in a way of adding data to the database on one server and triggering update notifications on the other servers. This work didn't focus on the implementation of Cloud part of the Inthouse solution.

5.2.1.3 App

Inthouse App is a part of the solution where the most of expenses were spent on. App is written in Java 8 syntax (with RetroLambda) - Android's native programming language, however, currently App's GUI is being rewritten to React Native while the core will stay in Java. In the iOS version we will use Multi-OS Engine to run Java core (Shared and App specific code).

In this work we focused on:

- implementation of controlling multiple controllers simultaneously
- support for Modbus-enabled controllers
- Facenika remote control support

5.2.1.4 Studio

It is written in full Java 8 using Swing and WebLaF UI libraries, sharing part of the core with other Inthouse parts (see 6.2). We licence thef WebLaF library for commercial use, since it provides very helpful features for creating nice Swing UI.

In this work we focused on:

- support for creating Configurations with multiple controller connections
- algorithms and UI for creating Modbus mappings

5.2.1.5 Inthouse Cloud Access Point firmware

Firmware for Inthouse Cloud Access Point is written in C++ and consists of two main binaries - first handles fetching data over Modbus protocol (Modbus client, see 6.1.3) and latter one handles upload of changed data to Inthouse Cloud (Cloud

client, see 6.1.4). Both applications run separately and transfer of the data from Modbus client to Cloud client is made by storing files in RAM using tmpfs file system. Data are stored in RAM until some level and then old data are being deleted. Currently, depending on number of changed values, in case of no Internet connection, we can store data for about 15 hours. We don't use persistent storage for storing changed values, because flash memory has very limited number of writes trough it's lifetime. When the connection becomes available In case Modbus connection is not available or USB-to-Serial adapter is not connected, application waits and checks for working Modbus connection.

Everything in this block is part of this work.

5.2.1.6 Facenika firmware

Firmware for Facenika Arduino board is written in C++ and supports all required features. Jump right to 6.4.2 to lean more.

5.2.2 Communication

There are several communication channels which should be discussed in this work. As you can see on figure 5.1 Inthouse system contains many connections to achieve requested complexity. Here we will describe the main ones and we will choose which are part of this work.

5.2.2.1 Inthouse Cloud ↔ Inthouse Cloud Access Point

Inthouse Cloud uses OpenVPN servers with self-signed certification authority. This authority generates (with revocation ability) new certificates (we use well-tried strong RSA certificates) for each device and therefore we have total control over connected Access Points. Every new successful connection calls a script which calls an API and stores a mapping of RouterID and corresponding OpenVPN IP address into database. We call this feature the Router DNS. Thanks to it we can route requests to Inthouse Cloud Access Points based on their RouterIDs.

Upon creating OpenVPN connection successfully, time of the Access Point synchronizes its time with multiple Inthouse Cloud servers over this connection using NTP. This is required to achieve precise timestamps of changed values uploaded to Cloud. Then bulks of unsent value changes are uploaded to Inthouse Cloud using REST API on Cloud For this we always use local OpenVPN IP addresses and we enforce transfer over OpenVPN network adapters so there are no data transfered different way then trough OpenVPN connection.

We also can connect to any port of any connected Inthouse Cloud Access Point (e.g. SSH, web administration) to be able to debug Access Point if necessary.

Everything related to Inthouse Cloud Access Point is part of this work and will be discussed in detail later.

5.2.2.2 Inthouse Cloud ↔ Inthouse App

For the connection between Inthouse App and Inthouse Cloud we use always HTTPS connection. Inthouse App selects random server from multiple DNS record. We use two types of the requests to REST API on Inthouse Cloud:

- stateless HTTP GET/PUT requests
- HTTP GET long-poll streaming requests (push updates)

Stateless requests are used for non-frequent requests (e.g. App/Configuration updates/download checks, license checks, resources fetch) or for poll request in case push updates are not available (e.g. HTTP requests forward to Siemens Climatix).

Long-poll streaming requests are used for pushing updates of frequently changed values from Cloud to Inthouse App. These values are fetched from Modbus-enabled device by Inthouse Cloud Access Point, sent to the Inthouse Cloud over a persistent connection and then streamed to mobile Apps. This is part of this work and will be discussed in detail later.

5.2.2.3 Inthouse Cloud Access Point ↔ Inthouse App

Direct connection from Inthouse App to Inthouse Cloud Access Point is made only in case of Ethernet forwarding to Siemens Climatix while being on LAN network. Since we can use Access Point as a Cloud gateway for Ethernet-enabled devices, we still need to be able to reach these devices from the LAN, since they are hidden behind Access Point. This connection is not encrypted due to PLC limitations and it's not part of this work.

5.2.2.4 Inthouse Cloud Access Point ↔ Siemens Climatix

Siemens Climatix supports multiple types of connection. Connection between Inthouse Cloud Access Point and Siemens Climatix can by made over Ethernet, USB or serial cable (Modbus).

For connection to the PLC we use Ethernet or Modbus. In this case, Ethernet is forwarded directly to Inthouse App when on one LAN network, or trough Inthouse Cloud for remote control. Modbus protocol is processed only inside of the Inthouse Cloud Access Point and for remote control we use our own set of technologies. Modbus protocol is an old protocol with strictly defined specification. We use Inthouse Cloud Access Point with USB-to-Serial adapter to control Siemens Climatix over Modbus.

Everything in this section in part of this work and will be discussed in detail later.

5.2.2.5 Inthouse App ↔ Siemens Climatix

We mentioned in the previous section that Siemens Climatix supports Ethernet connections.

Since Inthouse App is a mobile application, only reasonable communication way is Ethernet.

For a controlling over Ethernet we use JSON. Siemens Climatix supports only HTTP (insecure) protocol (with JSON not according to standards) requests for reading and writing values of data-points. We use this way to communicate with Climatix directly (IP port on Climatix must be reachable) or we tunnel these requests trough Cloud securely using Inthouse Cloud Access Point.

When controlling over Modbus protocol, we need to use Inthouse Cloud Access Point, as we mentioned in the previous section.

5.2.2.6 Inthouse Studio ↔ Siemens Climatix

The most advanced protocol which Siemens Climatix supports is Rainbow. It's a proprietary protocol implemented by Siemens's VB libraries. Therefore we can use in only on Windows platform and we use it for fetching controller's firmware application's data-point structure. Using Rainbow protocol we can communicate with Siemens Climatix over Ethernet or USB.

5.2.2.7 Inthouse Studio ↔ Inthouse Cloud

Inthouse Studio uses HTTPS REST requests to communicate with Inthouse Cloud. Currently we don't use stateful connections since we don't necessarily require immediate updates in Inthouse Studio.

5.2.2.8 Inthouse App ↔ Facenika

Currently only available way how to remotely control Facenika from Inthouse is trough secure serial Bluetooth connection. Inthouse checks values by short polling by sending commands (in JSON format) to Facenika. Facenika responds with a JSON structure containing all information about the current state. Everything connected to Facenika is part of this work and will be discussed later in a greater detail.

5.2.3 APIs

There are many APIs created and used by Inthouse system. Inthouse Cloud APIs are not part of this work but we will describe some of them here because we have referenced them in this work or because they are interesting.

5.2.3.1 Configurations API

This API serves for managing and downloading Inthouse Configurations and their tags. It's one of the most often used API in Inthouse. Each Configuration is bound with an owning Customer and so only this Customer can open and edit it in Inthouse Studio. This API always need a Configuration ID as one of the parameters and provides services like:

- put new Configuration to the Cloud (using Inthouse Studio)
- update existing Configuration (using Inthouse Studio)

- set Configuration Tags (using Inthouse Studio)
- list Configuration Tags (using Inthouse Studio)
- download Configuration
- check if Configuration update exists

5.2.3.2 Resources API

Resource API is used for management of Resource files. Each Resource is bound with an owning Customer and so only this Customer can open and edit it in Inthouse Studio and only Configurations loaded in Inthouse App which belongs to this Customer are eligible to request this Resource. Resources can be identified using their generated UIDs or their names (which much be unique for a single Customer). This API provides services like:

- put new Resource to the Cloud (using Inthouse Studio)
- download a Resource using it's UID or name
- list all Resources with or without data
- remove Resource from the Cloud (using Inthouse Studio)

5.2.3.3 Activations/Pairing API

This API serves for pairing an controller identification data with the particular Configuration and it's License. It's also used to check if the controller is licensed and paired. Result of a pairing (reactivating) is an activation token which is needed for Inthouse App to run.

5.2.3.4 PlcData API

This API is used for transferring Modbus data over Inthouse Cloud. It's used by Inthouse Cloud Access Point to push data to Inthouse Cloud and also by Inthouse App to download them. This API provides these services:

- put new data (snapshots or differential data) to the Cloud (used by Inthouse Cloud Access Point)
- send write request (used by Inthouse App) to Inthouse Cloud Access Point
- download a particular register values for a particular Modbus device connected to a particular Inthouse Cloud Access Point
- register for the push of new values for a particular registers on a particular Modbus device connected to a particular Inthouse Cloud Access Point

5.2.3.5 RouterDNS API

This API is not available from the outside and is used internally in the Inthouse Cloud to provide services like:

- pair an in-cloud IP address to the *RouterID*
- get the IP address using an provided *RouterID*
- list all *RouterID*s and their IPs (see 6.6.1 for more information)

5.2.4 Security and privacy

In Inthouse we don't store any personal data which can be connected to any sensitive data. Communication between all Inthouse components is secured on the transport layer. Only two people have console access to the Inthouse Cloud and are responsible for frequent security patches and maintaining minimal attack footprint. Each Inthouse Cloud Access Point has a unique strong root password which is stored outside of the Inthouse system. Each user uses their own ConfigID and/or RouterID and can change their access password. Communication with devices trough Ethernet (forwarded trough Inthouse Cloud Access Point) would need knowledge of the Climatix password, which can be changed by user and is not stored in the Inthouse Cloud.

6. Technical solution

In this chapter we will discuss each technical detail which was part of this work. To get an overall overview of Inthouse system and to examine description of all the individual parts (also those which were not parts of this work), return to the chapter 5

6.1 Inthouse Cloud Access Point

Most of the implementation and setup details of Inthouse Cloud Access Point are part of this work. For security reasons, not every security and/or implementation detail will be shared. Applications running on this small machine were written in bash script language or in C++ to achieve a high performance on a weak hardware.

6.1.1 OpenWRT custom build

Default OpenWRT build configuration was far away from expectations we required. It's possible to install many packages at the runtime, but in that case they consume much more space in the memory or they are precompiled without required features. Some of the main interesting features and modifications we changed in a build configuration are:

- enable `time, arpping` command in BusyBox
- remove commands like `wget, ntpd` from BusyBox and install them as regular packages
- install packages `libmodbus, libpthread, zlib, libstdcpp, openvpn, udev, top, uhttpd, socat, openssh-sftp-server, usb-modeswitch, wifitoggle, usbreset`
- install kernel modules `tun, leds-gpio, ledtrig-usbdev, lib-crc16`
- install drivers for various GSM USB sticks
- add `polarssl` and elliptic curves support into `openssl` package
- add `curl` package with `SSL` and `zlib` support

After we compile OpenWRT build with features and parameters we want, we must create a flashable image of the system. Also we need to make each image unique and already containing *RouterID* identification and unique certificates. We also must include our own binaries. For this task OpenWRT provides an image builder which is able to include any file or configuration into the resulting image. Everything included in the image will appear in a special read-only partition, which contains all the data compressed and without the possibility to change them, and therefore this file-system is very memory space-saving.

6.1.2 Connection to Inthouse Cloud

As mentioned in the previous chapter, OpenVPN is used to connect to Inthouse Cloud. Here we mention some OpenVPN configuration parameter which applies for our setup running on the clients - Inthouse Cloud Access Points. A few interesting OpenVPN configuration lines we use should be pointed out:

`proto 'udp'` - we have decided to use UDP as underlying tunneling protocol, since it is simpler, requires less resources of network components, and we can tunnel TCP over it flawlessly

`persist-tun` we use tun type of the OpenVPN network adapter. We use this option to force the existence of the adapter

`verify-x509-name "some field of certificate" name` with this option we can compare server's certificate meta-data. For example, we could only accept a certificate which contains a particular CN parameter.

`resolv-retry infinite` - default behavior of OpenVPN is to stop trying to resolve server's IP using its DNS name after a certain number of tries

`comp-lzo` - we use LZO compression to compress tunneled data. LZO is known for its low memory and CPU requirements and high speed of decompression

`auth-retry nointeract` - retry a failed authentication repeatedly without interactively asking for a credentials (password)

`tls-auth /path-to-the-file/filename.key 1` this key must be shared among all clients and a server. Using this key adds another layer of security. Advantage of this is that packets which are not signed up with this key can be discarded without much CPU power and therefore this feature is most useful against DOS attacks.

`tls-version-min 1.2` we enforce highest TLS version for control channel since we know that all our clients will support this

`tls-cipher HIDDEN-FOR-SECURITY-REASONS` we enforce particular control channel encryption cipher to maximize security. We know that all our clients will support the chosen cipher. Control channel is used to control data flow and to exchange the symmetric keys for the data encryption.

`auth SHA256` enforce using SHA256 algorithm to authenticate packets with HMAC [MK96] (default is SHA1)

`cipher HIDDEN-FOR-SECURITY-REASONS` enforces using the safest available algorithm as an encryption algorithm for a data packets stream (default is BlowFish)"

Server-side configuration is described in section 6.6.1. This configuration is a part of this work.

6.1.3 Modbus Client

Modbus Client is a standalone application written in C++ running as a part of Inthouse service in Inthouse Cloud Access Point. It serves for reading values over Modbus and dumping their updates. It uses `libmodbus` library [Rai] for parsing binary Modbus data. We use `libmodbus` library in Modbus RTU mode, since we are connected to Modbus device via serial port. To read more about the Modbus protocol and its specifications, return to 2.3. Since Modbus does not support any kind of notifications, we need to poll all the Modbus registers and track changes of their values.

At the beginning, application resets compatible USB adapters (sometimes necessary in case of stuck adapter drivers), sets up all necessary directories in RAM using OS-built-in `tmpfs` RAM file-system, where files will be stored, then initializes all necessary structures and libraries (`libmodbus`, `inotify`), hooks signal handler, initializes structure with Modbus registers snapshot and enters main reading cycle. There are some interesting facts we will talk about in this section.

6.1.3.1 Main cycle

After initialization, the application enters the main cycle. At the beginning of the cycle it checks if a USB-to-Serial adapter is connected. If it isn't, the application waits until it is. If the adapter later becomes disconnected, application returns to this point. After USB connection check, the application opens a serial connection - in Linux this means opening the device's file descriptor for reading/writing.

6.1.3.2 Writes

As the first thing after successful connection (or if already connected), the application processes pending write requests. Pending writes are stored in a dedicated folder by dedicated script (see 6.1.5). Writes are stored as files in JSON format containing set of `Register:Value` data. Filenames are timestamps, so the application knows how to order multiple writes. The application parses each file, executes writes using `libmodbus` library and removes corresponding files. At the end of the main cycle, the application waits for one second to reduce the load on both devices, but if some pending write arrives, application gets the notification from `inotify` handler (hooked on the folder dedicated for pending writes files) and starts processing write requests immediately.

6.1.3.3 Reads

When all pending write requests are processed, the application proceeds to read values of all Modbus registers in a defined range. We can request values for certain range of Modbus registers, yet as mentioned in 2.3, the response must fit into one Modbus RTU frame of maximum size 256B. Therefore we can request 125 of two-byte registers or 2000 of 1-bit registers in one request. Since we know that Siemens Climatix supports 2000 two-byte (1000 Holding and 1000 Input) registers, we need to split their reading into 16 (8 and 8) requests. The remaining 2000 of 1-bit registers (1000 Coils and 1000 Discrete Inputs) can be read in two requests, since mixing of register types in one request is not supported by the protocol.

Application caches all values from Climatix in its internal data structures and in every read cycle the fetched values are compared to the cached ones. If any change of value occurs in single read cycle, changed values are dumped to a file in JSON format which contains `Register:Value` entries of updated registers. This file also contains meta-data describing Modbus address of a slave device and file-descriptor address of USB-to-Serial adapter (in case of many) and is dumped to prepared folder in RAM.

At the first successful fetch of all register values and then every following hour, the application generates dump of all register values (also unchanged ones). This is to ensure data consistency in case of any problem at least at one-hour long granularity.

If there is less of free RAM space than 10MB, we start erasing old dump-files until we reach 10MB of free space. 10MB of free RAM space should be enough for all the necessary system functions. If application is unable to create 10MB of free RAM space by deleting all of the dump files (some other application is consuming RAM space), it waits until this amount of RAM space becomes available.

6.1.4 Cloud Client

Cloud Clients serve for uploading content of Modbus Client's dumped files onto Inthouse Cloud. It's a standalone application written in C++ running as a part of Inthouse service in Inthouse Cloud Access Point. It uses `libcurl` library to perform HTTPS requests to Inthouse Cloud and `gzip` library to compress sent data.

At the beginning, the application sets up all the necessary directories in RAM, hooks signal handler, hooks `inotify` watch handler on a folder where Modbus Client dumps its data and enters main cycle.

6.1.4.1 Main cycle

At the beginning of the cycle, the application uses `inotify` to get the number of new files which appeared in a target directory. If there are no new files, application sleeps and waits until Modbus Client creates some. When Modbus Clients dump a file to the destination folder, application is woken up by the system and proceeds to upload phase.

6.1.4.2 Uploads

In case we have thousands of Inthouse Cloud Access Points connected to Inthouse Cloud, the traffic load may be high. We try to solve this problem by compressing the whole upload batch payload at once, since we know that compressing bigger chunks is more effective because of algorithm's dictionary overhead. We created algorithm which reads files one by one and merges them into one big streamed array of JSON objects. This stream of uncompressed text data flows right into `gzip` where it is compressed on-the-fly. `gzip` produces new stream of compressed data which are fed into `libcurl` and streamed directly onto Inthouse Cloud. Since we use streams from the beginning to the end of this chain, the memory overhead for this task is very small, containing only internal stream buffers and some other small memory data structures used for compression and for communication.

We defined limits for one HTTPS upload as: maximum 100 files and maximum of about 1MB of compressed payload because they are reasonably big, do not overload any part of the system and data should pass most of networks without any problems. `libcurl` is forced to user OpenVPN's network adapter to ensure that no data will ever flow outside of Inthouse Cloud private network system. `/data` API is used for uploads from Inthouse Cloud Access Points. To ensure higher security, this API only allows requests from OpenVPN's network IP range.

6.1.5 Receiving write requests

We have configured `uhttpd` service in Inthouse Cloud Access Point and added new instance on a new port. A small script is called when the request to this port comes. It takes its input and outputs it to the file to the dedicated directory where Modbus Client waits for it. Thanks to firewall, this port is only available trough OpenVPN, therefore we do not need to strictly check the content of the data which arrives to this port, since only Inthouse Cloud has access to this port. To learn more about the incoming data, please return to 6.1.3.2.

6.1.6 USB forwarding

Controllers which support Modbus protocol and do not contain Ethernet port are only configurable over USB. That is why integrator often needs to connect the controller over USB to be able to flash mapping into controller. For this purpose we use VirtualHere application running in Inthouse Cloud Access Point which provides USB forwarding over network. We own a legal license for usage of VirtualHere in our product. We forward this USB communication through Inthouse Cloud right into integrator's PC. This feature is not implemented on PC side yet (no GUI).

The only downside of this feature is that USB standard requires low latency and jitter of the communication and therefore this feature may not work properly when one of the sides has inferior Internet connection.

6.1.7 Cross-compilation

As mentioned in 2.2, our device is equipped with MIPS-based CPU, therefore we need to compile our binaries for this architecture. To compile any binary which runs natively on the CPU we need a standard library. For Linux-based systems this library is called `libc`. `libc` library has several implementations. Currently OpenWRT operates using `musl libc` library, therefore a `musl`-compatible compiler is needed for making any executable binary. In the OpenWRT custom build configuration we can ask to generate a so called *toolchain*. It is a set of tools containing everything such as `musl`-compatible compiler and linker needed for compiling own native applications for corresponding OpenWRT. We compile our source codes with a `g++` compiler. Since we programmed in C++ language, we set `g++` to compile against `gnu++11`, which means we can use all the features of C++ 11 language. You can find all the build scripts and sources attached in *Attachment A*.

6.2 Shared

As we mentioned in the previous chapter, Shared is a part of the code which is shared across multiple parts of the system. We can consider this part the core of the system containing all the necessary algorithms and data types which can be platform independent. We spent a lot of energy on putting as much code into the Shared part as possible to make the platform-dependent parts lightweight.

One of the most crucial parts of the Inthouse are its Configurations.

6.2.1 Configurations

Configurations are data structures containing all the necessary data needed for creating a fully customized runtime appearances of the Inthouse App. To learn more general information about Configurations, please return to 5.2.1.1

Originally, Inthouse only supported one type of controller (Siemens Climatix over JSON) and only a single controller controllable within a single Configuration. Part of this work was adding support for new types of controllers and adding support for controlling multiple controllers of possible different types. Since adding new features intervened with an original format of Configurations, we were forced to introduce Configuration format versioning (for more information read 6.2.2.1).

For technical reasons we decided not to convert Configuration format versions in Inthouse Cloud (when adding new features), but to ensure backward compatibility in each client, such as Inthouse App and Inthouse Studio. Forward compatibility is not handled at all because everyone who has the ability to download a new incompatible version of a Configuration has also an ability to download a new compatible version of a client (Inthouse App or Inthouse Studio).

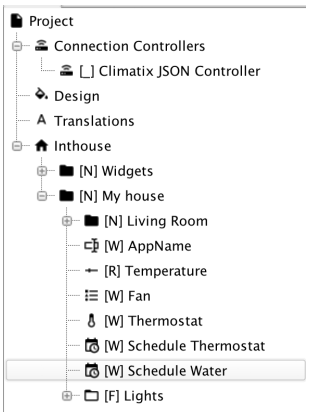


Figure 6.1: Configuration structure loaded in Inthouse Studio

6.2.2 Serialization and deserialization

Configurations are dumped in JSON format using `Gson` Java library.

Since objects inside of Configurations may have generic class types, we would not be able to deserialize proper type of a class when losing the class type information while serialization. The reason of this loss is that `Gson` does not natively store any class type information. To gain this feature we register custom `TypeAdapterFactory` which annotate every generic class with its specific class name. During deserialization, `Gson` uses this factory the opposite way to distinguish the proper exact type of a deserialized class.

6.2.2.1 Upgrading across Configuration versions

With the changes to Configuration format structure we have made in this work we needed to introduce versioning of Configuration formats. Every Configuration contains its format version. If the client (Inthouse App or Inthouse Studio) supports higher Configuration format version than the processed Configuration's format version, the conversion algorithm takes place and Configuration is upgraded to the client's version. A part of the conversion algorithm works on partially deserialized Configuration, `JSONObject` class type. JSONObject object may be a type of a set of `fieldNameString:JSONObject` tuples, or a primitive type like number, boolean, null, string, or an array of `JSONObject`s. From this object we can inflate deserialized classes using the data in a way that we pair class' fields with `JSONObject`s with corresponding `fieldNameString`. Editing partially deserialized classes is very easy and free-formed. You can find the upgrade algorithm attached in the *Attachment A*.

6.2.2.2 Multiple controller support

As a part of this work, we have implemented support for controlling multiple controllers at once in a single Configuration. To achieve this we had to make vast changes in the whole project, since almost whole software counted only with one controller. We created a separate group for controllers where Inthouse Studio's user define connection parameters for each controller he wants to control. Then the proper controller (from the created ones) needs to be selected for every item which serves for controlling.

When loading an exported Configuration into Inthouse App for the first time, it needs to be initialized. Initialization algorithm checks the availability of connection to each contained controller and also checks availability of each data-point (represented by control items) in every controller.

If connection to the controller is not available or the data-point inside of a controller is unavailable, control items which wouldn't work won't be displayed. This means that there may exist one equal Configuration for many different controller configurations and Inthouse App will always display only available items.

After using Inthouse App with a controller for the first time, identification (e.g. MAC address) of the controller is stored on Inthouse Cloud and the License of the Configuration is bound to this identification. This identification is later used to enforce connecting to the same controller even if connection parameters change.

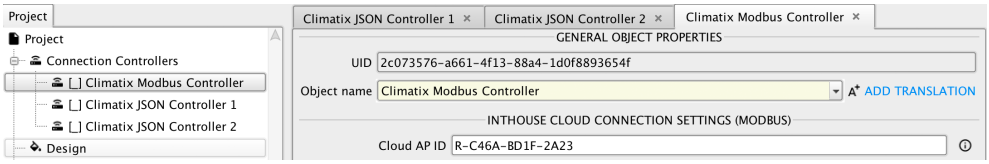


Figure 6.2: Multiple controllers definition in Inthouse Studio

6.2.2.3 Modbus mapping

Since Siemens Climatix only supports limited number of Modbus registers, mapping which publishes certain data-points and binds them to certain Modbus registers needs to be generated. As mentioned in 2.4, Siemens Climatix can publish any type of a data-point to any type of a Modbus register. As a part of this work we created a Modbus mapping generator which generates a Climatix-compatible mapping. The generator recreates this mapping on every data-point change in Inthouse Studio and this algorithm is a part of this work.

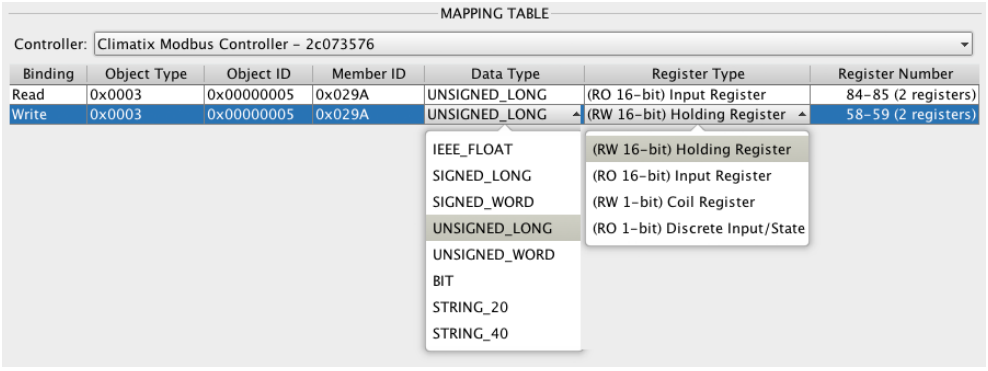


Figure 6.3: Modbus mapping table for Analog Object in Inthouse Studio

6.2.3 Climatix's Modbus data types

We mentioned in a previous section that Climatix provides more complex data types mapped to simple Modbus data types. As a part of this work we have created a set of methods for converting data read over Modbus to their representing values.

We created `ModbusRegister` class which represents one Modbus register (1b or 16b). Then we created `ClimatixRegister` class which represents complex data type mapped on regular Modbus registers. `ClimatixRegisters` are detecting changes on `ModbusRegisters` and process the resulting values in case of change of any underlying `ModbusRegister`. We put the data from `ModbusRegisters` into Java's `ByteBuffer` to fix their endianness and then we convert them to the target data type. These `ClimatixRegisters` are attached to the very core of the application (so called `Devices`) and notify the core with the value updates.

6.2.4 Controllers

In Inthouse code we call Controllers set of the classes which are responsible for communication with a physical controllers through Ethernet, Bluetooth, directly or via the Inthouse or Siemens Cloud. The main particular responsibilities are:

- network handling (addresses, ports, timeouts, encryption)
- connection handling (switching between direct and cloud connection)
- query handling (calling API with correct parameters)
- security handling (rejecting communication with foreign/unpaired/unknown controllers)
- request priority handling (write queries have higher priority than read queries)
- request queuing (queue requests, discard duplicate requests, make batch queries of possible)
- response low level parsing (parse responses to pair them witch requests)
- response delivery (deliver response to object who made the request)

Since for the HTTPS communication we use platform independent `OkHttp` library, we can put most of the Controllers code into the Shared part.

6.2.4.1 Types of Controllers

There is a separate implementation for every type of a physical controller. For example, with Siemens Climatix we can communicate via Ethernet or via Inthouse Cloud in two different API formats. Each of this formats needs its own implementation of the Controller. Facenika also needs its own Controller implementation, but Facenika's Controller is part of Inthouse App specific code, because implementation of code which uses any platform-dependent Bluetooth API is not possible.

6.2.4.2 FifoPriorityBlockingQueue

We created this data structure because we needed features of blocking queue, which automatically reorders its elements due to their priorities. This reordering must happen atomically. That means that after putting a new element into the queue, reordering must happen before it is possible to pop elements out.

Reason why we need this structure is the following. We have available only one concurrent connection to the controller, and we have two types of requests (at this moment) - read and write requests. Since write requests usually have higher priority than read request (because they usually come from user's interaction), we want to process them first. There are also some read requests that we want to have processed first (for example when it is important to process a read request right after a write request).

You can find the source code attached in the *Attachment A*

6.2.4.3 Poll and push communication type

As we already mentioned, Siemens Climatix does not support any open protocol which supports listening for updates. Therefore, in case of connecting to Siemens Climatix directly we need to poll it's API periodically. Controller itself is not responsible for polling values (the Poller which is implemented in Inthouse App takes this role).

Modbus-based Controller works slightly different though. Since Modbus register values are stored in database in Inthouse Cloud and Inthouse Cloud is fed new values by Inthouse Cloud Access Point, we can push updated values to any connected Inthouse App which registers for getting value updates using PlcData API (see 5.2.3.4). These push updates are send over open HTTP GET connection with very long time-out. This type of the communication is called long-polling, but in our case we leave the connection open until any timeout or error occurs.

6.2.5 Devices

Devices represent the very core of the application. To be less confusing for the user, in the graphical user interface we call them *Objects* (see fig. 6.4). Every device is an entity which stores the state of a corresponding data point (or set of data points) and handles all the state changes. There have been some minor changes done to this part of the software for purpose of this work, but we will not discuss them since the changes are not interesting and/or scattered.

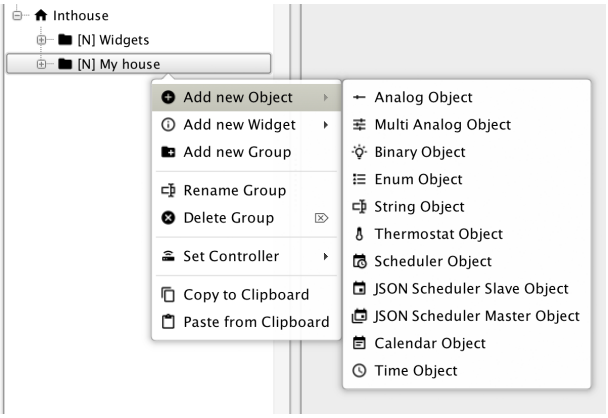


Figure 6.4: Devices in Inthouse Studio

6.2.6 Resources framework

To create a comprehensive picture of the software, we will describe one of the big parts of it here. Resources framework is not important for this work but the implementation is ours and it takes a significant part of the software. We consider any type of a file as a resource. In Inthouse we use this framework to distribute vector icons, raster images and whole Inthouse projects. We have implemented `ResourceCache` whose purpose is to synchronize resources with Inthouse Cloud. This synchronization supports downloads and uploads from/to Inthouse Cloud, efficient updates of individual or subsets of resources, efficient comparison of resources over network and creation of differential lists.

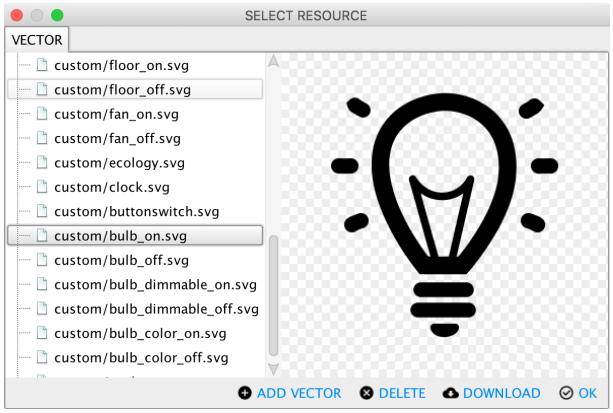


Figure 6.5: Resource chooser in Inthouse Studio

Thanks to Resources framework we can easily transfer media and synchronize projects between all company employees. Every operation can be performed on demand, asynchronously and the data are stored into persistent memory.

Figure 6.6: Status bar in Inthouse Studio

6.3 Inthouse App

As we have already mentioned, there have been many changes done to bring the features. Despite the fact that most of the crucial changes end in Devices part (in Shared part) of the application (see 6.2), some platform-dependent code has been added also to the Inthouse App part. This means that we try to hide the controller-specific details and connection-specific details and provide a graphical unified user interface.

6.3.1 Licensing

For every controller which is used with Inthouse App there must be a license. To check whether controller is licensed, the controller identification must be fetched and confirmed on Inthouse Cloud. We call this process *Controller Pairing*.

6.3.1.1 Pairing

Pairing Configuration with corresponding controllers is one of the most crucial security features of the system. Pairing needs to be done to enforce that communication will only be made with the controller that the license is purchased for. As well as that, it is possible that connection credentials are identical for two different controllers (e.g. unchanged default settings) and we need to distinguish them and communicate only with the right one.

6.3.1.2 License checks

Due to security reasons, Inthouse Cloud never sends any controller identification over any API. The only way to check if some controller is licensed for usage with Inthouse is to ask Inthouse Cloud via Activations/Pairing API (see 5.2.3.3) whether the fetched controller's identification is entitled for this usage. For this reason, at the first connection of Inthouse App instance, Inthouse App needs to fetch the identification data from the controller and confirm them on Inthouse Cloud through the pairing API.

6.3.1.3 Pairing multiple controllers

There is a logical problem when it comes to pairing multiple controllers in a single Configuration. As we already mentioned, every controller needs its own purchased license. Since the license numbers are transparent for Inthouse App, Inthouse App does not know which controller is paired with which license (there are multiple licenses used for a single Configuration, all of them entered in Inthouse Studio while exporting the Configuration). For this reason, Inthouse App needs to collect all the identification data from every controller and then confirm all of them together on Inthouse Cloud. Set of these identification data may be freely ordered, since we cannot enforce pairing of particular license with a particular controller due to the license number transparency.

6.3.1.4 Controller authorization

The information whether some controller is successfully licensed and paired is persistently stored in the application data. This information contains necessary identification data of the controller. This provides the ability to continue using Inthouse App without connection to Inthouse Cloud. At the beginning of every communication, and also repeatedly during it, we fetch these identification data and compare it to the stored one. In case of any identification conflict, the application stops communication and continues to fetch only identification data until the conflict disappears. This conflict results in displaying all values and controls as unavailable.

6.3.2 User interface

User interface of main screen has only changed in minor details. As we have already mentioned, we want to unburden users from technical details and provide a unified user control interface for every type of controller and connection type.

6.3.2.1 Launching App

If the App contains recent initialized Configuration it loads up directly to the main screen where the configuration is displayed. Otherwise the application stays on the launcher screen. If there was any Configuration downloaded in past, the list of these Configurations shows up. If Inthouse App does not contain any Configurations at all, user is asked to enter his Configuration ID by using keyboard or by scanning a QR code.

6.3.2.2 Connection details

However, the initialization process needs to bring some technical details to the user's sight. For example, a user must select the type of connection (from the list of the allowed ones) he wants to use. Integrator can pre-fill some fields (e.g. IP address and port, Inthouse Cloud Access Point ID, credentials) to simplify the initialization process for the user.

6.3.2.3 Multiple controllers

Connecting to multiple controllers needs a graphical user interface which provides ease of the initialization process. We created a set of screens containing fields for entering connection parameters, hints and status information. Then we created an algorithm which iterates over the set of all controllers and tries to connect to them (see section Pairing in 6.3.1). User is eligible to modify or fill connection parameters or to skip initialization of some controller. This algorithm also manages which screen should be shown to ask user to complete missing parameters.

This have been done by creating a `ScreensActivity` framework.

6.3.2.4 ScreensActivity

We have implemented our own framework for showing content to the user. This content is provided to framework in form of so called *Screens*. Every screen contains a simple definition of layout. This screen can be shown or hidden and the instance of the screen is notified about these actions. The framework also supports a stack of these screens, which provides ability to go to previous screen on back-press or to remove some screen from the stack. This framework can later be modified easily to support different platforms like ReactNative or Swing. Source codes are attached in *Attachment A*.

6.4 Facenika

After creating the first version of Facenika firmware and adding the support also into Inthouse, Facenika owner asked us to create and extension board to the current board and add support for PIR sensor and external wall switch, as well as to integrate 220 V relay and a simple USB charger (so that users can charge their phone on the Facenika shelf).

Facenika uses Arduino Nano as a main controlling unit.

6.4.1 Hardware

We used Eagle PCB software from Autodesk [Aut18] to create a fully working extension PCB. We have created two models - with and without USB charging capability. In figures 6.7 and 6.8 you can see rasterized visualization of both models.

On the input we used a capacitor 1000 µF/16 V to stabilize any possible current fluctuations when 220 V relay changes its state. As you can see in figure 6.7 and in *Attachment A*, for USB charging output we have used two 7805 linear

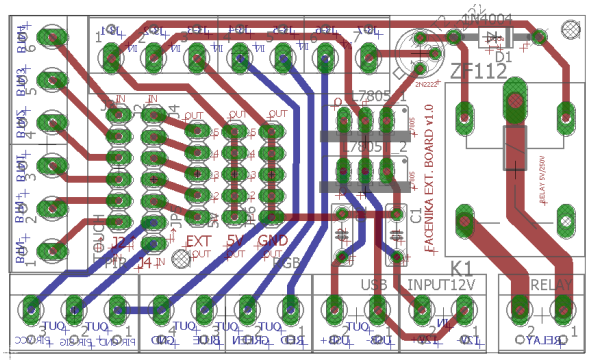


Figure 6.7: Facenika extension board with USB charging

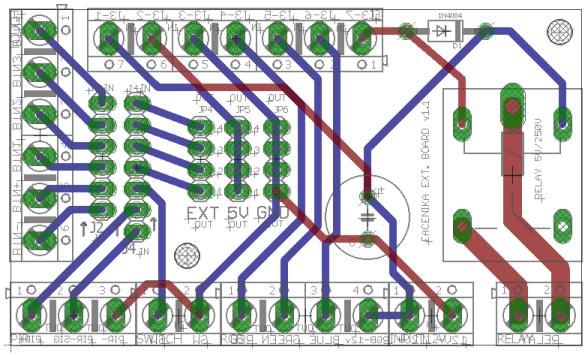


Figure 6.8: Facenika extension board without USB charging

voltage regulators to convert voltage of 12 V to 5 V. Since linear regulators convert excessive energy to heat, our regulators began warming up after connecting a USB device. In this setup, the heat was so high that we decided to also make a model without integrated USB charging capability, because the necessary heat sink would need to be very large and therefore expensive and space-consuming. We have come up with using a switching step-down voltage regulator, which usually already contains USB port and is much more effective, so the excessive heat is very low. In figure 6.9 we can see an example of used regulator.

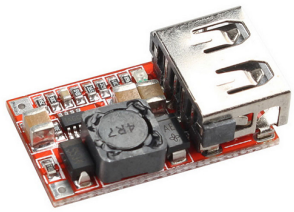


Figure 6.9: Switching step-down voltage regulator

As a PIR sensor, we decided to use a very small and cheap (but high-quality) sensor used in robotics which matched our needs perfectly (see fig. 6.10). This sensor has very low power consumption (it can be powered by Arduino controller) and has adjustable sensitivity up to 7 meters. When a sensor captures some motion, it puts signal to one of the I/O pins. This signal has adjustable length

and we set it to the shortest length possible. Since we can process this signal information in Arduino's firmware easily, we do not need the sensor to hold this information longer than necessary. We have connected its OUT pin with one of the Arduino's universal I/O pins.

Figure 6.10: PIR - used passive infrared sensor

For output for a wall switch we simply took one of the universal pins and the ground contact. We can easily monitor switch change by reading value of this pin.

6.4.2 Facenika firmware

We have implemented the whole Facenika's firmware which runs on Arduino board. It is written in C language and compiled with a compiler provided by Arduino.

The code contains two top-level parts - an initialization and a loop. In the initialization part we initialize all the defaults, data structures and set up all the I/O pins. We also load persistently stored data and initialize a Bluetooth serial framework. After the initialization method finished, the loop method is called repeatedly while the device is powered on. Apart from the communication part, our firmware supports several interesting features we will describe here further on.

6.4.2.1 Buttons handling

Facenika uses light-sensor based buttons with built-in infrared diodes. That means that the closer the finger is, the more light it reflects and the more light comes back to the sensor. We receive an analog value from each sensor as a 1-byte value (range of 0-255). We defined thresholds for registering a touch and unregistering a touch as two different values, to remove flickering of the press state while near threshold values. Then in each cycle we read these values and set the booleans describing press state for each button.

We support multi-press - multiple pressed buttons at once. In each cycle we decide which one of the combinations is pressed at the moment and also for how long. We call these combinations a *keystrokes*.

6.4.2.2 Commands

Commands are used to change some state of the device (turn on/off, change color, etc...). Commands can be generated out of keystrokes or can be received via

Bluetooth. The method for resolving commands out of the keystrokes generates commands depending on a particular key-press combination and its length. It also filters out invalid combinations and unwanted strokes if not all of buttons of a keystroke are pressed at the exact same time.

6.4.2.3 Features

We have implemented all the following features into the firmware:

- RGB light and relay on/off switching (possible separately)
- RGB linear and step dimming and linear and step hue change
- RGB wave/rainbow effect (continuous hue change) with speed setup
- timers for the wave effect, RGB switch, relay switch
- PIR sensor with adjustable timer, affecting RGB and/or relay (selectable)
- hardware buttons lock and enabling/disabling buttons completely
- wall switch with selectable effect (RGB and/or relay)
- hardware buttons calibration mode (some sensors may have slightly different sensitivity)
- showroom mode (resets most of the settings every few minutes)
- wet buttons mode (lower sensitivity and strong flickering filter applied)
- communication over Bluetooth (state sending, commands and keystrokes receiving)
- energy saving by sleeping between cycles
- delayed (for longer memory lifespan) dumping of the state to the persistent memory in case of power loss

Subset of these features can be used by using hardware buttons and all of them can be used using Inthouse App over Bluetooth.

6.4.2.4 Communication

Arduino is connected to a simple Bluetooth chip using two of its I/O pins. This connection is serial-based and Arduino's Bluetooth library does not provide the information whether Bluetooth connection is connected. Therefore we have decided to only implement a simple polling-based communication support, since we cannot elegantly register and unregister push listeners over Bluetooth.

Communication is done using a simple JSON RPC protocol (without notifications implemented). We use a simple JSON library, `ArduinoJson`, for generating JSON messages. Every request has its ID and a corresponding response is always returned. Facenika only receives requests and does not generate any of them. Every type of request has its own number (we do not use strings to reduce communication load) and the proper action is executed on Facenika's side

according to this type. List of these commands can be found in source code of Facenika's firmware source code (attached in *Attachment A*) in section *Bluetooth commands*. To achieve great simplicity, the state of the Facenika device is sent back as a response to the every request.

6.4.2.5 Inthouse App extension

We have created a fixed Configuration for use with Facenika devices. Currently, We support controlling only one device at a time. To control a different Facenika device, the user needs to select it from the menu. As you can see in figure 6.11, user can not only control Facenika's features but also emulate pressing of its hardware buttons (including long-press and very-long-press).

Inside, the Facenika Configuration contains one Facenika Device which contains small regular Devices (binary, analog, enum, see 6.2.5 for more information). This Facenika Device handles all responses and delivers sub-responses to the particular sub-Devices.

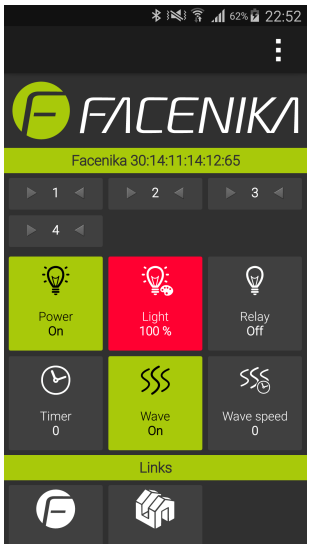


Figure 6.11: Inthouse App with Facenika Configuration

6.5 Inthouse Studio

We have already described most of changes in the Shared section (6.2). Implementation in pure Inthouse Studio code was mainly about adapting our GUI to these changes, for example GUI for the table for Climatix Modbus mapping, controller selection in each Device settings, modification of a project tree visualization to support adding multiple controllers, multiple license number fields in the dialog for export of the Configuration.

6.6 Inthouse Cloud

The only part of this work that has something to do with Inthouse Cloud is OpenVPN server configuration.

6.6.1 OpenVPN server

Here we point out some interesting options from the server-side configuration of OpenVPN. We leave out things we talked about in client-side section (see 6.1.2) and which are equal/symmetric to the server side. This configuration is a part of this work.

`ifconfig-pool-persist /path-to-some-folder/file.txt` we persistently store IP addresses of clients and set them the same at every new connection. This helps us maintain consistency of the list containing Cloud Access Point IDs matched to their current IPs.

`keepalive 10 60` this setting tells that OpenVPN sends ping packets every 10 seconds and connection is considered interrupted if no pong response is received within 60 seconds.

`dh /path-to-config-folder/dh.pem` file contains Deffie-Hellman parameters. These parameters are necessary for the encrypted key exchange. This parameter is mandatory, but size of DH parameters is up to our decision.

`user nobody` and `group nogroup` - after initialization, OpenVPN drops its privileges from the root privileges to the privileges of `nobody:nobody`. In case of any attack on OpenVPN and gaining a control over OpenVPN, attacker will not have many privileges to harm the system since root privileges were dropped.

`persist-key` must be used with previous element (dropping privileges). When OpenVPN drops the privileges it cannot load private keys again. With this option it is possible to restart OpenVPN without losing access to the keys.

`tls-server` designates server as a TLS main negotiator. Server enforces negotiation and should provide better entropy while generating keys.

`script-security 2` is necessary to enable calling our own scripts (on connection initiated)

`push "route X.X.X.X 255.0.0.0"` tells clients that OpenVPN network has a mask /8 and not /16 as they would get from DHCP.

`crl-verify /path-to-config-folder/crl.pem` tells OpenVPN where to find revoked client certificates. We can revoke a certificate if we decide to cut off some of the clients for any reason.

`client-connect /path-to-config-folder/connect-script.sh` - OpenVPN calls this script after the connection was initiated. This script is designed to call the API to pair RouterID with its current in-cloud OpenVPN IP.

Here we can see an example of our own DNS-like service, which provides us with a list of *RouterID*s paired with their in-cloud IP addresses. From IP addresses we can see that the first device is connected to a different server then the second

device. However, the routing and firewall is set in a way that every device is reachable from every Inthouse Cloud server. For security reasons Inthouse Cloud Access Point can only contact the server which it is connected to and not other servers nor devices in Inthouse Cloud.

```
root@cz:~# dnslist
[
    {
        "id": "r-0000-0000-0000",
        "ip": "10.0.0.99",
        "updated": "2018-05-07 23:11:41 GMT"
    },
    {
        "id": "r-1274-0000-0001",
        "ip": "10.3.0.26",
        "updated": "2018-05-03 13:13:44 GMT"
    }
]
```

7. Quality assurance and testing

The contract with Siemens demands the exact level of quality standards by several ways:

- giving a percentage of minimal supported Android devices there exist on the market according to Google
- defining a crucial parts of the service and their required behavior
- defining a severity of malfunction for selected parts of the service
- giving a deadlines for reparation for each severity level of malfunction
- defining a penalty for non-compliance with the quality standards

Therefore we are motivated to use quality assurance techniques to ensure required quality standard. No we will describe the crucial ways how we ensure the quality.

7.1 Extreme programming

In out team we use several techniques of extreme programming [Bec99] such as:

- continuous integration techniques (SCM tool, bug tracker)
- pair programming
- short software version iterations (small releases, sometimes every week)
- quick feedback (many of integrators have employees dedicated for Inthouse integration who provides us feedback right after the new release)
- frequent communication with the customer (we communicate with our customers on a daily basis)
- code formatting standards inspired by JPL Java Coding Standard [Lab14] (we have standards for indention, new lines usage, spaces usage, braces positioning, package hierarchy, lambdas usage)
- unit testing (we test our APIs reachability and correctness using external services, e.g. HostTracker)
- immediate crash reports and log analysis (our logging framework and Crashlytics)
- try-catch usage standards (if the problem is not fatal, never allow the application to crash, always upload an error report)

This helps us to produce high quality software in a short time in a very small team.

7.2 Continuous integration

We use Mercurial as a code repository and source control management tool. Inthouse repository currently contains around 2400 commits and more then 100 thousands lines of code. Along with Mercurial we use RhodeCode as a task and bug tracker to achieve highest possible management effectiveness. Code package structure is thoroughly created and maintained to achieve clear structure of the modules and units. We maintain implementation road-map which is frequently discussed with the customer and according to that we choose priorities and we focus always on the most crucial element.

7.3 Android Lint

We use IntelliJ IDEA IDE [Jet18] with Android SDKs and plugins. With these SDK comes also Android Lint which is a code inspection tool which helps to uncover possible flaws in code. We found out that this tool is very powerful in our development, because one of the most crucial implementation part belongs to our Android application Inthouse App.

7.4 Beta channel

We publish into beta and to stable release channel on Google Play and on Inthouse Partner Zone (web). We publish beta releases of Inthouse App and Inthouse Studio. All users get notifications about new versions. Users can register as beta-testers and they will get updates and notifications about new versions from beta channel.

We leave each release in a beta channel for at least a week. If we publish some large feature which is sliced into many releases, we use to leave all these releases of this feature in the beta channel until we finish the implementation and test this feature properly.

7.5 Runtime statistics

We have real-time runtime statistics about running application instances. On Android we use Crashlytics by Fabric [Fab18] (owned by Google, see fig. 7.1) which provides us information about instances in real-time and collects statistics about the service like version distribution, time spent in application, new users, number and reason of crashes, etc...

7.6 Crash reports

Crashlytics provides us an immediate notifications about crashes. If some serious bug pops out we would know about it in a matter of minutes. It also provides us the application state in the moment of crash, e.g. logs and stacktraces of each thread. Crashlytics has a support for ProGuard [Gua18] and decodes obfuscated stacktraces to readable form.

Figure 7.1: Runtime statistics from Fabric

7.7 Log uploading

We have implemented a framework for log messages uploading to Inthouse Cloud. Thanks to this we see real-time logs from each running instance (App, Studio and Cloud) in our database. We can search for crashes if some customer calls us that they have spotted some flaw in some of our products.

7.8 End-user feedback

There is an option in Inthouse App to submit a bug or feedback to us. Along with the user's message we can see the whole state of the application like logs and stacktraces and also some basic identification of the customer.

7.9 Testing

To increase the quality we test our application in several ways.

7.9.1 Stability testing

At our office we have several always-on devices running Inthouse App. Some of them runs some particular release without an interruption for several months. Before every release we test products with the set of different Inthouse Configurations which covers most of the supported features.

Considering a stability of Inthouse Cloud we monitor its services and APIs using a monitoring provided by HostTracker [Hos18].

Considering a stability of Inthouse Cloud Access Points, we have several test devices in the office running weeks without any interruption. Our hardware provides additional alarm which restarts a device if the operating system freezes.

7.9.2 Load and performance

Before every release we analyze our products using sampling and allocation tracking tools on a testing set of different Inthouse Configurations.

Load on Android devices depends on device's parameters. We do our best to make also an old Android devices (running Android 4.0 and higher) run Inthouse App flawlessly.

Load on Inthouse Cloud Access Point is very low, our executables utilize less then 20% CPU.

7.9.3 Field testing

We have testers in the real environment who test our beta versions of releases. These testers are usually employees of Siemens and employees of integrators who are dedicated for Inthouse integration. We communicate with these people on a daily basis.

Inthouse Cloud Access Points with Modbus feature are not in the field test yet.

8. Evaluation

In this chapter we will evaluate how successfully we achieved our goals and how we measure this success. By success we mean how expanded the solution is and how satisfied our users are.

8.1 Facenika in Inthouse

The implementation of Facenika's firmware and parts of its hardware and implementation of the extension for Inthouse App to support Facenika was finished and this solution entered a market about a year ago. Since then there have been around one hundred Facenika shelves sold and few users use Inthouse App with Facenika on a daily basis. We know these information from the Activations API (see 5.2.3.3) and from our runtime statistics (see 7.5).

Currently we don't know about any issues or flaws in the firmware nor Inthouse extension. From the manufacturer of Facenika we know, that users flawlessly use Facenika shelves and its features for months. However we suppose (from the nature of the solution) that users don't run Inthouse App as often as they use Facenika lights.

The complexity of the extension is reasonably high. The fact that no controlling is being done over a cloud helps to lower the complexity a lot. The most difficult part to implement was the firmware which needs to be very efficient and stable.

Number of users will depend on the success of Facnika in the future. Nowadays the revenue covers the expenses.

8.2 Modbus in Inthouse

The implementation of this extension is in the final phase before entering the market. We expect the release date to be in the next two months after at least one month of a field testing. There are still some GUI and security features missing. This means that no Modbus-enabled Inthouse Cloud Access Point is at any customer yet. However there are several Inthouse Cloud Access Points of the previous version released and providing Cloud access to Ethernet-enabled Climatix devices to our customers for around 2 years.

We are coming out with this feature because we got many requests from our customers to implement it. Many of our customers who sell devices with built-in Climatix sell 10% of Ethernet-enabled and 90% of Modbus-enabled Climatix controllers. These devices would be controlled the same way as the Ethernet-enabled ones, only the communication technology would be different. This means that there is a very high market potential for Inthouse Cloud Access Point and low time to market, because many of our current customers would add it to their catalog without spending high costs. Success of the Inthouse Cloud Access Point itself would bring a success also for the Inthouse App, since the Inthouse license for the usage with the Climatix device must still be applied.

8.3 Inthouse in general

Inthouse App for a usage with Ethernet-based Climatix devices is on the market since 2015. Since then we provide a stable experience for our end-users and more and more complex features for integrators.

Currently we support 99.7% of Android devices according to Android Distribution Dashboard [Goo18a].

Monthly usage of Inthouse App reaches hundreds of users and technicians of Siemens and integrators use Inthouse Studio on a daily basis. We communicate with these technicians (who represent our direct customers) and receive a feedback several times a week, we provide training courses to them.

From the stability point of view we receive one crash report a month in average (considering a stable version). These crashes are usually due to a very device-specific problems and often not reproducible. We usually fix most of the problems causing crashes in the beta version of applications.

9. Conclusion

In this work we analyzed the market where the system Inthouse takes place by defining its parts, use-cases, requirements and stakeholders. We also created a comprehensive background description which helps us to better understand the whole picture and reasons for facts occurring later in this work. Then we described the technical system overview by showing a technologies and deeper system parts and their roles.

Main part of this work deals with the implementation of two big elements. First of them is adding a support for Facenika devices into Inthouse, including implementation of the firmware for this devices and also designing minor pieces of hardware. Latter one is creating a way to enable Inthouse to control Modbus-enabled devices over the Inthouse Cloud.

Implementation of Facenika support was done completely and the solution has entered the market. Implementation of Modbus support is in final development phase and is preparing to enter the market.

We hope that implementation we wrote about in this work will bring Inthouse to the another level and that it will help Inthouse users to live an easier and more effective life.

Since Inthouse is a proprietary solution and the complete sources contain an important know-how, we attach and mention only the facts and sources which are absolutely necessary for this thesis.

We hope that reading this thesis will bring to a reader new hopes, ideas, inspiration and bravery on a field of software engineering, or open a way to start an own successful software project.

References

[Aut18] Autodesk. Eagle Homepage. 2018. URL: https://www.autodesk.com/products/eagle/.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. 1999.

[Fab18] Google Fabric. Crashlytics Homepage. 2018. URL: https://fabric.io/kits/android/crashlytics.

[Fib18] Fibaro. Fibaro Homepage. 2018. URL: https://www.fibaro.com.

[Fou18] The Linux Foundation. RethinkDB Homepage. 2018. URL: https://www.rethinkdb.com/.

[Gmb18] SmartHMI GmbH. SmartHMI Homepage. 2018. URL: https://www.smart-hmi.de/.

[Goo18a] Google. Android Distribution Dashboard. 2018. URL: https://developer.android.com/about/dashboards/.

[Goo18b] Google. Nest Homepage. 2018. URL: https://nest.com/.

[Gua18] GuardSquare. Proguard Homepage. 2018. URL: https://www.guardsquare.com.

[Hos18] HostTracker. HostTracker Homepage. 2018. URL: https://www.host-tracker.com/.

[Inc18] Blynk Inc. Blynk Homepage. 2018. URL: https://www.blynk.cc/.

[Jet18] JetBrains. IntelliJ IDEA Homepage. 2018. URL: https://www.jetbrains.com/idea/.

[Lab14] NASA Jet Propulsion Laboratory. JPL Java Coding Standard. 2014. URL: http://www.havelund.com/Publications/jpl-java-standard.pdf.

[MK96] R. Canetti M. Bellare and H. Krawczyk. Message Authentication using Hash Functions. 1996. URL: http://cseweb.ucsd.edu/~mihir/papers/hmac-cb.pdf.

[Ope18] OpenVPN. OpenVPN Homepage. 2018. URL: https://openvpn.net/.

[Org06] Modbus Organization. MODBUS over Serial Line, Specification and Implementation Guide. 2006. URL: http://www.modbus.org/docs/Modbus_over_serial_line_V1_02.pdf (visited on 02/12/2018).

[Org12] Modbus Organization. MODBUS APPLICATION PROTOCOL SPECIFICATION. 2012. URL: http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf (visited on 02/12/2018).

[Pro18] OpenWrt Project. OpenWRT Homepage. 2018. URL: https://openwrt.org/.

[Rai] Stéphane Raimbault. Libmodbus. URL: http://libmodbus.org/.

[Sie18] Siemens. Siemens Climatix Homepage. 2018. URL: https://www.siemens.com/global/en/home/products/buildings/hvac/oem/climatix/product-range.html.

[sro15] Inthouse Systems s.r.o. Inthouse specification. 2015. URL: https://inthouse.eu/specification/global.en.

[sro18a] Facenika s.r.o. Facenika Homepage. 2018. URL: http://facenika.com/.

[sro18b] Inthouse Systems s.r.o. Inthouse App - Google Play. 2018. URL: https://play.google.com/store/apps/details?id=eu.inthouse.app.

[sro18c] Inthouse Systems s.r.o. Inthouse Homepage. 2018. URL: https://inthouse.eu/.

[Tap18] s.r.o. TapHome. TapHome Homepage. 2018. URL: https://taphome.com/.

List of Figures

List of Abbreviations

API - Application Programming Interface
B2B - Business to business (trading between two commercial entities)
B2C - Business to customer (trading between commercial entity and terminal customer)
CPU - Central Processing Unit
DHCP - Dynamic Host Configuration Protocol
DNS - Domain Name Service
GSM - Global System for Mobile
GUI - Graphical User Interface
HTTP(S) - Hyper Text Transfer Protocol (Secured)
IDE - Integrated Development Environment
IoT - Internet of Things
I/O - Input/Output
IP - Internet Protocol
LZO - Lempel–Ziv–Oberhumer (data compression method)
MAC - Media Access Control
NTP - Network Time Protocol
OEM - Original Equipment Manufacturer
PLC - Programmable Logic Controller
RAM - Random Access Memory
ROM - Random Only Memory
RGB - Red Green Blue
RPC - Remote Procedure Call
SCADA - Supervisory Control And Data Acquisition
SCM - Source Control Management
SDK - Software Development Kit
SoC - System on a Chip
SQL - Structured Query Language
TCP - Transfer Control Protocol
UDP - User Datagram Protocol
UI - User Interface
UID - Unique IDentification
USB - Universal Serial Bus
USB - Universal Serial Bus
VPN - Virtual Private Network

Attachments

Following attachments are attached to this work

Attachment A

Disk DVD containing following items:

- necessary referenced source codes of Inthouse
- digital version of *Attachment B*
- digital version of this master thesis
- Facenika PCB layout files
- pictures and videos showing Inthouse products
- current version of compiled Android binary of Inthouse App (demo available in the application)
- current version of installation package of Inthouse Studio for Windows (license key needed for usage)
- README file

Attachment B

Official business presentation of Inthouse solution created by the author of this thesis.