Universidade do Minho

Embedded Systems

Masters in Mechatronics Enginering 2016/2017

Development of a home automation embedded system with remote control over the Internet

José Pires – pg29683

July, 2017

INDEX

INDEX OF FIGURES

INDEX OF TABLES

PREAMBLE

The main overall goal of this project is the acquisition of fundamental methodologies and tools to assist in the development of Linux-based embedded systems with networking and external hardware control capabilities.

The usage of a methodology is critical to correctly and conveniently tackle the complexity associated with an embedded system and the multidomain knowledge needed, namely in the hardware and software areas and in the overall project management required.

It is the opinion of the author that the correct usage of this methodology, and tools associated, paves the way for consistent and satisfactory deployment of an embedded system, meeting the customer/end-user requirements and constraints, independently of its complexity.

For this reason, it is intended to highlight the methodology, i.e., the process of conception, analysis, modeling, implementation and testing of an embedded system, in detriment of the overall complexity of the system.

Furthermore, extra knowledge had to be acquired, namely about internet based networks, the protocols which they run under, etc., and about access and control over the General Purpose Input/Output (GPIO) pins, available in the hardware target platform.

For the abovementioned reasons, the home automation embedded system envisioned is fairly modest in its features, but it is focused in the correct and accurate deployment, where features can be incrementally and safely added, supported by the use of the methodology.

1. INTRODUCTION

In this section it will be discussed the motivation behind this project and its goals, being presented the problem statement and some initial considerations about it.

1.1. Motivation and Goals

Nowadays, there is an increasing interest about remote and ubiquitous control of every device one has, at the distance of a fingertip. The household appliances are of a fundamental example of this, ranging from “smart” refrigerators that indicate the food shelf life and the stock present to food-processors that can cook our our meals according to predefined schedules and recipes, which the user can monitor remotely and query its status. However, there is a more embracing and systemic perspective, enabling the centralized and integrated control of every household appliance in one unique system, generically called home automation systems.

There are already some key market players developing home automation systems, namely Apple [1], Amazon and Google, offering a range of devices and solutions to build these systems. In the national market, telecommunication companies are offering an all-in-one product, adding this type of system to their portfolio [2].

Generically speaking, the home automation systems enable the end-user to control, monitor and automate every appliance in his/her household. More recently, these systems have been upgraded to include the remote “telecontrol”, where tele means at a significant distance, mainly Internet (GSM is also an option), as opposed to other wireless communication methods but with shorter range, like Bluetooth, Infra-Red remotes, etc. With the increased range, comes a greater ubiquity of control, allowing the end-user to control and monitor his/her house from virtually anywhere in the world with just an Internet connection; however, with the increased range comes also several security issues, enabling stranger and ill-intentioned people to do the same, which can cause havoc. The security topic is not the primary focus of the present work; though, it is extremely important to notice that security must be properly addressed in any embedded system, specially in on-line devices

(i.e., with internet connection) and in domestic applications where sensitive data can be exposed. For the same reasons, safety should also be taken into account, though is not the primary focus of the present work.

Noting these caveats, the main goal of this project is to develop a home automation system with remote control over the Internet, with just a sample of the features of the production ones, as a proof-of-concept, which can be later on incremented, following the project methodology.

As a proof-of-concept, three representative categories of home automation tasks were selected from an extensive range, namely:

- Lighting control and automation: turn the lights on/off at user's demand, or at a predefined schedule or finally via motion sensor detector. This aims to represent a reactive system to different stimuli, where the user can command (1st option), automate (2nd option) and monitor (3rd option); this last option can be useful to detect if a stranger has entered the premises and it would make even more sense with video surveillance capabilities.
- Room temperature control: monitor and control the temperature of a room. This aims to represent both a reactive system, giving feedback to the end-user of the room's temperature, and an active system, controlling the temperature in the room from the moment it is turned on.
- Doorbell: the end-user should be informed about the presence of someone at his/her door, deciding if it wants to allow the entrance to that person, like a housemaid, or not, signalling that nobody is at home.

The other reasons for selecting these subsystems involve:

- minimizing unneeded complexity, namely in the electronic components/connections for building it, as it is not the primary focus of the project;
- using diverse subsystems where, for instance, a different kind of interface to the hardware platform was utilized, like an Analogue-to-Digital Converter (ADC)
- using representative subsystems of a wide range of home automation systems' features.

Lastly, and relating to representativeness and simplicity, some simple hardware models were used, which mimic the aforementioned features of a home automation system, but don't add up unneeded complexity.

A final word to the follow-up of the methodology: it is a guideline, thus not all steps were executed, in which case it will be conveniently justified. As an example, the next step, as a preliminary way to obtain the system requirements, should be a market study, with some sort of benchmarking. This exceeds the scope of the present work; nonetheless it should be noted its relevance in the project development.

2. Theoretical Foundations

In this section some background is provided for the main subjects. Some technical concepts will be presented as they proved its usefulness along the project. Furthermore, it represents the minimum theoretical concepts that needed to be apprehended to successfully execute the project, namely the project's methodology, TCP/IP stack and multitasking in a Unix/Linux platform.

2.1. Waterfall Methodology

The waterfall model (Fig. 1) represents the first effort to conveniently tackle the increasing complexity in the software development process, being credited to Royce, in 1970, the first formal description of the model, even though he did not coin the term [3]. It envisions the optimal method as a linear sequence of phases, starting from requirement elicitation to system testing and product shipment [24] with the process flowing from the top to the bottom, like a cascading waterfall.

In general, the phase sequence is as follows: analysis, design, implementation, verification and maintenance.

- Firstly, the project requirements are elicited, identifying the key requirements and constraints the system being developed must meet from the end-user perspective, captured in natural language in a product requirements document.
- In the analysis phase, the developer should convert the application level knowledge, enlisted as requirements, to the solution domain knowledge resulting in analysis models, schema and business rules.
- In the design phase, a thorough specification is written allowing the transition to the implementation phase, yielding the decomposition in subsystems and the software architecture of the system.
- In the implementation stage, the system is developed, following the specification, resulting in the source code.

- Next, after system assembly and integration, a verification phase occurs and system tests are performed, with the systematic discovery and debugging of defects.
- Lastly, the system becomes a product and, after deployment, the maintenance phase start, during the product life time.

While this cycle occurs, several transitions between multiple phases might happen, since an incomplete specification or new knowledge about the system, might result in the need to rethink the document.

The advantages of the waterfall model are: it is simple and easy to understand and use and the phases do not overlap; they are completed sequentially. However, it presents some drawbacks namely: difficulty to tackle change and high complexity and the high amounts of risk and uncertainty. However, in the present work, due to its simplicity, the waterfall model proves its usefulness and will be used along the project.

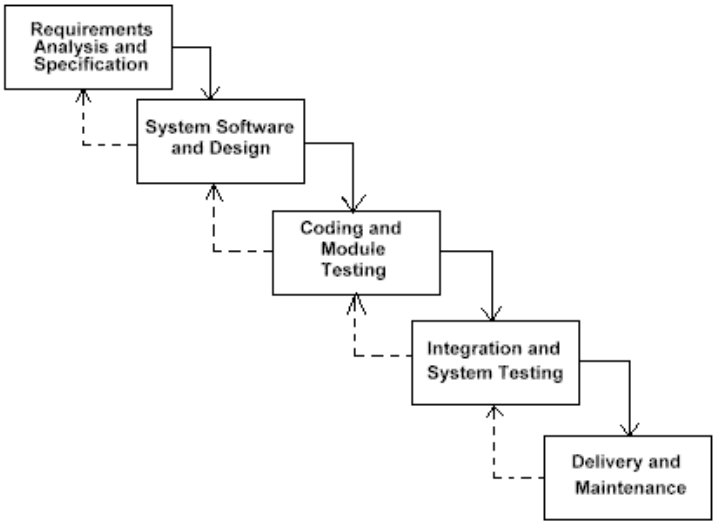


Fig. 1 - Waterfall model diagram

As a reference in the sequence of phases and the expected outcomes from each one, it will be used the chain of development activities and their products depicted in Fig. 2 (withdrawn from [5]).

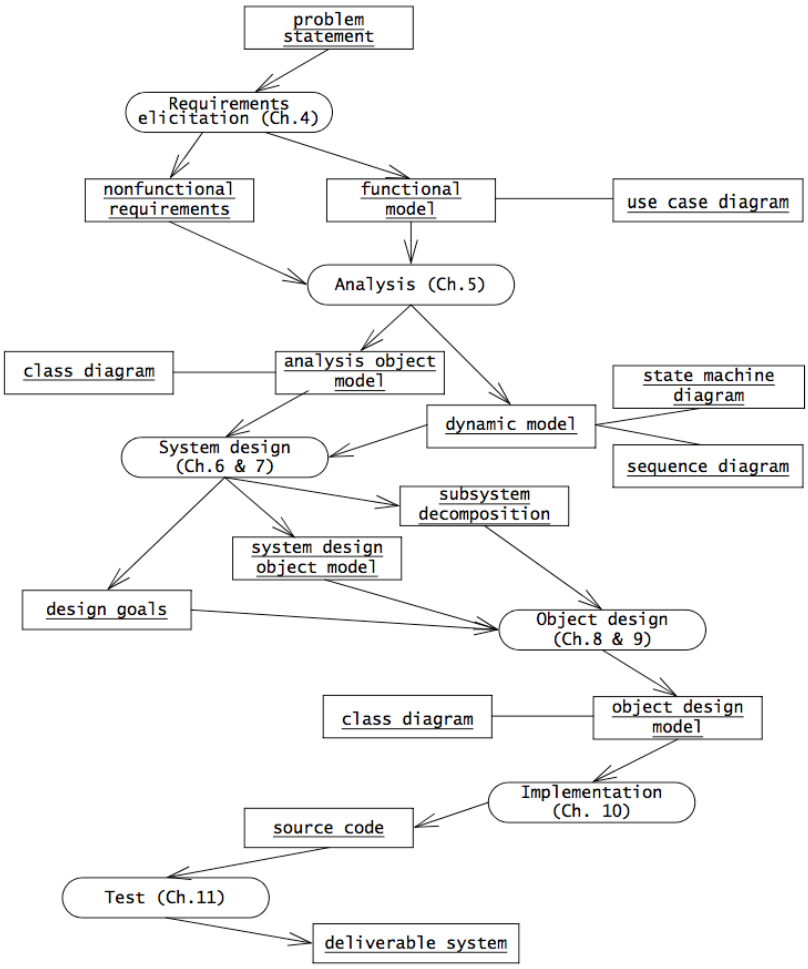


Fig. 2 - An overview of object-oriented software engineering development activities and their products. This diagram depicts only logical dependencies among work products. (Withdrawn from [5])

2.2. Unified Modelling Language (UML)

To aid the software development process, a notation is required, to articulate complex ideas succinctly and precisely. The notation chosen was the Unified Modelling Language (UML), as it provides a spectrum of notations for representing different aspects of a system and has been accepted as a standard notation in the software industry [5].

The goal of UML is to provide a standard notation that can be used by all object-oriented methods and to select and integrate the best elements of precursor software notations, namely OMT, Booch, and OOSE [5]. It provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design, deployment). System development focuses on three different models of the system (Fig. 2) [5]:

1. The functional model, represented in UML with use case diagrams, describes the functionality of the system from the user's point of view.

2. The object model, represented in UML with class diagrams, describes the structure of the system in terms of objects, attributes, associations, and operations.

3. The dynamic model, represented in UML with interaction diagrams, state machine diagrams, and activity diagrams, describes the internal behavior of the system.

2.3. TCP/IP

TCP is an acronym for Transmission Control Protocol and IP for Internet Protocol. It refers to two distinct protocols for internet-based communications, usually associated together, being part of the Open Systems Interconnection (OSI) Model (Fig. 3), which characterizes and standardizes the communication functions of a telecommunication or computing system, being agnostic to their underlying internal structure and technology.

A computer protocol is a standardized procedure for the exchange and transmission of data between devices, as requested for the application processes. The TCP provides services at the Transport layer, handling the reliable, unduplicated and sequenced delivery of data [6], while the UDP provides data transportation without guaranteed data delivery or acknowledgments. The TCP can be thought of a reliable version of UDP, generalizing. The IP part of the TCP/IP suite, providing services at the Network layer, is used to make origin and destination addresses available to route data across networks.

These protocols are applied in sequence to the user's data to create a frame that can be transmitted from the sending application to the receiving application. The receiver reverses the procedure to obtain the original user's data and pass them to the receiving application [6].

Another interesting fact, due to the technology agnostic aspect of the OSI Model, is that IP and the higher-level protocols may be implemented on several kinds of physical nets.

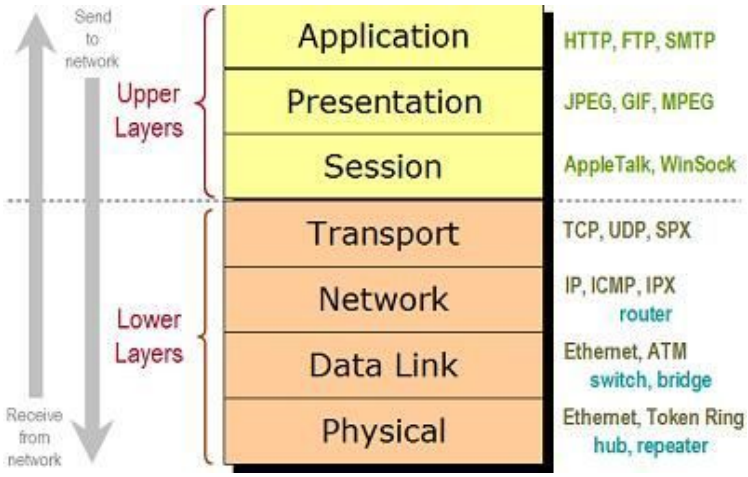


Fig. 3 - Open Systems Interconnection (OSI) Model

2.4. Socket

Computer systems implement multiple processes which require an identifier. As such, the IP address is not enough to uniquely identify the origin/destination of data to be transmitted, and the port number is added. This combination of an IP address and port number is sometimes called a network socket [7], allowing data to be delivered to multiple processes in the same machine - same IP address. It is the socket pair (the 4-tuple consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two end points that uniquely identifies each TCP connection in an internet [7].

In a broader sense, a socket can be described as a method of Inter-Process Communication that allows data to be exchanged between applications, either on the same host (computer) or on different hosts connected by a network [8], as a local interface to a system, created by the applications and controlled by the operating system, allowing an application process to simultaneously send and receive messages from other processes.

The Socket API was created in UNIX BSD 4.1 in 1981, with widespread implementation in UNIX BSD 4.2 [8]. It implements the Client-Server paradigm and implement several (standard) functions to access the operating system network resources, through system calls, in Linux [8].

2.5. Client/Server Model

The client/server model is the most common form of network architecture used in data communications today [9]. A client is a system or application that request the activity of a service provider system or application, called servers, to accomplish specific tasks. The client/server concept functionally divides the execution of a unit of work between activities initiated by the end user (client) and resource responses (services) to the activity request as a cooperative environment [9]. The client, typically handling user interactions and data exchange/modification in the user's behalf, makes a request for a service, and a server, often requiring some resource management (synchronization and access to the resource), performs that service, responding to the client requests with either data or status information [10].

An example of a simple client-server model using the Socket API, through system calls, is presented in Fig. 4. The operation of sockets can be explained as follows [8]:

- The socket() system call creates a new socket, establishing the protocols under which they should communicate. For both client and server to communicate, each of them must create a socket.
- Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place. Two sockets are connected as follows:
  1. One application, assuming the role of server, calls *bind()* to bind the socket to a well-known address, and then calls *listen()* to notify the kernel it is ready to accept incoming connections.
  2. The other application, assuming the role of client, establishes the connection by calling *connect()*, specifying the address of the socket to which the connection is to be made.
  3. The server then accepts the connection using *accept()*. If the *accept()* is performed before the client application calls *connect()*, then the *accept()* blocks.
- Once a connection has been established, data can be transmitted in both directions between the applications (analogous to a bidirectional telephone conversation) until one of them closes the connection using *close()*.

- Communication is performed using the conventional *read()* and *write()* system calls or via a number of socket-specific system calls (such as *send()* and *recv()*) that provide additional functionality. By default, these system calls block if the I/O operation can't be completed immediately. However, nonblocking I/O is also possible.

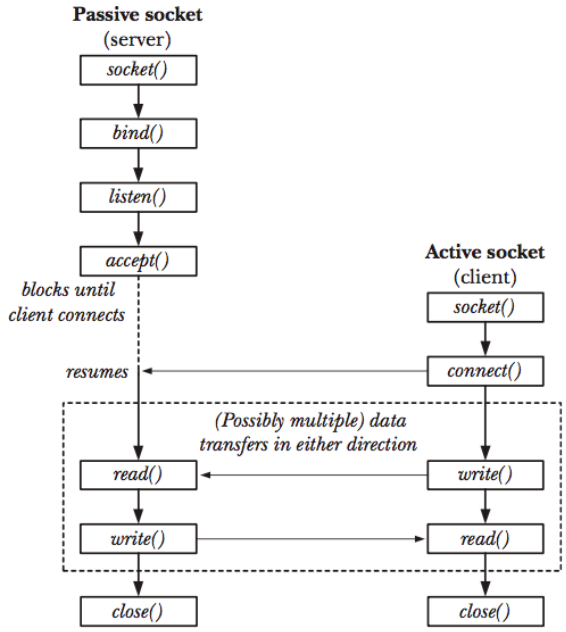


Fig. 4 - Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [8])

2.6. Concurrency

Concurrency is used to refer to things that appear to happen at the same time, but which may occur serially [11], like the case of a multithreaded execution in a single processor system. Two concurrent tasks may start, execute and finish in overlapping instants of time, without the two being executed at the same time. As defined in POSIX, a concurrent execution requires that a function that suspends the calling thread shall not suspend other threads, indefinitely.

This concept is different from parallelism. Parallelism refers to the simultaneous execution of tasks, like the one of a multithreaded program in a multiprocessor system. Two parallel tasks are executed at the same time and, as such, they require the execution in exclusivity in independent processors.

Every concurrent system provides three important facilities [11]:

- Execution Context - Refers to the concurrent entity state. It allows the context switch and it must maintain the entities states, independently.
- Scheduling - In a concurrent system, the scheduling decides what context should execute at any given time.
- Synchronization - This allows the management of shared resources between the concurrent execution contexts.

2.7. Pthread

Pthreads is a standardized model for dividing a program into sub tasks whose execution can be interleaved or run in parallel. It results from an effort of standardizing a threading API for POSIX (Portable Operating System Interface) compliant systems, the family of IEEE operating system interface standards, in which Pthreads is defined (the ”P” in Pthreads comes from POSIX). Vendors usually supply Pthreads implementations in the form of a header file, which can be included in the application, and a library, to which the program can be linked to [12]. With Pthread, the execution contexts are created by calling pthread_create(), the scheduling parameters modifiable while the thread executes and thread termination performed via a simple return or the pthread exit function. The primary synchronization mechanisms of the Pthread API are mutexes and condition variables [12].

3. REQUIREMENTS ELICITATION

The requirements elicitation is the 2nd step of the waterfall methodology, following closely the problem statement, defining the system in terms understood by the customer. There are several techniques to elicit requirements, namely: questionnaires, task analysis, scenarios and use cases [5]. In the present work, as no interaction is possible with the end-user, the first two techniques were not considered. However, one useful tool to understand a little bit better the product and the customer would be, as aforementioned in section 1.2., a market study with some sort of benchmarking to bridge the gap between the end user and developer.

As this task presented itself very time-consuming and being outside of the scope of this work, it was also not considered. The chosen technique, as supported by the methodology, was using Scenario-based design, yielding the use cases, as abstractions that describe a class of scenarios [5].

As the purpose of this work is a proof-of-concept system, the requirements being considered will be the bare minimum to illustrate the operation of the three home automation tasks mentioned in section 1.1.

The requirements specifications are then structured and formalized during analysis to produce a complete, correct, consistent and unambiguous model of the system – the analysis model – described in terms of its structure (analysis object model) and its dynamic interoperation (dynamic model).

3.1. Requirements

There are two types of requirements to consider:

1. functional requirements - directly related to the functionalities of the product, describing the user tasks that the system needs to support;
2. non-functional requirements – not directly related to the functional behaviour, describing the properties of the system.

3.1.1. Functional Requirements

As the major functional subsystems have been identified as part of the problem statement (section 1.1.), they will be used to more clearly identify the functional requirements.

- Lighting Subsystem
    - Enable the user to query and monitor the state of the lighting subsystem;
    - Enable the user to turn on/off the lighting subsystem;
    - Enable the user to automate on a time-basis the lighting subsystem;
    - Detect the presence of a person in the surroundings and turn on the lighting subsystem and notify the user about this event.
- Temperature Subsystem

- Control the temperature in a room based on a user-defined temperature set-point;
- Enable the user to query and monitor the state of the temperature control subsystem;
- Enable the user to turn on/off the temperature control subsystem;
- Enable the user to automate on a time-basis the lighting subsystem;
- Enable the user to configure the temperature set-point along an admissible range.

- Door-bell
  - Notify the user that a person is ringing the bell;
  - Notify the person at the door that the door bell is properly working by playing an audible bell-like sound;
  - Enable the user to remotely open the door;
  - Notify the person at the door that nobody is at home by playing an audible sound, if the user doesn't acknowledge the door-bell or if he chooses to decline the entrance to that person.
- Real-Time Clock (RTC)
  - Enable the user to define the date and time of the system;
  - Provide the system with real-time capabilities, enabling the accurate automation of the system;
  - Enable the user to obtain accurate time-stamped messages associated with system events;
- Overall system
  - Enable the user to establish a remote connection to his/her home automation system;
  - Notify the user if the remote connection establishment was not possible or if the system is down.

3.1.2. Non-functional requirements

- The login of the user in the system must be accomplished via an authentication method and all communications should be encrypted, as security is a critical feature;
- Low latency: the notifications to the user or acknowledgments of actions executed must not surpass 5 seconds;

- The device must operate even without user interaction, according to its presets;
- The device must log the notification data even if no internet connection is available, sending it to the user as soon as it is re-established;
- Friendly user interface.

3.2. Constraints

The project constraints can be classified in two types: the **technical constraints**, related to the technical aspects of the product, and the **non-technical constraints**, associated with the project management. Both types represent limitations to the project development that must be respected, addressing and overcoming the resulting problems.

3.2.1. Technical Constraints

- The target hardware platform is the Raspberry Pi;
- Use embedded operating system based on the Linux kernel;
- Use of the GNU toolchain;
- Use the TCP/IP stack;
- Use the General Purpose Input/Output (GPIO) pins on the target board;

3.2.2. Non-technical constraints

- Limited budget;
- Limited time: maximum of 15 hours/week;
- Limited manpower: only one individual;
- Meeting the specified deadlines;
- Limited technical knowledge about electronics and software

3.3. Project's Overview

The project consists of two main systems: the **Local system** and the **Remote system**, implementing the client/server architecture. The Local system will act as the server, providing services to the Remote system (client). The former system will

be the core of the embedded system, managing the access of the client to services/resources and controlling all peripherals related to the home automation tasks; the latter represents the remote interface to the Local system as a means to request access to services and be notified of relevant events. The Local system should also include some type of display for immediate data visualization and a physical interface as good measure to include even the non tech-savvy persons, but for simplicity it was considered that the remote interface to the local system (Remote system) should suffice in providing these features. In the same simplicity motto, the Remote system is intended to be a simple command-line interface (CLI), but it can be easily migrated to a Graphical User Interface (GUI), by tracing the triggering events and mapping them to the appropriate GUI elements.

3.4. System Architecture Overview

The system architecture overview is illustrated in Fig. 5. It consists of two main systems aforementioned: the Local and Remote systems. It is important to mention that the subsystems that decompose the Local system are physical models that intend to model and simplify the electronic connections necessary. The reasons behind this are practical ones: lighting up a bulb or a LED model the same behaviour and have basically the same amount of complexity; however, the light bulb adds bulkiness to the system and requires more power to operate, therefore, needing a bigger power supply.

- **Local System**:
    - Lighting Control subsystem – composed of a LED (to model the Light Bulb) and a motion detector sensor (in this case, a Pyroelectric Infra-Red (PIR) sensor).
    - Temperature Control subsystem – composed a sensor temperature (thermistor) and a cooling fan.
    - Door-bell subsystem – composed of a pushbutton (to model the door-bell button) and a buzzer (to mimic the door-bell sound).
    - Real-Time Clock subsystem – composed of a RTC to provide the system with real date and time.
    - Server subsystem – acts as an interface to the external world, handling all TCP/IP communications and enabling clients to request

services from the local system. It should also assure the data encryption to the out-going communications.

- Data storage subsystem – composed by the SD card. It enables data storage of user configuration and other relevant parameters to the system.

- **Remote System**, acting as a client: this system is an interface to the Local System, available as a Command-Line Interface (CLI), but which can be migrated to a GUI; for this reason, it is only available to platforms with access to a command-line, typically desktop (it was not conceptualized, for now, something like a smart-phone application).

Both systems communicate through the use of the TCP/IP protocols in a client/server paradigm.

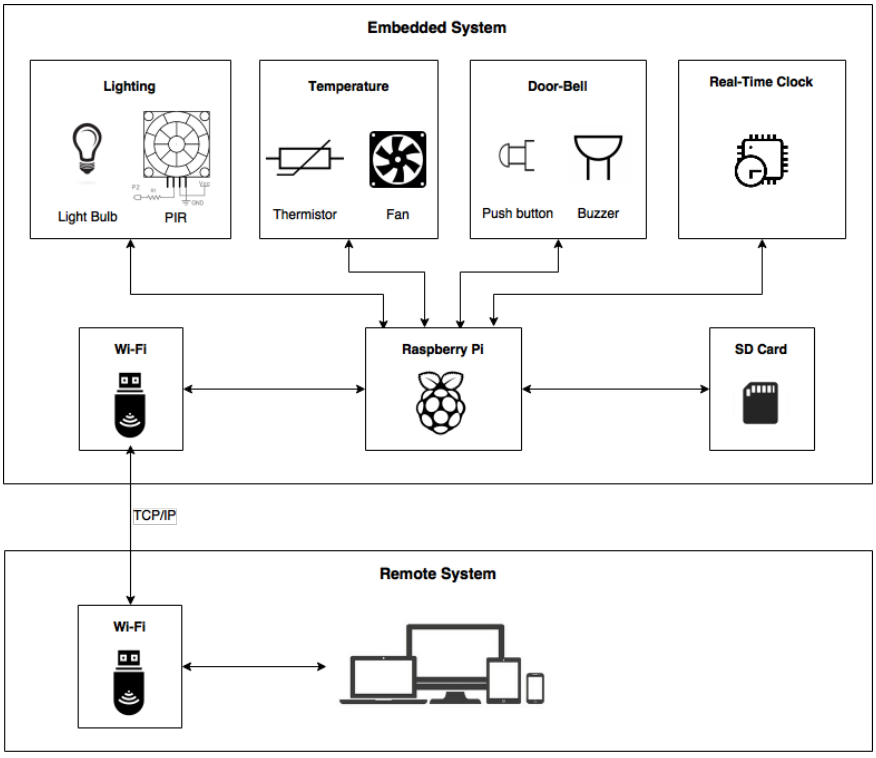


Fig. 5 - System Architecture Overview

3.5. Project Resources

An overview of the hardware and software resources that will be used in the project is presented. This resources could subject to change in the design and

implementation phases as new knowledge about the system is acquired, requiring new resources or discarding the already existent ones.

3.5.1. Hardware resources

- Raspberry Pi 3, model B, development board
- SD Card
- Wi-Fi USB dongle: starting from Raspberry Pi 3, the Wi-Fi is now built-in
- LEDs
- Pyroelectric Infra-Red (PIR) sensor
- Thermistor
- Analog-to-Digital Converter (ADC)
- Fan
- Push Buttons
- Buzzer
- Power Supply
- Enclosure

3.5.2. Software Resources

- Embedded OS based on the Linux Kernel
- C/C++ programming language
- Pthread Library
- WiringPi Library

3.6. Local System

3.6.1. Use Cases

The next requirements elicitation activities include: identifying actors, scenarios and use cases, refining the uses and identifying relationships among actors and use cases.

3.6.1.1. Actor Catalogue

Actors represent external entities that interact with the system, defining classes of functionality and role abstractions that do not necessarily directly map to persons. Thus, an actor can be human or an external system [5].

Some initial considerations are needed. The user interaction with the system shall always be done via the remote interface (remote system), except in the case of power failure, where the user is needed to turn the power back on. As far as external interactions are concerned, the local system will not distinguish between both user and remote system, except in the case of an inexistent network connection, where the remote system is supposed to address this, by queuing the data and sending it as soon as possible to the local system, providing the user with feedback. As this is a very specific case of client/server interaction, for simplicity here, it will only be considered the user as an actor.

Similarly, all interaction acknowledgments to the local system will be done through the server subsystem. As the server represents the interface to the local system, mainly concerned with handling the requests for services from the client, it will be considered as an integrated part of the local system and, therefore, it will not be distinguished. The only exception is, once again, when no network connection is available, being the server responsible to queue the data and transmit it to the client as soon as possible. Once again, as this is a very specific case of client/server interaction, for simplicity here, there will be no distinction between server and local system at this point.

The actor catalogue is presented in Fig. 6, composed of:

- **User** – represents any user with permissions to use the system.
- **Visitor** – represents the home visitors that can either ring the door-bell or trigger the motion detection module, associated with the lighting subsystem.
- **System Engineer** – represents the only entity responsible for the maintenance of the system and the only with ability to change critical system parameters.

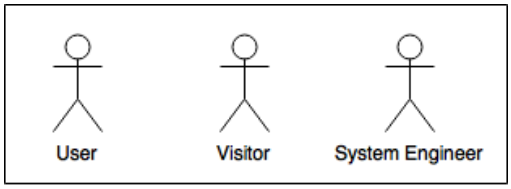


Fig. 6 - Local System Actor Catalogue

3.6.1.2. Scenarios

The 2nd step in the heuristics for determining the use cases is an identification of the main relevant scenarios, enabling the extraction of information from the application domain by using concrete examples. To describe a scenario, one must use natural language with specific instances names, like miguel:User, where miguel is a specific instance of an User and indicate the flow of events, resulting in a textual description.

All determined scenarios, alongside with its textual description, are presented in Appendix 1. For simplicity, here it will be presented a resume diagram indicating the aforementioned scenarios with a hierarchy relation established by a dashed enclosure (Fig. 7).

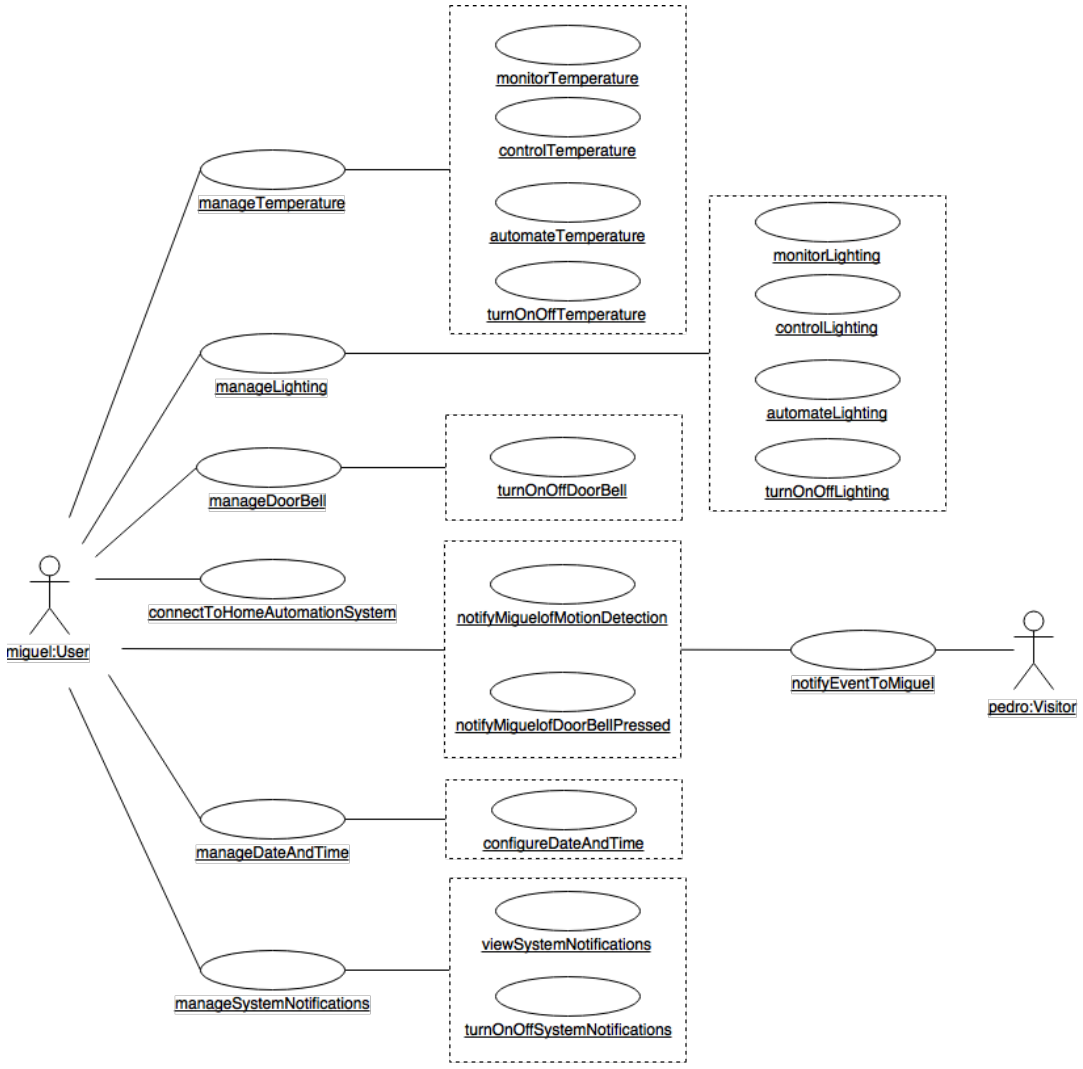


Fig. 7 - High Level Scenarios for the home automation system:
the dashed bordered scenarios represent the specific ones and are included for clarity

The main scenarios determined are: connectToHomeAutomationSystem, manageTemperature, manageLighting, manageDoorBell, manageDateAndTime and manageSystemNotifications, all of which initiated by miguel:User, and notifyEventToMiguel, which is initiated by pedro:Visitor and with participation of miguel:User. These high-level scenarios result from the aggregation of several low-level scenarios, represented by a dashed enclosure just for clarity purpose (it is not part of the UML notation). For example, the manageTemperature scenario includes monitorTemperature, controlTemperature, automateTemperature and turnOnOffTemperature.

A working glossary was also established as a means to ensure consistency in the specification and that developers use the language of the client. This working glossary should be updated along the project and an important example concept is presented in Table 1.

Table 1 - Working Glossary for the home automation system. Keeping track of important terms and their definitions ensures consistency in the specification and ensures that developers use the language of the client.

Notification	A notification is an update about the system activity, regarding its relevant events. Examples of this are: Initial and End time for automated tasks, external events like Motion Detection and Door Bell Pressed and logons on the system. It should include a descriptive message and a time-stamp. Data entries older than 30 days should be removed automatically.

3.6.1.3. Primary Use Cases

A use case specifies all possible scenarios for a given piece of functionality, being initiated by an actor and representing the complete flow of events through the system resulting from its initiation [5]. The primary use cases refer to the most relevant ones, whereas the secondary use cases refer to the least-important ones, like the ones initiated by the system engineer. The use case utilizes a textual description with a generic name, enlisting the participating actors and describing the flow of events, the entry and exit conditions and the quality requirements (this one after refination).

Again, the more detailed text-based description is included in Appendix 2, alongside with a hierarchical refining of use cases, and here it will be only presented

a diagram of the main primary use cases. As a side note, for simplicity reasons, from now on, the Home Automation System will be abbreviated to HAS.

In Fig. 8 is depicted the refined primary use cases model (zoomed in Fig. 70). It is possible to see two actors, User and Visitor, interacting with the system, initiating or/and participating in the use cases. The User initiates all uses cases, except the DetectMotion and PressDoorBellButton, that act like triggers, initiated by the Visitor. Both this use cases are extended by NotifyUser, notifying the User and logging a system message (LogSystemMessage). The other system messages with time-stamp information that the user requires are related to the connection to HAS (ConnectToHas) and the automation tasks (start and end times – Automate). The User initiates all use cases related to the system management namely ManageHAS, which includes ManageTemperature, ManageTemperature, ManageDoorBell, ManageDateAndTime, ManageSystemNotifications, ManageSettings and Logout, which are the core of the application domain.

The different color arrows represent the relationship include or extend, and were not all drawn to simplify the diagram. The use cases SwitchMode, SwitchState, Modify and Automate are factorizations to illustrate the main concepts (they are divisible by <Parameter>, where <Parameter> refers to the use case that includes, e.g., SwitchTemperatureMode, SwitchLightingMode, etc.).

The use case UpdateDateAndTime, extends both use cases ManageHAS and ManageDateAndTime, referring to the date and time needed for user's feedback, as the embedded clock would take care of automatically update date and time, just lacking retrieving it for user feedback.

There are two more use cases that are relevant to mention, as some considerations are needed. ConnectionDown use case extends all use cases, as this constitutes an important exception that hinders the communication between the User and HAS. This refers to User trying to connect to HAS or expecting some feedback from it. If trying to establish a connection is not possible, the user must be informed. If expecting some feedback, HAS should respond periodically to the User, just to acknowledge the internet connection is available. Otherwise, the user should be informed about the broken internet connection. Simply put, HAS needs to acknowledge periodically to the User that is available over an Internet connection to provide its services, otherwise the User will be informed that HAS is "down".

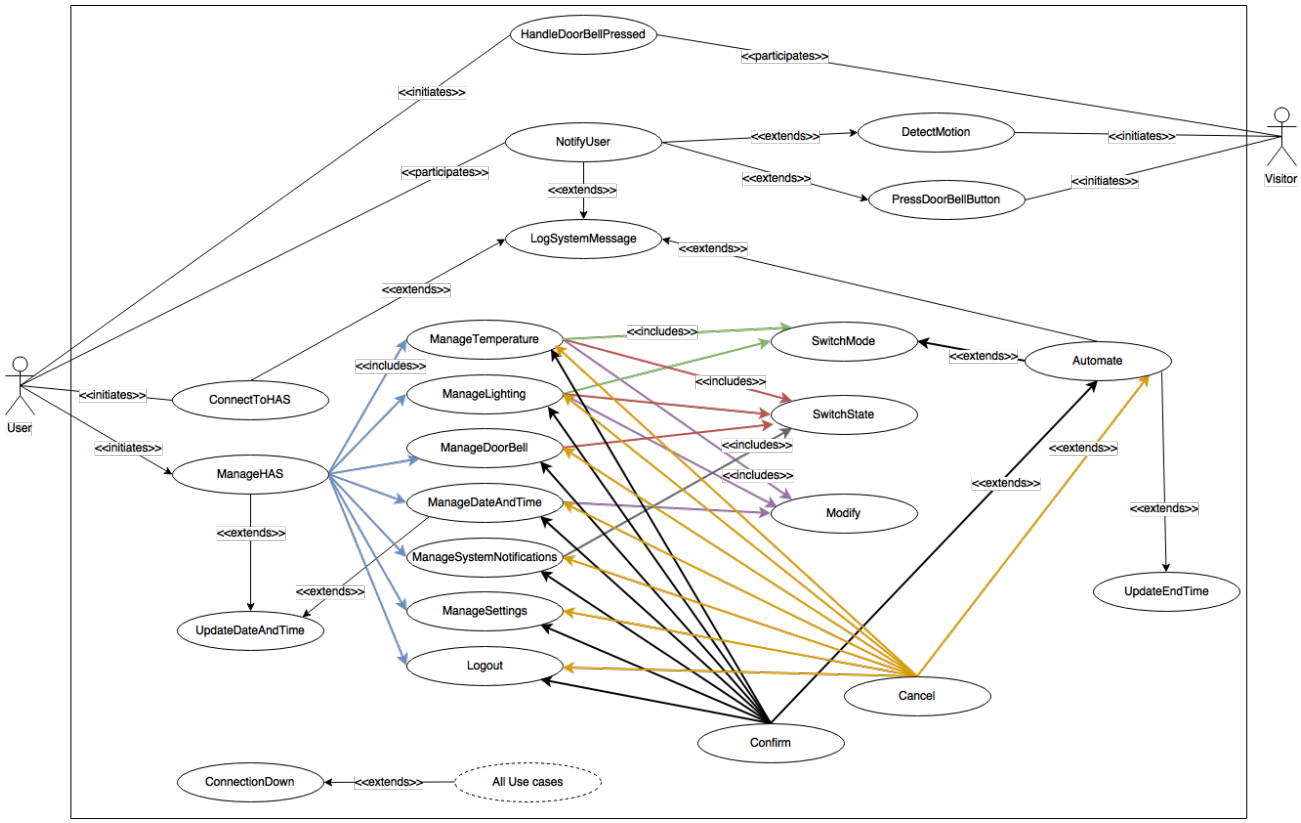


Fig. 8 - Refined primary use cases model (zoomed in Fig. 70): Two actors, User and Visitor, initiate or/and participate in the use cases. The different color arrows represent the relationship include or extend, and were not all drawn to simplify the diagram. The use cases SwitchMode, SwitchState, Modify and Automate are factorizations to illustrate the main concepts (they are divisible by <Parameter>, where <Parameter> refers to the use case that includes, e.g. SwitchTemperatureMode, SwitchLightingMode, etc.)

The final relevant use case is NotifyUser. It is not realistic to think that the User needs to be logged in to the system to receive these important notifications. These events can occur anytime and should be delivered to the User under any circumstances, as they can indicate, for instance, that a burglar is at the User's house. Several solutions could be used: if there is Internet connection, an email service could be provided in HAS to email the user, reporting these events. In a more critical situation, with no Internet connection available, a Global System Message communication (GSM) could be provided to send a text message to the User's phone. Otherwise, when the User connects back to the system, the notifications should pop-up indicating that these priority events have occurred. Furthermore, as data needs to be exchanged over TCP/IP, more effective methods could be used to handle the data, namely, like storing it in a database in an external HTTP server for global access. This could also solve the problem about establishing a connection, as both the User and HAS would connect to the same entity and

exchange data through there, freeing resources in HAS. As these solutions add an unneeded complexity, it is considered for practical reasons that the internet connection will be reliable (wired internet connection between the User and HAS will be provided in this case).

3.6.1.4. Secondary Use Cases

The secondary use cases refer to less important use cases that will not be thoroughly detailed; instead they are used to illustrate others actors like the system engineer and other tasks that seems insignificant like powering up or down the system, but that should be addressed to identify the due responsibilities.

In Fig. 9 are represented some of these secondary cases, namely:

- UpdateSystem: Describes the system engineer ability to update the software.
- ConfigureMachine: Demonstrates the possibility to change the network options.
- CheckIfMachineIsWorking: Describes the responsibility of the system engineer to verify the proper operation of the HAS system.
- SwitchPowerOnAndOff: Describes the responsibility of the User to power up and down the system.

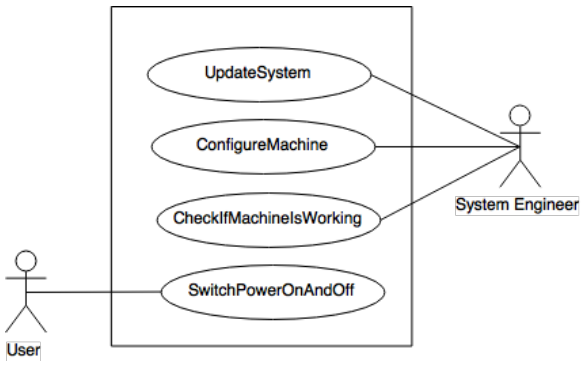


Fig. 9 - Secondary Use Cases for HAS

4. ANALYSIS

Analysis is the next step in the software development and represents the effort of producing a model of the application domain, the analysis model, that is correct, complete, consistent and unambiguous. As it is a model of the application domain it represents the user's view of the system; however, the developer now tries to formalize these concepts contained in the use cases, gathered in the requirements specification produced in the previous phase, in a consistent and systematic way, ensuring correctness and unambiguity as critical criteria.

The analysis model, produced in this phase, is composed of three individual models [5]:

- The **functional model**, describing the system functionality, represented by the use cases and scenarios;
- The **analysis object model**, describing the structure of the system, represented by the class and objects diagram;
- The **dynamic model,** describing the internal behaviour of the system, represented by the sequence and state-machine diagrams.

In Fig. 10 are depicted the analysis activities, described by a UML activities diagram [5]. This is the backbone guideline for the analysis phase presented in this section.

The use cases were defined during requirement elicitation, but they can be updated, removed or added some new ones, as more knowledge about the system is gained.

From the use cases, it is possible to define the participating objects, namely:

- **Entity objects**: representing the permanent information tracked by the system;
- **Boundary objects**: representing the interaction between the actors and the system;
- **Control objects**: responsible for executing use cases;

Then, the interactions between objects are defined and use cases are mapped to objects with sequence diagrams. The associations, attributes and aggregations are also identified and the state-dependent behaviour is defined.

Lastly, the model is consolidated, through the identification of relationships of generalization/specialization, and reviewed. It is also important to notice that analysis is highly iterative, like all software engineering development processes.

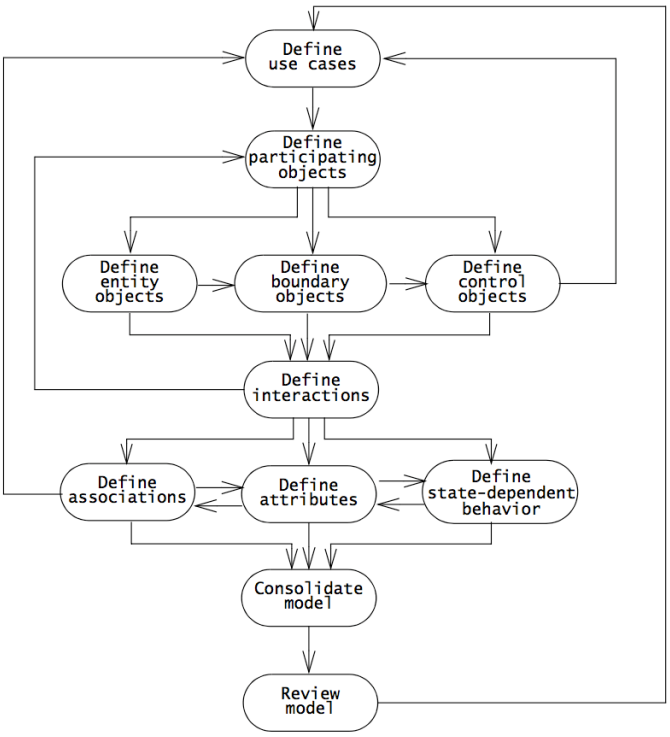


Fig. 10 - Analysis activities (UML activities diagram) (withdrawn from [5])

4.1. Participating Objects

In this section, the participating objects will be determined using some heuristics as suggested by Bruegge and Detoit [5].

4.1.1. Entity Objects

The entity objects represent the permanent information tracked by the system. The heuristics to identify entity objects from use cases are as follows [5]:

- Terms that developers or users need to clarify in order to understand the use case

- Recurring nouns in the use cases
- Real-world entities that the system needs to track
- Real-world activities that the system needs to track
- Data sources or sinks

Applying the above heuristics to all uses cases yielded the entity objects presented in Table 2 in a glossary form, with a clear description of the objects.

Table 2 – Glossary of Entity objects determined for HAS

User	Individual that utilizes and commands the Home Automation System and the associated services. Users are identified by UserID and Password.
Visitor	Individual that comes to the house, pressing the door bell button or triggering the motion detector. It is only identifiable as the source of external events, thus, not requiring additional information to be tracked.
Connection	Internet connection between the User and HAS. It is composed of a status (Established or Not Established).
Settings	System and User configurations available. They can be: UserID, Password, etc.
UniversalTime	Date and time defined in the system. It is updated on a minute basis and is composed of Year, Month, Day, Hours and Minutes.
State	State of a relevant subsystem with two possible values, On and Off. When off, it should disable its operation, with exception of the Lighting subsystem. When on, it can start its operation immediately (manual mode) or only when scheduled (automatic mode)
Mode	Mode of an automatable subsystem with two possible values, Manual and Automatic. When Automatic is selected, it allows the definition of the start and end times of the automated task.
Temperature	Temperature inside a room to be controlled. It is composed of the set-point (user-defined) and actual values. It is automatable.
Brightness	Brightness of the lighting in the house exterior. It is user-defined and assumed to be "accurate", not requiring it to be measured. It is automatable (e.g., to indicate someone is home).
Notification	Update about the system activity, regarding its relevant events, internal or external. It is composed of a descriptive message and a time-stamp. Depending on the type of event, they are handled differently (see Event description). Data entries older than 30 days should be removed automatically.
Event	Relevant occurrence in the system. Events can be external (Door-bell button pressed, Motion detected) or internal (start and end time of an automated task and log-on in the system). Internal and external events are registered in the System Notifications. Additionally external events are communicated immediately to the User.
Start/End Time	Time span during which an automatable task is active. It is composed of Hours and Minutes, disregarding the date, meaning that everyday, while the automatic mode is selected, the task will be executed.

4.1.2. Boundary objects

The boundary objects represent the interaction between the actors and the system, thus coarsely modelling the user interface with the system. The heuristics to identify boundary objects from use cases are as follows [5]:

- Identify user interface controls that the user needs to initiate the use case;
- Identify forms the users needs to enter data into the system;
- Identify notices and messages the system uses to respond to the user;
- When multiple actors are involved in a use case, identify actor terminals to refer to the user interface under consideration;
- Do not model the visual aspects of the interface with boundary objects (user mock-ups are better suited for that);
- *Always* use the end user's terms for describing interfaces; do not use terms from the solution or implementation domains.

Applying the above heuristics to all uses cases yielded the boundary objects presented in Table 3 in a glossary form, with a clear description of the objects.

Table 3 - Glossary of boundary object for HAS

UserStation	Computer platform (desktop, smartphone, etc) used by the User to connect and control HAS.
ConnectToHASForm	Form used for the authentication of the user in HAS. It is presented when user starts the remote software application in UserStation.
HASForm	Form to manage HAS. It is presented when user is authenticated in the system.
LightingForm	Form to manage the lighting subsystem. It is presented when the ManageLighting function is selected.
TemperatureForm	Form to manage the temperature subsystem. It is presented when the ManageTemperature function is selected.
DoorBellForm	Form to manage the door-bell subsystem. It is presented when the ManageDoorBell function is selected.
UniversalTimeForm	Form to manage the date and time of HAS. It is presented when the ManageUniversalTime function is selected.
NotificationsForm	Form to manage the system's notifications. It is presented when the ManageNotifications function is selected.
SettingsForm	Form to manage the system's settings. It is presented when the ManageSettings function is selected. It contains the username and the ability to change the password.
AutomateForm	Form to manage the automation of a lighting or temperature task. It is presented when the Automate function is selected from LightingForm or TemperatureForm, by switching the mode to automatic.
LogoutButton	Button to logout from HAS.

ConnectionDownNotice	Notice used for displaying the connection down information to the user. The user can only acknowledge it. It is triggered when no connection is available.
MotionDetectedAlert	Notice used for displaying the motion detection alert to the user. The user can only acknowledge it. It is triggered when motion is detected in the house perimeter
DoorBellRingDialog	Dialog used for informing the user that someone is at the door and allowing the user to decline or accept the request. It is triggered a visitor presses the door bell button.
ConfirmButton	Button used for confirming a modification in all subsystem and functionality managers and return to the one that originated it.
CancelButton	Button used for cancelling a modification in all subsystem and functionality managers and return to the one that originated it.
StateButton	Button used to switch between the two possible states: ON and OFF. It is used in the LightingForm, TemperatureForm, DoorBellForm and NotificationsForm. It disables all other functionalities, except the button itself. The default value is OFF.
ModeButton	Button used to switch between the two possible modes: MANUAL and AUTOMATIC. It is used in the LightingForm and TemperatureForm. Switching to AUTO, triggers the AutomateForm. The default value is MANUAL.
ModifyButtons (Up/Down)	Buttons used for modifying the values of temperature set-point, brightness, date and time in UniversalTimeForm, and start and end times in AutomateForm.
InfoLabel	Label used for displaying the values unmodifiable by the user, but still relevant, namely: connection status and real temperature.

Additionally, to aid the comprehension of the boundary objects, a more graphic representation was added, namely the user mock-ups, illustrated in Fig. 11.

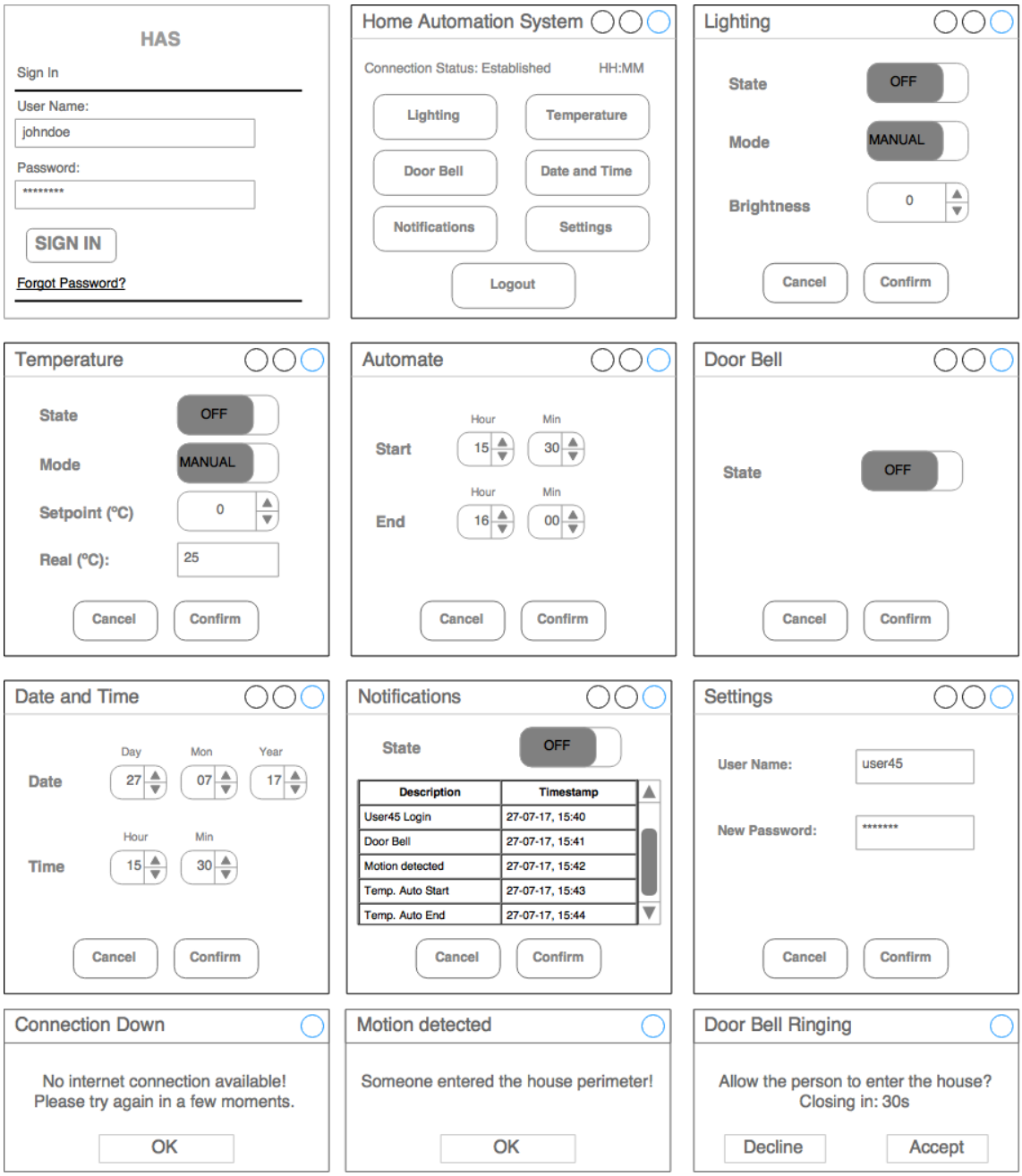


Fig. 11 - User mock-ups for HAS

4.1.3. Control objects

Control objects are responsible for coordinating boundary and entity objects, namely for collecting information from the formers and dispatching it to the latters. The heuristics to identify control objects from use cases are as follows [5]:

- Identify one control object per use case.
- Identify one control object per actor in the use case.

- The life span of a control object should cover the extent of the use case or the extent of a user session. If it is difficult to identify the beginning and the end of a control object activation, the corresponding use case probably does not have well-defined entry and exit conditions.

Applying the above heuristics to all uses cases yielded the control objects presented in in a glossary form, with a clear description of the objects. A syntactical rule was followed, consisting of appending the suffix <Control> to all control objects.

Table 4 - Control object for HAS

ManageConnectionControl	Manages the ConnectToHAS function on the UserStation. This object is created when User starts the remote software application in the User Station. It then creates an ConnectToHASForm and presents it to the User. After submitting this form by selecting the Sign In function, this object then collects the information from the form and forwards it to HAS. The control object then waits for a successful or unsuccessful authentication acknowledgement to return from HAS. If the former is received, this object presents the ManageHASForm; if the latter is received, it presents the ConnectionDownForm. If the CloseButton is selected it closes the application.
ManageHASControl	Manages the HAS system on the UserStation. This object is created when the User is successfully authenticated in HAS. It then creates an ManageHASForm and presents it to the User. After submitting this form by selecting a Manage option, this object presents the associated form.
ManageLightingControl	Manages the lighting subsystem on the UserStation. It is presented when user selects the ManageLighting option from ManageHASForm. If it is the first time being activated, it presents the default values (State = OFF, Mode = Manual, Brightness = 0). It then stores the previously selected values and waits for the User to switch state (SwitchStateControl), switch mode (SwitchModeControl), modify brightness (ModifyControl), confirm (ConfirmControl) or cancel (cancelControl). The switch state and modify brightness are immediately submitted to HAS, with the brightness value being updated to the user, waiting for HAS to acknowledge the information received. If switch state is switched to OFF, it disables all functionalities, except SwitchStateControl, ConfirmControl, and CancelControl. If the SwitchModeControl is switched to AUTO, the AutomateForm is presented to the user, otherwise, the brightness information is send to HAS and waits for acknowledgement. If the ConfirmControl is selected, the ManageLightingForm is destroyed and

	ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected values are sent to HAS and waits for acknlowdgement. Then, the ManageLightingForm is destroyed and ManageHASForm is presented to the user. While the User remains in the ManageLightingForm, the state can be updated via motion detector signal.
ManageTemperatureControl	Manages the temperature subsystem on the UserStation. It is presented when user selects the ManageLighting option from ManageHASForm. If it is the first time being activated, it presents the default values (State = OFF, Mode = Manual, Setpoint = 25). It then stores the previously selected values and waits for the User to switch state (SwitchStateControl), switch mode (SwitchModeControl), modify temperature set-point (ModifyControl), confirm (ConfirmControl) or cancel (cancelControl). The switch state and modify set-point are immediately submitted to HAS, with the set-point value being updated to the user, waiting for HAS to acknowledge the information received. If switch state is switched to OFF, it disables all functionalities, except SwitchStateControl, ConfirmControl, and CancelControl. If the SwitchModeControl is switched to AUTO, the AutomateForm is presented to the user, otherwise, the brightness information is send to HAS and waits for acknowledgement. If the ConfirmControl is selected, the ManageTemperatureForm is destroyed and ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected values are sent to HAS and waits for acknlowdgement. Then, the ManageTemperatureForm is destroyed and ManageHASForm is presented to the user. While the User remains in the ManageTemperatureForm, the real temperature is updated periodically.
ManageDoorBellControl	Manages the door-bell subsystem on the UserStation. It is presented when user selects the ManageLighting option from ManageHASForm. If it is the first time being activated, it presents the default values (State = OFF). It then stores the previously selected state value and waits for the User to switch state (SwitchStateControl), confirm (ConfirmControl) or cancel (cancelControl). The switch state is submitted to HAS, waiting for HAS to acknowledge the information received. If switch state is switched to OFF, it disables the door-bell actuation by the Visitor and, consequently, the DoorBellEvent. If the ConfirmControl is selected, the ManageLightingForm is destroyed and ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected state value is sent to HAS and waits for acknlowdgement. Then, the ManageDoorBellForm is destroyed and ManageHASForm is presented to the user.
ManageUniversalTimeControl	Manages the Date and Time subsystem on the UserStation. It is presented when user selects the

	ManageUniversalTimeControl option from ManageHASForm. If it is the first time being activated, it presents the default values (Year=00, Month=01, Day=01, Hour=00, Min=00). It then stores the previously selected values and waits for the User to modify each of these values, updating them to the User and sending them to HAS. If the ConfirmControl is selected, the ManageUniversalTimeForm is destroyed and ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected values are sent to HAS and waits for acknlowdgement. Then, the ManageDoorBellForm is destroyed and ManageHASForm is presented to the user.
ManageNotificationsControl	Manages the notifications subsystem on the UserStation. It is presented when user selects the ManageNotifications option from ManageHASForm. If it is the first time being activated, it presents the default values (State = OFF, and NotificationTable = empty). It then stores the previously selected state value and waits for the User to switch state (SwitchStateControl), confirm (ConfirmControl) or cancel (cancelControl). The switch state is submitted to HAS, waiting for HAS to acknowledge the information received. If switch state is switched to OFF, it disables all notifications from being appended to the NotificationList; conversely, if switched to ON, when an event is triggered, it appends the notifications to the NotificationList and while the User remains in ManageNotificationsForm, it also updates the NotificationTable. If the ConfirmControl is selected, the ManageLightingForm is destroyed and ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected state value is sent to HAS and waits for acknlowdgement. Then, the ManageNotificationsForm is destroyed and ManageHASForm is presented to the user.
ManageSettingsControl	Manages the system settings on the UserStation, namely related to the User account. It is presented when user selects the ManageSettings option from ManageHASForm. If it is the first time being activated, it presents the default values (User = user45, and New Password = 54resu). It then stores the previously selected values and waits for the User to change user name or change password. Then these values are submitted to HAS, waiting for HAS to acknowledge the information received. If the ConfirmControl is selected, the ManageLightingForm is destroyed and ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected state value is sent to HAS and waits for acknlowdgement. Then, the ManageNotificationsForm is destroyed and ManageHASForm is presented to the user.
ManageAutomateControl	Manages the Automate task on the UserStation. It is presented when user selects the automatic mode in

	ManageLightingForm or ManageTemperatureForm. If it is the first time being activated, it presents the system time for start time and end time with one more minute than that, otherwise it presents the last selected values. It then stores the previously selected values and waits for the User to modify each of these values, updating them to the User and sending them to HAS. If the ConfirmControl is selected, the ManageUniversalTimeForm is destroyed and ManageHASForm is presented to the user. If the CancelControl is selected, the previously selected values are sent to HAS and waits for acknlowdgement. Then, the ManageDoorBellForm is destroyed and ManageHASForm is presented to the user.
LogoutControl	Manages the logout from HAS. If selected, the ManageHASForm is destroyed and the ConnectToHasForm is presented to the user.
BrightnessControl	Manages the brightness value from HAS. If selected, the value presented to the User is updated and sent to HAS, as longs and it remains inside the limit bounds.
TemperatureSetpointControl	Manages the temperature set-point from HAS. If selected, the value presented to the User is updated and sent to HAS, as longs and it remains inside the limit bounds.
StateControl	Manages the state for the Lighting, Temperature, DoorBell and Notifications subsystems from HAS. If selected, the value presented to the User is updated and sent to HAS.
ModeControl	Manages the mode for the Lighting and Temperature subsystems from HAS. If selected, the value presented to the User is updated and sent to HAS.
ModifyControl	Modifies (increments/decrements) the value of Brightness, Set-point, UniversalTime and Start and End Times for AutomateForm. If selected, the value presented to the User is updated and sent to HAS.
ConfirmControl	Manages the flow between forms, destroying the present form and returning to the previous one.
CancelControl	Manages the flow between forms, sending the previously selected values on the present form to HAS and destroying it, returning to the previous one.
HandleConnectionDown	Manages the connection down event. When this event is triggered, it presents to the user the ConnectionDownForm. When the User acknowledges the information, it destroys this form.
HandleMotionDetectedControl	Manages the motion detected event. When this event is triggered, it presents to the user the MotionDetectedForm. When the User acknowledges the information, it destroys this form.
HandleDBButtonPressedControl	Manages the event of door bell button pressed. When this event is triggered, it presents to the user the DoorBellButtonPressedDialog, with the option to accept or decline, and a 30-second timer is initiated, updated every second while the User does not select any option or the timer doesn't run out. If the User selects the option Accept, a command to open the door is sent to HAS ("affirmative" buzzer sound) and

	the DoorBellButtonPressedDialog is destroyed. If the User select the Decline option or the timer runs out, a command is sent to HAS to signal nobody's home ("negative" buzzer sound) and the DoorBellButtonPressedDialog is destroyed

4.2. Sequence Diagrams

A sequence diagram maps the use cases to objects, showing how the behaviour of the formers is distributed by the latters, by modelling the interaction order between objects and assigning responsibilities to each object in the form of a set of operations.

The heuristics for drawing sequence diagrams from use cases are as follows [5]:

- The first column should correspond to the actor who initiated the use case.
- The second column should be a boundary object (that the actor used to initiate the use case).
- The third column should be the control object that manages the rest of the use case.
- Control objects are created by boundary objects initiating use cases.
- Boundary objects are created by control objects.
- Entity objects are accessed by control and boundary objects.
- Entity objects *never* access boundary or control objects; this makes it easier to share entity objects across use cases.

Applying the above heuristics to the main uses cases yielded the sequence diagrams presented from Fig. 13 to Fig. 17. As the software to draw the sequence diagrams has a different notation from the conventional one, a legend for these diagrams is presented in Fig. 12 and should be used to understand the following sequence diagrams.

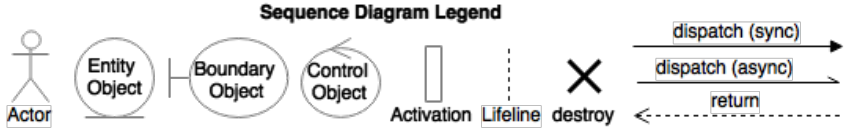


Fig. 12 - Sequence diagram legend

In Fig. 13 is depicted the sequence diagram for the ConnectToHAS use case. It starts with the User pressing the AppButton (the executable) that creates a control object which, in turn, creates the ConnectToHASForm; this is the form the User utilizes to login in HAS. After the User filling in the contents in the form and submitting it, the ManageConnectionControl object creates the User Credentials (an entity object), which it utilizes to verifyCredentials() in HAS. When HAS receives this request, it creates a control object for managing the credentials and checks for a match in the SavedUserCredentials. If the credentials match, this information is sent to HAS, which in turn sends the authorizedToConnect signal. Now, the ManageConnectionControl can try to connect to HAS and if the connection is established, it destroys the ConnectToHASForm and creates the control object that creates the ManageHASForm; this is the form the User will utilize for managing HAS with the connection status being updated to inform the User.

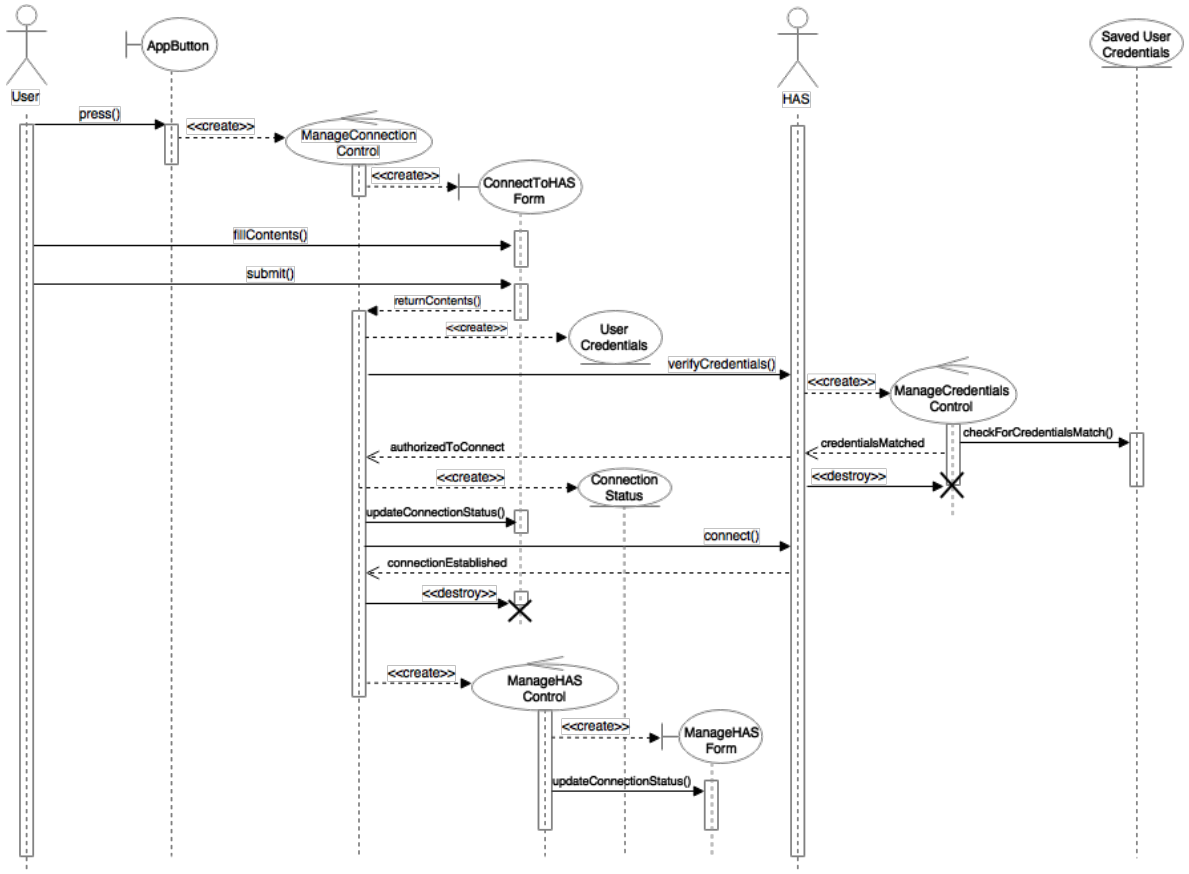


Fig. 13 - Sequence diagram for the ConnectToHAS use case

The HAS management is depicted in the form of sequence diagram in Fig. 14. The User starts, for example, by pressing the ManageLighting option in

ManageHASForm, creating a control object that creates and controls the ManageLightingForm. Next, this control objects creates the lighting parameters (entity object) and requests the HAS stored lighting parameters to fill the form. HAS, in turn, requests these parameters from the LightingManagerControl (created on the system start for managing Lighting in HAS) which accesses the stored lighting parameters. The information is cascaded to HAS, which sends it to ManageLightingControl to updateLightingParams() in the ManageLightingForm.

This could be enough to illustrate the interactions for Lighting subsystem originated from the ManageHAS functionality, as the remainder functionalities are the responsibility of the ManageLighting use case. However, it was also included the pressConfirm() or pressCancel() from the ManageLightingForm as an alternative path, to illustrate that the control returns to ManageHAS which, in turn, becomes available to manage other functionalities (temperature, door-bell, etc.). (Although HAS can manage several services “simultaneously”, the forms for User interaction are categorized by functionality, thus limiting the choices to a unique one and only from the designated places).

Therefore, when the User presses Confirm or Cancel from the ManageLightingForm, the form and control lighting objects will be destroyed. Additionally, if cancelled, the old lighting parameters are sent to HAS, which updates it and rewrites the stored parameters and an acknowledgment is sent to the ManageLightingControl object.

The remaining manageable functionalities by ManageHAS behave similarly and, therefore, were omitted for simplicity of the drawing.

The last functionality is the logout, triggered by pressing the logout option in ManageHASForm(). It then creates a control object to disconnect the User from HAS and, after acknowledgement, it destroys the ManageHASForm, thus closing the application.

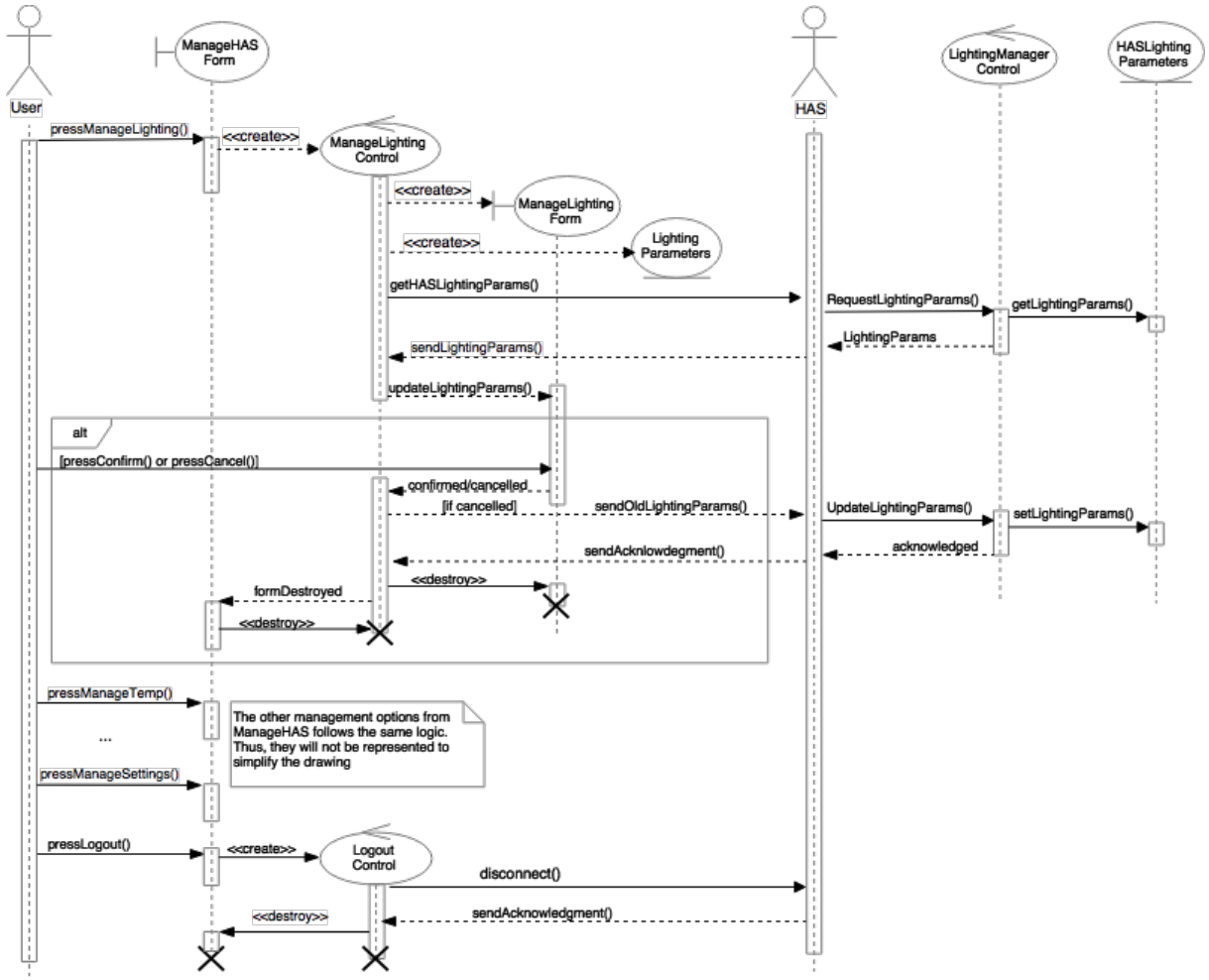


Fig. 14 - Sequence diagram for the use case ManageHAS

In Fig. 15 is depicted the sequence diagram for the use case ManageLighting. This use case was selected as being representative of the three management use cases ManageLighting, ManageTemperature and ManageDoorBell.

The basic interactions are as follows: the User selects one of three parameters - state, mode and value (brightness). The first two are switchable and the last one is modifiable (variable bounded value). The form acknowledges the interaction from the User to the ManageLightingControl object, which then creates the entity object with the parameter and request HAS to update the parameter value. HAS, in turn, forwards the request to LightingManagerControl, the control object responsible for writing the new value in HASLightingParameters. An acknowledgment is then sent to the ManageLightingControl which updates the parameter value in the form.

Two alternatives situations are considered: the first is related to switching the state, resulting in enabling/disabling mode and brightness functionalities; the second one is related to the mode: if the mode selected is the automatic one, the present form is destroyed and a control object is created to create the automate form.

Lastly, for completeness is added the pressConfirm() or pressCancel() option as a placeholder, but not detailed, as it was already in the ManageHAS sequence diagram.

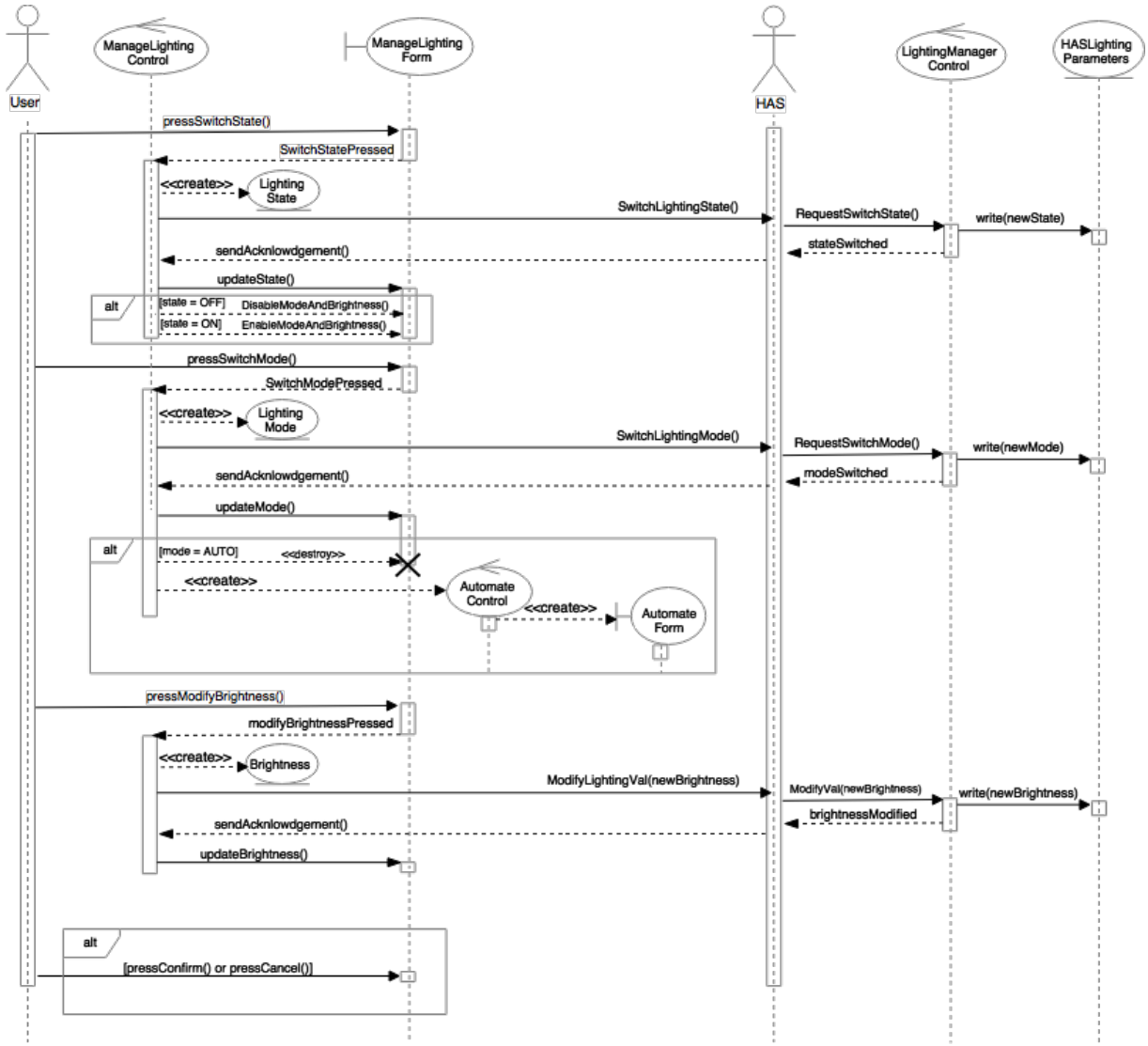


Fig. 15 - Sequence diagram for the use case ManageLighting

In Fig. 16 is depicted the sequence diagram for the use case ManageUniversalTime. Although very similar to the previous one, it was included here to introduce the Universal time control and data structures, as they are additionally used by the ManageSystemNotifications and AutomateTask.

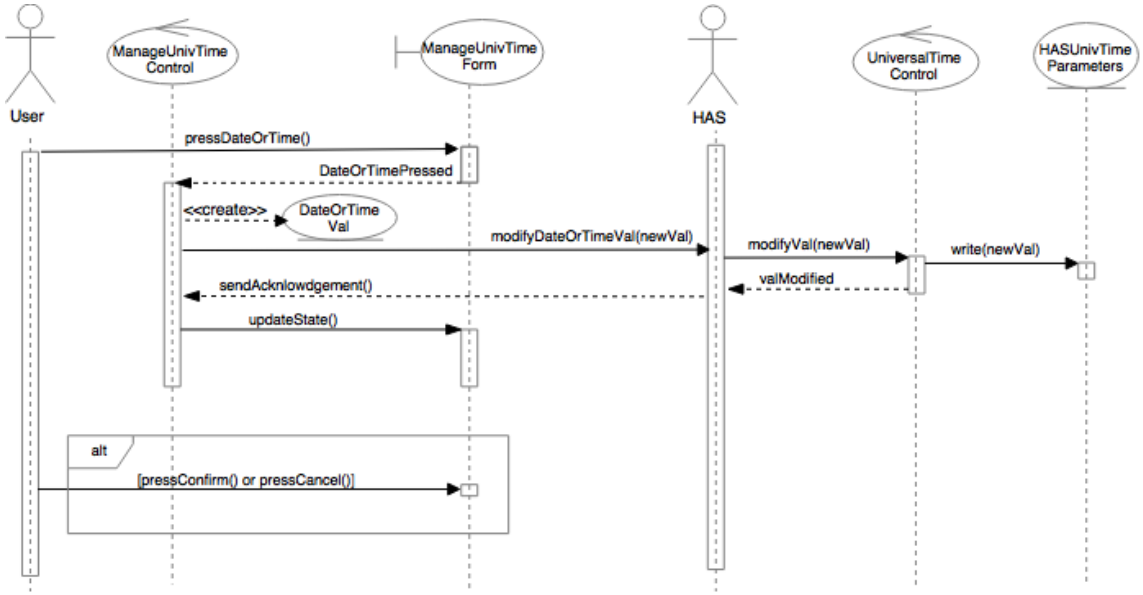


Fig. 16 - Sequence diagram for use case ManageUniversalTime

Next, it is presented the sequence diagram for the use case ManageLighting (Fig. 17). It is also very similar to the two previous ones; the difference resides in the trigger's source (ManageLighting or ManageTemperature), and the control object and data sink (Lighting or Temperature parameters).

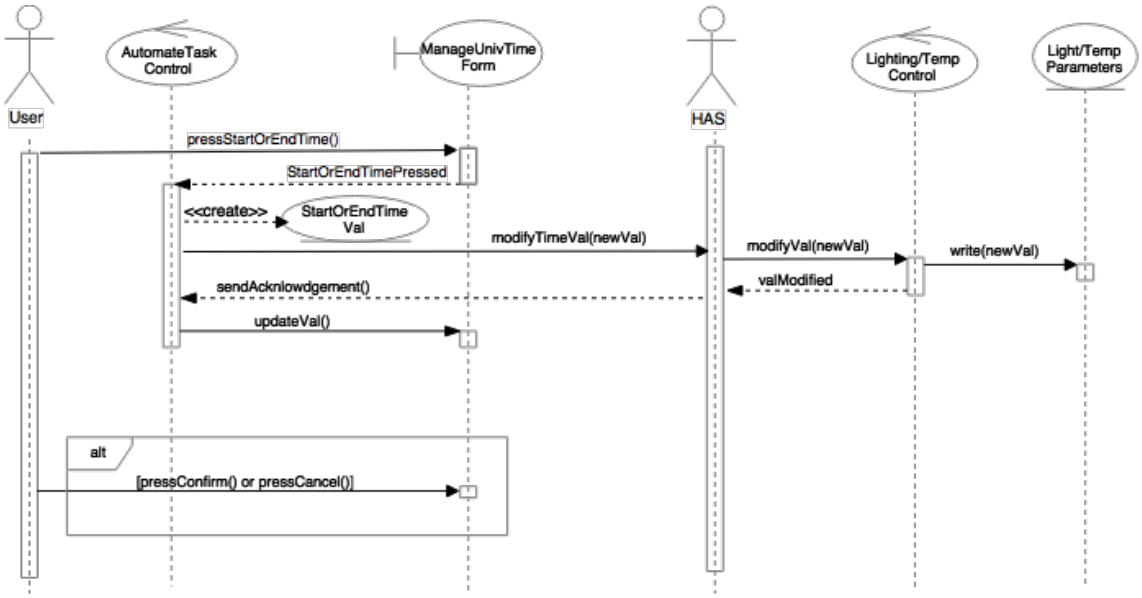


Fig. 17 - Sequence diagram for the use case AutomateTask

Lastly, it is illustrated in Fig. 18 (zoomed-in in Fig. 71) the sequence diagram for the use case DoorBellButtonPressed. This is one of the uses cases that can be initiated by the Visitor. It starts with the door-bell being pressed, which can happen at any time, being represented as an asynchronous event, as indicated by the arrow. Then, control and entity objects are created to signal the event to HAS. HAS, in turn, request the DoorBellControl to play the door-bell tone, which this object retrieves from the DoorBellParameters.

Then, HAS request the UniversalTime from UniversalTimeControl, which creates the appropriate data and returns it to HAS. Now, HAS creates an event with the type and time-stamp associated, which then broadcasts virtually simultaneously to both NotificationsControl, DoorBellControl and ManageDoorBellForm; the former simply appends the event information to the SystemNotifications data; the latter is responsible for managing and presenting the form to the user. It should be noted that this object is not created by HAS, but by the remote application, but to reduce the drawing complexity it was not added. For this reason, it should also be noted that the DoorBellControl and ManageDBForm operate asynchronously, synchronized when necessary by HAS.

The DoorBellControl disables the door bell button to prevent further door-bell rings and notifications and the ManageDBForm creates the Form Notification and presents to the User with the event, which can choose to accept or decline the

entrance in the house. Additionally, it also starts a timer, that after running out (User did not reply), should indicate nobody is home. Here, it was used the loop construct to represent that while the user does not reply, the timer should be incremented and shown to the user.

If the User replied (Yes or No), the form is closed and the associated tone should be fetched from the door bell parameters and played. Otherwise, when the timer runs out, the form is closed and a No answer is sent to HAS which requests the door-bell to play the associated sound (fetched in the same way). Lastly, the door-bell is enabled, allowing further door-bell rings and notifications (this will happen until the User disables the door-bell completely switching the door bell state to OFF).

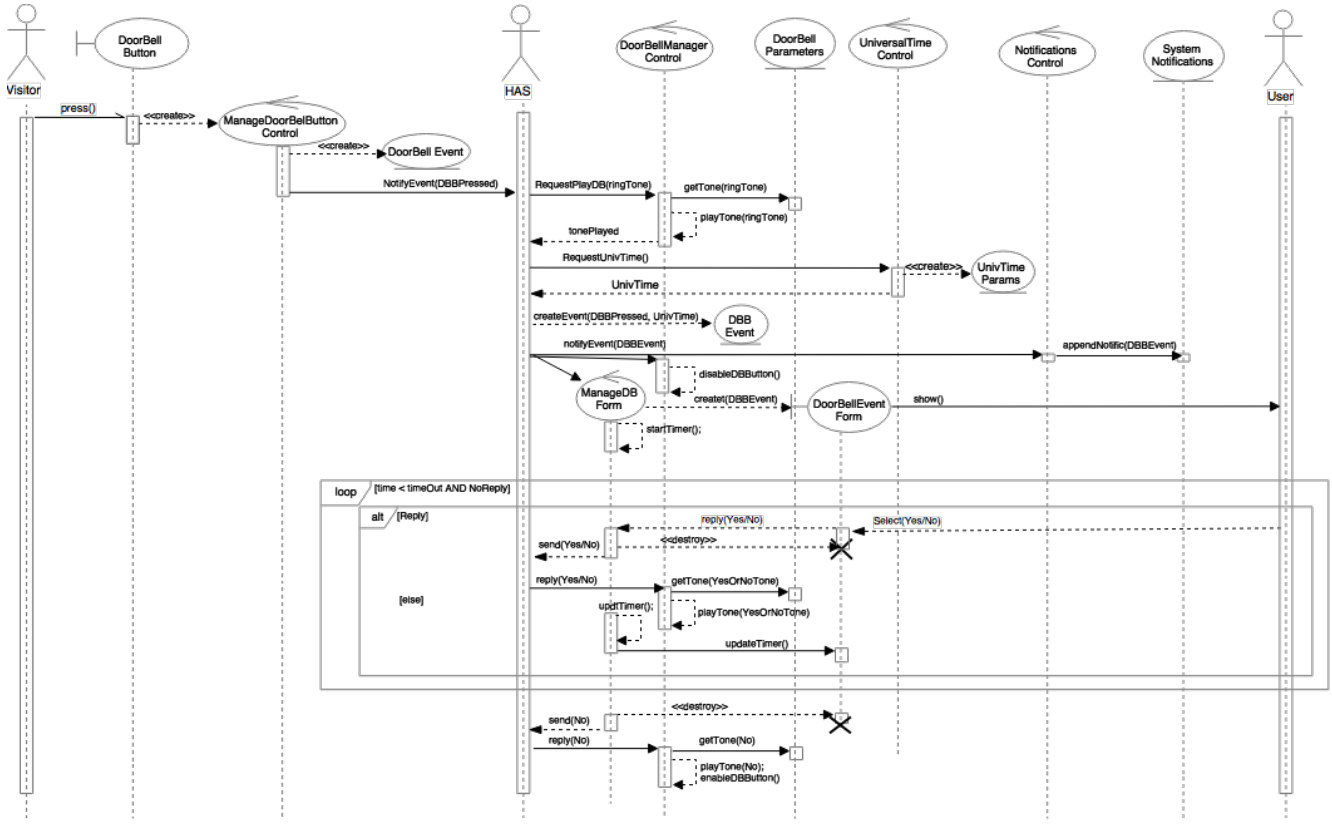


Fig. 18 - Sequence diagram for the use case PressDoorBellButton (zoomed-in in Fig. 71)

This sequence diagram is representative of: the other event triggered by the Visitor (DetectedMotion), which only informs the User and waits for a reply to close the form in the remote application; and the LogSystemMessage use case as it demonstrates the event creation, with retrieval of the time-stamp from

UniversalTimeControl, and its sending to the NotificationsControl which appends it to the SystemNotifications data.

4.3. Static Architecture

While the sequence diagrams allow to represent the interaction between objects along the time, the class diagrams allow to represent the interdependency between them, using associations and attributes. An association represents a relation between two or more classes with several properties: name, role and multiplicity. An attribute is a property of an individual object with a name and a type. With these concepts in mind, the class diagram for HAS was conceptualized, presented in Fig. 19. This model is an initial representation of the system and not a complete system specification, to be used as a design guide.

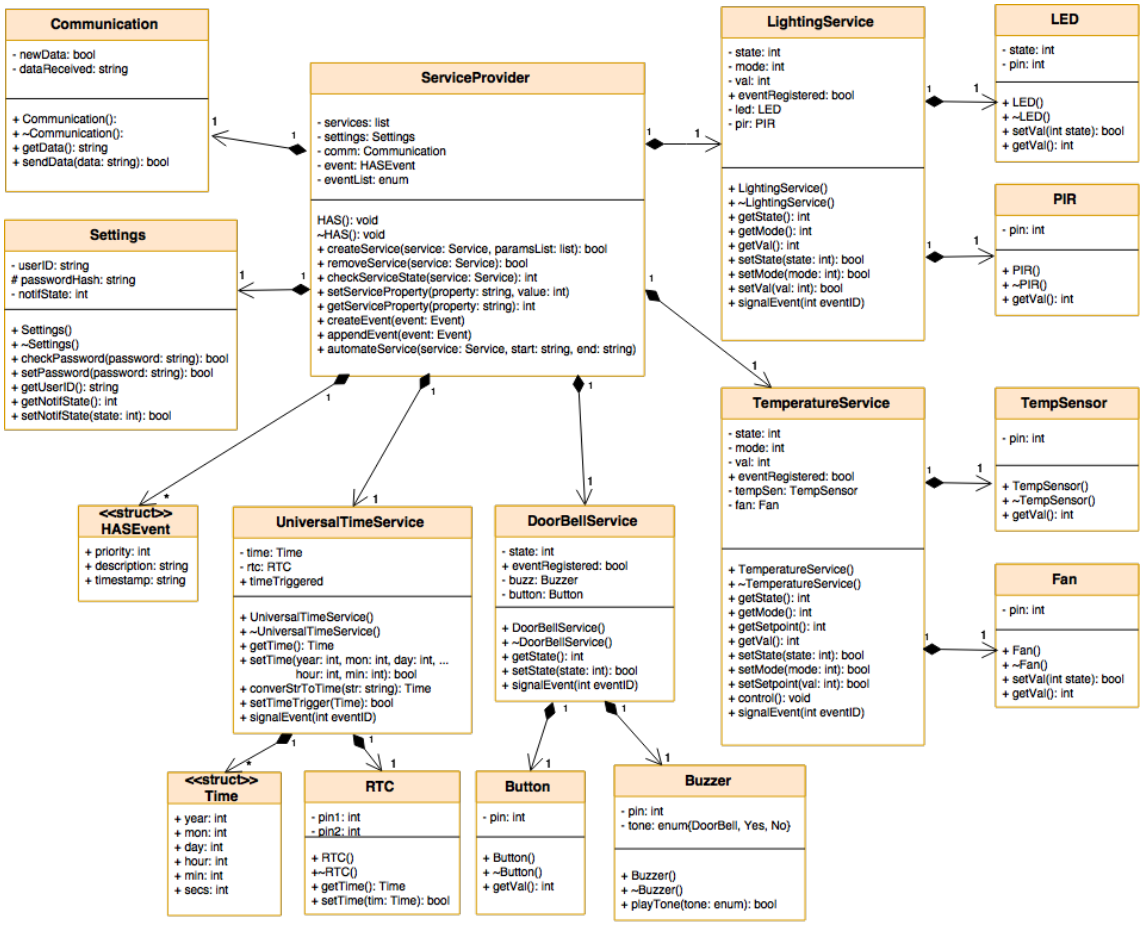


Fig. 19 - Static Architecture for HAS

By analysing the diagram, a central entity can be identified – the ServiceProvider. HAS, as a system, can be seen as a Server (portmanteau of service provider) as it aims to provide services to the User. Therefore, the ServiceProvider aggregates several services, namely: lighting, temperature, door-bell and universal time. However, to cope efficiently with its task, ServiceProvider needs to aggregate other functionalities, namely: the ability to communicate with the outside-world, otherwise, it would not have no one to serve; the settings, as it defines the user configurations that help to determine the behaviour of HAS or the information that it should make available at the its initialization or permanent data like the notifications or the user's password and how to access it; an event (HASEvent) structure to define the data that it should use, as ServiceProvider is the event handler, responsible for effective notification of priority events to the user and its feedback (if available), or the append of the least priority ones to the system notifications.

As a service provider, the ServiceProvider class should manage the services, creating or removing them, and querying or setting the properties, and keeping a list of the running services. From the abstraction perspective required in the analysis phase, this seems a good approach and could result, for instance, in the creation of an abstract class named Service, from which all services can inherit. This was done only partially, because, although UniversalTime is a service, it does not share a lot of attributes and behaviour with the Lighting and Temperature services; the door-bell also, but at a lesser extent, thus being considered as inheriting from service.

The services contain further aggregations that, although not necessary in this phase, aims to increase the completeness of the model, including the physical subsystem that are effective controlled by each service. The services are responsible for the proper operation of the service and for signalling an event to ServiceProvider which will handle it accordingly.

However, services are not responsible for the automation, ServiceProvider is, for the same reason mentioned before: ServiceProvider is the service provider and, as the Lighting and Temperature services requires the Universal Time service to send a trigger signal, HAS will handle that for them. Thus, HAS will automate the service and set a time trigger, which the UniversalTimeService will signal when occurred. As it can also been seen, AutomateService(...,...,...) uses strings to pass the time to the UniversalTimeService, requiring the latter to have a utility function to convert it

to a Time structure. This was made to minimize the access the dependencies and data access, as only UniversalTimeService should access the Time structure.

Lastly, it is also important the mention two aspects of aggregations: lifespan and multiplicity. The former refers to the proper term of aggregations, meaning that, in the present case, no object should outlive its creator; the latter says that all aggregator objects relate to no more than one instance of the aggregated objects, except for the two structures present: Time and HASEvent, as HAS can have multiple events and UniversalTime can handle several Time structures.

To finalize the discussion of the static architecture the inheritance relationships between classes were identified, based on shared attributes or behaviour, illustrated in Fig. 20. The LightingService, TempService and DoorBellService inherit from the superclass Service for the aforementioned reasons. Button, PIR and TempSensor are all specific types of sensor, reading some value from the outside world; Led, Fan and Buzzer are all specific types of actuators, being driven by a signal written on them.

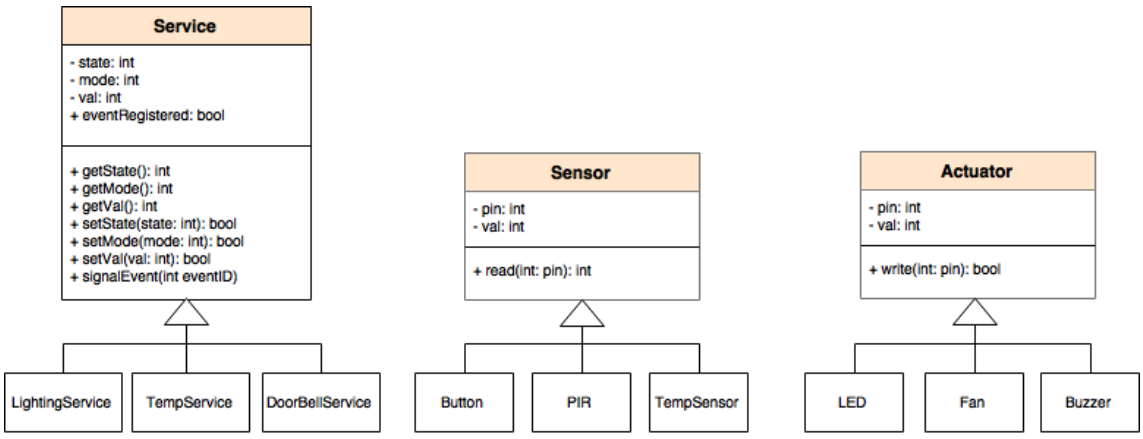


Fig. 20 - Inheritance relationships identified in the analysis model (UML class diagram)

4.4. Object Behaviour: State-Machine diagrams

The state-machine diagrams represent the behaviour from the perspective of a single object, enabling a more formal description and, consequently, identifying missing use cases. In the present section it will be described the behaviour of the main objects identified in the analysis stage in the form of state-machine diagrams.

A foreword is needed: the representation of the behaviour of the main objects is formalized using a Behavioural state machine, as described in the UML 2.5. specification, which an object-based variant of Harel statecharts, specifying the discrete behaviour of a part of the system through finite state transitions [25].

The notation should also be explained. A legend for the Behavioural State-Machine diagrams is presented in Fig. 21. In these type of state-machines behaviour is modelled as a traversal of a graph of state nodes connected with transitions. Transitions are triggered by the dispatching of series of events or the validation of some conditions (guards). During the traversal, the state machine could also execute some activities.

A state can have several activity labels, namely [25]:

- **entry** – behaviour performed upon entry to State (entry behaviour)
- **do** – ongoing behaviour that is performed as long as the modelled element is in the State or until the computation specified by the expression is completed (doActivity behaviour)
- **exit** – behaviour performed upon exit from the state (exit behaviour)

A **composite state** is defined as state that has substates (nested states). Substates could be sequential (disjoint) or concurrent (orthogonal). The UML specification defines composite state as the state which contains one or more regions, represented by the dashed lines. It can be seen that these concepts can be used to modelled concurrency and parallelism. As it lends itself useful, the concurrency is presented here, illustrated by the fork and join *pseudostates*. All disjoining states are activated at the fork if all transitions and conditions are validated; similarly, all conjoining states will only be activated if all previous states are active and the transition conditions validated. This notion is very important to understand the following diagrams.

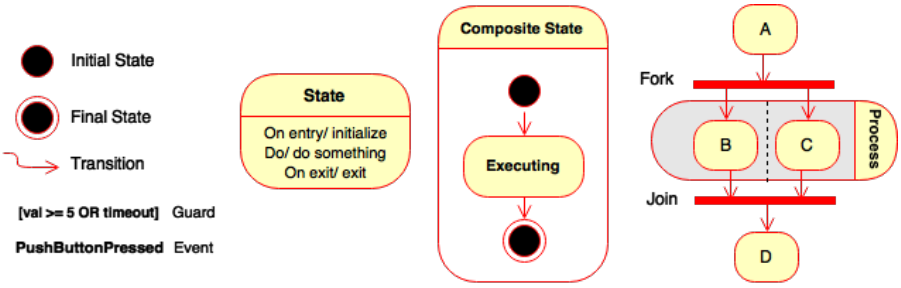


Fig. 21 - Legend for the Behavioural State-Machine diagrams

As the transitions are triggered by events (and validated guards), an event table was generated (Table 5), containing the main events identified, the system response, the source and its type. All events are generated asynchronously, meaning that they can happen at any given point in time without the system being particular aware of them.

Table 5 - Local system main events

Event	System response	Source	Type
Receive command from user	Handle command and acknowledge	Remote application	Asynchronous
Visitor pressed the door-bell button	Notify User and prompt him to open the door	Local system (Door-bell subsystem)	Asynchronous
Motion detected on the house exterior	Notify User	Local system (Door-bell subsystem)	Asynchronous
Automate task start/end	Signal the event to the service provider that forwards it to the appropriate service to start/end the task and append the notification to system's notifications	Local system (Universal Time subsystem)	Asynchronous
Connection down	Restart the communication channel	Local system (communication subsystem)	Asynchronous
Door-bell response timeout	Notify Visitor that nobody is at home by playing a "negative" sound on the door-bell	Local system	Asynchronous
General timeout	Stop the execution of a software component to prevent the system from blocking	Various	Asynchronous

In Fig. 22 is illustrated the global behaviour of the system. Although not necessary, it was generated as a by-product when trying to understand the communication between different objects and its interactions, and is very useful as global perspective of behaviour of the system as a whole.

The initial state of the system is the system starting up, with the system transitioning to the initialization if effectively the power-up signal is stable. On initialization entry, the system loads the settings, creates the needed services (the default ones and the ones based on the user's settings and finally creates a communication channel to the outside-world. The system will not transition immediately to the next states because, although the condition initializationCompleted is validated, only when all previous states are active, the

transition will be triggered. This is a wait-state for the creation of services that were triggered in the initialization by the createServ guard, and only when all services are created, the StartupServs flag is enabled and the system can evolve.

Then, the system execution will fork into three composites states, corresponding to the main classes of the system: Communication, ServiceProvider and Service. The reason behind this is, that HAS should only be concerned with providing services, leaving the communication and service operation to the appropriate process.

The ServiceProvider and Communication processes will transition to the Execution macro state, representing its inner working and will only exit if the system is powered off. The Service process, on the other hand, will do its work only when required by HAS to create the service, otherwise the service is allowed to constitute an independent process that will be accessed and removed in the executing macro state. The power-off signal will conduct the system to the shutdown state.

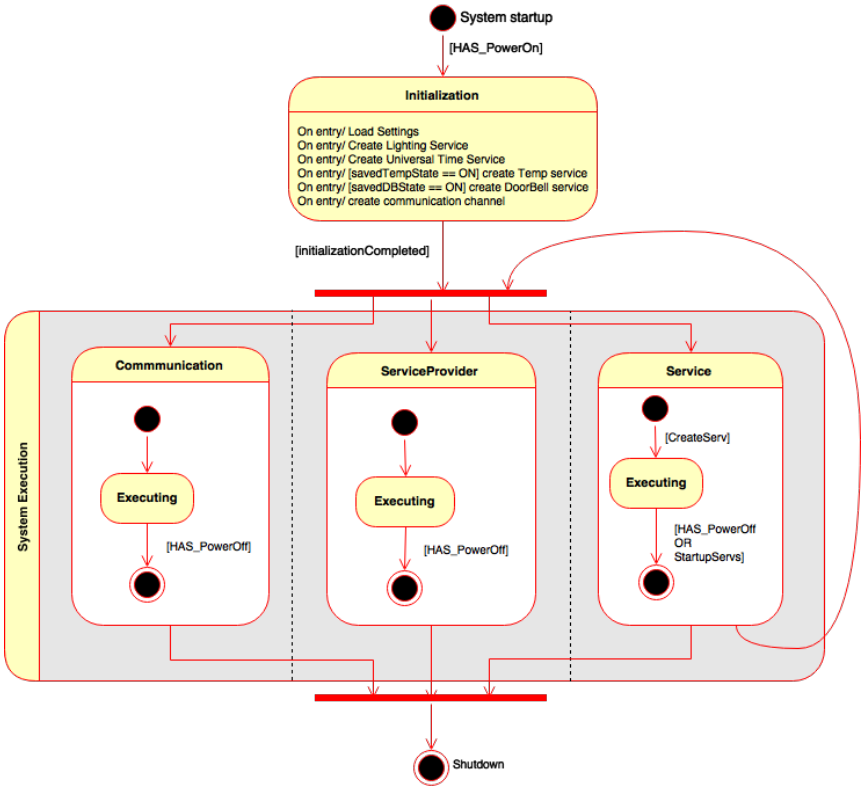


Fig. 22 - Global behaviour of the system

The communication process is composed of the following states: wait, connRequested, connected, readyToCommunicate, notCommunicating, UserMsgReceived and UserMsgToSend. The **wait** state sleeps (Do/ sleep), and when receives a connection (User connects) tries to connect (On exit/ connect). If the connection is accepted, it transitions to the **connected** state, otherwise returns to **wait.** When **connected** is entered it tries to authenticate the user by setting a flag visible to ServiceProvider, and if the credentials match, the communication process should receive a signal from ServiceProvider confirming this (authenticated guard).

Then, the process evolves to the **readyToCommunicate** state that sleeps. It should be triggered by the transitions relative to logout from the user, communication system down, message received from User and a message from ServiceProvider to send to the User. The **NotCommunicating** is blocking, meaning that if no communication is established, the process will keep on trying to restart the communication channel, blocking all execution, corresponding to the real situation, as the communication cannot work without a working communication channel.

The guards in the **UserMsgReceived** and **UserMsgToSend** are synchronizations messages between this process and HAS, allowing for the interchange of data and signalling that attention is required only when effectively needed.

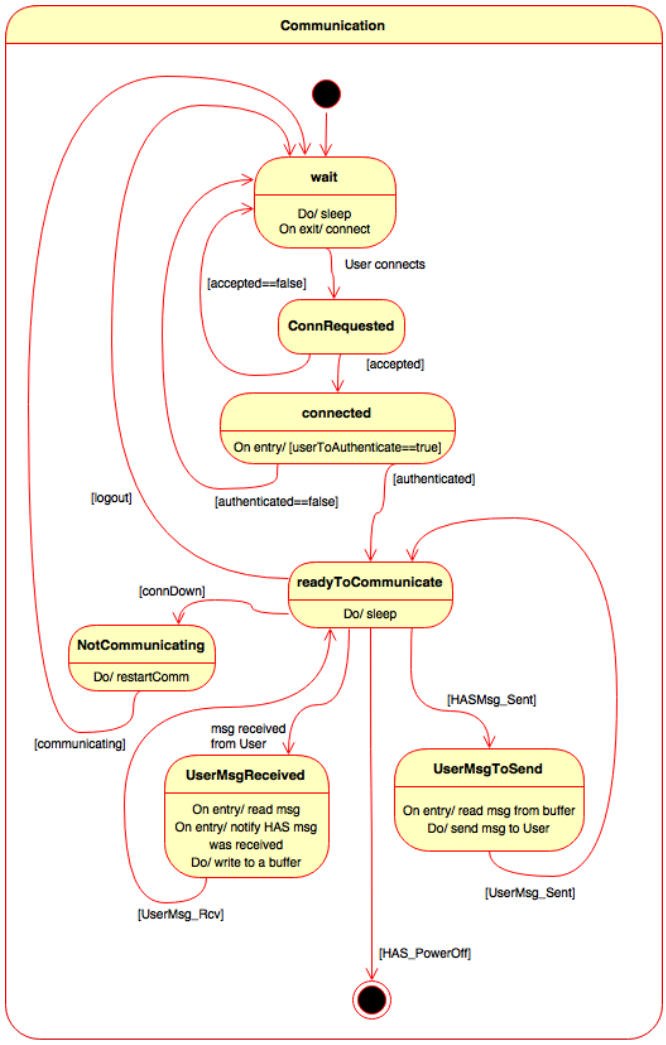


Fig. 23 - State-machine diagram for the Communication Class

The state-machine diagram for the ServiceProvider class is presented in Fig. 24. When evolving from the fork, it transitions to the **idle** state, meaning that it is ready to provide services. When a signal is received from the communication process indicating that the User has sent a message to ServiceProvider, the message is handled: if not valid the message is delivered to the communication process to send it to the User, otherwise it will go a dummy-state, the **MsgParsed** state, where it is delegated to the appropriate handler, based on the message contents: if it is a service request, it will be forwarded to the appropriate service to handle, and when done, feedback will be received from the Service process and it will be acknowledge to ServiceProvider; otherwise, if it a automate request, the parameters will be obtained and a UnivTimeService will be requested to retrieve time and the time

trigger will be set, so this can warn ServiceProvider to signal the appropriate service to wake up and do its job.

The remaining states refer to event handling. Both internal and external events are uncategorized, as they require the time-stamp that should be requested by ServiceProvider to the UnivTimeService. After being categorized its decided, based on its priority to notify the User (**Priority**) or simply append the event to the system notifications (**NoPriority**). Lastly, when a notification is requested it should be signalled to the communication to send the event information to the user.

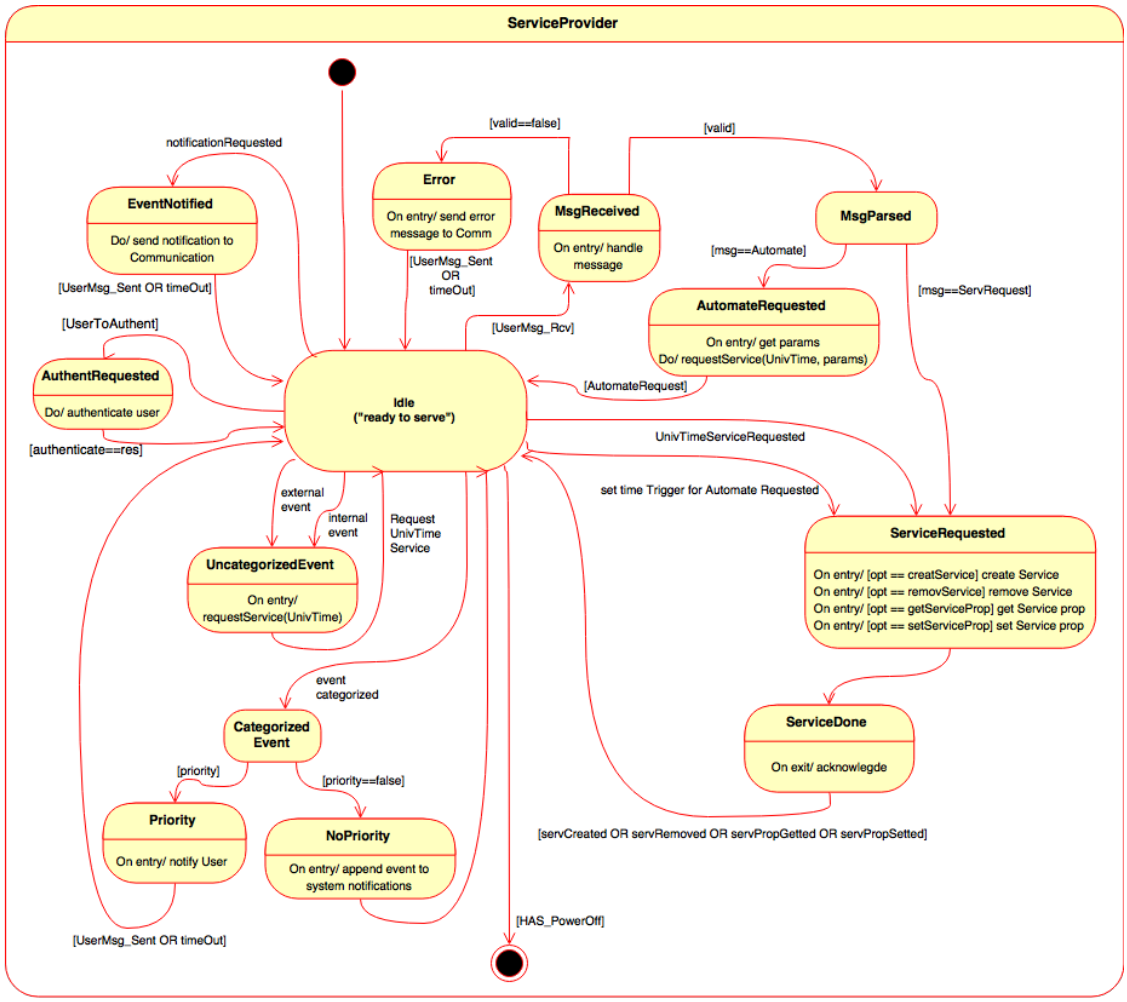


Fig. 24 - State-machine diagram for the ServiceProvider Class

The last state-machine diagram refers to the **Service** class and is depicted in Fig. 25. The service states follow the same logic as a process (in software terms). The service is created after being required by ServiceProvider or in the system start-

up, signalling a flag to ServiceProvider in case this was the request made. The **ready**, **running** and **waiting** state refer to the normal cycle of operations to access the processor. Connected to the **running** state, it is the **halted** state, where the ServiceProvider service should try to get or set the properties from the process, as the **running** state is the only accessible by the application, with the remaining two states being reserved for the processor access. As such, in the **halted** state, the actual service execution is halted by setting the appropriate flags to Service and freed by setting those appropriate to ServiceProvider. Lastly, the process is removed from execution and a flag is set to signal ServiceProvider to remove it from the list of currently available services.

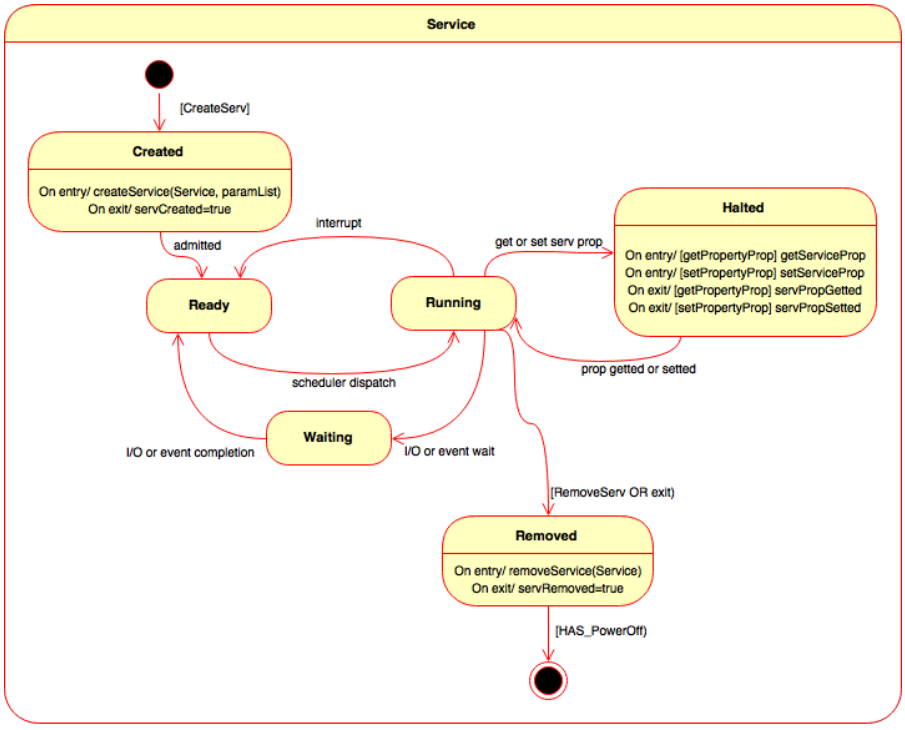


Fig. 25 - State-machine diagram for the Service Class

5. SYSTEM AND SOFTWARE DESIGN

System design is the transformation of the analysis model, representing the application domain, into a system design model, corresponding to the solution domain. During system design, the design goals of the project are defined and the system is decomposed into smaller subsystems that can be realized by individual teams. Furthermore, during requirements elicitation and analysis the focus was on the system functionality, whereas, in system design, the focus is on the processes, data structures, and software and hardware components necessary to implement it. The result of system design is a model that includes a subsystem decomposition and a clear description of each of these strategies [5].

The system design activities, used as a guideline in this chapter and illustrated in Fig. 26, are [5]:

- **Design goals definition**: Identifies the qualities that the system should focus on. Can be inferred from the non-functional requirements or from the application domain, or elicited from the client. The main criteria groups are: performance, dependability, cost, maintenance and end user.
- **Subsystem decomposition and definition**: Aims to reduce the complexity of the solution domain by decomposing the system into smaller parts, called subsystems, composed of a number of solution domain classes, encapsulating their state and behaviour. Typically, the subsystem decomposition is represented in a UML component or class diagrams, in the latter with subsystems represented as packages.
- **Hardware/software mapping**: Maps the functionality associated to a software component to the hardware (node) responsible for it. It is represented by a UML deployment diagram.
- **Persistent data management:** Determines the access, manipulation and storage of persistent data in the system, i.e., data that must "survive" the system's shutdown.
- **Access control policies definition**: Determines which entities can access and manipulate a given set of data.
- **Global control flow selection**: Determines the sequence of operations in the system, deciding which of those should be executed and in which order,

based on external events generated by an actor or on the passage of time. Three main types of control flow can be identified: procedure-driven, event-driven and threads.

- **Boundary conditions description**: Determines how the system should behave in the presence of limit conditions, namely: system initialization and shutdown, and exceptions. Exceptions are events or errors that occurs that occur during the execution of the system and can be caused by three different sources: hardware failure, software fault, and change in the operating environment.

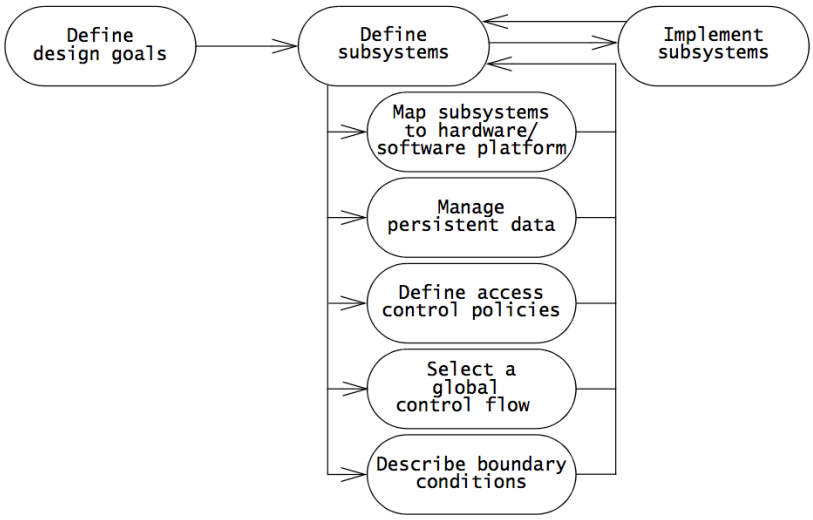


Fig. 26 - Activities of system design (UML activity diagram - withdrawn from [5])

5.1. Design goals

From the aforementioned groups of design criteria, only the first two, performance and dependability will be directly addressed, as the remaining three, cost (except hardware cost), maintenance, and end user, cannot be realistically estimated through the course of the present work. However, these groups of criteria are very relevant for the design orientation, specially the cost and end user, as these are the most visible aspects, namely: if it is cheap enough, so people are able to purchase it (and the producer still profits), and if the user likes the product, so he/she would consider purchasing it. It is also important to consider that in the design stage there is always a trade-off between the different set of criteria in order

to make the system feasible, as highlighted by the following software engineering aphorism: *"Good, fast, cheap. Pick any two".*

The three most relevant performance design criterion to consider are [5]:

- **Response time**: *How soon is a user request acknowledged after the request has been issued?* The so called input lag as several definitions and metrics depending on the type of user, like a video game player where the most higher frame-rate the better, to typical software applications like smart-phone apps. It is also important to note that most users are becoming more and more accustomed to faster response times, enabled by increasingly faster hardware, meaning that they are becoming less tolerant for lagging. A classic range of values, the "normal" user can tolerate is from the 50-200ms. However, in realistic terms, the range will be extended, as mentioned in the use cases description in the requirements elicitation to 1-2 seconds.
- **Throughput**: *How many tasks can the system accomplish in a fixed period of time?* To answer this question, it is important to refer the system's objective: the automation of a home comprised of 4 subsystems (lighting, temperature, door-bell and time). Thus, in any given time the user should be able to manage, directly or indirectly, at least these 4 subsystems, that should be able to coexist "simultaneously". Referring to the analysis classes identified earlier and recognizing that the system needs to communicate and be managed, one can add more two tasks. Hence, the system should be able to handle 6 tasks in the response time established earlier (1-2 secs), so, that from the user's perspectives, its requests are processed seamlessly.
- **Memory***: How much space is required for the system to run? Is memory space available for speed optimizations, or should memory be used sparingly?* This is a very important question, related to the trade-off between execution speed and available space. As the most restrictive requirement is the response time, and taking into account that the hardware platform selected uses an 8GB SD card, it is safe to try to optimize for execution speed, instead of memory size.

Related to the dependability criteria, two were selected as being the most relevant for the project, namely:

- **Robustness**: *Ability to survive invalid user input.* As aforementioned, the user must utilize a remote application to access and manage the home automation system. The commands issued on the remote application should translate to "messages" being passed to HAS for processing and actuation. Therefore, it should a shared responsibility between the remote application and HAS to determine the valid messages for the system, however, it is the ultimate responsibility of HAS, to validate those messages, meaning that if no valid message is sent to HAS, it should not trigger any unintentional action. Furthermore, it also important to notice that user input is normally conditioned by the user interface (UI), providing an excellent way of implementing these messages.
- **Security**: *Ability to withstand malicious attacks.* It is undeniably a very important topic to address specially in interned connected system, regarding specifically unintended usage of the system by third-party people and data security, namely personal information, like the credentials to access the system, and the data conveyed between remote application and HAS. Although this is not a primary focus of this project, it is important to identify good-practices to address this issue, namely: communication data encryption, meaning that, from the time the data leaves the remote application to the time it arrives its receiver, it travels encoded with a different cypher that should only be cracked by someone holding the cypher-key; encrypted user password: the user password is stored encrypted and the user password submission in the authentication process should be encrypted and compared to the one stored. In other words, the user password should not be stored in plain text, as it is the case for the wpa-supplicant configuration file in Raspberry Pi. This would dictate another two news class in the system: one for communication encryption and other for authentication.

5.2. Subsystem decomposition

The subsystem decomposition for HAS is illustrated in Fig. 27. The subsystems, composed of a number of solution domain classes, encapsulating their state and behaviour are represented as packages in the UML class diagram, with the dashed arrows representing the dependency between subsystems. The subsystems

identified are the service manager and depending on them are all the remaining ones: communication and every service (temperature, lighting, universal time and door-bell). There are some considerations to be made:

- the CommSubsystem includes a dashed class, Message, which could in the future be used for correct message handling, however, now this is not the case;
- the ServiceManagerSubsystem includes 3 classes: ServiceProvider, Settings and Event, with weak cohesion as they are not strongly related functionality-wise. This was used to condense classes as the Settings and Event ones are small in functionality.
- all peripheral subsystems are highly-dependent on the ServiceManagerSubsystem, represent high coupling.
- Ideal subsystem decomposition should minimize coupling and maximize cohesion; however, it is intentional, as it will be explained further on, when we explain the pipe-filter software pattern.

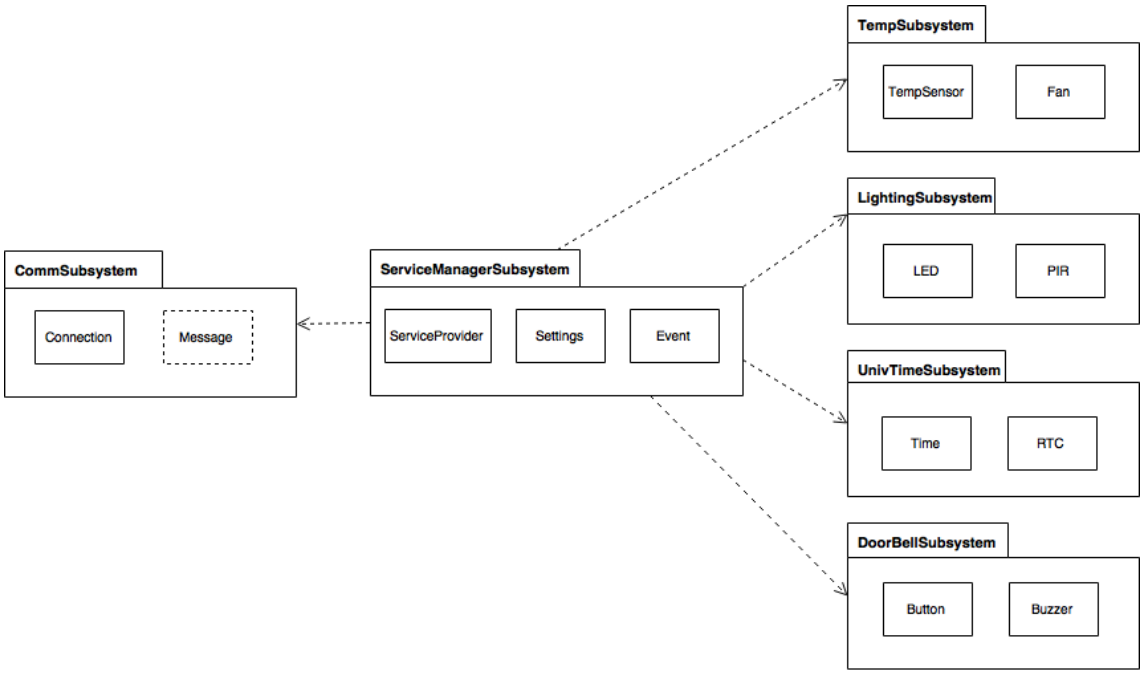


Fig. 27 - Subsystem decomposition of HAS (UML class diagram): subsystems are represented as packages and the dashed arrows indicate the dependencies between them

By looking at Fig. 27, it can be also seen that all the subsystems at the right represent some type of service as seen by the ServiceProvider (not confuse with the Service class). As such, these subsystems were packed in a major subsystem, called ServiceSubsystem, providing a standard interface for requests and responses between the ServiceProviderSubsystem and the ServiceSubsystem, as illustrated in Fig. 28. An interface is a set of operations of a subsystem that are available to other subsystems and is represented by the ball-and-socket connector, with the ball representing a provided interface and the socket a required interface. Thus, it is easily understandable by looking at the figure that the ServiceManagerSubsystem requires a ServiceRequest to the ServiceSubsystem, which the latter provides. The same holds true for the ServiceResponse. By doing this, the system coupling is reduced, as desired.

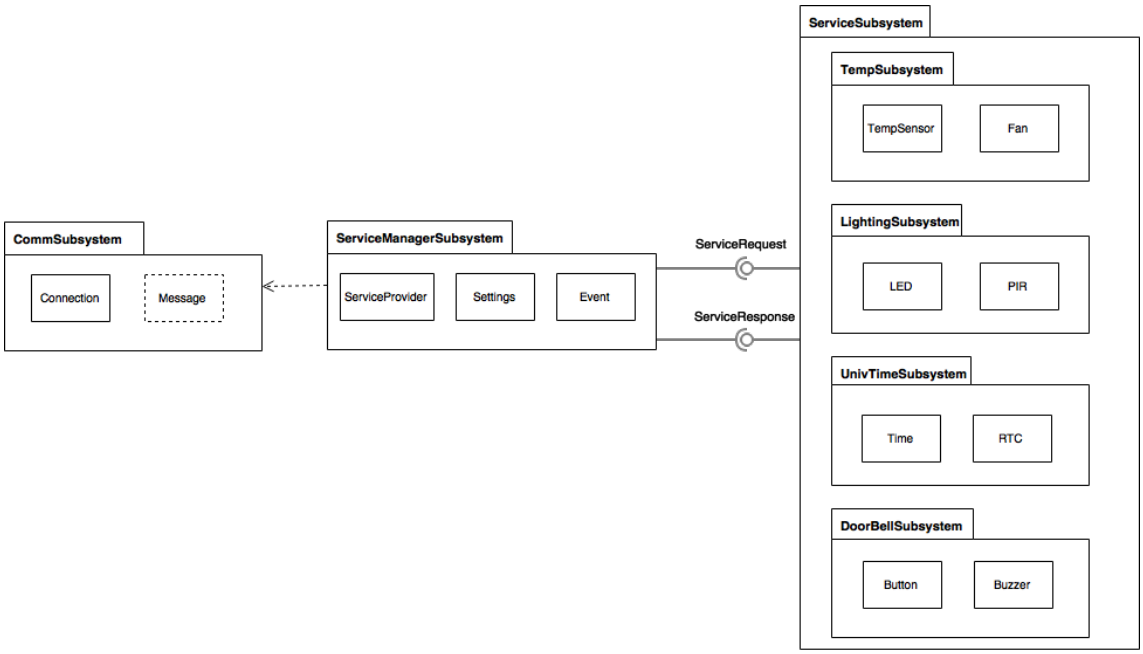


Fig. 28 - Subsystem decomposition of HAS (UML class diagram) improved with interfaces: provided interface is represented with a ball and a required interface is represented with a socket.

5.3. Hardware/software mapping

In the hardware/software mapping activity, the functionality associated to a software component is mapped to the hardware responsible for executing it. It is represented by a UML deployment diagram used to depict the relationship between run-time components and nodes. Components are self-contained entities that

provide services to other components or actors. The HASApplication and the TCP/IP Server, in Fig. 29, are two examples, represented by a component icon. A node is a physical device or an execution environment in which components are executed, such as a desktop computer, or myMac in Fig. 29, represented by boxes containing component icons. Furthermore, a node can contain another node, for example a device can contain an execution environment such as virtual machine.

The UML deployment diagram for HAS is depicted in Fig. 29, showing that a HASApplication can access, through a TCP/IP client, a TCP/IP server that provides information about the services managed by the HASServiceManager

. On the left-side it can be seen the two software components, HASApplication and TCP/IP server, being executed in the device myMac. On the right-side it can be seen the three software components - TCP/IP server, HASServiceManager, and Service - being executed on the myRaspberryPi device. This is not entirely accurate, as the Service here referred is a generic one from the ones being provided; in fact, there should be at least 2 and a maximum of 4 specific instances of services being executed. This was not added for generalization and visual clarity purposes.

The software components interact through the provided interfaces:

1. between the HASApplication and the TCP/IP client, and the TCP/IP Server and HASServiceManager through bidirectional message passing (not explicited yet);
2. between the TCP/IP client and the TCP/IP server through the TCP/IP protocol, bidirectional also (accurately the interface is provided the socket API);
3. between HASServiceManager through ServiceRequest and ServiceResponse interfaces (it is not defined yet; was used for consistency).

An interesting aspect of this particular diagram (Fig. 29) is that is enables the visualization of some software design patterns, namely: client/server for the TCP/IP client and server and pipe/filter for ServiceManager and Services, that could be extended for all message passing (mentioned in the point 1). Additionally, user interfaces typically follow the Model-View-Controller (MVC) design pattern [5]. The advantage of using software design patterns is that they are tested proven solutions for standardized sets of problems, paving the way for successful implementation.

The HASApplication will not be the primary focus of this project, hence, the MVC design pattern will not be detailed.

Relating to the client/server architectural style, a subsystem, the server, provides services to instances of other subsystems called the clients, which are responsible for interacting with the user. This architectural style is well suited for distributed systems that manage large amounts of data.

In the pipe and filter architectural style, subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs. The subsystems are called “filters” and the associations between the subsystems are called “pipes. Each filter knows only the content and the format of the data received on the input pipes, not the filters that produced them. Each filter is executed concurrently, and synchronization is accomplished via the pipes [5]. The most well known example of a pipe and filter architectural style is the Unix Shell. Pipe and filter styles are suited for systems that apply transformations to stream of data without user intervention, as is the case for the ServiceManager and Service communication. However, they are not suited for systems that require more complex interactions between components, such as an information management system or an interactive system [5].

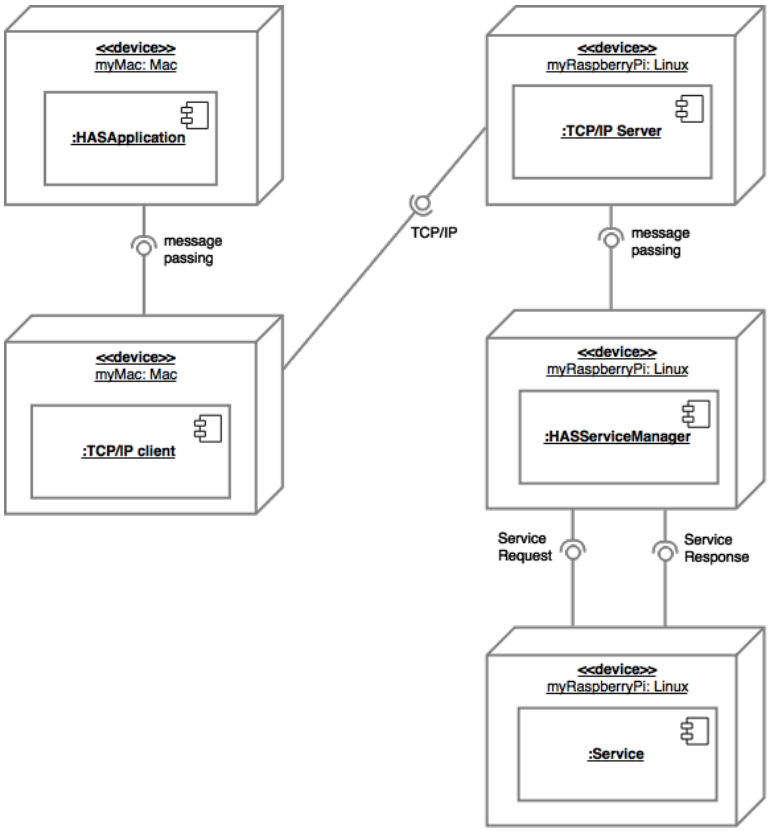


Fig. 29 - UML deployment diagram representing the allocation of components to different nodes. An HASApplication can access a TCP/IP server that provides information about the services managed by the HASServiceManager

5.4. Persistent data management

Persistent data outlives a single execution of the system, surviving the system shutdown and being readily available at system start-up. This is the case for the user settings, system notifications and services configuration. The latter refers to the case when the user started a temperature service with a 23ºC set-point; if the system is powered off and then turned on, it should restart the service with that set-point instead of the 25ºC default value.

There are, generally, three options for storage management:

- **Flat files**: Files are the storage abstractions provided by operating systems. The application stores its data as a sequence of bytes and defines how and when data should be retrieved. Files allow to perform a variety of size and speed optimizations, however, they required the application to address some issues, like concurrent access and loss of data in case of system crash.

- **Relational database:** Provides data abstraction at a higher level than flat files. Data are stored in tables that comply with a predefined type called schema. Although scalable and ideal for large data sets, they are relatively slow for small data sets and for unstructured data.
- **Object-oriented database:** similar to a relational database, but it stores data as objects and associations. It significantly reduces the time for the initial development of the storage system, however, they are slower than relational databases and more difficult to tune.

Taking into consideration that the data to be stored is relatively small and supported by the Unix motto that everything is a file (and what isn't, is a process), the data will be stored in flat files.

Additionally, it could be questioned the notifications case, but as no type of sorting in necessary (sorting is already done in a time-basis) or queries, flat files will suffice. Otherwise, another interesting solution related to the data representation could emerge, using a type of data interchange format like JSON (JavaScript Object Notation) or XML (Extensible Markup language), but would require a parser.

The last issue that is important to keep in mind is data corruption that could occur in the SD card from improper shutdown. A way to mitigate this would be to add a power circuitry that handles this specific type of event, signalling an interruption to the system, allowing from proper shutdown routines without compromising the data.

5.5. Global control Flow

The global control flow determines the sequence of operations in the system, deciding which of those should be executed and in which order, based on external events generated by an actor or on the passage of time. This is an important step, because in the analysis stage we assumed that all objects executed simultaneously, which its not true, as the system is constrained by the processor. Three main types of control flow can be identified [5]:

- **Procedure-driven:** operations wait for input whenever they need data from an actor, resulting mainly from system written in procedural languages.

- **Event-driven**: A main loop waits for an external event. Whenever an event becomes available, it is dispatched to the appropriate object based on the information associated with the event. It has the advantage of leading to a simpler structure and centralizing all input in the main loop, however it makes the implementation of multi-steps sequences more difficult to implement.
- **Threads:** Threads are the concurrent variation of procedure-driven control. It is the most intuitive of the three mechanisms, however, the debugging is extremely hard because it is difficult to track data and process execution over time.

Looking at the functional requirements and the deployment diagram (Fig. 29) it becomes obvious the need of threads, as the different services need to be executed "simultaneously" from the user's perspective, meaning concurrent access to the processor. The use of procedure-driven control is relegated for the testing of subsystems, as it makes this task fairly easier.

5.6. Boundary conditions

The boundary conditions are limit conditions for the system, thus, it is important to determine how the system should behave in these situations, namely: configuration, system initialization and shutdown, and exceptions.

Exceptions are events or errors that occurs that occur during the execution of the system and can be caused by three different sources: hardware failure, software fault, and change in the operating environment. The one presently addressed is the invalid message through the the display of an informative message to the user.

System initialization and shutdown were addressed previously, when using the state-machine diagrams as it proved itself useful.

The configuration was also addressed here, mentioning the loading of settings. The only thing that remained unclear was how would be the data retrieved. As aforementioned, the settings will be stored in flat files, so the loading of settings corresponds to the reading of the settings from a file to program memory.

5.7. Hardware Specification

5.7.1. Development Board

The Raspberry Pi 3 Model B (Fig. 30) is the development board chosen whose role in the system is compute and control all subsystems. It is a low cost board design around the Broadcom system on-chip (SoC) BCM2837, which includes a 1.2 GHz quad-core ARM Cortex-A53 CPU, with a 64-bit architecture, allowing it to run the full range of GNU/Linux distribution. Among other resources, it includes 1GB RAM, 4 USB ports, 40 GPIO pins, full HDMI port, Ethernet port, Camera Interface (CSI) and Display Interface (DSI), VideoCore IV 3D graphics core, Micro SD Card Slot and on-board Bluetooth Low Energy (BLE) 4.1 and Wi-Fi based on the BCM43438 chip [13].

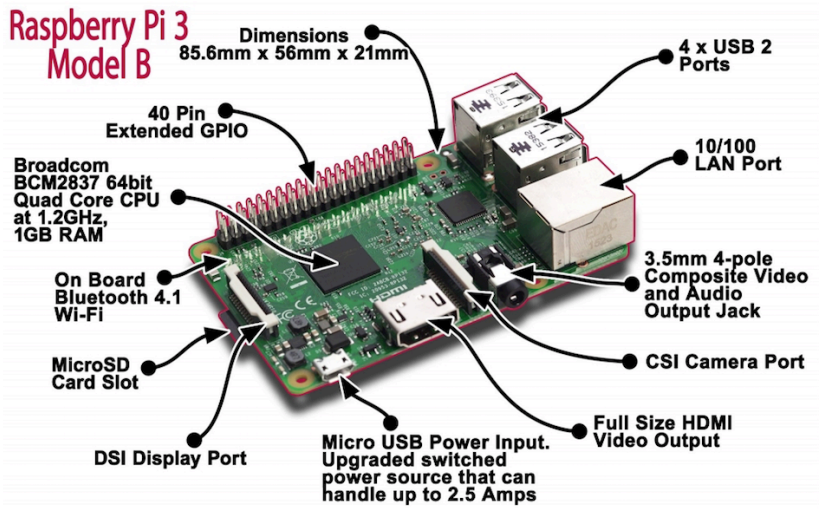


Fig. 30 - Raspberry Pi 3 Model B development board

5.7.2. Lighting Subsystem

As depicted in Fig. 5, the lighting subsystem is composed of a LED, modelling a light bulb, and a Pyroelectric Infra-Red (PIR) sensor for motion detection.

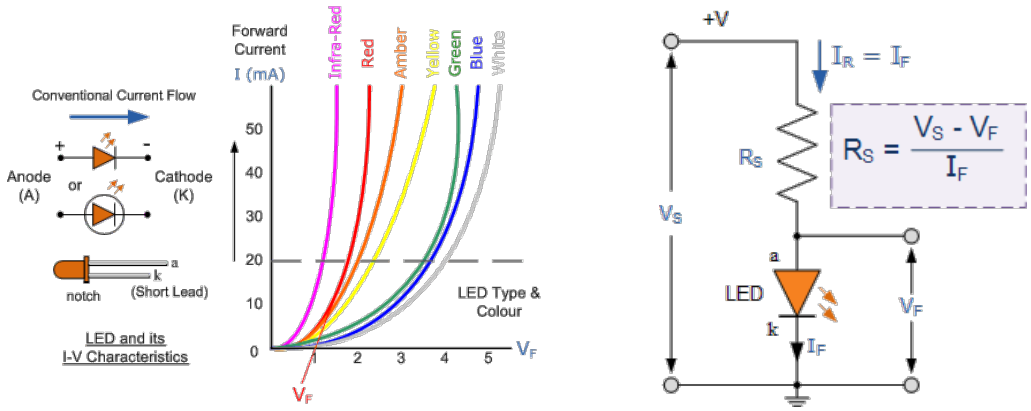
5.7.2.1. LEDs

The Light Emitting Diodes (LEDs) selected are for visualisation purposes, so they fall in the visible light category, emitting a fairly narrow bandwidth of visible light at different coloured wavelengths when a forward current is passed through them.

Depending on the wavelength, a different visible colour will be emitted, requiring a specific semiconductor material for that effect. With the different semiconductor materials varies the I-V characteristics for the LED, resulting at a different voltage drop across its terminals (Fig. 31a).

Before a light emitting diode can “emit” any form of light it needs a current to flow through it, as it is a current dependant device with their light output intensity being directly proportional to the forward current flowing through the LED. This characteristic can be used to manipulate the LED’s brightness, or, in the particular example it tries to model, to act as a dimmer for the lighting system, with a pulse-width modulated (PWM) technique.

As the LED is to be connected in a forward bias condition across a power supply it should be *current limited* using a series resistor to protect it from excessive current flow. The determination of the specific limiting resistor value for the LED was done using Ohm’s Law (Fig. 31b), with a voltage drop (VS – VF) equal to the board 3.3V output and the maximum current allowed per pin as unofficially documented in speciality forums for 3mA/pin on a total maximum of 50mA for all pins [14].



a) b)

Fig. 31 - LED Characteristics: a) I-V characteristics; b) determination of the limiting resistor value

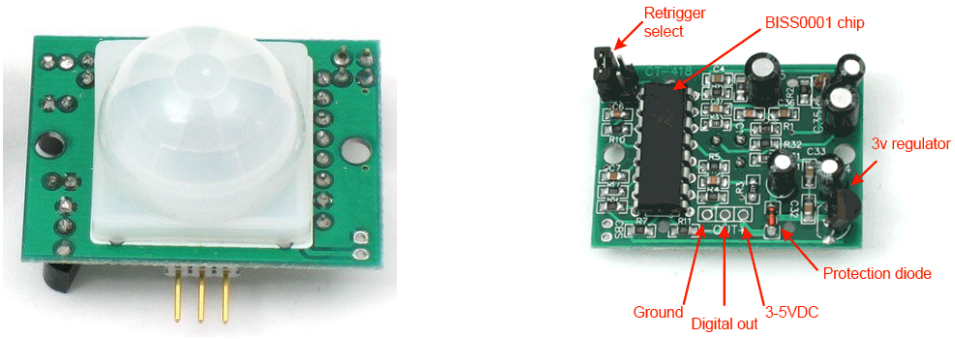
5.7.2.2. Pyroelectric Infra-Red (PIR) sensor

All objects at temperatures above absolute zero (-273.15 °C) emit infrared radiation. This principle is used by the Pyroelectric Infra-Red (PIR) sensor, being split in two halves in a motion detector configuration. The reason for this is one

wants to detect motion (change) not average IR levels, so the two halves are wired up so that they can cancel each other out; if one one half sees more or less radiation than the other the output will swing high or low [15].

So, basically, a PIR sensor detects changes in the amount of infrared radiation it receives, and when this amount is significant, then a pulse is triggered. This means that a PIR sensor can detect when a human (or any animal) moves in front of it.

Along with the PIR sensor, there is a supporting circuitry, mainly composed of resistors, capacitors, voltage regulator, protection diode and a digitizing Integrated Circuit (IC), namely the BISS0001, that takes the output of the sensor and does some minor processing on it to emit a digital output pulse from the analogue sensor [15], illustrated in Fig. 32.



a) b)

Fig. 32 - PIR sensor: a) Front View; b) Back View (PCB)

The pulse emitted when a PIR detects motion needs to be amplified, and so it needs to be powered at 5V/12V (in this case 5V). The output from the PIR IC will be a digital high-level pulse (3V) when triggered (motion detected) and a digital low-level pulse when idle (no motion detected), so the output pin can be connected directly to a GPIO pin. Pulse lengths are determined by resistors and capacitors on the PCB and differ from sensor to sensor The sensitivity is up to 6 meters with a 110° x 70° (azimuth and elevation angles). [15].

5.7.3. Temperature Control subsystem

As depicted in Fig. 5, the temperature subsystem is composed of a thermistor, to measure the room temperature, and a cooling fan to control it. As the output of

the thermistor is an analogue signal, an Analogue-to-Digital Converted (ADC) was added to obtain a digitized value that the Raspberry Pi could handle.

5.7.3.1. Temperature IC sensor

A thermistor is type of whose resistance is dependant on temperature, much more significantly than in standard ones, where it is difficult to measure, whereas in the thermistor case typical values are in the order of the 100Ω/ºC.

There are two main types of thermistors: NTC (Negative Temperature Coefficient) and PTC (Positive Temperature Coefficient), where the former are mainly used as a temperature sensor and the latter as a resettable fuse.

The NTC thermistor could have been used in the project, but the resistance variation with temperature is highly nonlinear, requiring the use of polynomial formulas or look-up tables.

As this posed as unnecessarily complex, a more straightforward solution was adopted, resorting to a common temperature sensor IC, namely the inexpensive TMP-36 (Fig. 33). The TMP-36 is a low voltage, precision centigrade temperature sensor. It provides a voltage output that is linearly proportional to the Celsius temperature and doesn't require any external calibration to provide typical accuracies of ±2°C over the –40°C to +125°C temperature range. Also it's very simply to use: supply an excitation voltage of 2.7 to 5.5 VDC and read the voltage on the Vout pin. The output voltage can be converted to temperature easily using the scale factor of 10 mV/°C [16].

Fig. 33 - Temperature IC sensor TMP-36

5.7.3.2. Fan

A cooling fan is used to model the cooling device for the temperature control subsystem. Cooling fans are typically used for refrigeration in computers, operating normally at 12VDC. Although, the Raspberry Pi does not supply 12V rail, some fans

can operate fairly well still with 5VDC being supplied to its terminals, meaning that some old computer fan could be used in this project. However, the author has a 5VDC operated fan from a previous project that can be used here (Fig. 34). It has a nominal output power of 1W, meaning that it consumes a nominal current of 200mA, far exceeding the limits of the 50mA total for the GPIO pins, remaining only the 5V rail as a possible supply source. As documented, provided the power supply has sufficient power output, 200mA on the 5V should be perfectly safe and achievable, as stated by a commercial application [17].

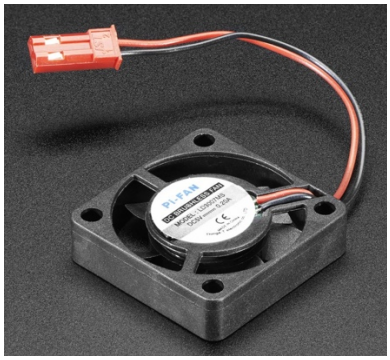


Fig. 34 - Cooling Fan 5VDC, 0.2A

The fan can be supplied directly from the 5V to work at full throttle; however, the rotational speed needs to be varied, depending on the temperature offset, to control the room temperature. Thus, a driving circuit is required, with a switching device controlled by PWM to efficiently regulate the fan's rotational speed. One example of such a circuit is present in Fig. 35 [18]. Q1, a MOSFET, will be switched on/off by the PWM signal from the board, C1 (capacitor) and D1 (diode) will smooth out transient while R1 (resistor) prevents the MOSFET from floating. The 3rd line in the fan represents the feedback signal from the Hall sensor present, acting as a tachometer, enabling the measurement of the rotational speed. In this case, this will not be used as it's not available, but foremost because it's not needed, since the feedback signal for control is the actual room temperature being measured.

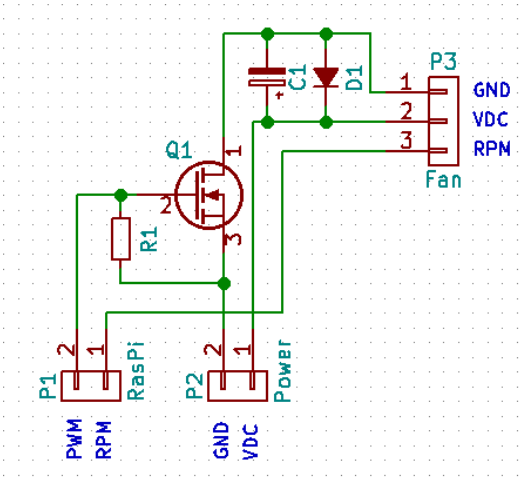


Fig. 35 - Fan driving circuit controlled by PWM

5.7.3.3. ADC

The TMP-36 IC temperature sensor outputs an analogue value. The development board used does not have an ADC on-board, requiring an external one, to convert this signal into a digital one that can be read by the board.

The ADC selected was the MCP3008, a 10-bit ADC, using the Serial Peripheral Interface (SPI) protocol, meaning that 4 pins will be required to read the signal, either in a hardware version (original pins for the purpose; fast, but not available in all distributions), or a software version (bit-bang implementation, pins can be any of the GPIO) [19].

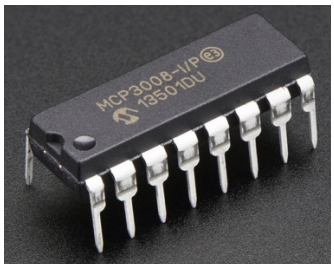


Fig. 36 - MCP3008 10-bit ADC

5.7.4. Door-Bell subsystem

As depicted in Fig. 5, the door-bell subsystem is composed of a push-button, to model the door-bell button, and a buzzer, to model the door-bell generation sound device.

5.7.4.1. Push-Button

There are several types of pushbuttons. The ones selected are tactile switches (Fig. 37), basically consisting of two pieces of metal that can be brought into contact with each other, allowing the flow of electricity when connected, and disabling it when disconnected.

There are some considerations to be done when addressing push-buttons. As the two sides being connected are connected to the GND rail and a GPIO pin, when the push-button is pressed a short-circuit will occur and, therefore, should exist a limiting current resistor. Furthermore, GPIO pins are driven by a tri-state buffer (high, low, and HiZ (high-impedance or floating)) with this latter state being the default. This means that the input will be left floating and any parasite signal can induce a pin state oscillation, which is especially concerning if one is using it on an interrupt-basis. For the aforementioned reasons, the GPIO pins internal pull-up resistor must be enabled on software to obtain proper readings.

The other important aspect relates to the use of the push-button as an interrupt. As tactile push-buttons are mechanical, they have a bouncing behaviour when the contacts are closed; i.e., when pressing the push-button the contacts don't close instantaneously and evenly, resulting in a train of switching pulses when the connection is being established, instead of a unique one. This is a physical characteristic of the switch that tends to get worse with the contact wearing and needs to be addressed by debouncing techniques to obtain a reliable switching pulse.

At this moment, it seems important also to raise the question of how to handle interrupt signals in anything other from bare-metal, as the operating-system poses issues to handle real-time behaviour. Furthermore, it will be important also to understand how to bring interrupts to the user-space, as this is intended to be handled in the kernel-space; the user cannot interrupt the processor, only the kernel can.

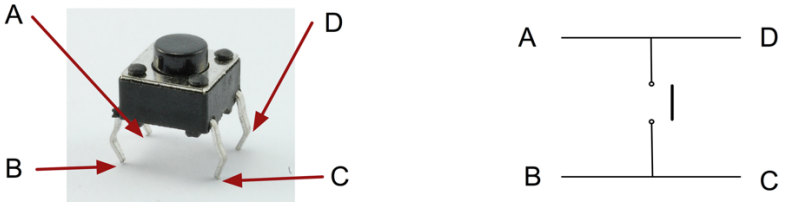


Fig. 37 - Pushbutton: Physical component (left) and schematic (right)

5.7.4.2. Buzzer

A buzzer is an audio signalling device, which can be mechanical, electromechanical, magnetic, electro-acoustic or piezoelectric. Typical applications include alarms, timers, and user input acknowledgment like mouse click or keystroke [20].

The buzzer selected is a piezoelectric one (Fig. 38), based on the inverse principle of piezo electricity, that is, the phenomena of generating electricity when mechanical pressure is applied to certain materials, with the inverse being also true: when subject to an alternating electric field a piezo electric material will stress or compress, according with the frequency of the signal, making the material vibrate and thereby producing sound. The driving signal is a square signal, whose duty cycle can be varied through pulse-width modulation (PWM) technique.

Fig. 38 – Buzzer

5.7.5. Real-Time Clock

The Raspberry Pi is designed to be an ultra-low cost, small size computer, so it does not include a lot of things that a typical computer has, like a little coin-battery-powered Real-Time Clock (RTC) module, which keeps time even when the power is off, or the battery removed. Instead, it is intended to be connected to the Internet via Ethernet or WiFi, updating the time automatically from the global ntp (network time protocol) servers.

When no network connection is available, the original Raspberry Pi will not be able to keep the time when the power goes out. To understand its importance, imagine the following scenario: the home automation system is programmed to control the room temperature between 9 a.m. and 6 p.m. A power failure occurs

during the night, but is re-established before 9 a.m., for instance, at 8:59 a.m. However, the internet connection remains down. When the system recovers, it will initialize with the default time and date, which for a typical POSIX compliant systems would be the iconic date of 1 January 1970, 00:00 GMT. Therefore, at the present time of the writing of this work, it would need 47 years to start to work. Obviously, this is a hyperbole, because, as soon as the internet connection was re-established, the date and time would be accurately set, but it makes up an interesting point.

For the aforementioned reasons, an external device is required, consisting of high oscillating frequency crystal with a backup power supply, typically in the form of a coin cell (Lithium mainly). The one selected is the DS3231 (Fig. 39), a low cost, high-precision I²C RTC with an integrated temperature-compensated crystal oscillator (TCXO) and crystal with an oscillating frequency of 32kHz [21]. It uses the I²C protocol, requiring only two wires to communicate and a kernel with the RTC module pre-compiled (nowadays it seems to come built-in) and I²C module enabled.

a) b)

Fig. 39 - DS3231 Real-Time Clock (RTC): a) Front view; b) Back view

5.7.6. Power Supply

As aforementioned, one of the reasons to try to keep the system as simple as possible, modelling the bulky and more power-hungry components as more modest ones with similar behaviour and functionality, was to minimize the volume and power supply requirements. Adding the estimated power consumptions of all the peripheral components connected yields roughly 500mA, which summed with the power consumption under heavy load (750mA) [22] results in 1.3A, meaning that a power supply with an output current above 1.3A and 2.5A can be used. The power supply chosen was the STONTRONICS T5875DV (Fig. 40), with a micro-USB plug and an output current and output voltage of 2.5A and 5V, respectively [23].

Fig. 40 - STONTRONICS T5875DV Power Supply

5.7.7. SD Card

The SD Card was selected according to the recommendations of the Raspberry Pi Foundation: a minimum capacity of 4GB and a minimum writing speed of 4MB/s, corresponding to class 4 [24]. Furthermore, for the Raspberry Pi 3 Model B, a micro SD card is needed. For these reasons, the PNY SDHC SD card, with 8 GB capacity, class 10, was selected (Fig. 41).

Fig. 41 - PNY 8GB, class 10, SD Card

5.8. Hardware Schematics

5.9. Pin Assignment

5.10. Software Specification

In the local system system implementation an off-the-shelf solution will be used for GPIO interaction, alongside with the POSIX APIs for TCP/IP communication (socket API) and threads implementation (pthreads). The interaction with the development board will be done through a GPIO access library, written in C for the BCM2835 SoC in the Raspberry, abstracting GPIO pins interaction through the familiar Wiring API (used in Arduino, for example) [26]. It has suitable wrappers for others programming languages, like C++.

In the analysis phase, section 4.3, Fig. 19, an initial class model was specified, although already containing some excessive details. Nonetheless, in this section, a more detailed and completed specification of the static and dynamic behaviour of the system is presented, providing a complete guide to its implementation through the specification of: the classes to implement, the system threads and its communication mechanisms, and a detailed description of the threads operations through the use of flowcharts.

5.10.1. Static architecture

The HAS system's UML class diagram will be presented in this chapter with the static architecture divided in four parts: service management; Wi-Fi communication between the remote application and HAS using the TCP/IP protocol; services provided by HAS to the end-user; and sensors/actuators, representing the interaction between the main board and external devices.

A foreword is needed, related to the improvement of the initial static architecture defined in Fig. 19, namely related to the concepts of **class implementer**, **class user** and **class extender**, when specifying the classes interfaces [5]. A class implementer is responsible for realizing the class under consideration, designing the internal data structures and implementing the code for each public operation; the class user invokes the operations provided by the class under consideration during the realization of another class, called the client class, specifying the boundaries of the class; and the class extender develops specialization of the class under consideration. Taking this into account, one can use the concepts of attribute **types**, **signatures**, and **visibility** of attributes and operation to materialise this: a type of an attribute specifies the range of values the attribute can take and the operations that can be applied to the attribute; a signature of an operation refers to the tuple of the types of its parameters and the type of the return value; the visibility of an attribute or operation prevents unintended access to data and operations by specifying who can access it and under which circumstances. UML defines four levels of visibility:

- **Private** attribute or operation: can only be accessed by the class in which it is defined and not by any subclasses or calling classes; intended only for the class implementer and denoted with the character symbol **–**.

- **Protected** attribute or operation: can be accessed by the class in which it is defined and by any descendant of that class, but not by any other class; intended only for the class extender and denoted with the character symbol **#**. It was used in the inheritance relationships between sensor and actuator superclasses and the respective subclasses.
- **Public** attribute or operation: can be accessed by any class constituting the public interface of the class; intended for the class user and denoted with the character symbol **+**. It was used for the attributes and operations made available to the ServiceProvider class.
- **Package** attribute or operation: can be accessed by any class in the nearest enclosing package, enabling a set of related classes (for example, forming a subsystem) to share a set of attributes without having to make them public to the entire system; it is denoted with the character symbol **~** (not confuse with the constructor symbol) and could be used for relating the UniversalTimeServiceSubsystem with the other services.

Another important topic is the implementation of constraints, which can be modelled in a formal language such as the Object Constraint Language (OCL), depicted as notes in the UML class diagram or in a textual form, containing three stereotypes: invariants, precondition, and post-condition, and should not be confused with the three concepts presented in the behavioural state-machine diagram, do, entry and exit, respectively, because all of the formers refer to conditions (guards or events in state-machine), where the latters correspond to actions when entering, remaining or exiting a given state. For invariants, the context for the expression is the class associated with the invariant (denoted by the keywords *self* in Java and *this* in C++), representing a condition that must always be verified, for instance, a valid value for brightness or temperature at the time of instantiation. For preconditions and postconditions, the context of the OCL expression is an operation, with the parameters passed to the operation used as variables in the expression; the precondition refers to the constraints that must be true before the execution of the operation like automateService with the automatic mode on; the precondition refers to the constraints that must be true after the execution of operation like when increasing or decreasing the values of temperature and brightness, with the new value being greater or smaller by 1 unit, respectively.

This last one is especially applicable to the UI deployed in the remote host, as it is responsible for handling and validating the user input.

The description of the class diagrams for the aforementioned 4 parts of the global static architecture will start bottom to top. In Fig. 42, it is illustrated the UML classes diagram for the Sensor and Actuator superclasses and the respective subclasses. The data (attributes) is made protected in the superclass to be accessible only from the subclasses and with public get/set methods to access and manipulate the data, following the idea of data encapsulation. There are two protected methods in the superclasses, referring to the read and write methods, respectively. This was done to prevent tampering in the lower level functions that directly access the hardware. Lastly, the subclasses only have to implement the appropriate constructors and destructors.

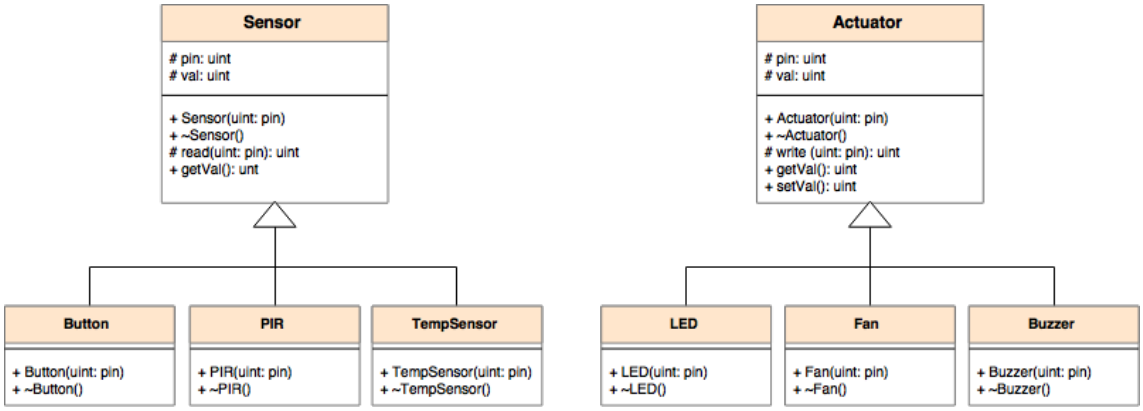


Fig. 42 - UML Sensor and Actuator Classes Diagram

The UML class diagram for the communication classes is presented in Fig. 43. As the Socket API is used a superclass named Socket was created from which both ServerSocket and ClientSocket can inherit. A foreword is needed, as the ClientSocket is to be executed in the remote host, as illustrated before in the UML deployment diagrams and for this reason is represented dashed. However, as the TCP/IP server and client communicate through sockets, this common class can be used to implement both in distinct hardware nodes.

The logic behind the communication classes follows closely the TCP/IP client/server architecture and the overview of system calls using the Socket API presented in Fig. 4. The parent class allows the creation of a socket and subsequent

function to handle the connection, namely the send and receive of data packets and close. It also includes a non-blocking option that can be used when the responsible threads cannot wait for its TCP counterpart. The bind, listen, and accept methods are used for initializing the connection on the server side, while the connect as similar purpose, but on the client side. These functions in the server side were made private, as the constructor should be used to start the connection, being initialized when the HAS system starts. To restart the server communication service (ServerSocket), a new instantiation should be done, taking into account that the previous one is destroyed in the process (singleton pattern), based for instance in a timeout. After the TCP handshake is performed, both sides can use the *send* and *rcv* methods to communicate with each other.

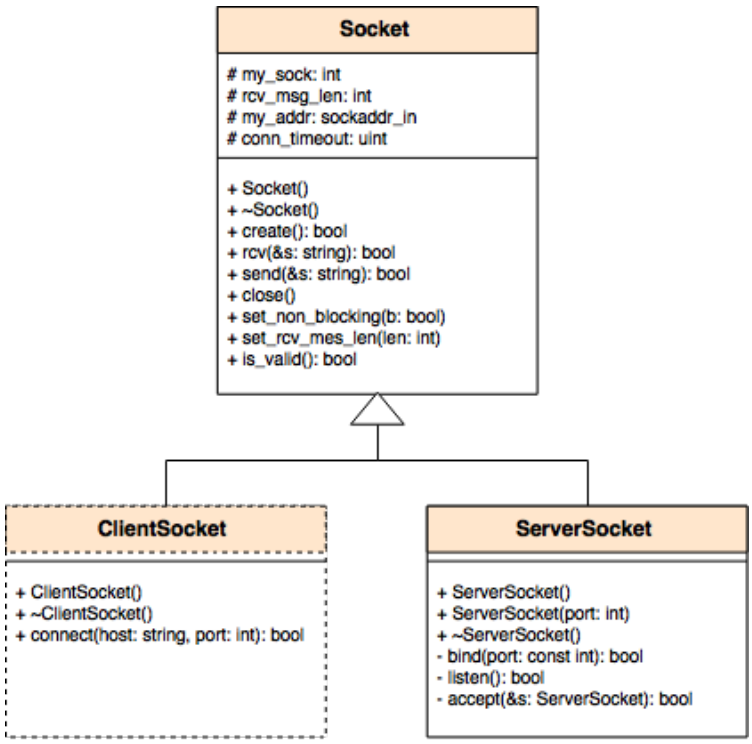


Fig. 43 - UML Communication classes diagram: Both client and server socket inherit from the superclass Socket; the client is represented dashed because it is deployed on a different hardware node than HAS

The UML class diagram for the services classes is presented in Fig. 44. The classes TempService, LightingService and DoorBellService inherit from the superclass Service that provides the basic functionality for setting and getting properties, signalling events and executing. The subclasses need to implement the constructors and destructors; additionally, the TempService class also contain the

methods and attributes related to the temperature set-point and a method to control the room temperature. The UniversalTimeService does not inherit from service as the functionality provided is different, but is contained in the same subsystem. It aggregates a time structure and a RTC object to provide its service.

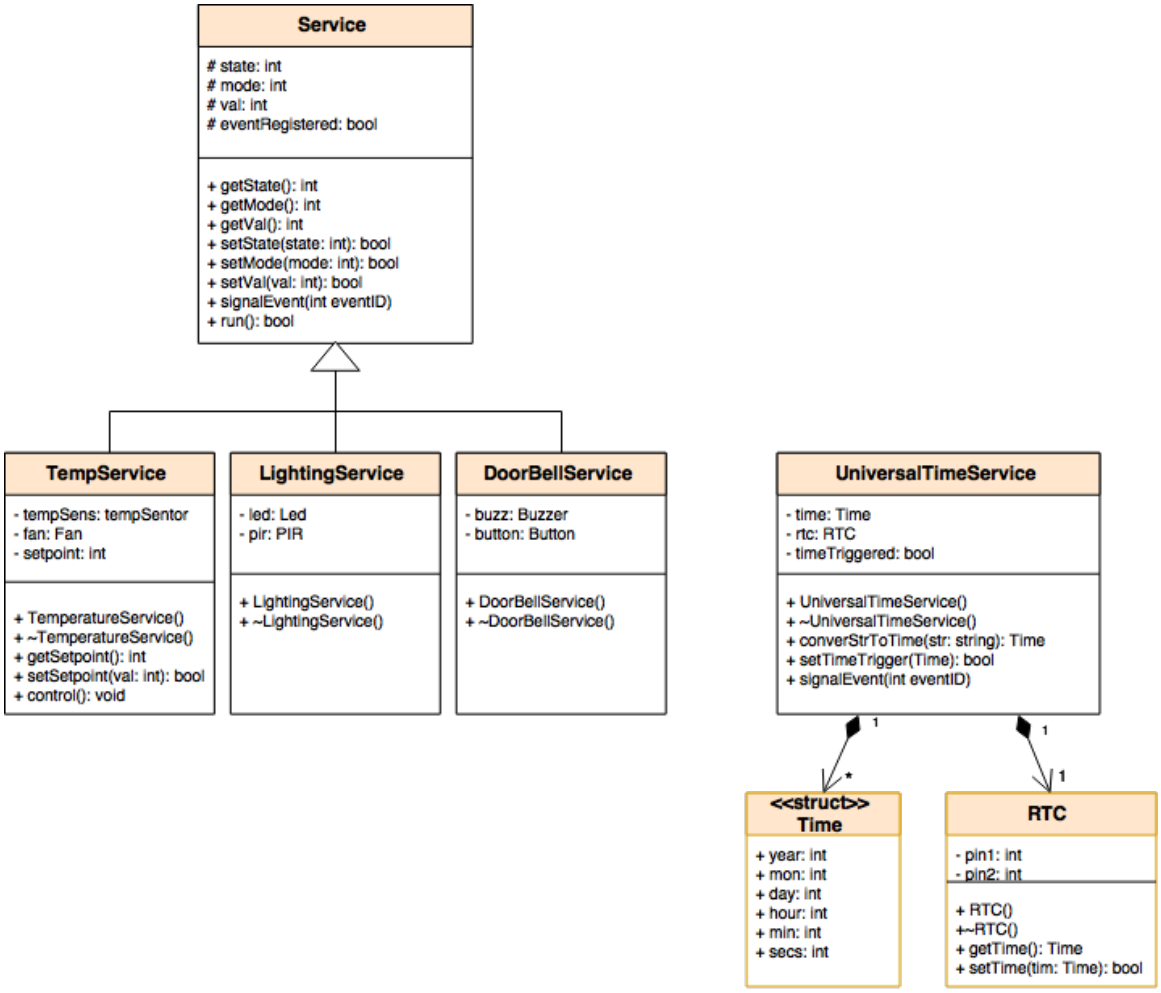


Fig. 44 - UML Services classes diagram

The UML class diagram for the ServiceManager subsystem is presented in Fig. 45. The ServiceProvider class provides the basic functionality for HAS, aggregating: all services indexed in a list and the interface to access and manipulate them, and additionally automate it; the server socket for communication with the outside world; the user settings and notifications and an event struct to handle the internal and external events.

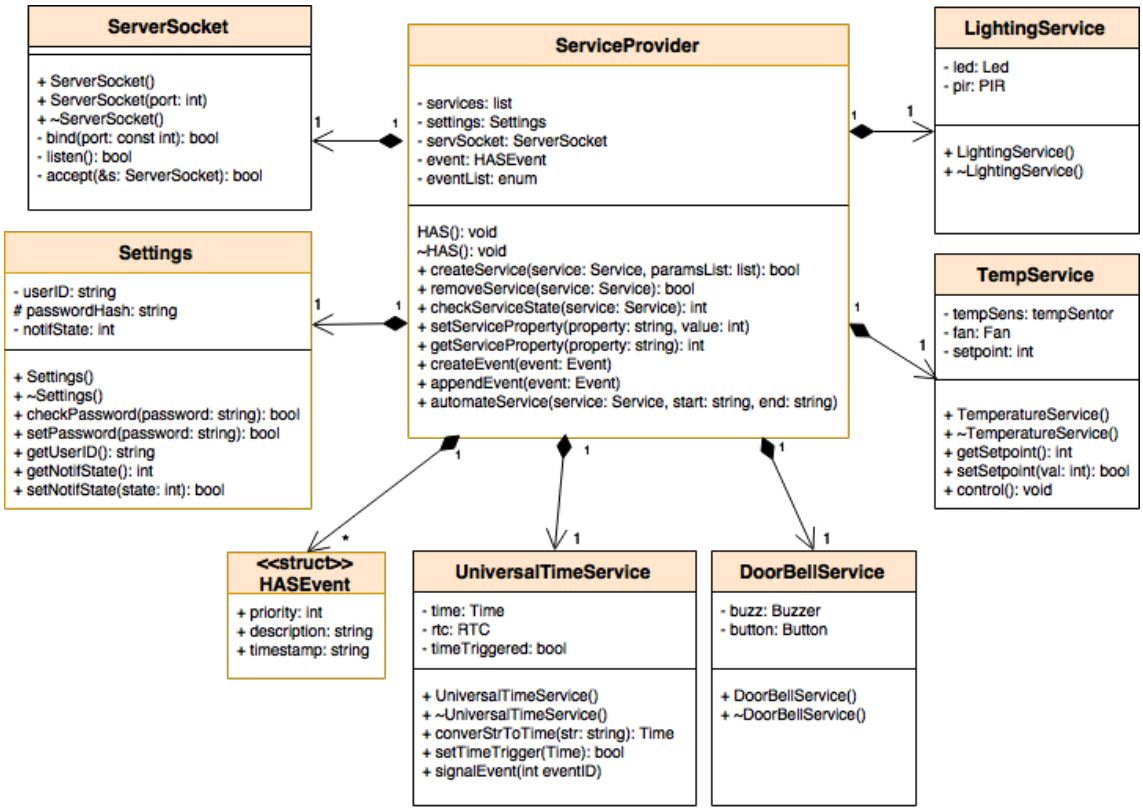


Fig. 45 - UML class diagram for the ServiceManager subsystem

5.11. Tasks

As aforementioned, the main idea would be to have several threads executing, responsible for the communication and services provided, as well with the service management. As such, the aforementioned classes should contain mechanisms to create and manage such threads, to communicate between them and to provide synchronization where needed. A foreword is needed on this topic: multi-threaded execution is very hard to debug and as the project has time constraints associated, threads will not be implemented. Moreover, the author would like to recognise that he does not sufficient technical knowledge to address this topic conveniently; however, it is also the intention of the author to acquire this knowledge in the future to conveniently implement it and benchmark this solution against other types of design.

Thus, the approach taken now is procedure-driven to implement the basic functionality of the subsystems, as indicate to do the subsystem testing. Moreover, this approach leverages on Unix/Linux process-oriented ideology, allowing for the easy implementation of several independent processes, corresponding to the four

main subsystems of HAS and the communication channel based on the TCP/IP protocol.

5.12. Flowcharts

In this section the flowcharts of the main software components will be presented. The flowchart for a generic service containing a server socket is illustrated in Fig. 46. Again, it follows closely the system calls using the socket API presented in Fig. 4. The service with the server socket will be executed in perpetuity, being launched when the system is started-up, and will only exit if aborted or powered-off. First, the GPIO is configured to allow hardware access through the wiringPi API. The socket is configured, being created, binded to a well-known address and notifying the kernel, through the listen() system call, that it is ready to accept incoming connections. The server then accepts the connection from the client to that socket. If the accept() is performed before the client application calls connect(), then the accept() blocks, reason for the inclusion of the timeout. After the client connected, it waits for the arrival of a new message. The message is then read, parsed and a handled. If an acknowledgment is due to the client, a message will be send using the send() system call. The generic handler is the predefined process for each service, being individually addressed ahead, with all other logic being maintained.

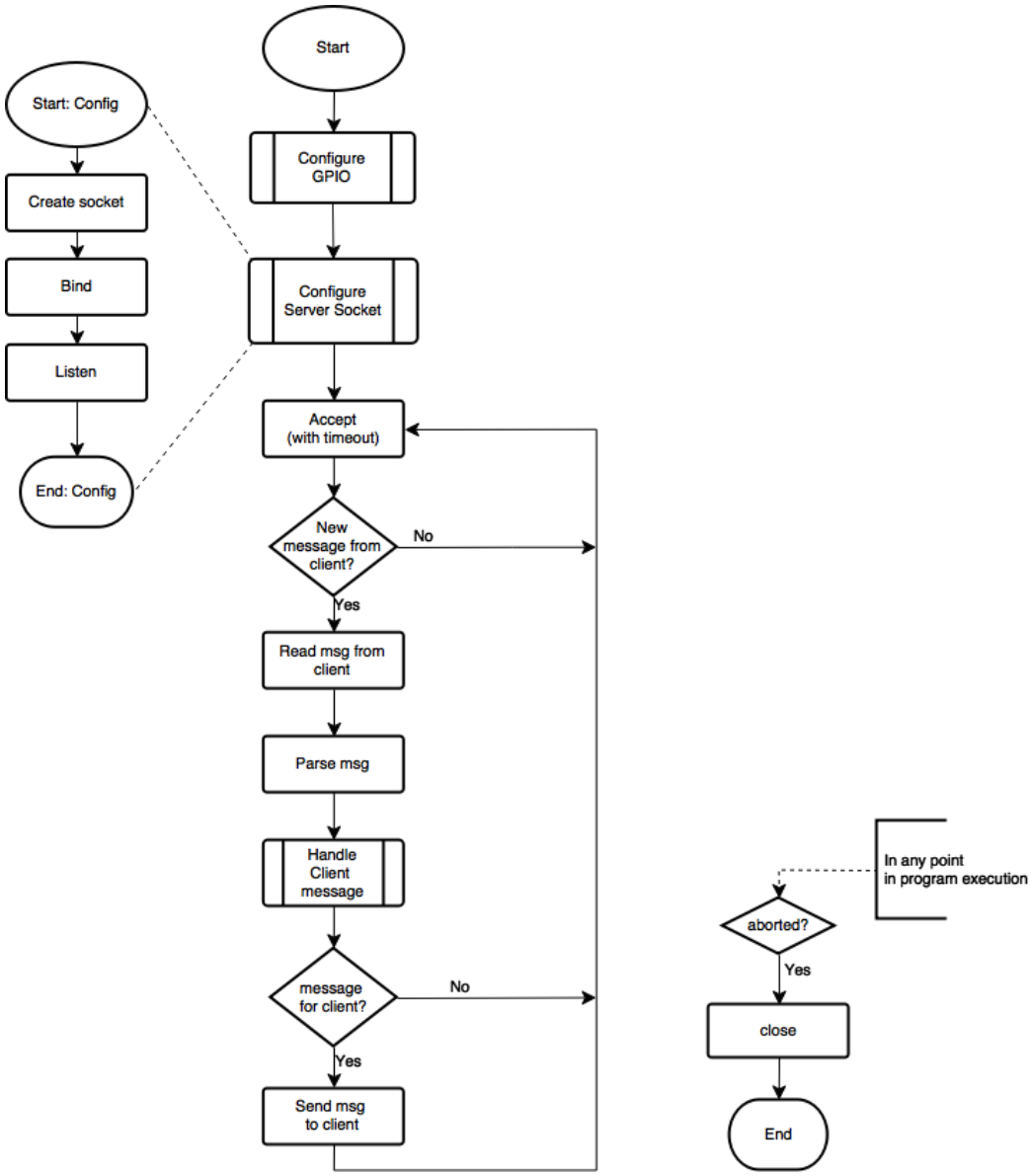


Fig. 46 - Generic service flowchart containing a TCP/IP socket

The flowchart for the client, executing on the remote host, is illustrated in Fig. 47, following the Socket API. The socket is configured, i.e., created and connect to the address of server socket. The user is prompted to issue a valid "command", being checked if it is not to quit the application. The message is sent to server and if a return message is sent by the server it will be display, for example, to acknowledge the command issue was successful. It is noteworthy to mention that the client is agnostic about the message contents and does not perform any sort of validation on user's input (except for quitting the client) and expects the server to do this, as it is the ultimate gatekeeper for the system.

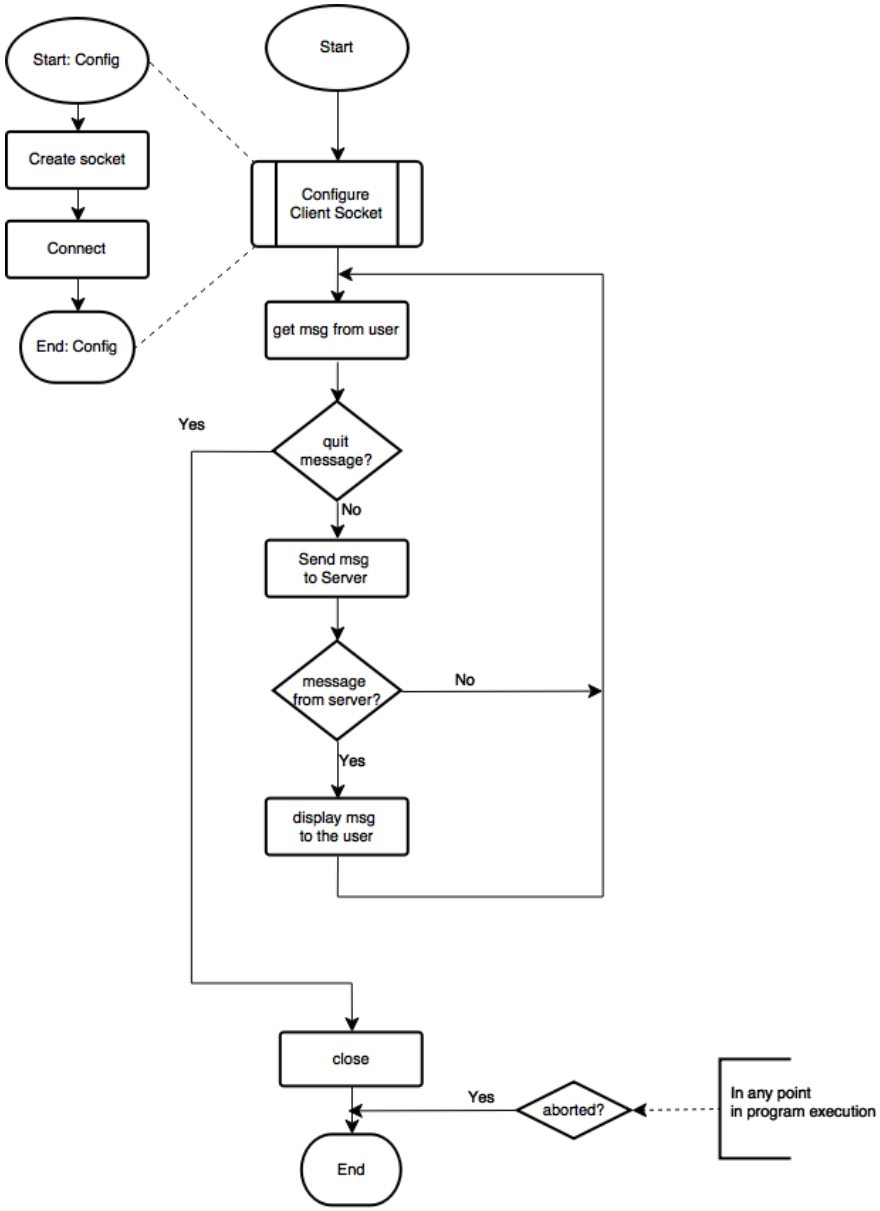


Fig. 47 - Client flowchart (executed on the remote host)

In Fig. 48 is illustrated the flowchart for the door-bell handler. If the door-bell is not pressed the process is suspended from execution a determined amount of time, corresponding to a polling technique. Otherwise, the door-bell will ring and the door-bell button will be disable. Then, a message is sent to the user and a timer starts. If the user answers before the timer runs out the message will be processed, and if it is affirmative the buzzer will play an “affirmative” sound; otherwise, as in the case of the timer running out, the buzzer will play a “negative sound”. After this, the button will be enabled, allowing for further presses, until the user switches state to

off (not depicted here). It should be noted that this is just the door-bell handler and that it is used in conjunction with the server socket, parsing and communication to accomplish the overall goal.

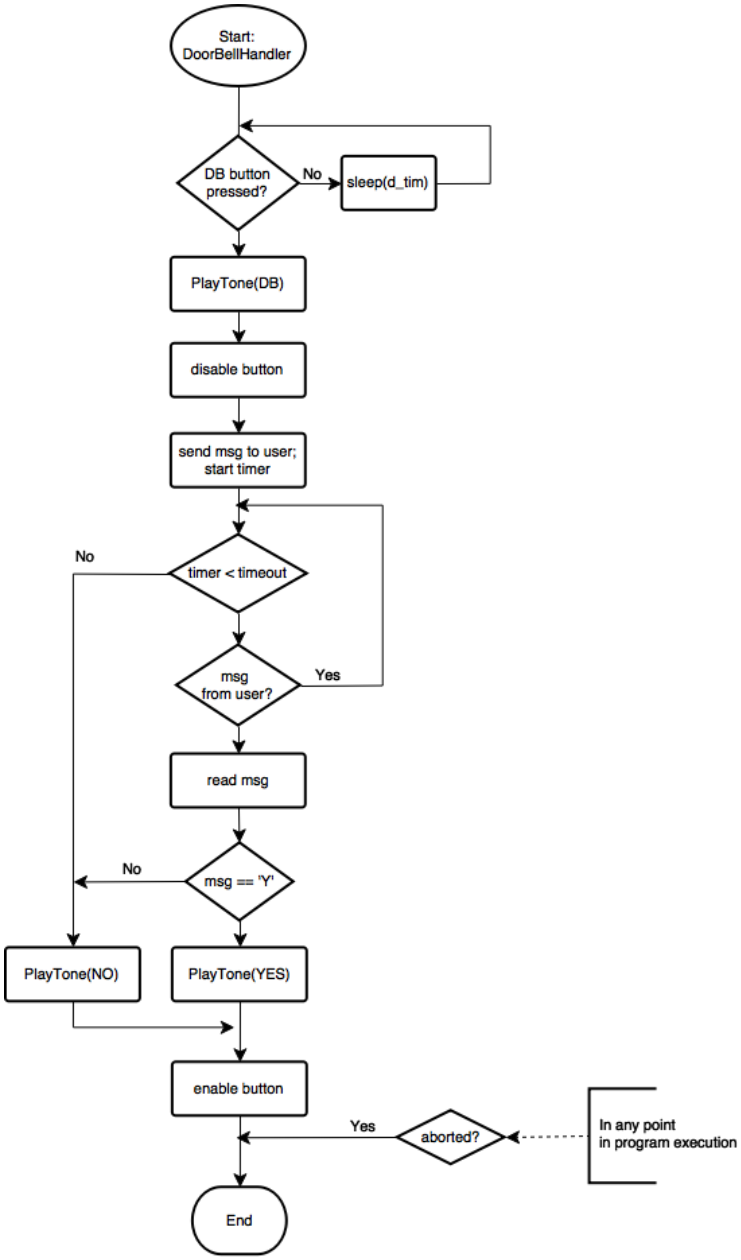


Fig. 48 - Flowchart for the door-bell handler

In Fig. 49 is illustrated the flowchart for the door-bell handler. If the PIR is not pressed the process is suspended from execution a determined amount of time, corresponding to a polling technique. Otherwise, the light will be turned on. Then, a

message is sent to the user and a timer starts to prevent further activations of PIR during a coherent time. After this, the light will be turned and the cycle can be restarted. It should be noted that this is just the lighting handler and that it is used in conjunction with the server socket, parsing and communication to accomplish the overall goal.

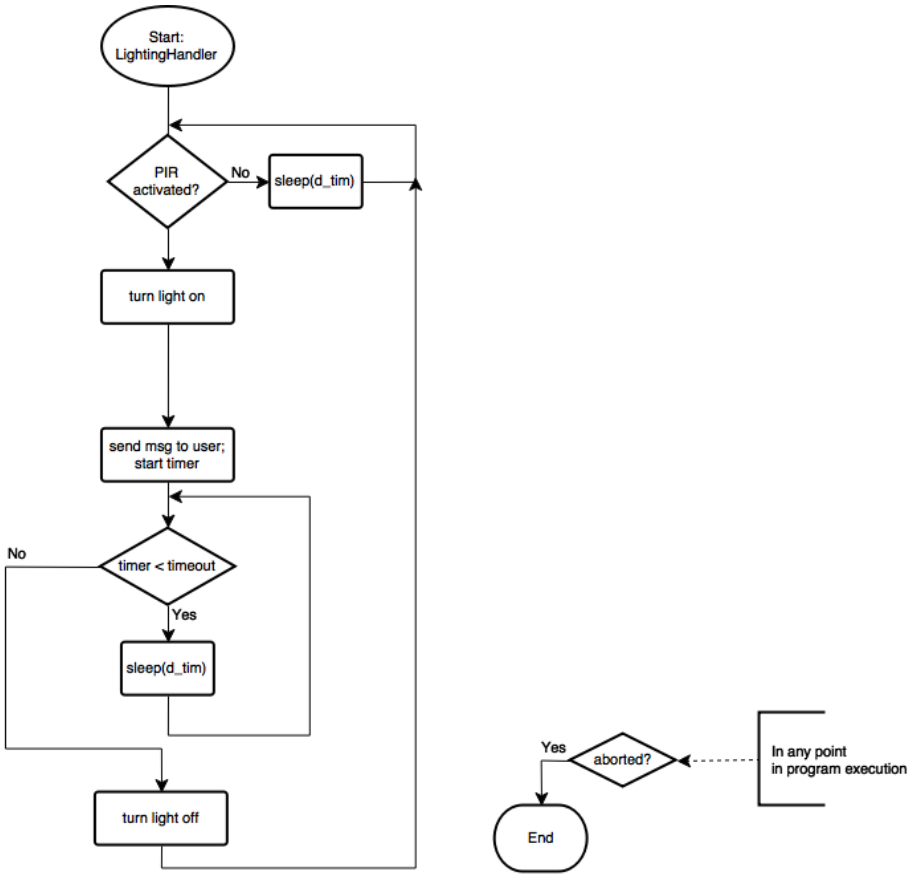


Fig. 49 - Flowchart for the lighting handler

In the Fig. 50 is depicted the flowchart for the temperature control feedback loop. The temperature sensor is read and if it's time to compute, the output value is calculated using a PID algorithm, and the value is written to the actuator, the fan.

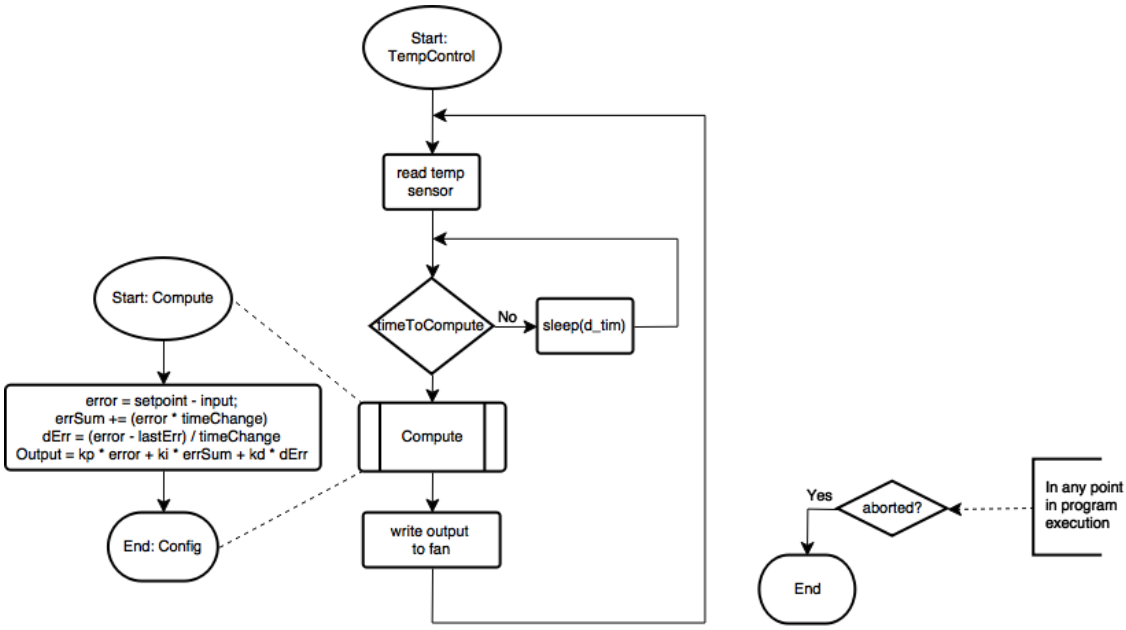


Fig. 50 - Flowchart for the temperature control loop

6. IMPLEMENTATION

The next step in the development process is the implementation, consisting of in the mapping of the design models into actual code. In this section it will be presented the hardware and software implementations and the system configuration.

6.1. Hardware

Regarding the hardware, all electronic circuitry was assembled in a breadboard for easy and individual subsystem testing. Due to time constraints, no advancements were made in trying to develop a printed circuit board to wrap the electronic connections or in a case to enclosure all the hardware.

6.2. Software

The first step towards implementations consisted in setting up all the needed tools for the development. Two approaches were followed: cross-compilation and native compilation. For the first, a virtual machine running a Linux distro (distribution based on the Linux kernel) with Ubuntu flavour, Ubuntu Mate was installed on the host computer, as it seemed more user-friendly and was able to generate suitable code for the Raspberry Pi target outside of the box. For the second approach, a Linux distro suited for the ARMv7 of the Raspberry Pi 3 was installed in the SD card, namely the Raspbian Jessie Light, a light-weight version of the "normal" distribution, without graphical user interface, minimizing the resources utilization. The latter was preferred, due to the need of testing the hardware and the internet connection.

Thus, the next step was configuring the native platform for implementation and testing. The toolchain for compilation is installed outside of the box, namely the gcc compiler, debugger and related tools. The next step was to include the needed libraries, namely the WiringPi library for GPIO access and manipulation.

Another important aspect was the internet connection setup. The network interfaces to access the internet were configured editing the *wpa_supplicant.conf* file and appending the network SSID and password, resulting on data exposure and security liability, as the password is stored in plain text. Next, a dynamic IP addressed had to be configure to allow access based on the host name, as the eduroam network does not allow static IP addresses. Hence, Avahi was installed, which is an implementation of zeroconf, a set of techniques that automatically creates a usable IP network without manual intervention or special configuration servers [27]. Now, the Raspberry Pi could be accessed using the syntax <*hostname*>*.local*, similar to Bonjour clients on Apple, for example.

After this step, the hardware platform was ready to start the implementation. As a facilitating extra step, a secure shell (ssh) service was also configured to allow headless operation, i.e., the Raspberry Pi did not need to be connected to an external display, as the host display would suffice.

Starting the actual implementation, the client-server architecture was implemented in both hardware nodes, namely an Apple Macbook and the Raspberry Pi, to test the TCP/IP communication between both devices. No virtual machine was needed as the Darwin based flavour of the Berkeley Software Distribution (BSD), a Unix-like system, utilises the same socket API with slight nuances. In Listing 1 and , it is illustrated the server and client implementation stubs, used for testing a LED blinking.

Listing 1 – Server implemention in C to test a LED blinking

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
//#include <error.h>
#include <strings.h>
#include <unistd.h>
#include <arpa/inet.h>

#define ERROR          (-1)
#define MAX_CLIENTS    (2)
#define BUFFER_LEN     (256)

// LED Blink
#include <wiringPi.h>
#define LED   18

// helper function to print error info and exit the program
void p_error(char *msg)
{
  perror(msg);
  exit(ERROR);
```

```
}

// msg handlers
int msgHandler( char* );
int ledHandler( char* );

int main (int argc, char *argv[] )
{
  // Variable initialization
  struct sockaddr_in server_addr, client_addr;
  int sockfd, new_con; // socket file descriptor and new connection
  unsigned int port_nr, sockaddr_len = sizeof(struct sockaddr_in);
  int data_len;
  char buffer[BUFFER_LEN]; // allocate a buffer to receive and send msgs

  // check the arguments
  if (argc < 2)
      p_error("Please input the port nr.\n");

  // 1. Create a TCP socket
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
      p_error("Server socket: \n");
  printf("Socket created...\n");

  // Initialize the buffer array to be 0
  bzero( (char *) &server_addr, sizeof(server_addr) );

  // Initialize sockaddr_in structure before binding the socket
  port_nr = atoi(argv[1]);
  server_addr.sin_family = AF_INET; // IPv4 family
  server_addr.sin_port = htons(port_nr); // Host TO Network Short
  server_addr.sin_addr.s_addr = INADDR_ANY; // listen to all interfaces (0.0.0.0) on the
designated port
  //bzero(&server_addr.sin_zero, 8); // do an 8-byte padding

  // 2. Bind the socket to a specific port and IP address
  if ( ( bind(sockfd, ( struct sockaddr *)&server_addr, sockaddr_len ) ) < 0 )
      p_error("bind: \n");
  printf("Binding done...\n");

  // 3. Listen (wait) for a connection: Initialize a wait queue of connections to
  // this socket
  if ( (listen(sockfd, MAX_CLIENTS)) < 0)
      p_error("Listen: \n");
  printf("Listening...\n");

      // Initalize GPIO
  //  wiringPiSetupGpio ();
  //  pinMode( LED, OUTPUT );
  // Repeat the accept connection, write/read data and close connection forever
  while(1) // better signal handling required
  {
      if( (new_con = accept(sockfd, (struct sockaddr *)&client_addr, &sockaddr_len)) < 0)
              p_error("Accept: \n");

      // else connection accepted; print informative message
      printf("New client connect from port nr. %d and IP %s\n", ntohs(client_addr.sin_port),
inet_ntoa(client_addr.sin_addr) );
      data_len = 1;

      while(data_len) // while the client is connect
      {
              // receive the msg
              data_len = recv(new_con, buffer, BUFFER_LEN, 0);

              if( data_len) // if msg is not empty
              {
                      // echo the msg
                      send(new_con, buffer, data_len, 0);
```

```
                        buffer[data_len] = '\0'; // null-terminate the char array so printf
can work properly
                        // print an informative msg in the server side
                        printf("\nSent msg: %s", buffer);// no need to append '\n', because
buffer already hasit

                        // blink the LED
                        if( msgHandler(buffer) )
                                printf("Command executed successfully!\n");
                        else
                                printf("Command invalid!!\n");

                        //buffer = 0;
                }
        }
        // Client has disconnected; inform it
        printf("Client disconnected\n");

        // Close the connection
        close(new_con);

    }

    return 0;
}

int msgHandler( char* msg)
{
    if(msg[0] == 'L')
        return ledHandler(msg);
    else
        return 0;
}

int ledHandler( char * msg)
{
    int returnVal = 1;
    if( msg[1] == '1' )// LED ON
        digitalWrite(LED, HIGH);
    else if ( msg[1] == '0' )
        digitalWrite(LED, LOW);
    else
        returnVal = 0;

    return returnVal;
}
```

Listing 2 - Client implementation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
//#include <error.h>
#include <string.h> // for raspberry pi it is used instead of strings.h
//#include <strings.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netdb.h> // for gethostbyname()

#define ERROR          (-1)
#define MAX_CLIENTS    (2)
#define BUFFER_LEN     (512)

// helper function to print error info and exit the program
```

```
void p_error(char *msg)
{
  perror(msg);
  exit(ERROR);
}

int main (int argc, char *argv[] )
{
  // Variable initialization
  struct sockaddr_in server_addr; // this will contain the address of the server we want to
connect to
  int sockfd, data_len; // socket file descriptor and new connection
  unsigned int port_nr, sockaddr_len = sizeof(struct sockaddr_in);
  struct hostent *server; // this is a pointer to a struct of type hostent, which defines
a host computer in the internet.
  char buffer[BUFFER_LEN]; // allocate a buffer to receive and send msgs

  // check the arguments
  if (argc < 3) {
      fprintf(stderr, "usage: %s hostname port\n", argv[0]);
      exit(ERROR);
  }

  port_nr = atoi(argv[2]);

  // 1. Create a TCP socket
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0 )
      p_error("Server socket: ");
  printf("Socket created...\n");


  server_addr.sin_addr.s_addr = inet_addr(argv[1]);
  if(server_addr.sin_addr.s_addr == -1 )
  {
      // gethostbyname() version
      // takes a host name in the internet as an argument and returns a pointer to a
      // hostent containing information about that host. If the structure is NULL,
      // the system could not locate a host with this name.
      server = gethostbyname(argv[1]);
      if(server == NULL)
      {
              fprintf(stderr, "ERROR: no such host\n");
              exit(ERROR);
      }
      bcopy( (char *)server->h_addr,
             (char *)&server_addr.sin_addr.s_addr,
             server->h_length);
  }
  // Filling the serv_addr struct. Because the field server->h_addr is a char string
  // we use the function:
  //  void bcopy( char *s1, char *s2, int length);
  // which copies length bytes from s1 to s2
  bzero( (char* ) &server_addr, sizeof(server_addr) );
  server_addr.sin_family = AF_INET; //IPv4 family
//    bcopy( (char *)server->h_addr,
//           (char *)&server_addr.sin_addr.s_addr,
//           server->h_length);
  server_addr.sin_port = htons(port_nr);

  // 2. Connect to the remote server
  //  - The client needs to know the port_nr of the server, but not its own.
  //    This is typically assigned by the system when connect is called.
  if( connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
      p_error("ERROR connecting");

  printf("Now connect to server IP %s in the port %d\n", inet_ntoa(server_addr.sin_addr),
ntohs(server_addr.sin_port));

  int exit = 0;
```

```
    while(1) // better signal handling required
    {
        printf("Please enter the message or q to quit: ");
        bzero(buffer, BUFFER_LEN);
        fgets(buffer, BUFFER_LEN - 1, stdin);

        if(strlen(buffer) == 2) // see if the user wants to exit
                exit = (buffer[0] == 'q' ? 1 : 0);

        if(exit) break;

        // write message to socket
        data_len = write(sockfd, buffer, strlen(buffer) );
        if(data_len < 0) p_error("ERROR writing to socket");

        // read reply from the socket
        bzero(buffer, 256);
        data_len = read(sockfd, buffer, BUFFER_LEN - 1);
        if(data_len < 0) p_error("ERROR reading from socket");
        buffer[data_len] = '\0';
        printf("msg received: %s\n", buffer);

    }

    printf("Disconnected from server.\n");
    return 0;
}
```

The server and client were compiled using gcc and tested. It was observed that the LED blinked successfully at user command.

The next step to assure proper implementation was to understand on to execute the server program at startup, so that the user could require its services. A shell script was created to automate the server initialization and it's presented in Listing 3. It uses the default shell interpreter, bash (Bourne Again Shell), to interpret the script at the designated path (a global path is safer) and calls the server program with the port parameter value designated (4545).

Listing 3 - Shell script to automate the server initialization

```
#!/bin/bash
#create a port variable and assign a value to it
port="4545"
#call server program with the port nr. specified
# - the path to the program should be global (it is safer)
/home/pi/app/server $port
```

Starting from the Jessie version of the Raspbian OS, systemd became the preferable way to manage services, which is a trend is other Linux distros. Before, it would suffice to add the shell script to */etc/inittab* file, as init is the first user-space

process to be executed. systemd takes a very different approach from the sysvinit scheme of organizing init scripts into directories by runlevel. Instead, it uses unit files to describe services that should run, along with other elements of the system configuration. These are organized into named targets, like *multi-user.target* and *graphical.target*.

Thus, a unit file was written, illustrated in Listing 4. This defines a new service called "Server TCP/IP" and requesting that it is launched once the multi-user environment is available. The "ExecStart" parameter is used to specify the command we want to run. The StandardInput and StandardOutput are used to specify the streams accessible by the process, i.e., from where to read the data and where to dump it. The permissions on the file were then set to executable and root-level access. The last thing to do was to configure system by enabling the service, illustrated in Listing 5.

Listing 4 - Unit file for running the server process at startup

```
[Unit]
Description=Server TCP/IP

[Service]
# ExecStart=/home/pi/app/server 4545 # -> it works, but it best to encapsulate it in a .sh
script
ExecStart=/bin/sh -c '/home/pi/app/server 4545 >&2'

StandardInput=fd:sockio-stdin
StandardOutput=fd:sockio-stdout

[Install]
WantedBy=multi-user.target
Alias=serverApp.service
```

Listing 5 - Systemd configuration to enable the server service

```
sudo systemctl daemon-reload
sudo systemctl enable serverApp.service
```

The same methodology was used to implement and execute at start-up the remaining services (lighting, temperature and real-time clock), just changing the ports the server will be listening to and that the clients must be aware. In that sense, the idea is to connect 4 different clients by invoking the client 4 times with the ports expected by the respective servers.

7. CONCLUSIONS AND FUTURE WORK

The project, although simple in functionality, proved itself to be a challenge, but also an excellent opportunity to employ good practices, methods and methodologies to development of Linux-based embedded systems with networking and external hardware control capabilities. In that sense, the main overall goal was accomplished.

The usage of the proposed methodology enabled to tackle the complexity associated with the development of an embedded system and provided a whole new perspective to think about these kind of systems, starting from the application domain, i.e. the end-user/client perspective, to the solution domain, i.e., the developer perspective.

The requirement elicitation proved to be a very labour intensive activity, but also proved to be very useful in the definition of the functionalities and constraints of the system, through the identification of the use cases. An excessive detail was used at the time, proving the author's inexperience with this approach, and increasing the complexity unnecessarily.

In the analysis stage, a formalization of the application domain view was accomplished by the generation of the analysis model, composed of three individual models: the functional model, describing the system functionality, represented by the use cases and scenarios (additionally a mock-ups were used to help clarify the functionality and the access to it from the user's view); the analysis object model, describing the structure of the system, represented by the class diagram; and the dynamic model, describing the internal behaviour of the system, represented by the sequence and state-machine diagrams. Again, an excessive detail was used, stemming from the requirement elicitation, especially in the static architecture and dynamic behaviour. Nonetheless, it also helped to identify more use cases and to clarify the interactions.

In the design phase the system began to gain shape. The design goals were defined and the system was decomposed into subsystems and defined. The UML class diagram help to clarify the subsystems and interfaces. The hardware/software mapping was accomplished through the use of UML deployment diagrams, defining clearly which software components would be executed and in which hardware

nodes. The persistent data was identified, as well as a strategy to manage it. The global control flow was selected and the boundary conditions described. Next, the hardware and software specification were done. Regarding the latter, the static architecture of the system was designed, addressing the cohesion and coupling principles. The hardware/software mapping and the static architecture allowed to clarify the software components role, representing the view of the HAS system as a service manager, providing services to the user and communicating with him/her through a TCP/IP communication channel. Next, the tasks were mentioned as the desirable model for executing the software control flow. However, due to its complexity and to time constraints and lack of technical knowledge from the author, the vision was abandoned and an alternative one, procedure-driven, was proposed. Lastly, the flowcharts for the main subsystems were presented.

Regarding the implementation, the hardware was deployed in a breadboard for easy testing. The steps for software implementation were also mentioned, concerning the basic implementation of the server/client architecture running on the local and host, which constitute the core for the system operation and its execution at system's start-up as desired, as 4 identical services can be deployed on the same number of ports, that the user can use to connect.

A last word is needed regarding the end-result. It is clearly not a very successful implementation of the system design, and for that reason the author must also recognise a lack of technical knowledge that, associated with time constraints, didn't allow a better quality. However, it was a major opportunity to learn and also to identify faults and ways to improve in the future.

With that said, one can address the future works. There are a lot of improvements to be made. It is the author's will to try to implement the specified design with tasks and benchmark it against other solutions and also successfully built a home-made automation system that he can use at his home.

REFERENCES

[1] https://www.apple.com/shop/accessories/all-accessories/homekit (accessed in 13 July 2017)

[2] https://www.meo.pt/pacotes/meo-smart-home (accessed in 13 July 2017)

[3] SOMMERVILLE, Ian. Software process models. *ACM computing surveys (CSUR)*, 1996, 28.1: 269-271.

[4] CUSUMANO, Michael A.; SMITH, Stanley A. Beyond the waterfall: Software development at Microsoft. 1995.

[5] BRUEGGE, Bernd; DUTOIT, Allen H. *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. 3rd Ed., Prentice Hall, 2004.

[6] CARNE, E. Bryan. *A professional's guide to data communication in a TCP/IP world*. Artech House, 2004.

[7] WRIGHT, Gary R.; STEVENS, W. Richard. *TcP/IP Illustrated*. Addison-Wesley Professional, 1995.

[8] KERRISK, Michael. *The Linux programming interface*. No Starch Press, 2010.

[9] HANSON, M. David. The Client/Server Architecture. *Server Management*, 2000, 3.

[10]https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.1.0/com.ibm.cics.tx.doc/concepts/c_clnt_sevr_model.html (accessed in 14 July 2017)

[11] BUTENHOF, David R. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[12] NICHOLS, Bradford; BUTTLAR, Dick; FARRELL, Jacqueline. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[13] https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/ (accesed in 16 July)

[14] https://raspberrypi.stackexchange.com/questions/60218/what-are-the-electrical-specifications-of-gpio-pins/60219#60219 (accessed in 16 July)

[15] https://learn.adafruit.com/pir-passive-infrared-proximity-motion-sensor?view=all (accessed in 16 July)

[16] http://www.analog.com/media/en/technical-documentation/data-sheets/TMP35_36_37.pdf (accessed in 16 July)

[17] https://www.adafruit.com/product/3368 (accessed in 17 July)

[18]https://www.domoticz.com/wiki/Raspberry_pi_fan_control_and_monitoring_with_bash (accessed in 17 July)

[19] https://learn.adafruit.com/reading-a-analog-in-and-controlling-audio-volume-with-the-raspberry-pi?view=all (accessed in 17 July)

[20] http://www.futureelectronics.com/en/passives/buzzers.aspx (accessed in 17 July)

[21]https://www.maximintegrated.com/en/products/digital/real-time-clocks/DS3231.html (accessed in 17 July)

[22] http://blog.pimoroni.com/raspberry-pi-3/ (accessed in 17 July)

[23] http://www.farnell.com/datasheets/2020825.pdf (accessed in 17 July)

[24] https://www.raspberrypi.org/documentation/installation/sd-cards.md (accessed in 17 July)

[25] http://www.omg.org/spec/UML/2.5/ (accessed in 24 July)

[26] http://wiringpi.com/ (accessed in 26 July)

[27] http://www.elinux.org/RPi_Advanced_Setup (accessed in 26 July)

APPENDIX 1 – SCENARIOS (DETAILED DESCRIPTION)

The scenarios are described extensively here, subdivided by the respective subsystems.

Temperature Subsystem

Scenario name	monitorTemperature
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the temperature relevant information: state, mode (manual or automatic), temperature setpoint and actual temperature. 3. If Miguel remains connected, the actual temperature is updated.

Scenario name	controlTemperature
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the temperature relevant information: state (on/off), mode (manual or automatic), temperature setpoint and actual temperature. 3. Miguel modifies the mode or the temperature setpoint. 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the actual temperature is updated.

Scenario name	automateTemperature
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the temperature relevant information: state (on/off), mode (manual or automatic), temperature setpoint and actual temperature. 3. Miguel turns the automatic mode on and defines the operation schedule (start and end times). 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the the actual temperature is updated.

Scenario name	turnOnOffTemperature
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the temperature relevant information: state (on/off), mode (manual or automatic), temperature setpoint and actual temperature. 3. Miguel switches the system state (on/off or vice-versa). 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the actual temperature is updated..

Lighting Subsystem

Scenario name	monitorLighting
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the lighting relevant information: state (on/off), mode (manual or automatic) and light brightness. 3. If Miguel remains connected, the state can be updated if Pedro is coming to the house, triggering the motion detector sensor.

Scenario name	controlLighting
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the lighting relevant information: state (on/off), mode (manual or automatic) and light brightness. 3. Miguel modifies the mode or the light brightness. 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the state can be updated if Pedro is coming to the house, triggering the motion detector sensor

Scenario name	automateLighting
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the lighting relevant information: state (on/off), mode (manual or automatic) and light brightness. 3. Miguel turns the automatic mode on and defines the operation schedule (start and end times). 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the state can be updated if Pedro is coming to the house, triggering the motion detector sensor

Scenario name	notifyMiguelOfMotionDetection
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Pedro comes to the house, triggering the motion detector sensor. 2. The light is turned on and the state is updated. 3. Miguel is notified of Pedro's presence outside of the house.

Scenario name	turnOnOffLighting
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the lighting relevant information: state (on/off), mode (manual or automatic) and light brightness. 3. Miguel switches the system state (on/off or vice-versa). 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the state can be updated if Pedro is coming to the house, triggering the motion detector sensor.

Door-Bell

Scenario name	turnOnOffDoorBell
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the door-bell relevant information: state (on/off). 3. Miguel switches the system state (on/off or vice-versa). 4. Miguel saves the modifications or discards it..

Scenario name	notifyMiguelOfDoorBellPressed
Participating actor instances	miguel:User, pedro:Visitor
Flow of events	1. Pedro presses the door-bell button. If the door-bell subsystem is turned on, the door-bell produces an audible sound and the state switches to off, disabling further button presses from Pedro. Otherwise, the door-bell does not ring and the scenario is aborted. 2. Miguel is notified of Pedro's presence at his house door. 3. Miguel acknowledges the door-bell, allowing Pedro's entrance at his house and the door-bell generates a specific affirmative sound. 4. Miguel acknowledges the door-bell, but does not allow Pedro's entrance at his house, or does not acknowledge the door-bell after 30 sec., and the door-bell generates a specific negative sound ("something like: nobody's home!"). 5. The door-bell state is switched to on, enabling other people to press the door-bell button.

Real-Time Clock

Scenario name	configureDateAndTime
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel obtains the RTC relevant information: current date and time. 3. Miguel modifies date and/or time. 4. Miguel saves the modifications or discards it. 5. If Miguel remains connected, the current date and time are updated.

Overall System

Scenario name	connectToHomeAutomationSystem
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel establishes a connection to the home automation system and obtains the relevant information: connection status, current date and time, all available subsystems, system notifications option and a logout option. 3. Miguel cannot establish a connection to the home automation system and obtains an error message describing the problem.

Scenario name	viewSystemNotifications
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel establishes a connection to the home automation system and obtains the relevant information: connection status, current date and time, all available subsystems, system notifications option and a logout option. 3. Miguel selects the system notifications option and all relevant events are presented with timestamp information. The data entries over 30 days should be removed.

Scenario name	turnOnOffSystemNotifications
Participating actor instances	miguel:User
Flow of events	1. Miguel accesses the home automation system ~~(via remote client)~~ and authenticates itself. 2. Miguel establishes a connection to the home automation system and obtains the relevant information: connection status, current date and time, all available subsystems, system notifications option and a logout option. 3. Miguel selects the system notifications option. 4. Miguel switches the system notifications option on/off or vice-versa.

APPENDIX 2 – USE CASES (DETAILED DESCRIPTION)

In this section, the main use cases are described extensively. As many uses cases are similar, only changing the parameter to which they refer to, the Lighting subsystem was used as an example for the Manage<Parameter> use case, where <Parameter> is Temperature, DoorBell, DateAndTime and SystemNotifications.

Use case name	ConnectToHAS
Participating actors	Initiated by User
Flow of events	1. The User inputs the credentials: username and the password and tries to connect. 2. HAS looks up the credentials for a match. If they match, the User is logged on the system (include use case ManageHAS) and a log message is appended to the system notifications with a time-stamp (extended by the use case LogSystemMessage). Otherwise the connection is refused.
Entry conditions	• The User has internet connection.
Exit conditions	• The User is logged on the system, or the connection is refused (it is extended by the use case connectToHAS).
Quality Requirements	• The connection to HAS is acknowledged within 20 seconds, with the login on the system or an connection error message.

Fig. 51 - Use case ConnectToHas. Under ConnectToHas, the left column denotes actor actions and the right column denotes system responses.

Use case name	ManageHAS
Participating actors	Initiated by User
Flow of events	1. HAS responds by showing the User the services can be managed, system functionalities, connection status and current date and time. If the User doesn't select any option, but remains logged in, the date and time are updated (extended by the use case UpdateDateAndTime). 2. The User selects the lighting management (include use case ManageLighting). 3. The User selects the temperature management (include use case ManageTemperature). 4. The User selects the door-bell management (include use case ManageDoorBell). 5. The User selects the date and time management (include use case ManageDateAndTime. 6. The User selects the system notifications management (include use case ManageSystemNotifications). 7. The User selects the settings management (include use case ManageSettings).
Entry conditions	• The User is logged into HAS.
Exit conditions	• The User has logged out (include use case Logout).

Fig. 52 - Use case ManageHAS

Use case name	ManageLighting
Participating actors	Initiated by User
Flow of events	1. HAS responds by showing the User the previously configured values and the command options (the User cannot manage the lighting if he/she does not see what to manage). (It is extended by the use case DetectMotion). 2. The User modifies the brightness of the lighting (include use case ModifyLightingValue). 3. HAS responds by changing the actual brightness of the light and updates the value presented to the user. 4. The User switches the lighting subsystem state (include use case SwitchLightingState). 5. HAS responds by disabling/enabling all functionalities of the lighting subsystem (except SwitchLightingState) and updates the state presented to the user. 6. The User switches the lighting subsystem mode (include use case SwitchLightingMode). 7. HAS responds by switching the actual mode of the lighting subsystem and updates the value presented to the user.
Entry conditions	• The User is logged into HAS and selected the Lighting option.
Exit conditions	• The User has quitted the Lighting option, saving or discarding the modifications done.

Fig. 53 - Use case ManageLighting

Use case name	SwitchLightingMode
Participating actors	Initiated by User
Flow of events	1. The User switches the lighting mode. 2. HAS responds by switching the lighting mode and updates the value presented to the user
Entry conditions	• The User is logged into HAS and selected the Lighting option.
Exit conditions	• The User has quitted the Lighting option, saving or discarding the modifications done.

Fig. 54 - Use case SwitchLightingMode

Use case name	AutomateLighting (extends SwitchLightingMode)
Participating actors	Initiated by User
Flow of events	1. The User switches to the Lighting Automatic mode. 2. HAS responds by providing the User the option to indicate the initial and end times of the lighting automated task. 3. The User modifies the start and end times of the lighting automated task, increasing or decreasing time. 4. HAS responds by updating the values to the User
Entry conditions	• The user has switched to the Lighting Automatic mode.
Exit conditions	• The User has quitted the Automate Lighting option, saving (extended by the use case LogSystemMessage) or discarding the modifications done.

Fig. 55 - Use case AutomateLighting

Use case name	UpdateEndTime (extends AutomateLighting)
Participating actors	Initiated by User
Flow of events	1. The User increases the starting time of the automated lighting task. 2. HAS responds by updating the starting time and also the end time of the automated lighting task. The end time should be, at least, 1 min superior.
Entry conditions	• The user has increased the starting time of the automated lighting task.
Exit conditions	• The User has quitted the Automate Lighting option, saving or discarding the modifications done.

Fig. 56 - Use case UpdateEndTime

Use case name	SwitchLightingState
Participating actors	Initiated by User
Flow of events	1. The User switches the lighting state. 2. HAS responds by switching the lighting state and updates the value presented to the user
Entry conditions	• The User is logged into HAS and selected the Lighting option.
Exit conditions	• The User has quitted the Lighting option, saving or discarding the modifications done.

Fig. 57 - Use case SwitchLightingState

Use case name	SwitchLightingOff (extends SwitchLightingState)
Participating actors	Initiated by User
Flow of events	1. The User switches off the lighting. 2. HAS responds by disabling the lighting service and the related functionalities.
Entry conditions	• The User had previously switch on the lighting.
Exit conditions	• The User has quitted the Lighting option, saving or discarding the modifications done.

Fig. 58 - Use case SwitchLightingOff

Use case name	SwitchLightingOn(extends SwitchLightingState)
Participating actors	Initiated by User
Flow of events	1. The User switches on the lighting. 2. HAS responds by enabling the lighting service with the parameters selected and the related functionalities.
Entry conditions	• The User had previously switch off the lighting.
Exit conditions	• The User has quitted the Lighting option, saving or discarding the modifications done.
Quality Requirements	• If the lighting mode is manual, the lights should start within 1 second, otherwise, only when it is indicated by AutomateLighting use case.

Fig. 59 - Use case SwitchLightingOn

Use case name	ModifyLightingValue
Participating actors	Initiated by User
Flow of events	1. The User modifies the lighting value, increasing or decreasing it. 2. HAS responds by updating the value presented to the user, if it is inside the lighting brightness range. If the manual mode is selected the actual physical Lighting brightness value should be updated immediately.
Entry conditions	• The User is logged into HAS and selected the Lighting option.
Exit conditions	• The User has quitted the Lighting option, saving or discarding the modifications done.
Quality Requirements	• If the lighting mode is manual, the lights should start within 1 second, otherwise, only when it is indicated by AutomateLighting use case.

Fig. 60 - Use case ModifyLightingValue

Use case name	UpdateDateAndTime (extends use cases ManageHAS and ManageDateAndTime)
Participating actors	Initiated by User
Flow of events	1. HAS responds by updating the time and date of the system.
Entry conditions	• HAS has been initialized once (connected once to the power supply) and 1 minute has passed.
Exit conditions	• The battery power supply is removed from the HAS system.

Fig. 61 - Use case UpdateDateAndTime

Use case name	DetectMotion
Participating actors	Initiated by Visitor
Flow of events	1. The Visitor comes close to the house, within the range of the motion detection sensor. 2. HAS responds by turning the light (extended by the use case NotifyUser).
Entry conditions	• HAS is powered on and operating.
Exit conditions	• HAS is powered off.
Quality Requirements	• The lights should turn on under 2 seconds.

Fig. 62 - Use case DetectMotion

Use case name	PressDoorBellButton
Participating actors	Initiated by Visitor
Flow of events	1. The Visitor presses the Door Bell button. 2. HAS responds by producing an audible sound (extended by the use case NotifyUser) and disables the door-bell until the User acknowledges this or until a time period has not passed.
Entry conditions	• The Door-Bell subsystem is turned on.
Exit conditions	• HAS is powered off or the Door-Bell subsystem is turned off.
Quality Requirements	• The door bell should ring under 1 second.

Fig. 63 - Use case PressDoorBellButton

Use case name	NotifyUser (extends use cases DetectMotion and PressDoorBellButton)
Participating actors	Initiated by Visitor Communicates with User
Flow of events	1. HAS responds by sending a notification to the User and by logging the notifications in the system notifications.
Entry conditions	• The Visitor has triggered the motion detector or has pressed the door bell button, there is internet connection and the User is logged in to the system.*
Exit conditions	• HAS is powered off.
Quality Requirements	• The notification should be sent to the User within 5 seconds.

Fig. 64 - NotifyUser

* It is not realistic to think that the User needs to be logged in to the system to receive these notifications. These events can occur anytime and should be delivered to the User under any circumstances. Several solutions could be used: if there is Internet connection, an email service could be provided to email the user, reporting these events. In a more critical situation, with no Internet connection available, a Geo-Service Messaging (GSM) could be provided to send a text message to the User. Otherwise, when the User connects back to the system, the notifications should pop-up indicating that these priority events have occurred.

Use case name	HandleDoorBellPressed
Participating actors	Initiated by User Communicates with Visitor
Flow of events	1. The User acknowledges the door-bell pressing and allows the Visitor to enter by sending an OK to HAS. 2. HAS responds by playing an affirmative sound to the Visitor to signal that is entrance is allowed. 3. The User does not acknowledge the door-bell pressing within a time period or denies the entrance to the Visitor. 4. HAS responds by playing an affirmative sound to the Visitor to signal that is entrance is allowed.
Entry conditions	• The Visitor has triggered the motion detector or has pressed the door bell button.
Exit conditions	• HAS is powered off.
Quality Requirements	• The time period for acknowledgment should not exceed 30 seconds.

Fig. 65 - Use case HandleDoorBellPressed

Use case name	Confirm (extends ManageTemperature, ManageLighting, ManageDoorBell, ManageDateAndTime, ManageSystemNotifications, Logout and Automate)
Participating actors	Initiated by User
Flow of events	1. The User selects the confirm option in the extended use cases. 2. HAS responds by returning to the higher-level menu.
Entry conditions	• The User have done some modifications in the extended use cases.
Exit conditions	• HAS is powered off or the internet connection is lost.
Quality Requirements	• The time period for acknowledgment should not exceed 2 seconds.

Fig. 66 - Use case Confirm

Use case name	Cancel (extends ManageTemperature, ManageLighting, ManageDoorBell, ManageDateAndTime, ManageSystemNotifications, Logout and Automate)
Participating actors	Initiated by User
Flow of events	1. The User selects the cancel option in the extended use cases. 2. HAS responds by replacing the values modified meanwhile by the User, by the old ones before these modifications.
Entry conditions	• The User is using one of the extended use cases.
Exit conditions	• HAS is powered off or the internet connection is lost.
Quality Requirements	• The time period for acknowledgment should not exceed 2 seconds.

Fig. 67 - Use case Cancel

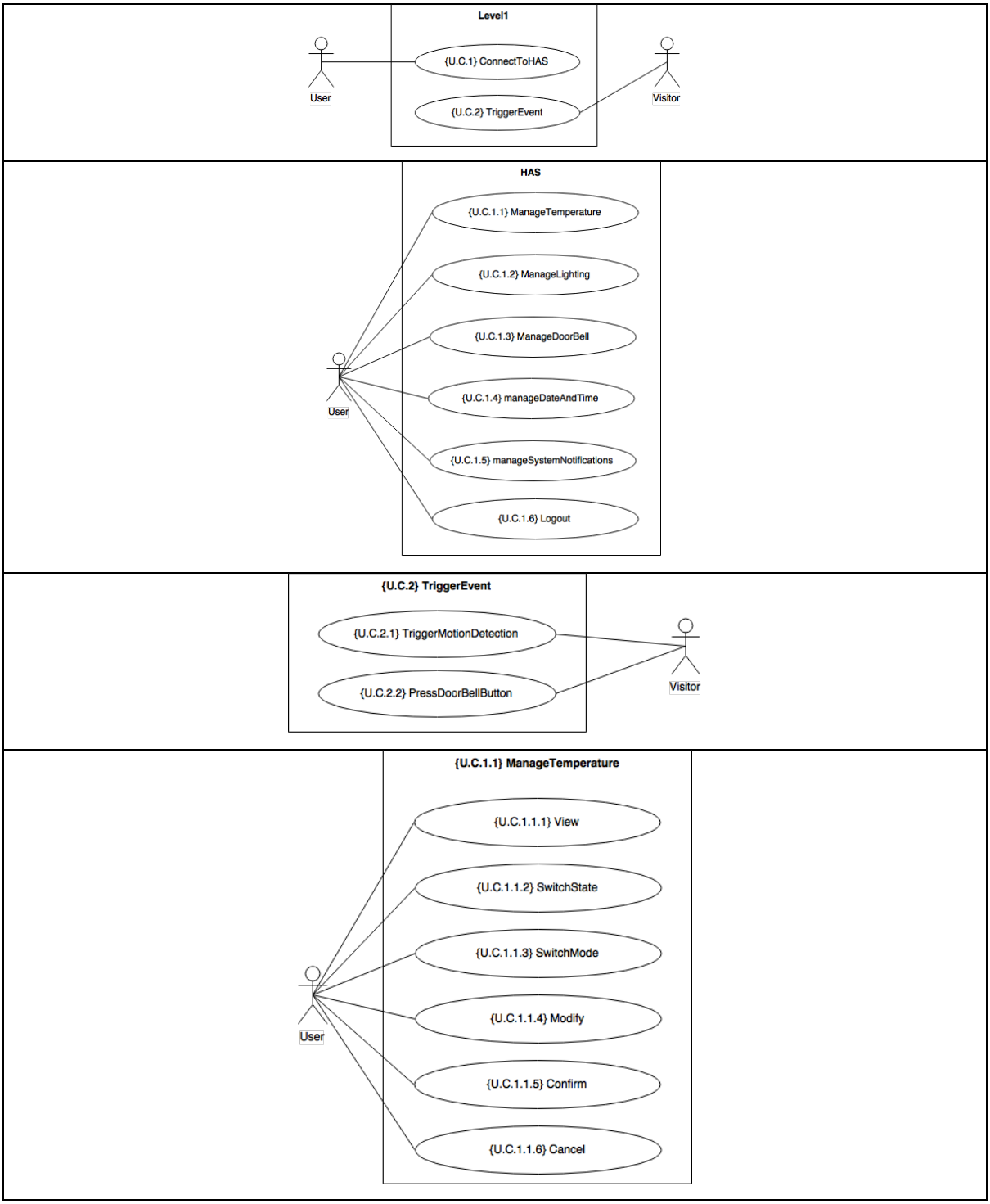
Use case name	LogSystemMessage (extends AutomateTemperature, AutomateLighting, ConnectToHAS and NotifyUser)
Participating actors	Initiated by User or Visitor
Flow of events	1. The description of the relevant event should be appended to the system notifications with time-stamp information. 2. HAS responds by adding the description of the relevant event with time-stamp information to the system notifications.
Entry conditions	• The User or Visitor use one of the extended use cases.
Exit conditions	• HAS is powered off.
Quality Requirements	• System notifications older than 30 days should be removed.

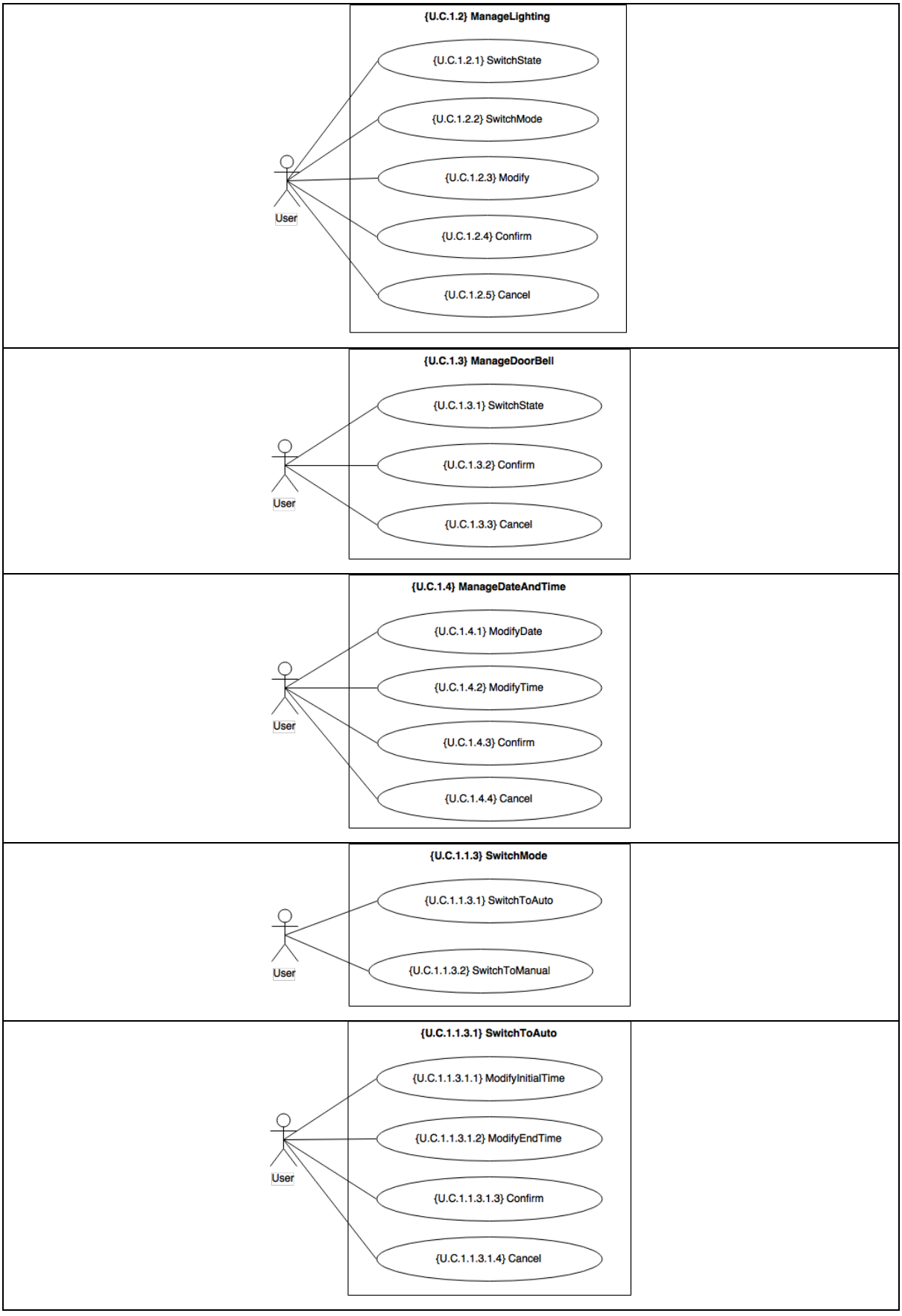
Fig. 68 - Use case LogSystemMessage

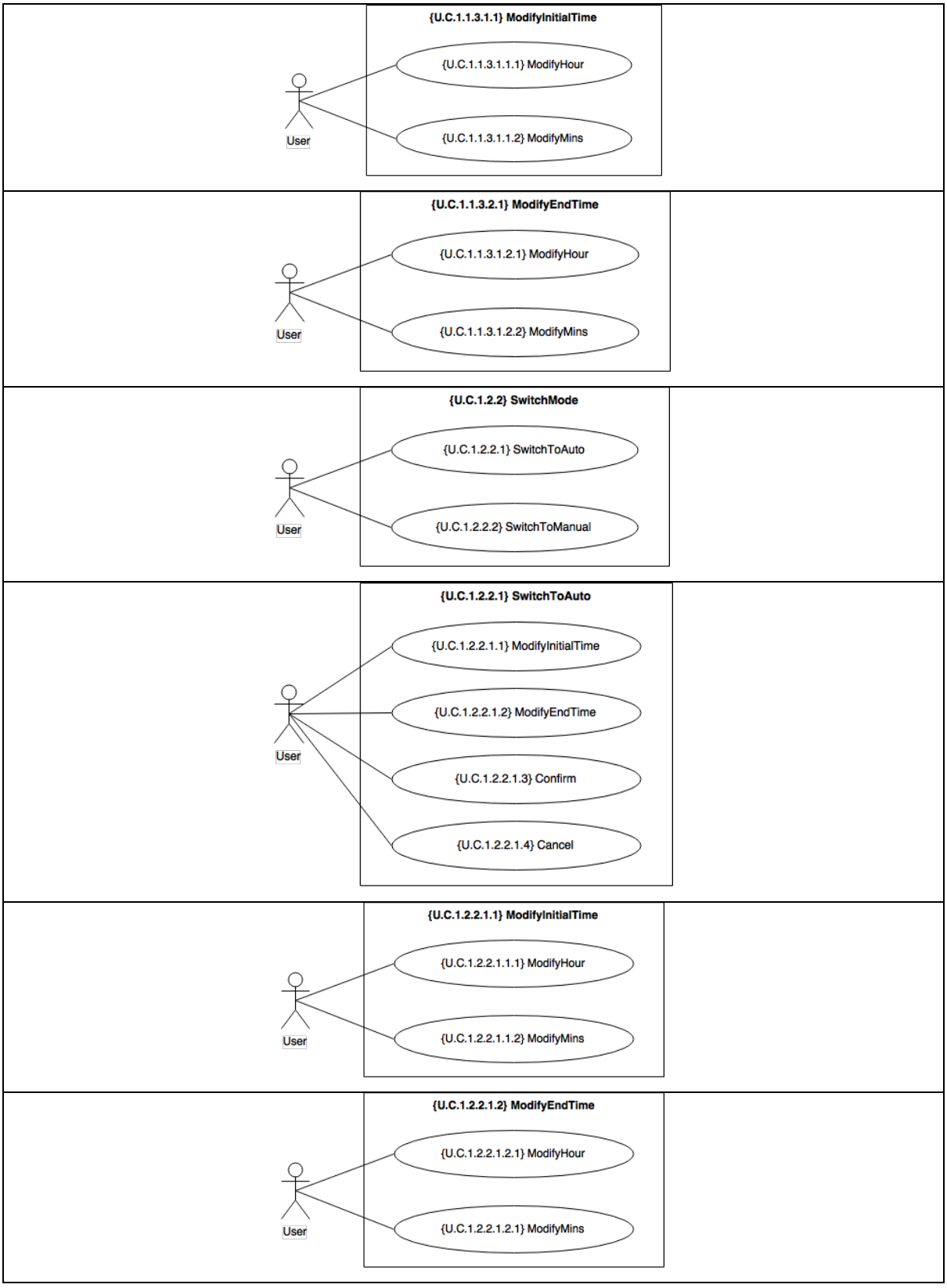
Use case name	Connection Down (extends all use cases)
Participating actors	Initiated by User or Visitor
Flow of events	1. The User tries to connect to HAS or is expecting some feedback from HAS. 2. If trying to establish a connection is not possible, inform the user. If expecting some feedback, HAS should respond periodically to the User, just to acknowledge the internet connection is available. Otherwise, the user should be informed about the broken internet connection.
Entry conditions	• The internet connect in HAS is down.
Exit conditions	• Informative message is displayed to the user.
Quality Requirements	• If no internet connection in HAS is available within 20 seconds, warn the user.

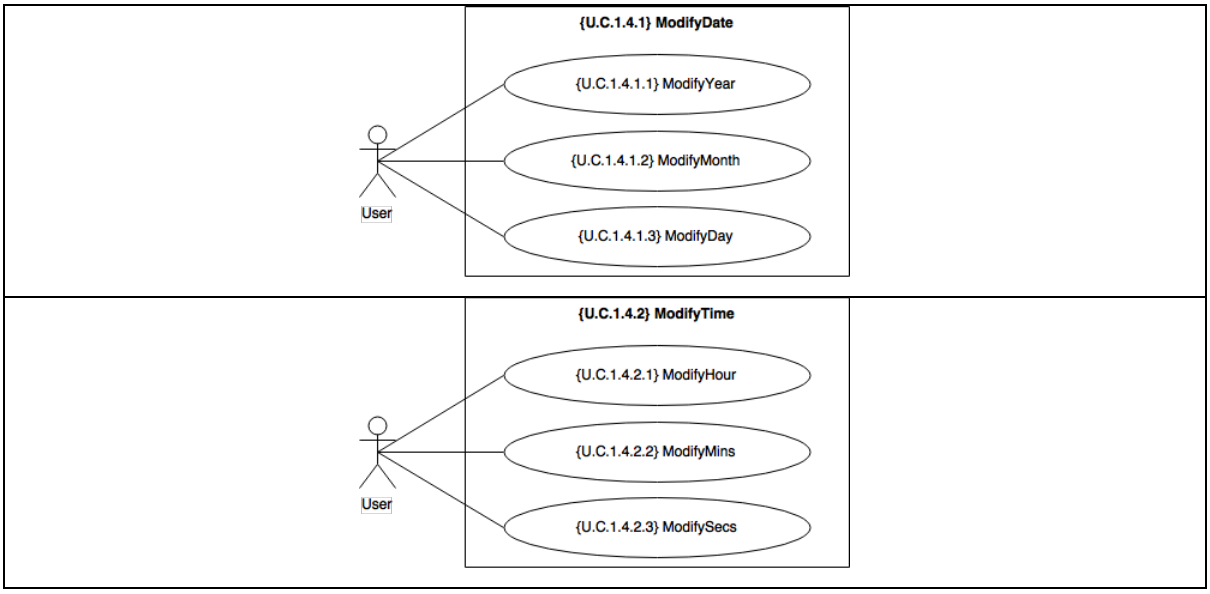
Fig. 69 - Use case ConnectionDown

Table 6 - Discriminated Use cases




{U.C.1.2} ManageLighting
{U.C.1.2.1} SwitchState
{U.C.1.2.2} SwitchMode
{U.C.1.2.3} Modify
{U.C.1.2.4} Confirm
{U.C.1.2.5} Cancel
User
{U.C.1.3} ManageDoorBell
{U.C.1.3.1} SwitchState
{U.C.1.3.2} Confirm
{U.C.1.3.3} Cancel
User
{U.C.1.4} ManageDateAndTime
{U.C.1.4.1} ModifyDate
{U.C.1.4.2} ModifyTime
{U.C.1.4.3} Confirm
{U.C.1.4.4} Cancel
User
{U.C.1.1.3} SwitchMode
{U.C.1.1.3.1} SwitchToAuto
{U.C.1.1.3.2} SwitchToManual
User
{U.C.1.1.3.1} SwitchToAuto
{U.C.1.1.3.1.1} ModifyInitialTime
{U.C.1.1.3.1.2} ModifyEndTime
{U.C.1.1.3.1.3} Confirm
{U.C.1.1.3.1.4} Cancel
User


{U.C.1.1.3.1.1} ModifyInitialTime
{U.C.1.1.3.1.1.1} ModifyHour
{U.C.1.1.3.1.1.2} ModifyMins
User
{U.C.1.1.3.2.1} ModifyEndTime
{U.C.1.1.3.1.2.1} ModifyHour
{U.C.1.1.3.1.2.2} ModifyMins
User
{U.C.1.2.2} SwitchMode
{U.C.1.2.2.1} SwitchToAuto
{U.C.1.2.2.2} SwitchToManual
User
{U.C.1.2.2.1} SwitchToAuto
{U.C.1.2.2.1.1} ModifyInitialTime
{U.C.1.2.2.1.2} ModifyEndTime
{U.C.1.2.2.1.3} Confirm
{U.C.1.2.2.1.4} Cancel
User
{U.C.1.2.2.1.1} ModifyInitialTime
{U.C.1.2.2.1.1.1} ModifyHour
{U.C.1.2.2.1.1.2} ModifyMins
User
{U.C.1.2.2.1.2} ModifyEndTime
{U.C.1.2.2.1.2.1} ModifyHour
{U.C.1.2.2.1.2.1} ModifyMins
User


{U.C.1.4.1} ModifyDate
{U.C.1.4.1.1} ModifyYear
{U.C.1.4.1.2} ModifyMonth
{U.C.1.4.1.3} ModifyDay
User
{U.C.1.4.2} ModifyTime
{U.C.1.4.2.1} ModifyHour
{U.C.1.4.2.2} ModifyMins
{U.C.1.4.2.3} ModifySecs
User

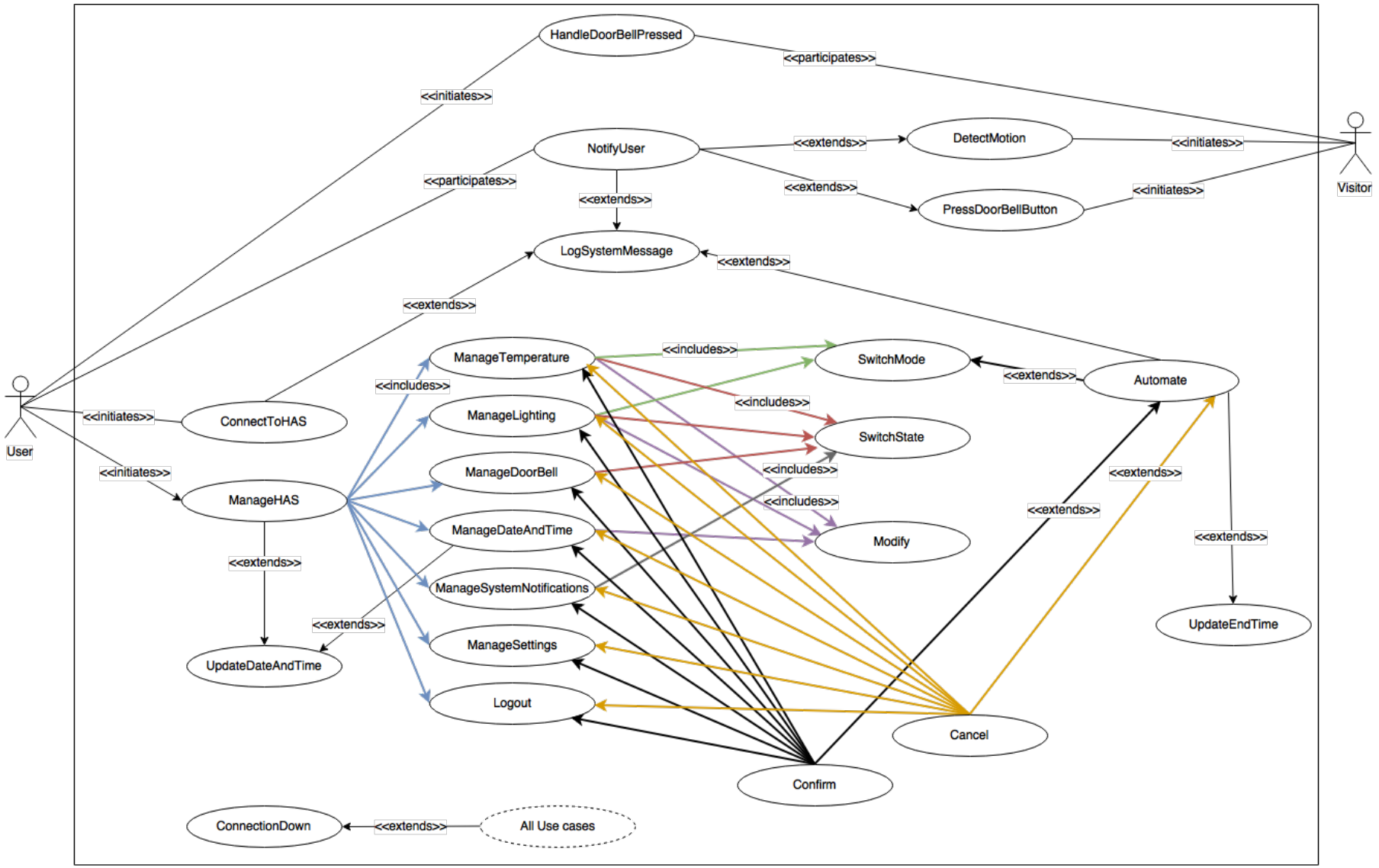


Fig. 70 - Refined primary Use cases model (ampliation of Fig. 8)

APPENDIX 3 – SEQUENCE DIAGRAM (USE CASE PRESSDOORBELLBUTTON)

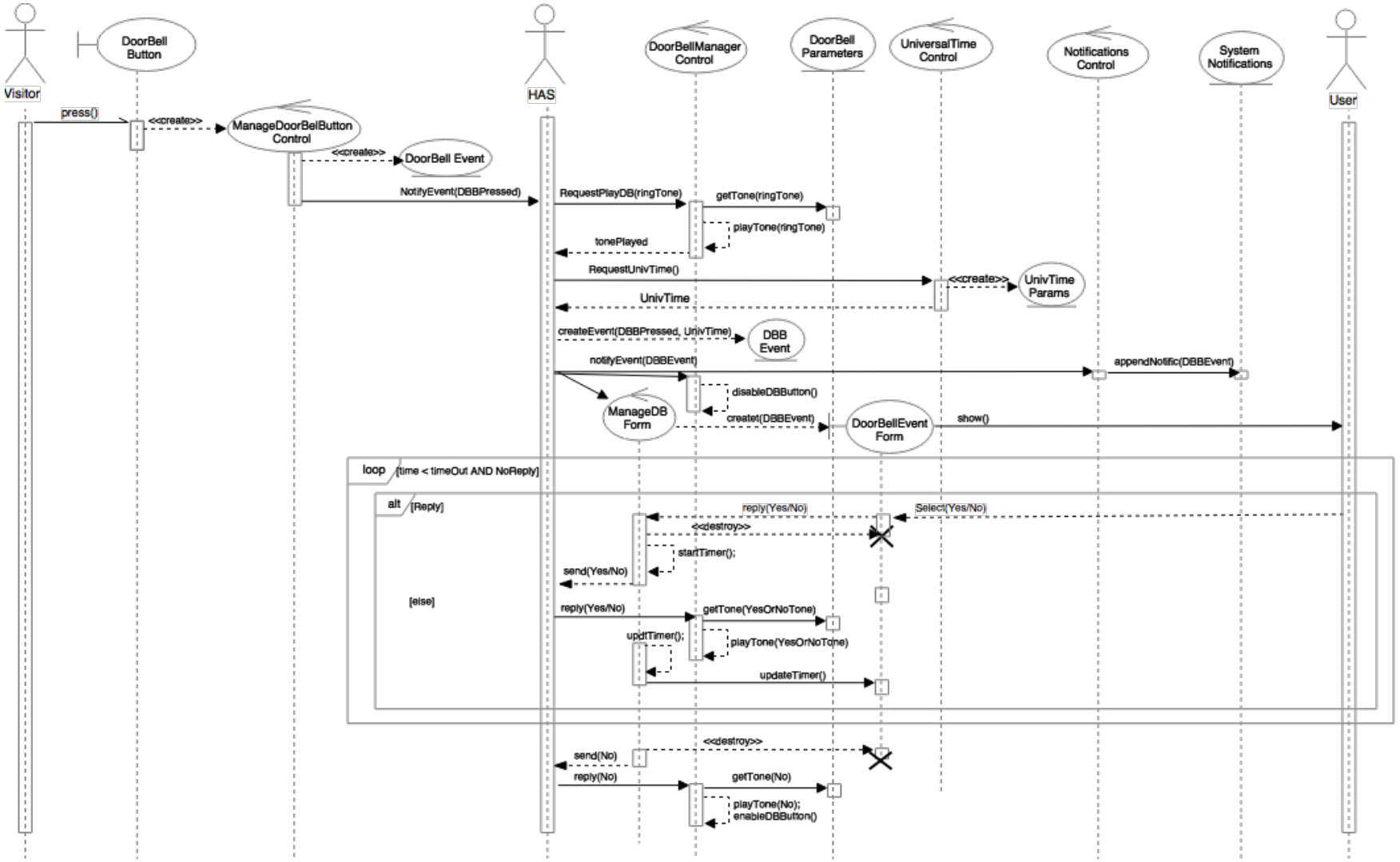


Fig. 71 - Sequence diagram for the use case PressDoorBellButton (zoomed-in version of Fig. 18)