Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg East
http://www.cs.aau.dk

Title:
Distributed and Fault-Tolerant Home Automation System

Project period:
SS10, Spring Semester 2013

Project group:

Members:

Donatas Poteliūnas

Tihomir Georgiev

Sami Zahran

Supervisor:
Associate Professor Dr. Arne Skou

Pages: 109

Finished: 11 June 2013

Abstract:

The field of Home Automation Systems has been researched for a long time and it still receives a lot of focus today. It is believed that it will constitute a main part of the home of the future. Most of the current systems look at the problem from a local point of view and place basic functionality as the first priority. While a very comprehensible choice, these systems are however very vulnerable to failure problems and do not scale very well. Our goal is to create a solution for a fault-tolerant and distributed home automation system. We will focus on rendering failures transparent to users so they will not have to intervene themselves on the system. On the other hand, our completely distributed system will be able to scale on a large number of users and areas to address the problem of the deployment of very large home automation systems.The industry can leverage this solution as well. We will not limit ourselves to a theoretical solution but will also provide a functioning prototype to demonstrate the result of our work.

Contents

1 Introduction

1.1 Overview of Domotics

General introduction

Home automation or the so called domotics refers to the automation of the home and some of its domains, such as consumer electronics, lightning, heating and so on. The aims of a domotic system are, among other things, providing increased comfort and energy preservation to a household. Usually high prices and vendor specific incompatible home automation devices are among the most common reasons why the home automation domain has not been massively adopted on a global scale. In our previous report called "Seamless Integration of Devices into HomePort", we have described all of the benefits and drawbacks of using a home automation system, which we are going to discuss about in the next part. Below we are listing the benefits and disadvantages of the current home automation systems cited from our previous semester project [18].

Benefits

By considering that most of contemporary houses in industrialized countries possess facilities like electrical power, telephones, TV sets, etc, one can easily see how big the benefits would have been if there was an automatic system able to control all these things at the same time in an integrated manner. Through a few examples, we will actually demonstrate real possibilities and advantages of such systems. The simplest example would be simple actions such as a remote controller for lighting or heating. These are actually quite popular and used, especially in places like hotels or modern office buildings. A level higher would be relationship between different elements of the home automation. Appliances sending notifications one way or another when a task is finished or there is a need to take actions or decision concerning them. One of the most classical examples would be intelligent rooms, which would be able to sense if a person is present in the room and set the light on or off depending on the information gained from the sensors. Another similar but more sophisticated example would be that the room would have (through the data coming from several different sensors) enough information to know who exactly the person in the room is, enabling possible settings such as appropriate lighting, temperature, music levels, TV channels, turning on specific appliances or even greeting the person and asking for vocal input of setting preferences. Another possibility would be to use blinking lights to alert about a fire detected by a smoke detector, or some changes like stopping the TV or the radio to render the user more aware of the situation. Other possible interests would be in energy saving, for example using sensors to detect human presence and adjust

light, heating, ventilation level, etc. Applying such a method widely could result in enormous financial savings in places such as large urban areas.

Drawbacks of current home automation systems

According to [4], there are four great barriers that need to be overcome before home automation might become widely adopted:

- High cost of ownership. Even If one uses the cheapest hardware and builds the system on his/her own and thus reducing the expenses (a few hundred dollars), this still implies a high cost in time invested and a need for great expertise. For people installing a home automation system by themselves, although a lot of time has to be spend to implement it, most of the time cost are still not negligible: from a few thousand dollars to tens of thousands dollars. As for people using an outsourced service from a company, prices are even more important. Most of the time it never goes under ten thousand dollars, going as high as more than a hundred thousand dollars. From this, we can deduce that today's home automated systems are clearly not accessible to the largest part of the population.

- Inflexibility. The market of the home automation system's hardware is comprised of a large number of brands and vendors. The main problem is that while this large number of different devices only covers one specific part of the home automation (e.g. the lighting system, the media system), they are not made to be integrated easily with other systems. From that, it is easier most of the time to have several independent systems, each for one specific use, instead of a single integrated system managing everything. Current systems also provide either monitoring or control functionality, but not both. In this case a user again needs to buy and install systems from two different companies. It should be noted that most existing systems are proprietary and closed. This implies that they are not very customizable and the user is constrained to use it the way it has been thought of, without taking into account case to case users and households. Another problem would be the need of structural changes in the household necessary for installing hardware and wires used for the home automation.

- Poor Manageability It is quite difficult to make hard rules of automation because people and users change their preferences over time. Moreover, the system could have more then one user with unique needs. The music they programmed to be played everyday during the time they are taking a shower might become a bother on a day when the user isn't in the mood to listen to music or if he takes his shower at a different time from the supposed one. It

is also hard to integrate guests into the system and most of the time guests or casual users are afraid of a system they don't understand and don't want to come close to it in fear of disturbing the system. Although it isn't a specific problem to domotics, bugs are one of the reasons why home automation is unreliable, because it is harder to correct errors in this domain. Complex user interfaces and the necessity to have a knowledgeable consultant available are also part of the poor manageability of these systems.

- Security. One of the most interesting features of home automation is the possibility of remote control of the system. But this poses a big security issue because software and Internet processes are hackable. Most of the time users are afraid of using their system to automate door locks and cameras since they are highly security sensitive aspects. Giving temporary access to other people, for example nurses or grandparents, is also difficult with the current existing systems.

Aside from these surface problems, some deeper ones remain. The problems arising from multi-users for one home automation system are to be taken into consideration. Since the system contains more than one set of preferences, scripting conflicts and ambiguities in expected behaviour will arise [28]. A different issue would be what is called the "Good Mom" syndrome. The concept is that members of the household have specific jobs that give them the feeling that they are useful, productive and valuable within the family. Thus, they might not want a simple electronic system taking away their jobs and values from them [51]. Finally, some researchers argue that even solving the above-mentioned problems wouldn't be enough as they believe that there are no compelling use cases for domotics. To be clearer, it is the fact that most people don't really mind getting up and turning a switch off manually instead of using an automated system or a remote controller device.
On top of all of these drawbacks listed above, we can say that most of the available systems are not capable of integrating new devices after deployment. Moreover most of them do not provide a mechanism for safe and secure execution of each system.

Because of all of these drawbacks, home automation middleware systems have been developed, which solve the interoperability issues between vendor specific protocols. Moreover, these middleware systems provide additional benefits to the end users, as they eliminate the binding of a user to a specific vendor. This enriches users' choice on devices, as they can purchase devices from different vendors, install and configure them through their home middleware system.
Cited from our previous project [18], we have listed the benefits and disadvantages of current home automation systems. Now we are going to further elaborate on

the disadvantages of the centralised home automation systems.

Currently, most of the existing home automation middleware systems are centralised [25] [7] [54]. The ones that could be deployed in distributed fashion [57] [21] do not provide fault tolerance mechanisms. On one hand, existing centralized systems imply single point of failure. What is more, centralized systems suffer from scalability issues. On the other hand, the distributed solutions available on the market can only be deployed on the local network and what is more important, they do not provide any fault-tolerance mechanisms for dealing with unpredicted phenomena and situations. Furthermore, these distributed solutions do not point out the principle of their operation in a distributed environment and how they coordinate their actions in order to shape up an uniform business process. This reduces the scope of home automation systems which can be used only in environments without any critical requirements. Currently, industrial companies are investing solid amounts of money in order to implement solutions that can meet their needs. Such needs usually are related to the automation of the entire industrial production process and the coordination of software applications running on some personal computers and embedded devices. Because a lot of aspects are highly critical in these industrial applications, they require solid solutions which can deal with extreme situations. One category of these industrial systems is the Supervisory Control And Data Acquisition (SCADA) [55] systems. Such a SCADA solution usually is quite expensive for the companies to implement, and some companies invest enormous amounts of money in order to implement it. Home automation middleware solutions could potentially be thought of substitutes to SCADA systems, only if they could prove to be a solid alternative. In order to do so, they need to be capable of providing fault-tolerant solutions which will firstly trigger correct outputs based on their logic and secondly would be capable of providing hardware fault-tolerance. This is why, a new approach to interoperability is needed, which can provide a cheaper, flexible, fault-tolerant and customisable solution through high level policies, which could also potentially be used in other domains than the home automation domain, such as the industry as we have already described in the use case of substituting SCADA systems with a home automation system.

1.2 Objectives

The goal of our project is to develop a distributed home automation system which will provide mechanism for participation of potentially isolated physical subsystems. These subsystems could become isolated due to the failure of an intermediate device a.k.a. as bridge in the world of home automation, or home automation middleware. Furthermore, distributed home automation middleware will enable the inclusion of new services in the home automation on demand regardless of scala-

bility concerns and also will provide a mechanism for execution of certain business logic in a fault-tolerant way. Here are the main objectives of distributing home automation middleware:

- To make home automation systems more reliable. In order to do this, we will implement fault-tolerance features in home automation systems. These are already known in distributed systems but we will need to apply these to domotic systems. We can apply software fault-tolerance techniques together with hardware fault-tolerance techniques, in order to build robust distributed home automation systems providing high level of reliability and extensibility.
- To make it scalable. We will distribute the load of the system among participating nodes, which is not possible in centralised systems. In this sense, a node is an entity which can participate in a home automation middleware by communicating with other nodes.
- Find a suitable architecture for a distributed home automation system. A lot of distributed architectures exist currently. We will need to find something which could fit into the concept of home automation system and adapt it correctly.
- Address security issues. We want to avoid third party malicious entities to be able to connect to our distributed home automation system, read or tamper information sent and received by the nodes within the network.
- Orchestrating all the nodes participating in the distributed home automation system, such that they can form a whole business process.

1.3 Motivation

A lot of effort has been put on adding interoperability between proprietary protocols operating in different home automation systems manufactured by different vendors. This has been done for the sake of reducing the overall cost of installation of a home automation system, increase the ease of use of such a system, and providing the end user with rich collection of home automation devices from different vendors. Thus the end user could afford to buy and set up on its own or with little help of averagely skilled professional a home automation system and customise its behaviour. Most of these systems have been designed and developed just for providing interoperability between components at home, and others have more features, such as an engine executing certain constraints or a state machine, in order to automate the home through a home automation specific language. Some of the home automation inter-operable systems present on the market today are AMIGO [17], DomoNet [60], DomoEsi [25], HomePort [24], Extendible

Protocol Independent unit Controller (EPIC) [54] and Open Services Gateway initiative (OSGi) [7] . These systems can connect subsystems operating different proprietary or open source protocols. Some of these protocols are Universal Plug and Play (UPnP) [21], LonWork , X-10 [66], ZigBee [9], Zwave [10], UPB [59], etc. Even though such systems bring a lot of advantages, they bring with them their shortcomings correspondingly. DomoNet, DomoEsi and AMIGO, HomePort, EPIC, OSGi all have centralised components. The only exception to the centralised approach is HomePort [24], but in its latest release [57], it is has a centralised component as well. In this case the centralised component could yield the following negative implications:

- Single Point of Failure - If the middleware system fails, this brings down the communication and the overall interaction between the different subsystems. The accessibility of these devices via the central component is becoming practically impossible. Even in case of a distributed solution such as the HomePort [57] , a failure in one instance of the middleware node could isolate its associated physical subsystems.

- Lack of Fault Tolerance - A fault tolerance mechanism related to middleware failures is not supported, since there is only one central component. Furthermore, some of these systems, such as HomePort and EPIC, provide a run-time engine for execution of a home automation logic. This means that if a node or the middleware (if centralised) does not execute the composition logic correctly, this might not trigger the correct output, or even worse, it can trigger an inappropriate actuator at an inappropriate time. For instance, an alarm system could be turned off while a family is enjoying their holiday or a burner could be switched on while there is nobody in the house, which sometimes might have disastrous consequences. Or if such a system is to be used in the industrial or medical world, it can have even worse consequences for the intended users as it could even lead to a disaster.

- Potential performance bottleneck - Centralised solutions, such as DomoEsi and OSGi, can not scale up to provide a high degree of Quality of Service (QoS) to the end user. For instance, let us imagine a scenario where such a system is deployed in a huge corporation with thousands of devices simultaneously running. If these devices are to be accessed very frequently by a large number of users, the centralised component can slow down the access to a certain device.

- Scalability issue - Such a centralised approach, evident in some of the already mentioned home automation middleware systems, can bound the maximum number of devices that can participate in the system.

Based on all of these shortcomings of a centralised approach in the home automation middleware systems, it is apparent that a decentralised solution would fit better in a home automation interoperability middleware system. Our motivation is to design and build a distributed home automation middleware system, which will be capable of dealing with all of the above mentioned issues related to a centralised approach and provide the end user with a scalable, fault-tolerant inter-operable home-automation middleware system.

1.4 Example

Let us consider few situations where the need for a fault-tolerant distributed home automation system would arise. First is a situation where a home automation system would be a critical part of a global system. This could be the case in the medical field for example, where it would not be acceptable for the system to completely shut down, be it due to a software problem or a hardware problem. This implies that if the central control part of the system fails, there would be a mean to recover from the failure without having to shut down anything and without isolating a part of this home automation system. Let us assume that a medical system is using subsystems manufactured by different vendors and their own specific vendor communication protocols are interfaced through a home automation middleware system. If the system is centralised, a fault in the middleware could isolate the different subsystems, and thus they would not be able to communicate and turn on some important actuator in case of change of state of a certain device. Even in the case of a distributed home automation middleware system, the consequences would be slightly more favourable, but there still would be at least one vendor specific subsystem isolated from the rest. While in a regular household, this could not lead to very severe consequences, in the industrial world, this is not acceptable. This is where the fault tolerant part of the system would be necessary to let the isolated device participate to another node of the system, without affecting the normal operation of the rest of the subsystems.
Let us take the same example of a medical system running different vendor specific subsystems interfaced and connected through a home automation middleware. In this example we will not distinguish between centralised and distributed solutions, since the example considers a scenario where both are behaving pretty much in the same way. Let us imagine that the already mentioned home automation middleware has the capability to execute a certain logic, applied over the entire system. In this case, if a hardware failure occurs in the node executing the logic, it could trigger the incorrect actuator and in the medical or industry worlds this is not acceptable. Now let us look at another example. In that case, the home automation is comprised of a very large system with a considerable amount of nodes. Only one central control part would not be able to handle the large number of nodes.

The system would need to possess several control parts, this is where a distributed system would become interesting. Instead of relying on a very powerful central part, the fact of distributing the control system would allow to reduce the load on each of them and even distribute the load depending on the situation. In case of failure of one central part, another one could take on the load of the failed one for the system to continue to function.

1.5 Hypothesis

All of the negative implications coming from a centralised solution or coming from the distributed solutions available on the market have led us to think about a distributed solution to address the home automation interoperability issue and provide fault-tolerance. By developing a fault-tolerant distributed home automation system, we can provide a scalable, fault-tolerant and self-configurable solution to the home automation interoperability issue. Firstly, we claim that a distributed home automation middleware system can provide a very flexible solution, where the user can join and release home automation middleware nodes on demand. Secondly, the distribution of the middleware can provide a foundation for the development of a mechanism that will mask or eliminate a potential hardware failure or software error in one of the distributed nodes. Thirdly, providing a distributed mechanism for fault-tolerance and removing the upper bound of potential devices that can be added to a home automation middleware system can change the intended use of these existing systems which can become applicable even in the industry. And last but not least, implementing hardware and software fault-tolerance mechanism over a distributed home automation system could even change the intended use of such systems and allow them to be used for instance in the industry or in the medical world.

1.6 Methodology

Our goal is to provide a solution for distributing a home automation system and rendering it fault-tolerant. We will first answer the question about the interest of this subject. After this, we are going to put together the different questions we need to answer during this study. First of all, is it possible to distribute a home automation system and can we make it fault-tolerant? Then, what are the best existing methods to do this currently? Finally, how can we use what we learned to adapt it to a home automation system?
To address these problems, our methodology will be the following:

- We will first start by introducing the subject of home automation, its interest

and our goal in more details.

- In a second step, we will analyse works related to our subject. We will search for other similar projects or projects having some common points with our study, analyse their research, the techniques they used and their results.
- From that point, we would get a better overview over the state of the art related to our subject. That would allow us to define clearly our requirements for accomplishing our goal.
- After having determined our requirements, we would have to study the current existing technologies, techniques and methods related to our subjects. Studying this will enable us to determine what is interesting, and what is really possible in the concept of home automation systems, and how the strengths of distributed systems could be accommodated in the context of home automation.
- After these global theoretical studies, we would get into a more practical part of our work where we would design our solution, including all of what we learned before based on the analysis and the related work.
- From this design, we would be able to start implementing a prototype of our distributed home automation system. We will describe and document our implementation in detail to make it as open and easy to use as possible.
- This prototype would let us do some experiments to confirm that we are meeting our requirements.
- Finally, after concluding that our solution meets our goal, we will make a general conclusion about our work, its result and our contribution to the home automation domain.

We have already described the reasons why the home automation interoperability middlewares have not been intended to be used in a domain different than the home automation and we have already described what needs to be done in order to suit them in domains where they initially have not been intended to be used in. In our Master's Thesis, we will analyse what needs to be done in order to suit the home automation systems into other domains and we will provide a design and an open source implementation of such a system. The report is organised as follows: In the next chapter 2, we shall proceed by describing the works related to our project available in the world of home automation. Next, we shall set the necessary requirements 3 in order to develop a fault-tolerant distributed inter-operable middleware. In 4, we will analyse existing fault-tolerance techniques and means for

distribution of the system. 5 captures the design of our system. 6 documents implementation details about our project. Finally, we will perform some experiments 7 and make a conclusion 8 about our project.

2 Related work

In this section, we will evaluate and compare home automation middleware systems available on the market or developed by non-profit organisations. The section will elaborate on each of them individually and it will summarise their advantages and corresponding shortcomings.

2.1 UPnP

The UPnP [21] or the Universal Plug and Play is an architecture for adding interoperability between devices manufactured by different vendors compliant to the UPnP standard. The UPnP follows peer-to-peer distributed architectural pattern and works over IP, so each UPnP device needs to be capable of communication over TCP/IP protocol. Furthermore, each device needs to run a Dynamic Host Control Protocol (DHCP) client and upon start-up needs to look for a DHCP server. If a DHCP server isn't present on the network, the device alternatively must use a technology called AUTOIP, which is a server-less method for enabling devices to obtain a unique IP address within a local network. Basically, the standard consists in exchange of messages between UPnP peers taking part of the network.
In order to produce devices compliant to the UPnP standard, the vendors need to implement the so called UPnP Device Architecture, consisting of the following communication protocol stack depicted in figure 1:

- UPnP Vendor - These are messages having vendor specific information.
- UPnP Forum and UPnP Device Architecture - Envelops the vendor specific message with wrapper defined by the UPnP forum.
- Simple Service Discovery Protocol (SSDP), Simple Object Access Protocol (SOAP), General Event Notification Architecture (GENA). Each UPnP device can have multiple services available and a state domain. SSDP defines a procedure for device's service discovery. SOAP is chosen as the protocol for exchanging device specific information in a structured way. Thus it is readable to other UPnP devices. GENA's role in this picture is to provide a mechanism for event-firing and event-notification of interested devices. Usually the devices notified about a fired event are the so called control points, which are UPnP devices capable of getting the state of a certain device in a UPnP network.
- UDP/TCP, IP, HTTP - HTTP is used as a data transfer format. Then the message is wrapped in User Datagram Protocol datagrams or TCP segments and finally the message is chopped and wrapped into IP packets and is sent over the IP network.

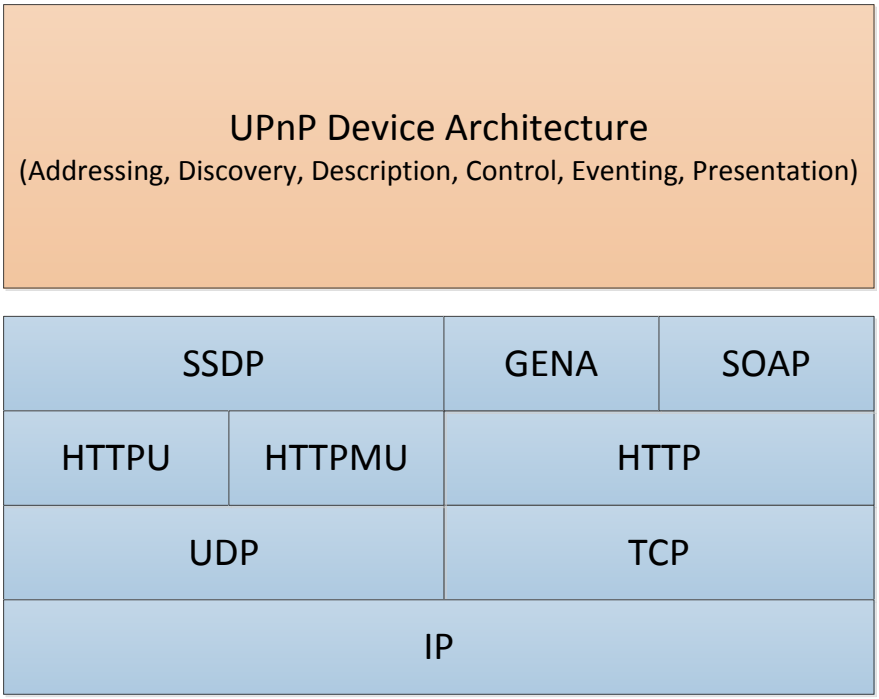

Figure 1: UPnP Protocol Stack

Devices in the UPnP architecture can be controlled devices or control points. Now let's describe how the interaction between a controlled device and a control point is conducted and what are the sequence of actions that are performed by the parties for a device to be attached in order to participate in a UPnP network and then manipulated by a control point.

- Upon boot-up of a UPnP device A, it is assigned an unique IP address.
- If A is a controlled device, it advertises its available services over the network or if A is a control point, it searches for services available on the network. In this message, a URL is provided to the control point.
- This URL is used by a control point device in order to acquire relevant information about a device's actions and feasible states. Other relevant information included in the device's description is expressed in XML, which might be a description of other embedded devices contained by the controlled UPnP device, vendor ID, product ID, etc.
- Then a control point is up and ready to send actions over SOAP to the

controlled devices. An action forces changes in the local state of the device and the device in response sends the result back to the control point.

- The devices can send events to subscribed control points.
- The control point is capable of retrieving a device's User Interface modelled as a web page. Of course, the device itself needs to provide the user interface presentation to the controlled point.

The UPnP supports only devices that have IP capabilities on board. However, most of the home automation devices do not have any IP capabilities, because they usually are devices with very restricted resources. In this case, these home automation devices require an additional bridge component which can add the IP capabilities and implements the UPnP stack. Now let's see how UPnP serves as a base to some of the existing inter-operable middleware.

2.2 AMIGO

The first of systems that we are going to discuss about and leverage the UPnP standard is called AMIGO [17]. It stands for Ambient Intelligence for the Networked Home Environment. AMIGO is a middleware trying to link different types of devices at home in one large network. It is a very general piece of software enabling networked devices to inter-operate with each other. The devices can be from different domains such as home automation, consumer electronics, personal computing and so on. AMIGO consists of the parts depicted in figure 2:

- Service Discovery Protocol (SDP) Detection and Interoperability (SDI) - This component makes it possible for multi service advertising and access, regardless of the service discovery procedure used by the network services.
- Service Interaction Interoperability - Enables interaction between services regardless of the broker advertising it. In this sense, a broker is a device that advertises services compatible with specific protocols.

In a sense, the AMIGO inter-operable middleware is a middleware for enabling interaction between services advertised by different brokers. Considering the home automation domain, it does not provide solution to it. In the paper presenting AMIGO, an example is given, where the home automation domain is participated in the AMIGO network through implementation of an UPnP bridge.
In any case, in the AMIGO architecture there are two centralised components which are the AMIGO core and the potential bridge between home automation domain and UPnP. Thus the system could experience scalability problems and on top of that if one of the centralised components fails, the devices attached to it will remain isolated from each other.

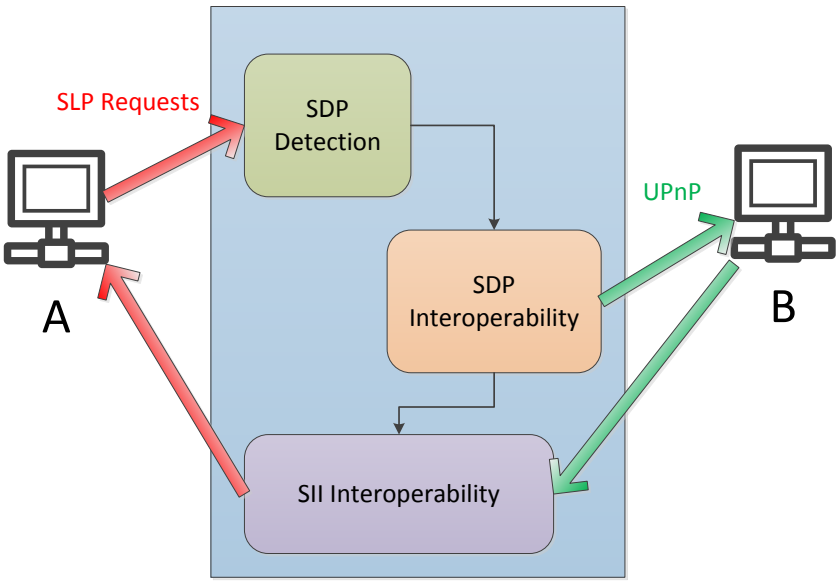

Figure 2: The Amigo interoperable middleware core

2.3 DomoNet

Yet another architecture presented is the architecture of DomoNet [60]. In general, the DomoNet architecture leverages the Service Oriented Architecture (SOA) to home automation. In order to use DomoNet, one would need to model each device as a web service and thus one could access it as simply as calling a web service. The components in the DomoNet architecture are having the so called TechManagers a.k.a. application gateways for different kind of protocols. Moreover, if N different protocols need to incorporated into the design of DomoNet, then only N TechManagers are required. The TechManagers are the components linking the domotic specific protocols to web services corresponding to the devices that need to be accessed. According to the design of DomoNet, the web services need to be installed on one or more web servers. Each of the TechManagers needs to register at all web services available on the home network. The web services in turn provide information about other TechManagers that have been registered at them. Thus all TechManagers can command any device, just by knowing what kind of web services they are advertising in the network.
Each device can be manipulated by a TechManager with the means of the DomoML language. It is an XML SOAP [62] based language, used for sharing messages be-

tween a TechManager and the device web services.
Now let's follow up with the sequence of operations, that are executed upon TechManager's boot-up:

- Get a list of all device web services available on the network using Universal Description, Discovery and Integration [63].
- Register with all devices of interest.
- Publish all devices, available at its own subsystem, as web services.
- Get all devices available in the registered device web services.

It is worth mentioning that each TechManager also makes sure that it creates virtual devices in the subsystems where it is required. For example, the UPnP architecture is distributed and each UPnP device is having its own IP address and port. This means that the UPnP TechManager needs to create virtual devices within the UPnP network of all other devices existing in the other subnetworks.
The DomoNet provides an interesting architecture for Domotic middleware.
Firstly, scalability issues are not present as more web servers can be added on demand.
Secondly, as it can be inferred from the description of DomoNet architecture, the single point of failure problem here is mostly overcome but the architecture does not provide any means for execution of run-time logic, which could potentially model the system as one big home network and set some constraints and relations to the components of all subnetworks.
Thirdly, as a run-time engine for the execution of the home automation logic is not present, software fault-tolerance is not applicable in this system.
And finally we can state that, as we previously mentioned, the single point of failure problem is mostly eliminated but not quite. If a web server hosting a web service's device crashes, this would isolate the web services and their corresponding devices from the home network. DomoNet does not provide any mechanism for re-participation of devices to a new TechManager, in case of a failure of their TechManager.

2.4 EPIC

EPIC stands for Extensible Protocol Independent unit Controller [54], and it is a project initiated by the Aarhus University, Denmark. The project relies on user driven innovation in its foundation. The aims of the projects are to achieve easy addition of new components from different subsystems manufactured by different vendors and running different protocols. The main idea behind this framework is

to provide the end user with an easy way of using home automation devices at affordable prices, which should be easy to install and configure. The developers of home automation devices can use this framework for easy development of new components.
The main components of the EPIC platform are visualized in figure 3:

- GUI - The GUI is implemented using Windows Presentation Foundation (WPF) [42] and Windows Communication Foundation (WCF) [41] technologies. All the underlying devices available at home are captured and modelled in a protocol independent way by the core and further propagated up to the GUI, which displays the available home facilities to the end user and their states.
- Core - The relations between devices are represented as tasks and evaluated upon physical devices' change of state. The core consists also of: a GUI manager which interacts with the GUI, a Device List and a Task Manager.
- Sensor System - The sensor system is similar to a DomoNet's TechManager. Its role is to bridge a subsystem's specific protocol to EPIC's core.

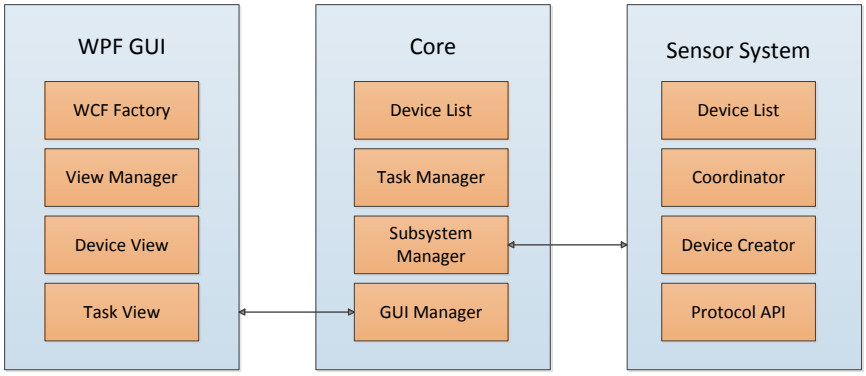

Figure 3: The EPIC system overview

The EPIC system is a home automation middleware system that provides interoperability between devices manufactured by different vendors and taking part of systems running vendor specific protocols. EPIC's target, as we said, is to provide the end user with a friendly way of setting up a home automation system. The system characterizes itself by its run-time task execution engine. In other words,

it provides a mechanism for defining a logic and composition logic execution upon device's state change event. However, the system is not capable of software-fault tolerant execution of the composition logic. The system could potentially have its core as a single point of failure, or its core could turn to be a performance bottleneck if the system is to be deployed in an industrial corporation having many devices and each of them is accessed more often than what the core component is capable of dealing with. The system has been designed with scalability and security in mind, as it decouples the middleware core and the sensor systems. This means, the sensor systems and the core could run on different physical machines. This eliminate scalability concerns that we take into consideration to some extent. Finally, the system does not provide any means for re-participation of devices that have previously been attached to a failed sensor system.

2.5 DomoEsi

DomoEsi[25] is a project initiated by the department of Automation and System Engineering at the University of Seville, Spain. The aims are providing a low cost home automation solution to the ordinary people, where one could set up a home automation system. The project is exclusively based on the UPnP standard. It basically provides a UPnP control point and software bridges between some existing vendor specific systems and UPnP standard. Thus it makes compatible some vendor specific systems to the UPnP standard. The project characterises itself with allowing the end user to install a low cost home automation system. The software bridges provided by the DomoEsi system to the UPnP standard include:

- X-10 [66] - The bridge is very flexible in its use, as it can work on its own, without the need of UPnP control point to control it.
- Infrared interface - This interface lets the user turn on and off a device through interaction with the GC-100 device [14], which is a device which can link consumer electronics and even alarm systems in a home network. Thus the GC-100 device can receive commands by a UPnP control point and resend these commands to a device that has an Infrared interface. This device can accept commands through its Infrared interface and these commands can be relayed to the UPnP network. Thus events can be fired in the UPnP network and the subscribed control points can handle these events.
- Nintendo Wiimote [43] - This remote controller device can be used for controlling devices running in the DomoEsi network. The device is interesting because there are multiple ways to control something with it, as it has ac-

celerometers. This can for example be used by handicapped people to easily control their home appliances.

- ZigBee [9] - It is an alliance of companies defining a set of standards for low cost wireless communication. The ZigBee compatible devices have been invented and applied in the health care, telecommunications, home automation domains and more. Furthermore, the ZigBee standard operates in the open 2.4 GHz frequency range and defines three types of nodes taking part in a wireless mesh network:
  - ZigBee Coordinator
  - ZigBee Router
  - ZigBee End-Device
- IPDomo - All of its components work under the UPnP standard and the system consists of a UPnP control point and cards interfaced through Ethernet UTP cables for controlling lights, videophone, alarm, etc. This is a system that is readily used in the DomoEsi as it is UPnP based. It allows a DomoEsi control point to control the above mentioned IPDomo cards and the other way around - the IPDomo control point can control the DomoEsi devices.
- Voice recognition - The user can enter voice commands to the system and this will raise UPnP events.

In short, the DomoEsi is a system which needs to run on a single computer and this approach, as previously mentioned, suffers from the single point of failure problem. Another disadvantage of DomoEsi is the lack of a composition logic execution engine. This means that software fault-tolerance techniques are not applicable. If the DomoEsi daemon fails, this will isolate some of the UPnP integrated components through the DomoEsi system.

2.6 OSGi

Standing for Open Services Gateway initiative [6] [7], OSGi is a Java based architecture, which aids software engineering activities by providing a framework for building loosely-coupled Java architectures. OSGi has emerged as a solution to the home automation vendor interoperability issue, but later has been adopted as a general framework for building component based software. In its nature OSGi represents a framework for building software systems with reusable components. Of course, as the volume of a software system grows, the number of component dependencies grows as well, so this model by itself does not guarantee the success

of a software project. Furthermore, these reusable components hide their implementation to the rest of the components available in a system and communicate with them based on service calls.
Before we go into architectural details about the OSGi architecture, let's first describe a couple of important components related to the Java platform, and to the OSGi framework:

- Java Archive (JAR) file - A file that puts together many Java classes and meta data associated with these classes. Important note about JAR files is that a JAR file is visible to all other JAR files.

- Bundle [23] - They are building units of OSGi based software system. They are dynamic components as they can be loaded and unloaded at any point in time. A bundle is just an ordinary JAR file and the bundle's manifest describes the bundle with attributes such as: version, Import and Export Java packages, bundle name, etc. OSGi JAR files are called bundles.

OSGi provides a layered architecture consisting of the following layers which can be seen on figure 4:

- Execution Environment - determines what classes are available to be used.

- Modules - In an ordinary Java program, a JAR file is visible to all other JAR files. The Modules' layer hides bundles, and bundles share only the components that they explicitly point that they want to be shared. On the other hand, modules layer imports automatically all needed components in a Bundle.

- Life Cycle - This layer takes care of dynamical addition, removal, activation and deactivation of bundles.

- Services - The services layer is facilitating the interoperability between bundles. In the service layer there is a component called *service registry*. When a bundle is loaded, it registers its services within the service registry. A bundle can also get services from the service registries, based on some interface or even listen for registration of services.

The OSGi architecture has emerged as a centralised system, where the bundles are running in the same virtual machine. According to the latest specification [8], OSGI could distribute the services and use them as they were on the same machine with the help of newly introduced *distribution provider* component. The distribution provider is capable of creating the so called *proxy* which can import a service hosted on a remote OSGi instance. The distributed provider of the remote host

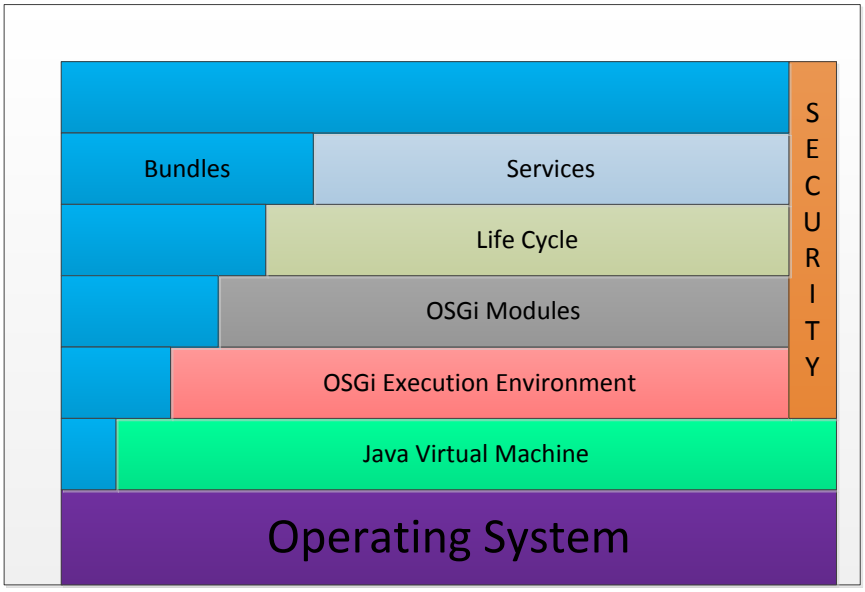

Figure 4: OSGi Architecture

in turn must create the so called *endpoint* in order to be capable of exporting services. The endpoint describes the local services that need to be invoked remotely through WSDL.The endpoint and the proxy wrap each call as SOAP messages according to the WSDL description of the service to be invoked.This leverages the Remote Method Invocation (RMI) [45] inter-process communication approach.
To sum-up, OSGi provides a dynamic system which finds its role as a home automation middleware. If all vendors were to build translation bridges as OSGi bundles, then the vendor specific interoperability problems would be eliminated. However, the traditional OSGi architecture suffers from a single point of failure issue. On the other hand, the distributed OSGi architecture eliminates this problem to some extend. However, there are certain issues associated with the Distributed OSGi. First of all, there is no mechanism for automatic re-participation and auto-recovery of a set of OSGi frameworks, i.e. in this system it is not clarified how one could set-up a distributed home environment, where all devices could participate in one big home network and new framework nodes can be participated on demand. Secondly, the OSGi platform does not provide a run-time composition logic execution engine. This implies that a software fault-tolerance mechanism for redundant execution of the composition logic is not applicable and this imposes a risk of unintentionally triggering an incorrect actuator at home. Furthermore, if a bundle being a bridge to a subsystem experiences a permanent fault, the subsys-

tem will remain isolated without human intervention. Even though a distributed implementation of OSGi exists, this distributed architecture is still centralised. If the centralised OSGi component fails or even if a distributed bundle fails, this could isolate bundle's corresponding devices, or isolate all the devices from each other.

2.7 Jini

Jini [52] is a very interesting networking concept. The goal is to push some of the OS functionalities onto the Jini network while being as transparent as possible for the user.
With Jini, the user is able to plug any device (printer, scanner, etc) into the network and Jini will take care to inform any other device in the network of the presence of a new element. This new element will also immediately be ready for use. This is possible because Jini stores each device's driver within its network. When a machine needs to use a specific device within the network, the driver of this device will automatically be downloaded to the machine. This means that the OS of the machines connected with the Jini network do not need to possess drivers for external devices anymore.
The network also allows for interoperability between operating systems: a new device within the network will be shared and accessible from any operating system.
Jini is divided into four different parts:

- Directory Service.
- JavaSpace.
- Remote Method Invocation (RMI).
- Boot, Join and Discover Protocol.

The Discovery Service is used to register new devices within the network. As it can be seen in the list before, Jini uses Java, which means that the devices that wish to be connected to the Jini network need to support Java. Objects of devices are placed in the JavaSpace and accessed with the Remote Method Invocation. The Boot, Join and Discover Protocol is used by devices and users to advertise themselves within the network which allows them to then register themselves.
Jini was a very brilliant idea and it worked very well. However, it was never widely adopted because of some elements like oppressive licensing, the fact that is was completely based on Java and RMI and its complexity. It has also a small issue of scalability because it was a centralized model although it doesn't appear so. That is because one of the components, the lookup service, which handles the

communication between the service and the client, is a centralized element and so do not scale. However the communications themselves are certainly decentralized. Within the scope of our project, Jini could become really useful with the fact that it stores each device's driver within the network. In case of a node failure in our network, we could make use of this functionality to rapidly transfer all devices attached to the failed node to another node very easily. This would be an interesting choice for the fault-tolerance part in our solution, while Jini also provides some distributed properties, like its communication part.

2.8 HomePort

HomePort is yet another home automation middleware solving the "vendor protocol heterogeneity" issue. The first version of HomePort [24] gives an overview of the system and its architecture, and its second version [57] slightly differs from the first one as it provides more details about the design and the different parts in it.
In the design of the first version of HomePort which can be seen on figure: 5, there were four layers:

- Device layer - This layer is device specific and the other Layers in HomePort do not make any assumptions about it. More specifically, this is the layer which withholds the physical devices themselves.

- Bridge layer - This layer provides a link between the vendor specific devices and the upper layers. In other words, it converts vendors specific commands into HomePort understandable commands and the other way around. A physical device is accessible via the bridge layer over IP.

- Service layer - This layer acts as a gateway for a number of bridges. Its responsibility is to play a web server role, to model the devices and expose them to the Internet through HTTP using a Restful architecture. Each physical device is modelled as a web resource using the HTTP protocol. It can be manipulated by performing a HTTP PUT request on it or its state can be checked by performing a HTTP GET request. For example one can perform a PUT request and can turn the heating 30 minutes before it enters the room and thus the room will already be warm when he or she enters their front door.

- Composition Layer - In this layer the home automation devices are grouped using a composition specific language to represent the full set of devices in a household. Furthermore, some sort of dependencies and constraints could be applied and this could aid the automation of the modern home.

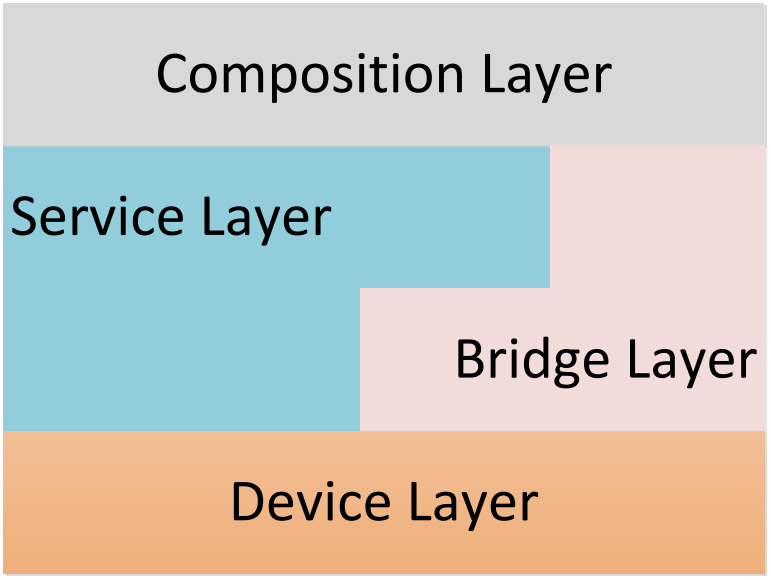

Figure 5: Initial HomePort Architecture

It is worth mentioning that this architecture could be further fine tuned if needed. It can work either without a bridge layer or without a service layer. In the former scenario, the gateway (running the service layer) could "talk" directly to the physical devices. In the latter scenario, the bridge could be removed from the architecture and the service discovery mechanism would need to be incorporated. The second version of the HomePort [57] has a slightly different design and all the functionalities mentioned in this report have been implemented. Furthermore, all the details about how goals are accomplished are fairly well described and the project implementation also has an open source repository [58], where more insight could be gained.
The latest design of the HomePort project can be seen on figure: 6. It consists in the following modules:

- Adapters module - The goal of the adapters module is to provide a bidirectional mapping between vendor specific protocols and the HomePort core. For each vendor specific subsystem, there is a corresponding adapter which handles the communication and translation between the HomePort itself and the vendor specific subsystems. This module corresponds to the bridge layer in the initial HomePort architecture. However, there are some differences between them such as that the adapter module is intended to be run on the same physical machine. The bridge layer in turn is accessible via the

TCP/IP protocol stack and it could be run on a different machine. Another difference is that the adapter layer is mandatory while the bridge layer is optional and could be eliminated in some instances.

- Configuration module. It allows the user to find some of HomePort parameters.
- Events module - It takes care of notifying interested parties upon a change in a device's or service's state. The current implementation exhibits from a W3C Server Send Events [61] draft. An interesting observation about this module is that it can notify even a HomePort client upon a service's state change.
- Local Service Discovery - This module makes possible the advertisement and discovery of services in the local network. Two different discovery protocols are available on this module and namely these are the DNS Service Discovery (DNS-SD) and the Simple Service Discovery Protocol (SSDP).The use of DNS-SD means that HomePort can discover the ZeroConf described services on the local network. It can also discover the services available on the local network and participate them into the services available on HomePort. UPnP supports ZeroConf compatible devices as well.
- Web Server - This module makes it possible for clients to get and modify device states out of the local network through the HTTP protocol.
- HomePort Services - These are all services contained in HomePort daemon which are advertised on the local network and can also be accessed through the web server. Each service is described with a unique name, ID, vendor ID, device ID associated with the service, etc.
- Access Control - This module consists of a look-up table withholding information for different clients about their corresponding rights and restriction in terms of getting's and setting's service's state.
- Log module - This module is present on the system for the sake of providing the application developer with the necessary information in order to debug the application. The log information can also be accessed through the web browser, but the use of a secure connection is recommended if data in it is considered to be sensitive.

In the latest release of HomePort as well as in its initial release the single point of failure problem has been addressed. In the latest release of HomePort, one could instantiate multiple HomePort instances which can work together if run in the

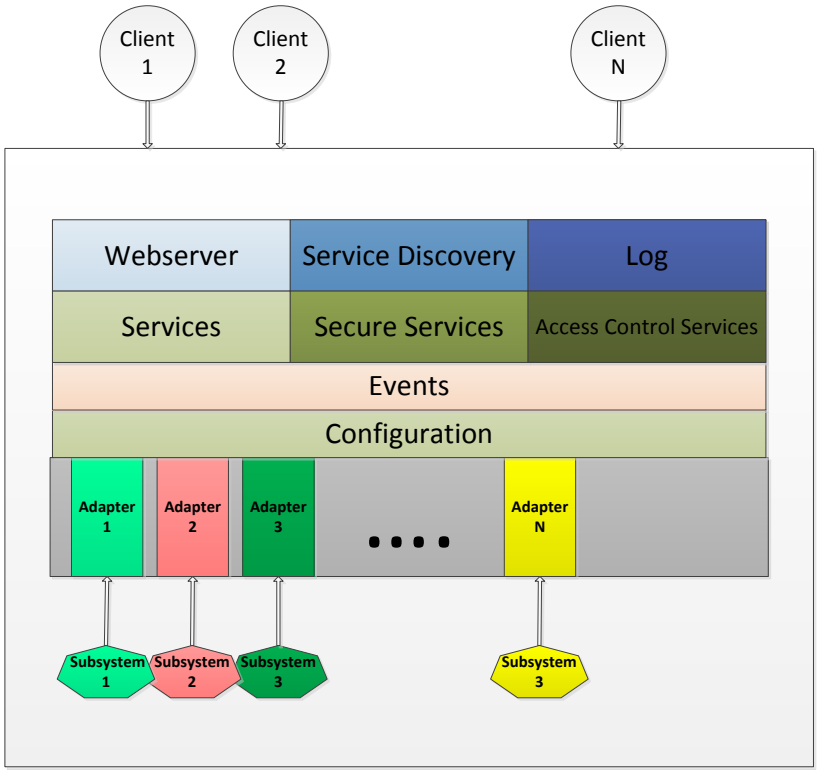

Figure 6: Latest HomePort Architecture

same local network. However, neither the initial HomePort architecture nor the latest one addresses the issues of the re-participation of devices in case an instance of HomePort fails. In this case the devices interfaced through the failed instance of HomePort will be practically isolated from other devices connected to other HomePort instances run in the same local network. Furthermore, none of them provide any mechanism for software fault tolerance. By software fault tolerance in this sense, we mean a mechanism for redundant execution of composition logic. Of course, as it can be seen from the latest release of the HomePort architecture, the composition logic is not mentioned as it was in its initial architecture. Nevertheless, we give the deserved credit to the authors, as such a module is currently being under development.

2.9 SCADA

Standing for Supervisory Control And Data Acquisition, SCADA [55] is a widely spread industrialised used architecture for monitoring remote objects through sensors and triggering remote actuators. Moreover, a SCADA system gathers data from sensors and presents these data to an operator through a Human Machine Interface (HMI). Sometimes some SCADA deployments could even take some actions according to the state of sensed objects. These data are presented in a comprehensive form to the operator, depending on the meaning and the interpretation of the sensing data.
A typical SCADA system consists of the following components which are shown in figure 7:

- Main Terminal Unit (MTU) [15] - This is the "heart" and the "main brain" of the SCADA system. This component also acts as a central repository for the sensed data from the micro-controllers

- Remote Terminal (Telemetry) Unit (RTU)/ Programmable Logic Controller (PLC) - A SCADA system typically has several of these components, which in a sense act as the "arms" and the "legs" of a SCADA system.The PLCs and the RTUs are mutually replaceable components and they need to follow the set of industry standards according to whom they communicate through potentially diverse media with the MTU.

  - PLC [19] - These devices are micro-controllers, having multiple sensors, actuators and Real-Time Operating System (RTOS) [12]. The RTOS executes the simplest type of task scheduling and in particular the execution of the so called ladder logic in an infinite loop. In the beginning of each iteration, a procedure called sanity check is performed in order to examine all the available hardware for potential hardware faults. If the sanity check passes, then the states of I/Os are examined and finally the ladder logic is executed.

  - RTU [65] - There are two subdivisions of RTU: Remote Telemetry Unit and Remote Terminal Unit. The former kind of RTU is a very simple I/O multiplexed device with almost no computation abilities and is used as a slave device capable of interfacing its sensors and actuators to a remote master controller, which determines the logic of the particular application. The latter kind of RTU is a more flexible device, as it can interface the I/Os to a controller, execute some local logic and triggers some actuators based on the local logic. What is more, such kind of devices can examine I/Os through interrupts.

All of these devices are interchangeable and the choice of which one to use in a certain case depends on what one wants to achieve.

- Communication infrastructure - This component determines the underlying physical medium and the protocol used for communication between a RTU/-PLC and the MTU.

- Human Machine Interface - It corresponds to the MTU data gathered and is represented to an operator who can monitor the state of sensed phenomena or objects and take some actions.

Some SCADA deployments provide fault tolerance in a number of ways. A MTU central server can be duplicated and end-devices could potentially have back-up links connected to both the master and the back-up MTUs. The master MTU is capable of executing some logic and triggering some actuators based on the logic while the back-up MTU provides only a back-up storage for the acquired data. Another type of fault-tolerance used in SCADA is the RTU fault tolerance shown in [39], where a control point uses 2 or more redundant multiplexers with the same set of sensors to interface the I/Os to a main controller, which gathers the data from the redundantly deployed sensors and sends it wrapped as Controller Area Network communication protocol packets to the main controller.
We can make a certain analogy between the centralised component presented in most of the protocols and architectures described above and the MTU component in a SCADA system. As we already described, none of the above mentioned centralised components has a feature ensuring centralised component fault-tolerance. However, there is such kind of mechanism that can be employed in the MTU of some SCADA instances. A back-up SCADA MTU together with back-up links will ensure that the data acquisition process will not stop in cases where the master MTU fails. However this technique does not provide any means for load balancing of end-devices. Even though recently open industry standards have emerged related to development and deployment of SCADA systems, another major disadvantage of SCADA systems still is the vendor specific communication protocols used for communication between a MTU and a PLC/RTU. This is another reason why we believe that a cheap and flexible home automation inter-operable middleware could potentially have its impact on the way that engineers design industrial solution for remote monitoring and data acquisition.

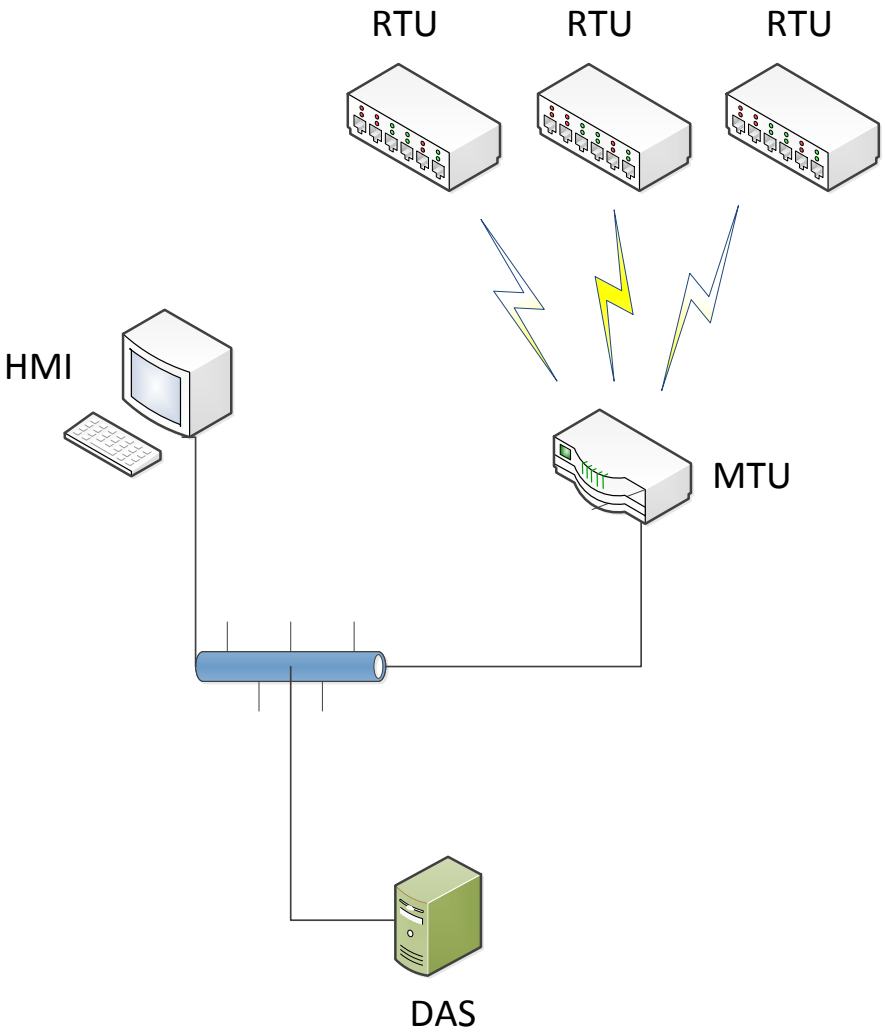

Figure 7: The SCADA architecture

3 Requirements

The mandatory requirements that a home automation middleware system must meet are:

- Provide interoperability between vendor specific subsystems.
- Each subsystem must interact with the middleware system in a safe way e.g. if the subsystem crashes this must not affect the middleware system.
- Centralised Point of Access - The system must provide at least one centralised access point. Via this interface any subsystem can be prompted to turn on or off one or more of its actuators. Furthermore, this centralised point of access must have an engine which can execute certain composition logic and take some actions when an actuator in any system is triggered or some specific information is read via a sensor.

Along with these requirements that are inherent to a home automation middleware system, a distributed one, that could potentially be used in the industry, must meet the following additional requirements:

- Continuity - The system must be able to operate continuously even in case of marginal failures.
- Self-recovery - If a failure in one of the distributed nodes of the system occurs, the system must be capable of self-recovery and participate isolated subsystems to a new node, such that the overall behaviour of the system stays unaffected.
- Fault-tolerance - The system must ensure that transient hardware or software errors do not affect the correct execution of the composition logic available in the system. An error in the execution of the composition logic can result in disastrous consequences as it can trigger a critical actuator incorrectly.
- Security - As the system must be secure and not allow for unauthorised nodes to get the information and commands transmit back and forth in the system.
- Scalability - This means that the system must remain stable even when the number of its elements changes. It should not affect the performance under a pre-defined threshold.

4 Prerequisites

In this chapter, we will analyse and present what of the existing concepts and methods will be needed in our project, based on our requirements.
In the Taxonomy and Terminology appendix A we present a taxonomy, according to which we will express fault-tolerant concepts and architectures. Firstly, we will start by discussing about different means for dealing with faults, then we will analyse various strategies for failure remedies. Thirdly, we analyse means for building a distributed system that could be utilized by our design. Finally, we conclude the chapter by taking various decisions about out design.

4.1 Failures and Failure Remedies

In this chapter, we will analyse potential types of faults that could cause a service failure and the types of failures that can occur in a system. Then we shall proceed by describing what fault tolerance means, and what fault removal and fault forecasting are.

Failures

Before speaking about fault tolerance, we must first explain what is a failure in an information system. [5] classifies the types of failures in the following groups:

- Service Failure - Delivered service is not correct. As we are mainly going to analyse service failures, we are going to describe the different subclassifications of service failures:
  - Content and Timing Failures - Content is not as expected or it has not been delivered on time.
  - Halt and Erratic Failures - A service has either been temporarily halted or has been delivered in an irregular state.
  - Consistent and Inconsistent Failures - The former types of failures delivers the same wrong service to all of its consumers, and the latter types deliver different type of services, including correct service to its consumers, a.k.a. Byzantine Failures.
  - Minor and Catastrophic Failures - Catastrophic failures could even result in fatal consequences, and the minor failures usually does not have a large scale of impact on the surroundings.
- Development Failure - A fault made in the development stage can occur in the use phase. We are not going to go into details about these type of

failures, as elimination of failures should be subject to the engineering team developing a system.

- Dependability Failure is when a very frequent service failure is presented.

We are mostly going to analyse a service and dependability failures and remedies to these failures. As we are analysing fault-tolerance applicable in a home automation interoperable middleware, we are going to present failures, which are also referred to as service failures.
Different sources present different taxonomy for classifying failures. There are two different kinds of component failures [64]:

- Byzantine Failures. *"The component can exhibit arbitrary and malicious behaviour, perhaps involving collusion with other faulty components".*

- Fail-stop Failures. *"In response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and then stops"* .

As we have investigated that type of failures that can occur in a system and we have already classified and built a failure taxonomy, we are going to discuss in more details about failure protective measures categories that we have briefly mentioned above. As we consider fault prevention to be more of an engineering task that needs to be taken into account when a system is implemented, we are going to elaborate more on the other three types of categories.

Fault Tolerance

- It is a mechanism for detection of errors and system recovery from these errors. A fault-tolerant mechanism consists of:

- Error detection - Detecting an error process takes place simultaneously along with the regular system's operation.

- Recovery - This procedure takes place upon detection of a service failure. It puts the system in a previous most recent non-faulty state. In turn, it consists of:

  - Error Handling - This procedure executes in turn the following procedures:

    * Rollback - Brings the system to the most recent non-fault state.
    * Rollforward - The system takes the current state, eliminates the errors and moves to the next state.

    - Compensation - The mechanism which systematic usage aids error masking.
  - Fault Handling - It consists in the execution of the following procedures:
    - Diagnosis - A fault is diagnosed first in order to detect the errors.
    - Isolation - The faulty components are excluded from the system.
    - Reconfiguration - New components to replace the faulty ones are taken from a set of redundant components.
    - Reinitialisation - The system is updated to the new state forced by the reconfiguration phase.

Fault Removal

The fault removal can be applied during Development phase and during use phase. The Fault removal strategy is achieved via the verification of certain system properties. If an important fault tolerant property has not been satisfied, the component causing this needs to be identified and finally it needs to be corrected. Such kind of verification could be performed for instance via model-checking tools such as [22] and [36]. If an error needs to be corrected during use phase, a system maintenance is used, which could be:

- Preventive - Prevents a system from experiencing a system fault.
- Corrective - Removes errors that had caused a system fault already.

Fault Forecasting

Fault forecasting is achieved by means of qualitative and/or quantitative evaluation of a system's fault occurrence.

- Qualitative - Classifies and ranks the event that lead a component to a failure state.
- Quantitative - Probabilistically determines what the likelihood of a failure occurrence is.

The aim of our project is the design and development of a distributed, reliable, and fault-tolerant home automation middleware. Thus we are ultimately interested in analysing existing fault-tolerant mechanisms and architectures rather then fault forecasting. It would also be a good point to verify that the design of our system satisfies certain fault-tolerant properties such as guaranteed delivery of a viable result of a logic execution in the distributed home automation core components to the vendor specific devices. This is why in this chapter we will analyse existing fault-tolerant mechanisms and architectures.

4.2 Redundancy

Redundancy mechanism could be applied over hardware and/or software components.

Hardware Redundancy

A component is duplicated or triplicated, so it provides a high probability of hardware component availability and software logic execution accuracy. Common hardware fault-tolerant architectures implemented on a multi-chip architectures are revised in [35], applicable in the automotive industry, but these can be used elsewhere as well. Then all these architectures are considered for providing fault tolerance in systems on a single chip (SoC). We list, analyse and compare these techniques in terms of economical impact, performance and reliability:

- Lock-Step Dual Processor Architecture. Two processors execute the same code, where one of them drives all the calculations and outputs and the other one executes the code on the background. Both of them use the same memory and bus. The results are compared by a comparing logic module. If there is a difference in the output result, an error is detected.
  - Advantages:
    - Performance is really fast in case that all tasks are critical. A critical task is a task with a strict deadline.
  - Disadvantages:
    - Bus and Memory errors are not checked. So additional technique needs to be applied to insure the system against bus and memory corruptions.
    - The faulty processor can not be determined without the use of an additional mechanism.
    - Reduced performance in case that only a few tasks are critical, since the processors are executing the same code.
- Loosely Synchronized Dual Processor Architecture. Two processors running on different chips and using different memory. Both of the systems are running Real Time Operating System (RTOS), which synchronizes and compares their output. The output is compared by means of cross-checking the results. Sanity check is applied in case of a mismatch. It checks the hardware and determines the failed one. Critical tasks are duplicated, while non-critical ones are executed in parallel.
  - Advantages:

    * Only a small subset of all tasks are to be critical, increased performance.
    * Buses and memory failures do not require additional failure detection techniques.
  - Disadvantages:
    * If most of the tasks are critical, then this method is less effective than the Lock-Step architecture, because of the sanity check and the cross check.
- Triple Modular Redundant Architecture. This architecture is similar to the Loosely Synchronized Dual Processor Architecture but with three processors, running in a lock-step in order to determine the correct result which is based on a majority voting. This can be used to determine when a single CPU fails. The bus and memory failures require another mechanism for ensuring fault tolerance.
  - Advantages:
    * Performance is pretty much as fast as the Lock-Step architecture.
  - Disadvantages:
    * Bus and memory errors are not checked. So additional techniques need to be applied to insure the system against bus and memory corruptions.
    * Expensive, since it requires three processors.
- Dual Lock Step Architecture. This architecture takes characteristics from the Loosely synchronized and the Lock-step architecture. On top of the loosely synchronized architecture, in each of the systems a CPU checker is added, which runs the same code as the master CPU and a monitor component checks the validity of the result.
  - Advantages:
    * Sanity check is no longer needed.
    * Software design errors can be prevented.
    * Fail silence capabilities.
  - Disadvantages:
    * Expensive solution, since it requires two autonomous systems, each of them having two processors and memory.
    * If most of the tasks are not critical they can not be executed in parallel. However in our case, non-critical tasks can be executed in parallel, since the three CPU's in our scenario will be actually distributed nodes of the system.

- SoC Fault-Tolerant Architecture. 2-sets of 2 CPU, which are Master and checker correspondingly. The memory consists of 4 banks - 2 for code and 2 for data. Instead of a bus, a cross bar [47] is used which ensures availability of memory access.
  - Advantages:
    - Shared memory at lower cost.
    - Fail-operational capability is an option.
    - Fail-silent mode is available, which can increase performance as critical tasks can be executed in parallel.
  - Disadvantages:
    - Error-correcting code is required for the cross bar and for the memory fault tolerance.

[37] gives an example of a general fault-tolerance mechanism combining the TMR with watchdogs. A watchdog in this sense is a mechanism where a third party component monitors a program's execution flow for the sake of detecting faults or errors. Here is the methodology and the steps taken to develop this kind of mechanism.
We have one component which is a single point of failure. First of all, we need to replicate this element within the system. Then there are three classical kinds of faults.
The first one is when the element enters a deadlock state in which the element cannot take any action. In this case the solution is to put a watchdog on each element that is activated periodically (kicked) as long as nothing abnormal happens. When a fault occurs, the element deadlocks and stops kicking its watchdog. As a consequence, the watchdog timer grows until it exceeds a predefined value (we say that the watchdog overows) and that will trigger the element to restart.
A fault of the second sort would cause the element to output incorrect data. To detect such a value fault, we need to use a component called a voter. We assume that at any time only a small amount of the elements fail (in the case of three elements, only one could be faulty), the voter will detect which element has failed and trigger a restart. It is also possible for the voter to mask the incorrect output from the element which failed.
In the third case, a element fails by going into a livelock state. In this state, the element is not able to output any result; but it still keeps kicking its watchdog from time to time. As such, the watchdog timer does not overows and the element failure cannot be detected by the watchdog. That is why in this case, the fault detection can be done by the voter that will see an empty value from the element which had a failure. Thus we are converting an omission fault into a value fault.
We also do not want that the voter become a single point of failure, which would

diminish the global fault tolerance of the component. To avoid this, we can use more voters. That is why we need to design a component, called the arbiter, to detect a fault in a voter and pull out the result from the voter that has not failed. In this situation, the arbiter is also able to take responsibility to trigger the restart of failed elements.

All of these architectures are designed for the sake of providing a reliable end result of execution of certain logic. However, hardware redundancy could also be applied on replicating communication links. By replicating the communication links between system components, we can make sure that a component will not become isolated from the rest of the components.

Software Redundancy and Software Fault Tolerance

In the the previous subsection, we have discussed about a variety of hardware faults and means for remedying these faults. In contrast to the physical faults that occur, because of a different reason and usually affecting only one component at a time, software faults are time invariant and in most of the cases for the same input, the same result is output. This means that the use of one of the methods for providing hardware fault-tolerance can not ensure against service failures triggered by software faults. In other words, for any input N, the same output X will be produced by the same software executed on a different hardware. In this subsection, we will explore how this issue can be overcome and how the application can provide mechanisms for handling faults caused by software entities. Another issue that a system could experience is when a component is put in a *crash state*. Such kind of states are irreversible and a component can not be reverted to a safe state if such a fault occurs. In this subsection, we address this issue as well.

Software Fault Tolerance Software fault tolerance manages faults occurring on a software level. One source of a software's fault are software bugs, another source of software fault could be a software vulnerability. The former source of software bugs can be overcome with the use of the programming methodology called N-version Programming [33]. In its nature, it represents different versions of software logic or task, made by different independent software teams and this logic is redundantly executed. The software teams are possibly coming from different background and they have nothing in common, except that they develop applications based on the same requirements. The final applications, developed by the teams, are all executed and a majority voter element compares their output and accepts the majority of equivalent outputs as the correct result. However, a number of experiments have been made in order to verify the effectiveness of this method and the results gathered have not been very encouraging [34]. Neverthe-

less, we are considering this method as the only alternative that could increase the probability of delivery of a reliable system.

Software Redundancy Software Redundancy could be applied in order to replicate applications on different physical machines for the sake of recovering an application in case of failure.
It is a method which completely removes failed components and dynamically loads new ones instead at run-time. For this sake, all elements are duplicated across other components and then loaded on demand.

Dynamic Recovery

[44] Software and hardware redundancy provides one way for building a robust fault-tolerant system. Another way of building such system addressing different use case scenarios, is dynamic recovery.
The biggest advantage of this technique is that it does not require any redundant elements, like in the hardware redundancy. Instead, there is only one component running without back-up and obviously this solution is less expensive. However, it requires special fault-detecting techniques. When a subcomponent of a component fails, the component should be able to detect the fault before it has negative consequences to the overall system and potentially replace the failed subcomponent with a spare one, or isolate the component and continue working without it.
In the previous section, we have analysed various hardware and software fault-tolerance mechanisms and architectures. In the next section, we are going to analyse existing approaches for distribution of system components that could potentially be utilised by our design in order to build a reliable, self-recoverable and fault tolerant distributed home automation middleware.

4.3 Distribution of Nodes

In this section, we will analyse different approaches to build a distributed fault-tolerant system. The distributed system that we are going to build needs to be completely scalable, secure and fault tolerant to hardware and/or software failures as we have described in 3. The most scalable solutions known these days are the peer-to-peer distributed systems. In these systems, a single point failure is not present. This is the reason why we are going to analyse various peer-to-peer solutions that we could utilise in our design. We start by explaining basic concepts used in building a distributed peer-to-peer system.

Peer-to-Peer Systems

Peer-to-Peer networks consist of a dynamic set of distributed components a.k.a. nodes. Each node offers a certain set of services and/or resources to all other nodes and all nodes are equal. Examples of peer-to-peer systems are Skype [49], Gnutella [67], Napster[53] and KaZaA [26]. All of these systems are Peer-to-Peer based systems, each of them having its own architecture. Three kinds of peer-to-peer systems can be distinguished:

- Centralised - These systems, which are Napster-like, have a centralised dedicated server keeping information about the shared content by the peers. When a peer wants to access a certain type of resource, its query is sent to the centralised server and the server replies with all the information needed by the query initiator. This information is used by a peer in order to connect to another peer providing the resource and to access this resource.
- Hybrid - This approach represents a hierarchical peer network, where there are two kinds of nodes: ordinary peers and super-peers. The network of super-peers is organized in a decentralised way and all the ordinary peers are participated to a certain super-peer. An example of an application exhibiting this architectural pattern is Skype [49].
- Decentralised - In this approach, advertising and searching for resource is done in a completely decentralised way. In addition, applications employing this architecture are capable of participating new nodes automatically and recovering from nodes leaving the network automatically as well.

Distributed Hash Table (DHT)

General Description of DHTs and their properties A Distributed Hash Table [32] is one example of decentralized distributed system. It is based on a service similar to hash tables in order to look for information or participating nodes, with a classical (key, value) pair system. The system is build so that any node can easily find the value corresponding to a specific key. However, the mapping from the keys to the different values is distributed in the nodes. This means that the system is a lot more adaptable to changes in participants and as such can scale to a very important number of nodes while handling without trouble all new arrivals, departures and failures which could happen.
DHTs are often used to build complex and efficient services. One of the most successful example is the distributed peer-to-peer file sharing network BitTorrent [27]. Due to their concept, DHTs allow for a certain number of interesting properties:

- The system is autonomous and decentralized. There is no node which acts as a central coordinator.

- The system is fault tolerant. Incoming, leaving or failing nodes do not impede the functionality of the system.
- The system is highly scalable. It is possible to function well even with millions of nodes participating.

To achieve these results, the DHTs use the fact that each node only need to coordinate with a certain number of nodes, which are often its neighbours. This means that with each change in the system, only a small amount of work is necessary to be done by a single node for the system to still function efficiently.
The way DHTs are working is the following:

- At the basis is an abstract keyspace. Most of the time this is done in the form of a set of 160-bit or 128-bit strings.
- This keyspace uses a partitioning scheme to split its ownership between all nodes included in the system.
- The different nodes are then linked by an overlay network, which makes it possible for them to find the owner of any key within the keyspace.

The different steps when a DHTs is used are the following:

- When a value is stored, the DHT often uses the SHA-1 hash function [16] to generate the 160-bit or 128-bit key, while a message is dispatched to all nodes participating.
- This message is forwarded from node to node by using the underlying architecture of the overlay network. This is done until the node responsible for the key is reached.
- The reached node then keeps the key and data and this allows other members to retrieve the contents through another hashing of the name of the file. With the result obtained by the hashing, the client asks the DHT to find the data corresponding to it via a get message, which is a query for the network to get some return value.
- This message is routed through the overlay network until it reaches the corresponding node which has responsibility for the key. This node will then reply with the data that the client was searching for.

The largest part of DHTs are using a variant of consistent hashing [30] to direct the keys to the corresponding nodes. Consistent hashing uses a function $\delta(k_1, k_2)$. This is corresponding to an abstract distance between the keys k_1 and k_2. Each

node possesses a single unique key which is its ID. As such, a node with an ID of i_x will possess every keys k_m where i_x is the closest ID, using the the $\delta(k_m, i_x)$ function to determine the distance.
Consistent hashing is very useful because it means that removing or adding a node in the system will only cause a specific set of nodes with neighbouring IDs to change. This allows all the other nodes to remain unchanged, thus limiting the effect to a very small part of the system. This is a big difference to normal hash tables where such a change induce most of the keyspace to be remapped. With this system, consistent hashing allows for a large number of arrivals, departures and failures with an efficient support.
DHTs are using an overlay network to connect their nodes. The way the overlay network is working is the following: all nodes in the system keep a certain numbers of links to some other nodes. The sum of these links constitutes the overlay network. For any key k, there exist a node ID owned by a node such that this node possesses k or possesses a link to another node which is closer to k, using the keyspace distance described before.
A greedy algorithm is then used to route a message to the node possessing a key k. A greedy algorithm uses the problem solving heuristic to do the locally optimal choice at each step in order to find a global optimum. As such, each step will send the message to a neighbour closer to k. The destination is known when there is no closer neighbour to send the message to: the current node is then per definition the owner of k. This system ensures that the maximum route length is minimal and that the maximum number of neighbours for a node is also minimal. This allows the request to be handled fast, which is the biggest advantage of this network.

Pastry Pastry [3] is a distributed overlay network handling automatic object location in a decentralised peer-to-peer manner. It consists of dynamically joining and leaving peers, that we shall call nodes. It is a self-organising network, which handles itself nodes joining and leaving the network automatically.
Each Pastry node **A** characterises itself with the following attributes:

- Node ID - This is an unique number which is randomly chosen and usually is formed by applying a hash function over a node's public key or its IP address. The node ID is 128 bits long and it is randomly chosen. This ensures a very high probability for uniform distribution of nodes in *node ID space* and consequently a very high probability for the geographically randomised distribution on Node IDs. This is very beneficial for keeping the virtual network alive even in cases of unpredicted emergencies in certain geographical area.
- Routing table - A node's routing table consists of $O(\log_{2^b} N)$ rows and 2^{b-1}

records in each row. In the first row, **A** stores nodes' IDs which have the same first digit as **A**. In the second one, the records have the first two digits in common and so on. **b** is a configuration parameter which determines the digit encoding and it usually is 4, which means that the digit encoding is $2^b = hexadecimal$.

- List of immediate neighbours. An immediate neighbour **B** to the node **A** in this sense, is a node which is close to node **A** in terms of proximity metric, which usually is expressed in IP-based routing hops. In order to form the list of its immediate neighbours, a node **A** is using some sort of heuristics. Each node has 2^b or $2 \cdot 2^b$ immediate neighbours. Ultimately, the neighbourhood table is not used in routing messages, but it is maintained for the sake of determining node A's relative physical location.
- Leaf set - This set contains K nodes in the Pastry network whose node IDs are numerically closest to the node ID of node **A**. Moreover, this set consists of an equal number of nodes with bigger and with smaller node IDs compared to **A**. Each node has 2^b or $2 \cdot 2^b$ nodes participated in its leaf set.

As we have described what attributes a Pastry node possesses, now we shall discuss what happens when a message needs to be sent from one node to another one.

Pastry's Message Routing When a node **A** receives a message M, it first looks-up in its leaf set and checks whether the node is there. If the node is within its leaf set, it is redirected to its destination, which is the node which node ID is numerically closest to the key of the message. If M's key does not fall within the range of **A**'s leaf set, **A** looks in its rooting table and forwards M to the node which shares longest common prefix with M's key. This way, each time when M is redirected to a node **B** from node **A**, M is getting closer and closer to its destination until it reaches it. Pastry rooting complexity expressed in routing hops is $O(\log_{2^b} N)$, where N is the number of peers within the network, 2^b as mentioned earlier is the the digit encoding. Usually the digits encoding is hexadecimal so b=4.

Tapestry Tapestry DHT [2] is yet another structured peer-to-peer overlay. It has *160 bit* ID space, which allows for multiple applications coexistence and each application is assigned a unique identifier A_{ID}. The nodes and objects in the network in turn are assigned identifiers as well - node IDs and Global Unique Identifiers(GUID) respectively. It is a Pastry-based DHT and it is very similar to Pastry, but on the other hand there are some differences between them which we shall discus here.
Here are the points in which the Tapestry DHT differs from Pastry:

- Backup neighbours. As in Pastry, the node ID's are assigned in a randomised manner. However, Tapestry maintains multiple node ID's for each of its entry in its routing table. Furthermore, these nodes are sorted according to proximity metric. Thus when a message M has a common prefix with multiple nodes in an entry in its routing table, M is routed to the node with the least proximity metric e.g. IP routing hops.

- Lack of Neighbourhood set. While Pastry has a set of nodes which are its neighbours in terms of geographical location and this is measured with IP routing hops, Tapestry does not possesses such a thing.

- Lack of Leaf set. Tapestry does not have a leaf set. Instead, it provides two distinct types of functions in its API, where the first one seeks for an object which is hosted at a certain node and the second one routes a message directly to a node with a node ID.

Tapestry's Message Routing Tapestry's Routing Algorithm is exactly the same as Pastry's routing algorithm and we are not going to discuss it again. The performance is the same as in Pastry $O(\log_{2^b} N)$, in case of consistent routing tables without any blanks. However, in the general case Tapestry would be more efficient than Pastry since it keeps backup links for each entry in the nodes' routing tables. The second reason why we find Tapestry more efficient is because it sorts out the links for each entry in its routing table according to the Round Trip Delay metric. A relative disadvantage of Tapestry could be its bigger routing table which has size of $O(c \cdot d \cdot \log_{2^b} N)$, where c is a the maximum number of backup links that an entry could have and d is the same configuration parameter as we have described in the description of Pastry.

Chimera Chimera [13] is a light weight implementation of a Structured Overlay implemented in *C*. It first has emerged as an implementation of Tapestry and it eventually changed its name to Chimera. In its latest implementation it uses some properties from both Pastry and Tapestry DHTs. In its implementation it combines the strengths of both DHTs as it has Pastry's leaf set and Tapestry's back-up links sorted according to a proximity metric.
Considering the fact that we are aiming at the development of a middleware which will be applied in the home automation domain, Chimera would be an excellent choice for us to apply in our project, since it is implemented in *C* and this means that no extra effort needs to be put there in order to apply it to our design. Another benefit of using Chimera would be that it uses the strengths of the above revised DHTs. Now let's move on and revise another DHT called PGrid.

P-Grid P-Grid [31] is a peer-to-peer lookup system which is based on a virtual distributed search tree. It is structured in a way similar to standard distributed hash tables. Each node contains a part of the complete tree. The positions of the nodes participating are determined by their paths, which correspond to binary bit strings. These binary bit strings represent a sub-part of the overall tree and that sub-part is what the node is responsible for. Fault-tolerance is implemented by the fact that several nodes can be responsible for the same path.
Each node also contains data about a minimum of one another node which possesses the other side of the binary tree at the corresponding level. Thus in case one node cannot fulfil a specific request, it will forward the request to a node that is closer to what is being researched. For example if node 1 possesses a binary prefix of 00, it will redirect all requests starting with 1 to a node possessing a binary prefix of 10 or 11, which is in its routing table. If a request starts with 0 then it will check the next binary number, if it is 0 then this node will be responsible for it and if it is 1 then it transfers it to a node owning the binary prefix of 01.
Another interesting point of P-Grid is that nodes' paths are not determined from the start but change dynamically by negotiating them with other nodes part of the network. P-Grid can then be considered as a decentralized and self-organizing structure adapting to nodes incoming, leaving or failing. This allows for great load balancing within the network, where imbalance can be detected and corrected by dynamic replication of data.
While most P2P systems do not possess update mechanisms which are functioning with data replication because the data shared is static, P-Grid tries to correct this with its update algorithm. This algorithm is based on rumour spreading, which can give a probabilistic guarantee that data will continue to be consistent and compatible without destroying the self-organizing nature of P-Grid. The algorithm is effective and based on push/pull gossiping schemes. It takes into account very unreliable and replicated environments and can deal with realistic situations where we have most of the nodes not connected to the network.
To handle changes due to dynamic IP within the system, P-Grid designed a decentralized and self-maintained peer identification service, which is working even on environments which have a low availability of the nodes. The way it is working is that P-Grid stores the IP mapping in itself and update it when the IP address is changed. This way P-Grid is able to have an updated view of IDs/IPs at any time.
P2P systems are also subject to a lot of frauds, because participants are mostly strangers to each other. To thwart this, P-Grid built a decentralized trust management model. This model is able to analyse past actions between participants and issue a probabilistic assessment of whether some participant cheated before. This enable the use of the trusting grade, which other participants can use to determine

if someone is trustful or not, with a system similar to a reputation.

4.4 Group communication

Introduction to the concept

Group communication [29] is the process of using a network to allow for one-to-many communication. A lot of distributed systems are based on Remote Procedure Call (RPC) [11]. However RPC only allows for one-to-one communication. We could simulate one-to-many communication with RPC by sending a single message several time to several different receivers. But this solution has some flaws. What happens in a situation where some group members are not reachable? How do we deal with the latency between the first message sent and the last one? That is how group communication is different from RPC:

- It is possible to organize groups in different ways.
- Groups can be contacted in multiple ways.

Another interesting point is that groups are dynamic. Groups can be created when needed and can be deleted after that. As for processes, they enter the group through a membership request and can then later leave it. It is also possible for a process to be a member of different groups at the same time. Groups can also be overlapped.
The important facts are that messages are sent to groups without knowing the number of members in the group and the address of the members in the group. When a packet is sent to a group address, all members in the group get to receive it (multicasting). In case multicasting is not available to the network, then it is possible to use a broadcasting solution, which allows to contact every member of the network. If both options are not available, it is possible to send messages with unicasting: multiple packets are sent to multiple receivers in the group and it is necessary to know the address of each members, contrary to multicasting and broadcasting where one packet is sent to several receivers.
There are different kinds of groups:

- Closed groups: only the members of the group are able to have communication with the group.
- Open groups: anyone can communicate with the group.
- Peer or flat groups: all members of the group are equal and it is completely symmetric. There is no single point of failure but it is more complex to take decisions, with the need of voting algorithms.

- Hierarchical groups: there exists one master in the group which is taking simple decisions for the group (coordinator). However in this case, the loss of the master stops the group from working, with the necessity of electing a new coordinator for the group.

Fault-tolerance in group communication

Let us see now how group communication can be applied for fault-tolerance principles. Group communication allows for failure masking and replication, because the group will replicate some of its members/processes and if some of them fail, the rest will still work.
It is also possible to implement reliable multicast like for example RMTP [50] with the help of group communication. As the name implies, reliable multicasting is to implement a protocol to provide a reliable way of using multicast.
What is more, to avoid issues of scalability in reliable multicast, it is possible to only ask for negative acknowledgements from the receivers, or to use non-hierarchical feedback control, as it was done in the Scalable Reliable Multicasting (SRM) protocol [20]. In this protocol, receivers only send negative acknowledgements, not only to the sender but to the entire group. This means that any other process intending to send a negative acknowledgement can stop it, presuming that since there was already a negative acknowledgement, the multicast will be sent again. To make it work even better, all receivers use a random delay before sending their negative acknowledgements. This allows for easiest suppression of the other negative acknowledgements which didn't get send yet. In this protocol, all receivers are treated equally: they all have the same chance to send their negative acknowledgements, hence the appellation non-hierarchical feedback protocol.

4.5 Security

In this subsection we will discuss about general security techniques that we would need to accommodate in our project, in order to make the project feasible to use in reality. First, we will start by discussing about different means for authentication of newly attached distributed middleware nodes and then we will proceed by providing an analysis of existing approaches to securing data, sent from one node to another.

Authentication

We would need to accommodate an authentication protocol in our system, such that a distributed node attempting to perform an operation over a resource, like requesting a home automation service, can acquire a permission to do so. The

second reason why we would need to accommodate such a mechanism is to ensure that only trusted nodes can participate to the network.

Authentication of New Nodes The authentication of nodes attempting to join an instance of our distributed home automation system is essential. One approach to authenticate joining nodes, is to ask them for a shared network name, network password and then let them access a resource or request a service etc. This approach has the disadvantage of transmitting the networks password upon each resource request and communication between two nodes. Another approach would be to have a dedicated centralised server taking care of the authentication process and issuing certificates to the successfully logged nodes. These certificates can be used to ensure that a node has successfully logged-in and it is trusted. Indeed, such an approach can verify that a node has valid permissions to request a specific resource e.g. a service in our case. An example of protocol using such an approach is the Needham-Schroeder authentication protocol [46].

End-to-End Security

Each sensitive piece of information, send from one node to another, must be unread, untampered and unmodified by a third party malicious node. This is where end-to-end security mechanisms comes into play. There are a number of ways that can be used to provide end-to-end security such as:

- Stenography. With this approach, sensitive data is hidden in insensitive data and transmitted over the network by a sender node. The receiver in turn knows that the received insensitive data contains within itself the sensitive data. Moreover, the receiver knows exactly how to find it, and what the meaning of the data is. For example, sensitive authentication credentials can be hidden within insensitive data, e.g. an audio .mp3 file. The sender inserts the credentials at a specific place in the .mp3 file, known by the receiver. The receiver in turn knows where the password has been inserted by the sender and thus it extracts from the contents of the .mp3 file. The obvious disadvantage in this mechanism is, that the data can be altered by a third party malicious node.

- Anonymity-based networks. This mechanism relies on the fact that sensitive data can be chopped up into pieces and transmitted over the network using different routes. Thus a third party malicious node would not know exactly who the receiver is, and what the data is. This approach is often used in anonymous peer-to-peer networks. In this case, if a node A wants to transmit a piece of sensitive information to node B, it chops the information into many random pieces. These pieces of information are forwarded to node B using

different routes. This way a third party malicious node is unaware of what the content of the whole message is and who the receiver is. Obviously this method suffers from the same issue as the stenography as it is vulnerable to malicious altering of information.

- Security through obscurity. This method relies on the unawareness of malicious nodes that a sensitive information is transmitted. Usually this concepts has been often used in vendor specific protocols. Typical example of the use of security through obscurity are the previous generation specific SCADA systems, where vendors used to develop their own protocols and potential attackers were unaware of the internal structure, encoding, etc. However, this method proved to be not very efficient as an attacker can gain knowledge of the transmittance protocol used by means of traffic analysis, for instance.

- Symmetric Key Encryption. This mechanism is based on shared secret between a sending node S and a receiving node R. When a message M needs to be transmitted from S to R, S encrypts M with the shared key and sends it to R and R in turn decrypts it with the same key using an inverse function. The advantages of this approach is that it has been widely used for a number of decades and that it is lightweight compared to Asymmetric Key Encryption algorithms

- Asymmetric Key Encryption. The disadvantage of symmetric cryptographic algorithms is that a mechanism for the secret key exchange is needed. In reality this is difficult to accomplish. The asymmetric encryption algorithm is capable of encrypting a message M with a public key Pu known by everybody. The other communication end point will decrypt it with Private key Pr and thus can restore the original message M. The main disadvantage of this approach is that it is about 100-1000 time computationally more expensive compared to a symmetric key encryption algorithm. Example of algorithms based on the asymmetric key encryption are the RSA [48] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [38]. Some protocols, such as the Transport Layer Security Protocol (TLS) [56], are using a hybrid approach. The TLS in particular uses an asymmetric key encryption only for secret key exchange and afterwards, the transmitted messages are encrypted and decrypted using a symmetric key encryption algorithm.

From all of these methods we are going to use a hybrid approach between asymmetric and symmetric approach in order to ensure end-to-end security between two communicating end point nodes. Thus we will guarantee reduced computation cost and at the same time we are going to take advantage of the asymmetric cryptography's benefits.

Digital Signatures and Digital Certificates

Digital Signature Digital signature is a mechanism for verifying a message's authenticity. It relies on asymmetric cryptography algorithms, such as the RSA and the ECDSA are. The algorithm works as follows: a message M is given as an input to hash function $F(M)$ and $F(M)$ produces a hash value M_{digest} a.k.a. message digest. Then an entity E_1 encrypts M_{digest} using its private key and produces an encrypted message digest $M_{digest_{enc}}$. This value is called M's signature M_{sign} and it is transmitted along with the message M. Another entity E_2 can verify that the message M is authentic in the following way: firstly, it decrypts M_{sign} with E_1's public key and this produces M_{digest}. Secondly, E_2 calculates M_{digest_2} using the same hash function $F(M)$. Finally, it compares M_{digest} and M_{digest_2} and if the values are equal, the message is authentic, otherwise it is not.

Digital Certificate The digital certificate [40] a is short written statement signed by a trusted *certification authority*'s (CA) private key. CA's public key is available to everybody, so anyone can verify a message signed by CA, by using CA's public key. We have already described the procedure of signing a message using the digital signature mechanism.

4.6 Design Choices

In this section, we are going to discuss about the design choices that need to be made. We will take a variety of decisions such as the scale of the network, the way of devices participate to a middleware node, software fault-tolerance of composition logic execution, static or dynamic distribution of middleware nodes, what level of security needs to be applied and so on.

Method for Physical Device Participation

Methods for physical device participation could vary depending on the physical media used for communication between the physical devices themselves and their corresponding bridges (adapters). For instance, if a vendor specific subsystem is interfaced to the home automation middleware system, through a Universal Serial Bus (USB), the vendor gateway to the middleware could be connected only to one node at a time. This implies that if a device is connected to a middleware node and the node crashes or leaves the network at some point, the vendors subsystem itself needs to take some actions to switch between the adapters. We need to provide a clean and easy to use Application Programming Interface (API) to the developers of the bridges (adapters). This API can be used to switch between directly connected devices. In the general case scenario, point-to-point connections

need to be avoided. If we assume that the vendors of adapters avoid to use such kind of point-to-point physical connections, then all the decisions on which middleware node needs to used, can be taken in the middleware nodes themselves. For instance, if a physical device is interfaced to its corresponding adapter through Ethernet or Wi-Fi, the decision on which node is handling the interaction with the physical device could be taken by reaching a consensus among the distributed nodes. For the sake of providing the developers with an uniform way of handling the point-to-point connection issue we would need to provide an API which will apply abstraction to hide all the details around the physical media used. This API must be used in an uniform way regardless of the physical media used, in order to provide a mechanism for context switching in the physical nodes themselves.

Means for Adapter Mobility

As we mentioned in the previous design choice 4.6, the physical devices need to be participated to an alternative node B in case of failure or unavailability of its primary node A. In this case, we could use the approach where all the adapters are available on all nodes. This solution, in fact could be applied only in small-scale home automation middleware systems. In case of a system deployed on a large scale, this solution is practically applicable only if expensive servers are dedicated for this task. Another solution could be to store the adapters only at a subset of all the nodes available in the system. This solution is more flexible, more scalable and cheaper. In the ideal case scenario, all the networked nodes will be homogeneous and the adapters' executable binaries will be backed-up on some nodes; they can be serialised and sent to a node that needs it on demand. In the general case, our design would need to deal with heterogeneous nodes. In this scenario, there are two choices. The first one has been widely adopted in the world of software development, and is the use of virtual machines such as *Java Virtual Machine* and *.NET's Common Language Runtime (CLR)*. In this case the adapters' code could be compiled to a machine independent code (Java bytecode or Microsoft Intermediate Language (MSIL)) and an adapter could be sent to a node on demand and compiled by a just-in-time (JIT) compiler. However, we need to consider a home middleware solution which could potentially be ported to devices with restricted pieces of hardware and these machine independent solutions are computationally demanding, since a virtual machine needs to be ported on these devices. The ideal implementation of such a system will use a C language implementation, which is less demanding than the above mentioned virtual machines. The only reasonable option left is to use a mechanism which will back-up the adapters compiled with a cross-compiler and targeted at different platforms. Even though requiring extra memory space, we find this approach mandatory. This is why we are going to use the latter approach for the adapter mobility where we will store various

pre-compiled adapter versions targeted at different platforms.

Use of Group Communication

Sometimes a node would need to communicate with a group of other nodes. For instance, it might want to send the state of an actuator modelled as a service in the context of home automation middleware to a group of nodes. In this case, it will be more efficient if the node sends a multicast to a reserved IP address, to whom the group of nodes have already subscribed to. To be more precise, the node which possesses the service, that the other nodes want to subscribe to, should first create a multicast address. After this, it will broadcast the service and the multicast address within the network. The nodes which want to subscribe for the event would just need to tell the network that they are interested in receiving the messages from this multicast address. This way, the next time the service node needs to inform its subscribers, it will just need to send it to the multicast address and all of the nodes subscribing will receive it.
However, multicast is very unreliable and messages can often get lost. We want to avoid this in our project and that is why we are going to use reliable multicast. In particular, the Scalable Reliable Multicasting (SRM) protocol [20] that we describe in 4.4 seems to be the best solution because it only uses negative acknowledgements and this does not overload the network with a lot of acknowledgements. This can be very useful in cases where the network comprises a very large number of nodes and wide multicast groups.

Adapters Redundancy Factor

As we have mentioned in the previous section 4.6, we need to provide a mechanism for adapters redundancy and we have made our choices related to the means for adapters redundancy. In this case, factor corresponds to the number of replicas of the adapter. Now we need to decide on the adapters redundancy factor. On one hand, the redundancy factor must not be too large. Large factor implies a lot of space taken by the adapters' storage. On the other hand, the factor does not need to be to small either. A very small factor might make an adapter unavailable at some point. This is why we need to have a mechanism for adjusting this factor dynamically at run-time, through an interface. This feature could "fine-tune" this parameter in order to employ a variety of deployments. In one deployment scenario, this middleware system could be used in an environment with dedicated home automation computing devices. Obviously, in this case the redundancy factor needs to be very small. In another deployment scenario, the middleware system could consist of unstably participated devices, joining and leaving the network all the time (high churn rate). In this case, the redundancy factor needs to be

very big, in order to provide availability of all the adapters at any moment with a probability very close to 100%.

Static versus Dynamic Distributed Home Automation

The home automation middleware system could be assembled from statically allocated designated nodes or dynamically joining and leaving self-organising set of nodes. In the first case each node needs to know the IP addresses of the other nodes in the network and the available services of the other nodes. The dynamic approach in turn provides a very flexible solution as one can connect random number of low-end computing devices to play the role of home automation middleware and these devices could be used for something else as well. The main drawback of the former solution is that it does not scale. It is obvious that such a solution will not scale in case that a network is initially intended to have a small number of bridges in it but having its number of bridges progressively increasing. This implies that this system would not be able to perform as usual and it might not be able to take the demanded load. This is why we are going to use the Dynamic Distributed approach of middleware nodes. For this sake we will take advantage of the Overlay networks that we have described in 4.3.

End-to-End Security Mechanism

We must decide on the approach that we shall use for securing the communication between two communicating nodes. As the symmetric and asymmetric encryption algorithms have proven to be the most robust types of end-to-end security, widely used in the industry, we are going to reduce our design choice to choosing a method between these two.
An hybrid encryption protocol will be our choice. As the network will consist of a number of peers and the peers will share a secret e.g. network name and network password, symmetric encryption algorithm could be easily facilitated with the use of the shared key known by all nodes in the network. Such an approach could be beneficial in a way that it is 100 to 1000 times computationally cheaper than an asymmetric encryption algorithm. However, encrypting all of the exchanged messages using the network name and the network password is not a good approach, as traffic analysis tools can analyse the transmitted messages and break the password. This is why a hybrid approach will be a very robust solution. With such an approach, a node will store other nodes' public keys in its routing table along with their node IDs and IP addresses. The public keys will be used to establish the secure channel with which the nodes can exchange a shared key. This shared key will be used to encrypt/decrypt subsequent data exchanges between the two parties. This approach can be used when a node A needs to send very large pieces

of information to node B. Asymmetric key exchange algorithm will be used in case that the exchanged information is not large e.g. a simple service request.

Centralised or Distributed Authentication

The second security aspect that we need to decide on is which security method we will use in our system in order to ensure that only trusted middleware nodes can join the network, retrieve and alter information from it. This can be done either by a dedicated authentication server or by the so called "chains of trust".
The first approach obviously is not very scalable, as it also suffers from single point of failure which could be avoided by means of using a back-up authentication server. Even in this case, this solution is quite expensive. On the other hand, a centralised solution could be a very robust solution, as the authentication server could issue certificates to the newly participated nodes, verifying their public keys (We have given our arguments for choosing a protocol in which a public key cryptographic algorithm will be used). Moreover, the authentication server could sign these certificates and other nodes could verify the authenticity of any node within the distributed middleware using the Needham-Schroeder authentication protocol [46]. The benefit of having a dedicated server is obvious, since any node could verify the authenticity of other nodes and the private key used for signing certificates will be known only by the authentication server.
The second approach is to use an authentication service running at each node. With this approach, any node **A** could be authenticated at any other node **B** which is already part of the network using a secure authentication service available at any node. The credentials used are common for all of the nodes in the network. These credentials could be the home automation network name and password and the public key of the adjoining node. Then **B** will store **A**'s public key on multiple distributed nodes using hashing and DHT's services. **A**'s key will be used when **A** becomes part of the Distributed Hash Table of another node. Each entry in the distributed hash table of each node has a Node ID, node's IP address and node's public key. So when something is to be transferred from a random node **R** to **A**, **R** is using **A**'s public key to encrypt the data. If **A** needs to return a service state or some data stored in the network, it needs to verify **R**'s public key, which was received by **A** along with the data sent by **R**. In order for a public key to be verified, a query is sent, whose routing complexity is $\log_{2^b} N$ hops. This point is the first disadvantage. Of course, only the node having a resource needs to verify **R**'s public key. The second disadvantage of having a distributed authentication service is that it will double the routing complexity in terms of hops. An ordinary query's routing complexity is $\log_{2^b} N$ and the query complexity with a public verification by an end node will become $\log_{2^b} N^2$. So if an ordinary query takes 5 hops to arrive at its destination, for a query with a public key veri-

fication it will take 10 hops. From this perspective a centralised log-in server is a way better approach. It should be noted that a node can not be issued a certificate on its public key. The algorithm presented compares the actual public keys, which is a sufficient check to verify the public key's validity. The biggest advantage of a distributed authentication service is that it eliminates the single point of failure which is presented in the former solution. In the end, the doubled number of extra hops is not going to affect the scalability of the distributed hash table while the use of a centralised authentication server could affect the scalability of the system, if a lot of nodes are presented on the network. There is another disadvantage of using such a distributed authentication service.
Even though **A**'s public key is stored in the network, a certificate could not be issued for its public key since someone needs to sign the certificate verifying **A**'s public key and thus this solution does not eliminates all security risks. A malicious node **Ma** could send a faked public key to a node **R** within the network. Even though **R** could query the network in order to verify that **Ma**'s public key is not trusted, there is the risk that **Ma** could fake the verification, since **Ma** itself could sent a verification to **R**.
This is why we need to emulate the Needham-Schroeder's centralised authentication protocol on a distributed scale. With this approach, a shared private key **Pr** available at all nodes' authentication service could sign a certificate **CeA** verifying **A**'s public key using **Pr**. Then this certificate could be returned to **A**. **A** is going to provide this certificate in subsequent data exchanges with other nodes available in the distributed middleware service. Then any node in the network would be able to verify the certificate itself in 0 routing hops. **A**'s public key along with its certificate will be used by other nodes to verify resources requests signed by **A** with its public key. Moreover, **Pr** will be disclosed to **A** as well upon authenticating **A**. Thus **A**'s secure service will be able to sign certificates to newly joined nodes as well. The advantages of using such an approach are obvious:

- The authentication service will be highly distributed, self-organised and will eliminate the single point of failure presented at the centralised solution.
- The additional hops required to verify someone's public key evident in the second approach are eliminated.
- A node's public key could be certified and then sign requests checked by any node as well. Which also will be beneficial when a node is attempting to change something in the network.
- The network authentication public key will be disclosed to anybody. This public key can be used by a node attempting to connect to the network, in order to encrypt its credentials and sent it securely.

- Each node will have its own public/private key pair, which will eliminate the need of using the network name, network password pair as a key for symmetric encryption between two nodes in the network. Even though slower compared to a symmetric approach, this method will be beneficial in the way that the network's name/password will not be used as an encryption key every time that two nodes want to interact with each other. This in turn will reduce the thread of having a program that can analyse the traffic in the network and thus infer the network name/password.

Of course, there is an obvious disadvantage when using such an approach. If a malicious node *Mallory* somehow steals the network name and the network address, this could compromise the whole network, since the private key of the authentication service will be disclosed to *Mallory*'s authentication service and eventually *Mallory* could issue public key certificates to infinitely many malicious nodes. However, this thread is existing in all of the three approaches. In the first approach, if *Mallory* knows the network name/network password, it can share it with infinitely many malicious nodes as well. It is obvious that the latter approach is the most suitable choice, since it combines the strengths of the former two approaches and eliminates some of their disadvantages. This is why we are going to use the latter authentication protocol described.

Composition Logic Fault-Tolerance Method

We have analysed various fault tolerance architectures in 4.2. From all of above described architectures, the one that would best fit our needs is the Triple modular redundancy. Some of these architectures provide only checks for result accuracy. The triple modular redundancy software fault-tolerance scheme could provide checks for result accuracy. Moreover, it can decide which result is the correct one based on a majority voting component.

Overlay Network Decision

We have revised a variety of overlay networks in 4.3 and in 4.6 we have given our arguments for choosing a dynamic node distribution with the use of overlay networks. Now we would need to decide on which overlay network we are going to use in our design 5. Chimera seems to be the most decent choice in our case, since it combines the useful features from Pastry and Tapestry that we need to accommodate in the distributed middleware. First of all, Chimera has the leaf set feature which characterizes the Pastry protocol, and secondly and most importantly it has redundant links under each rooting table entry and these links are sorted according to a proximity metric, which is inherent from Tapestry. This

means that the routing will consider IP hops and this is very important as we would like to ensure quick service discovery.

5 Design of the Distributed Middleware for Large Scale Automation Domains (DMLSAD)

In this chapter, we are presenting the design of DMLSAD. We will first introduce the high-level view of the distributed home automation middleware and then we shall proceed by describing each components in details and the high level interaction between the various system components.

5.1 Overview

Service discovery and usage

The backbone of the DMLSAD is the overlay network. As we have argued our choice in 4.6, we are going to use Chimera DHT. Each of the nodes in Chimera is a home automation middleware itself. Therefore each of the nodes is responsible for handling the communication between some vendor specific devices and the front end of the system. Instead of having a dedicated front end component, each of the nodes in DMLSAD can be used as a home automation entry point for client applications. Moreover, requests to the overlay network can be performed via different nodes simultaneously.
When a new node "applies" for joining the network, it contacts an existing node in the network. The applicant node is asked to provide its credentials. Once logged-in, the node is assigned a node ID and it advertises its services in the network. Its services are digested using a hash function applied over their names and the pair $<$*service hash value, node's IP address*$>$ is stored in other N nodes whose node IDs are numerically closest to it. When a service request S is performed over a node A in the DMLSAD network by a client application, node A calculates a digest value S_d over the requested service name and queries the network in order to discover the node hosting the service. This node is one of the N nodes numerically closest to S_d which host the service node's IP address or DNS - name. Once a node B from the set of nodes N is discovered, it replies to node A with the IP address of node C hosting S. Sequentially, node A forwards the query to node C, C in turn performs the request over S and returns the result back directly to the client.

Adapters Distributed Back-up and Subsystems Fault-Tolerance

The overlay network is also used for backing-up adapters in a decentralised fashion. When an adapter is loaded, a hash value is calculated over its name, and it is forwarded to the N numerically closest nodes to the adapter's hash value in the network. When an adapter is needed, a hash value is calculated by the node A that needs the adapter and then the network is queried in the same way as it is for the

service discovery. As a result, A receives the suitable adapter and loads it as we have mentioned in 4.6. Using our clean and easy to use API, the newly installed adapter at node A attempts to participate the free vendor specific subsystem to it.
There is a Chimera extension in our design that we are accommodating in order to notify the nearby nodes in case of node A's failure. We are going to use Pastry's neighbourhood table as the above mentioned Chimera extension and thus when a node A fails, all of its neighbours will detect that it has failed. Furthermore, they will query the network for the adapters that the failed node had before. Using the newly installed adapters, the neighbourhood nodes will then try to participate the adapters' corresponding devices to the set of devices accommodated by them. Figure 8 graphically illustrates the distributed high-level design of DMLSAD. Figure 9

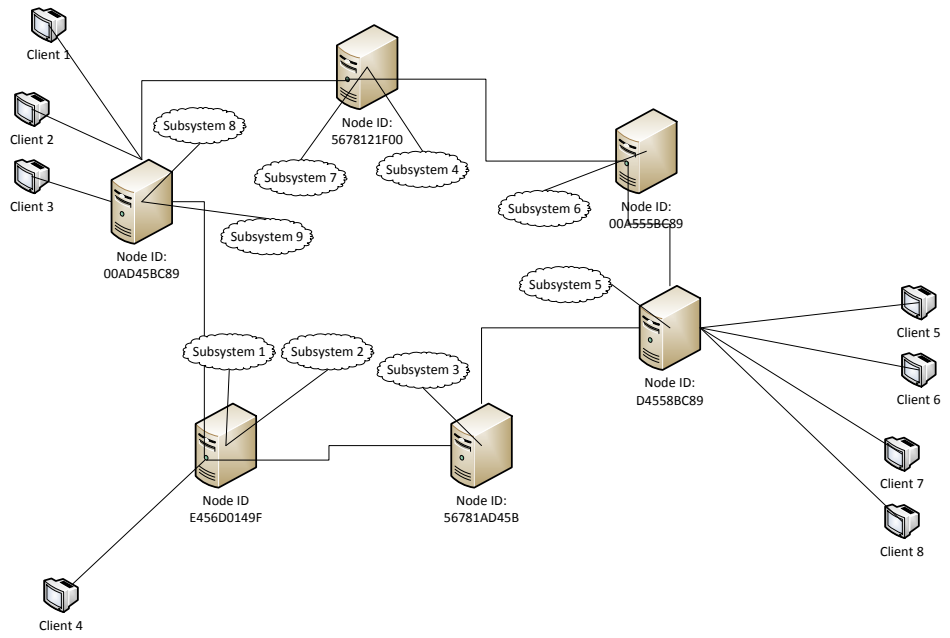

Figure 8: Distributed Middleware for Large Scale Automation Domains High Level Architecture

shows the layers and components available at each node

5.2 Components

In this subsection, we will describe the role of each component in DMLSAD. We will take a look at each logical component in the system. As all the nodes are

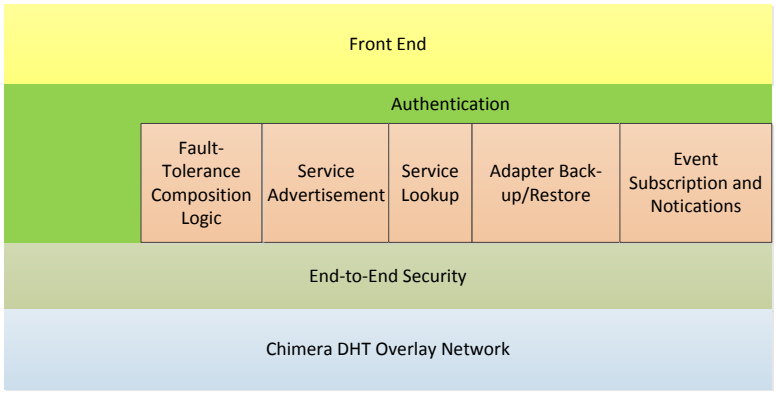

Figure 9: Distributed Middleware for Large Scale Automation Domains Nodes Layered Design

equal, the logical components could be deployed at any node.

Front End Component

As it can be exhibited from figure 8, clients can interact with the network, using any node in the network that has such capabilities. One DMLSAD deployment option is to incorporate front end capabilities in all the nodes participated in the overlay.
The first role of the front end is to provide an entry point for the "outside world" to the services provided by the system. Moreover, when a client requests a service, the front end is the one responsible for locating the node handling the communication with the vendor specific subsystem, providing the service and propagating the result back to the requester. Furthermore, through a session based approach, the front end can send events back to the clients attached to it in case of having a client subscription for an event of service state change. In order to do so, the node that acts as the front end for one or more clients must subscribe to the node hosting the service for receiving service change state event notifications on the behalf of one of its clients interested in receiving such notifications.

Authentication Component

The authentication protocol used in our design uses a distributed authentication service running at each node. We have given our supportive arguments for choosing such an approach and not using a centralised authentication server in 4.6. This approach is applied in such a design, in order to overcome the dedicated

single point of failure, presented on a dedicated authentication server. Moreover, the authentication component needs to be present at all the nodes in instance of DMLSAD. The authentication is the component verifying that a newly participated node is coming from a viable source and it makes sure that only certified nodes can read and change the state of DMLSAD. The following figure 10 represents the authentication mechanism upon a node joining the network.

First, when a node A attempts to join an existing instance of DMLSAD, it

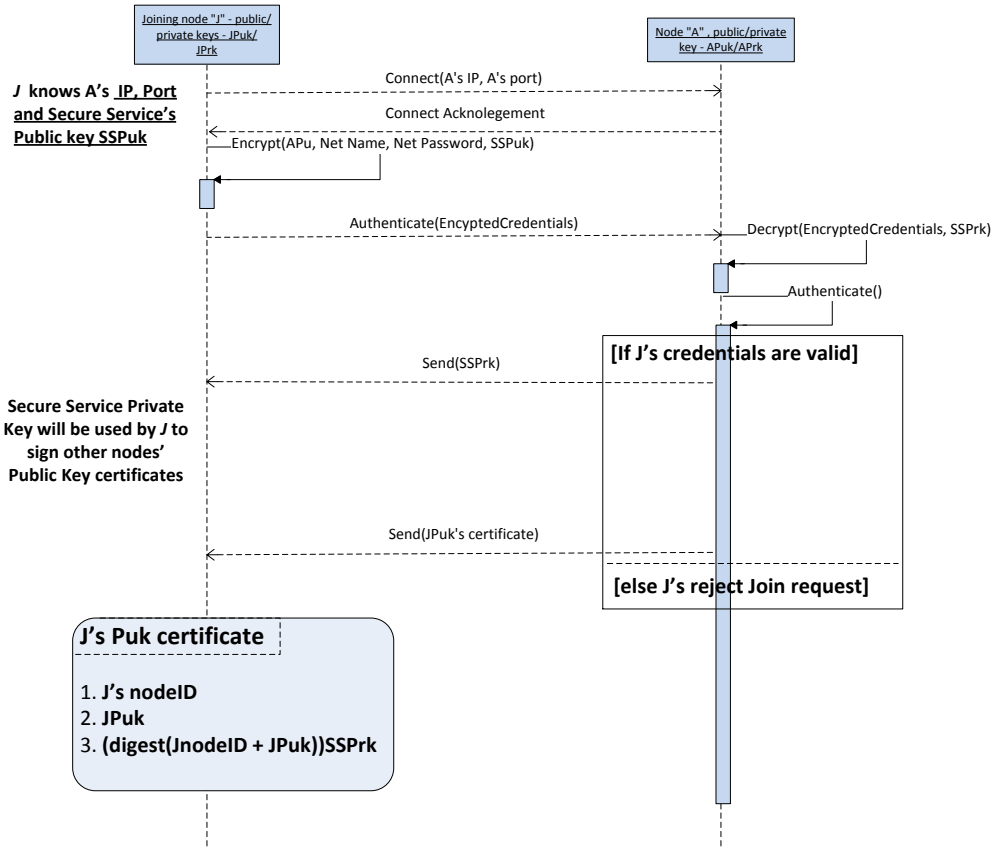

Figure 10: Authentication of new nodes

needs to provide its credentials to a DMLSAD's existing node B. A logs-in with a DMLSAD's name/password and provides its public key encrypted with network's authentication public key Pu. Following this approach, a new node A can be added to an instance of DMLSAD as easily as providing an existing DMLSAD node B's IP address, network name, and network password. On top of the network name and network password, A provides its public key to B as well. B, in turn issues a public key certificate $A_{Pu_{Ce}}$ to A, verifying that A has already successfully logged

within the network. B also sends the authentication private key to A which is used for signing public key certificates. This private key will be used by A's authentication service, when a new node attempts to connect to the network, using A's authentication service. Furthermore, A's public key certificate is used every time when it wants to interact with DMLSAD. This process is further used and explained in End-to-End security component.

End-to-End Security Component

All the communication between communicating nodes is encrypted, such that no unauthorised third party can tamper or read the information transmitted. In 4.6, we have provided our arguments for choosing a hybrid protocol using both asymmetric and symmetric encryption algorithms. A prerequisite for using such an algorithm is that each node has public/private key pairs issued by the authentication component. All the information transmitted from node A to node B is encrypted. In order to do so, Chimera's distributed hash table needs to be upgraded. The upgrade concerns the addition of an attribute that will be stored in each of the routing table's entries. In the original implementation of Chimera, each routing table entry consists of a node id and a node IP address. We are adding a third attribute, which will be the node's public key. This public key is used by a node A when something needs to be transmitted to a particular node B. Indeed B is in A's routing table and A has B's public key.
In this design, there are two scenarios. Different types of protocols are used in both of them. The first scenario is when a message needs to be routed from node A to node B e.g. a service request. In this case, all the intermediate nodes that get the message before the message arrives at B are using only asymmetric encryption algorithm. This is so because the message that needs to be routed would usually be very short, e.g. a service name. The second scenario is when A requests an adapter from B. A hybrid protocol is used to transfer the adapter in this case. The hybrid protocol uses asymmetric key encryption only for establishing the communication session and exchanging the secret key. Sequentially, this secret key is used by both parties for encrypting the pieces of information when sending the actual adapter.

Public Key Verification Figure 11 shows the overall process of public key verification upon a request. When a node A wants to interact with DMLSAD, its public key needs to be verified. This is why we are incorporating a mechanism for verification that a public key belongs to a node that is already within the network. In order to do so, a node R, which receives a request by another node S, verifies the identity of S by using its authentication service's public key. To accomplish this, a node calculates a digest value of the concatenated string of the node ID

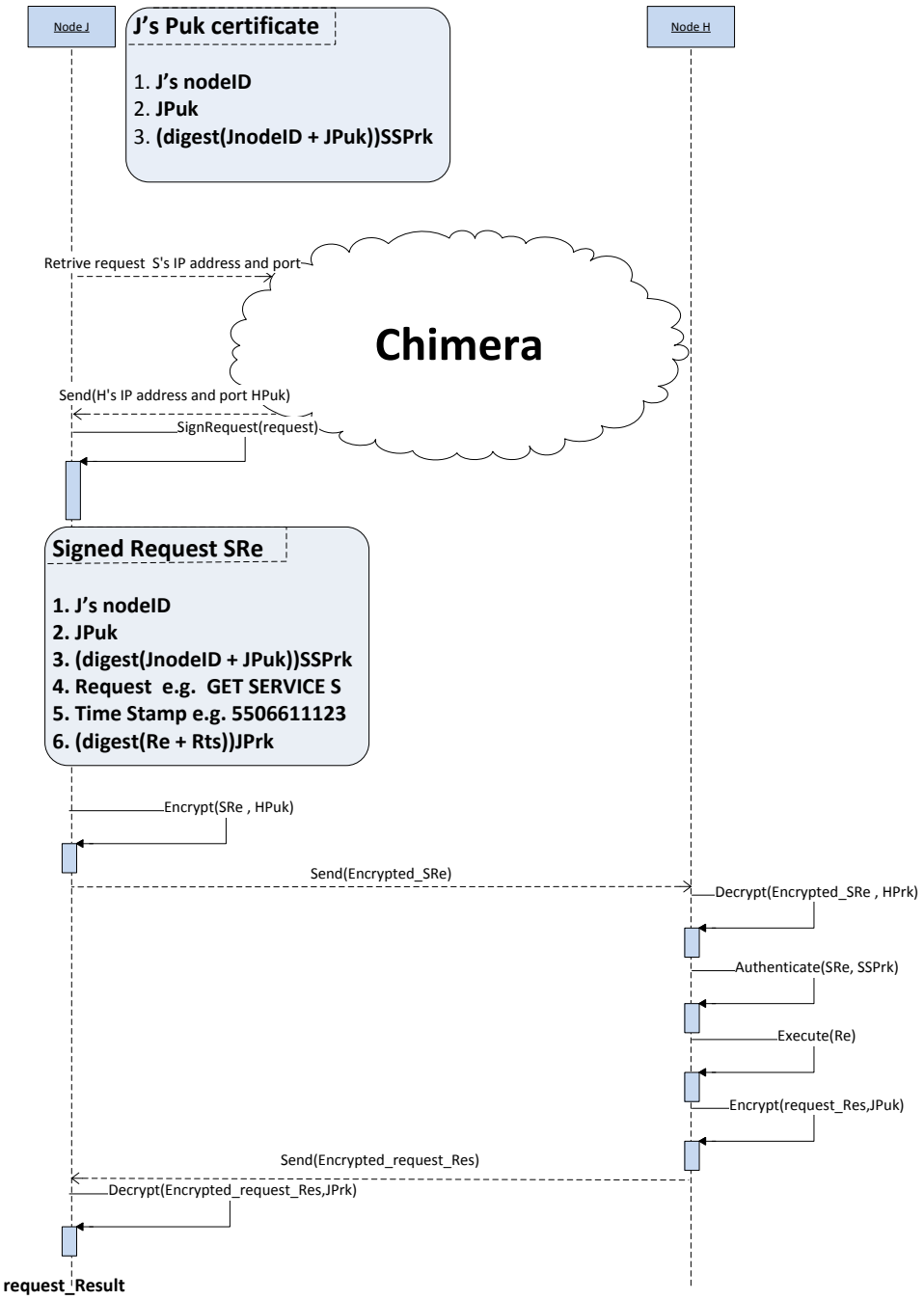

Figure 11: Request Authentication

and node's public key. Then it decrypts the certificate's signature and compares the two values. If they match, then the certificate is valid, otherwise it is not. A's public key certificate has the following attributes:

- A's node ID.
- A's public key A_{Pu} .
- A's public key certificate signature - $(digest(A_{nodeID} + A_{Pu}))_{SS_{Pr}}$. This is a secure digest value which is a result of applying a secure hash function SHA2 over the concatenated string of the node ID's and node's public key and this value is encrypted by the authentication services' private key.

The verification mechanism is applied every time when something is requested e.g. service or adapter. Moreover, every network request is signed by A. A's signed request, in this sense, is a certificate signed by A's private key. Indeed, the signature of each operation can be verified by any other node using A's public key. We have shown above that any node can verify A's public key as well. In this case, we have a message signed by A's private key, which can be verified by another node, using A's public key. The signed request, issued by A, has the following attributes:

- A's node ID.
- A's public key A_{Pu} - A's public key is used for verification request and also for backward encryption of returned result to A.
- A's public key certificate signature - $(digest(A_{nodeID} + A_{Pu}))_{SS_{Pr}}$.
- A's request - Re - Request from Chimera e.g. a service request, an adapter request, etc.
- Request time stamp - R_{ts} - This time stamp is checked by the node that executes the actual service. The time stamp is a number, based on *A's* local time. It protects against "replaying" attacks. If there is no time stamp, a malicious node *Mallory* could intercept a signed request and send it again at a later stage. Once A's request is executed, the node executing the request will store the time stamp and A's node ID. Every time when a request is to be executed, the time stamp will be used to determine that a request has not been executed before.
- Return result public key Cl_{Pu} - This attribute is optional and can be used if a client requests a service and has established a secure communication. This key can be used by an end node to encrypt a service result and return it to the client which initiated the request.

- Return result node's IP address - It is used by the node B having the requested resource. B uses this address for backward communication purposes. If this field is blank, then B uses A's IP address for returning the service result to A. Moreover, if this field is blank, A's public key is used as well and the return result public key is ignored.

- Request signature - $(digest(Re + R_{ts} + Cl_{Pu}))_{A_{Pr}}$. A hash function applied over the concatenated string of the time stamp, client's public key and the operation itself, encrypted with A's private key. Note that this signed request can be decrypted using A's public key which is the third attribute in the signed request. In turn, A's public key is signed with the network secure service's private key and can be verified using the secure service's public key available at any node.

The signed request will ensure that only trusted nodes can use network's services. Note that only the nodes having a resource are verifying the request. This is very efficient, since the intermediary nodes which are visited upon request, do not need to parse the certificate's contents and verify its signature, which is a computationally expensive operation.

Service Advertisement Component

This component propagates node A's services into the network upon A joining DMLSAD middleware network. To do so, A's primary task is to calculate a hash value K over each of its services' names. This hash value K is in fact called *key* in the context of routing overlays. Then A forms a $<K,$ *A's IP address*$>$ key-value pair. Using K, A propagates its IP address into DMLSAD and associates this IP address with K.
In order to store the metadata about a service in DMLSAD, A uses its routing table to find N numerically closest nodes in the network where a service metadata will be stored. N is a configuration parameter, which can be fine tuned in order to determine the number of replicas. On one hand, N must be big enough to provide availability of the already mentioned services metadata with a probability very close to 100%. We gave our supporting arguments to this point in 4.6. On the other hand, it must not be very big since it will require more space for storing the service metadata.
In order to store $<K,$ *A's IP address*$>$, A finds N numerically closest nodes in the following way.

1. A determines whether K is within the range of its leaf set, and if so $<K,$ *A's IP address*$>$ is forwarded to the node B in the leaf set, whose node ID

is numerically closest to K. B in turn saves <*K, A's IP address*> and re-forwards it to N nodes in its leaf set where half of these N nodes are smaller than B's node ID and the other half are bigger than B's node ID.

2. If K is not within the range of A's leaf set, it looks in its routing table and finds the routing table entry E which has the longest common prefix with K. Then A forwards <*K, A's IP address*> to the node stored in E that is closest to A in terms of IP hops.

3. This procedure is recursive and it converges when a node X finds that K is within the range of its leaf set. Then the node X executes the first step of this procedure.

4. The procedure is individually executed for each service offered by A.

Following this procedure, all the services' metadata will be stored in DMLSAD.

Service Lookup Component

The service lookup component takes care of finding a service's location on the network. This component needs to be accommodated by all the nodes in the network. It queries the distributed middleware system using Chimera's message routing mechanism. When a service S needs to be found by node A, A first looks-up in its locally available services. If the service is available in its local set of services, the procedure is over. Otherwise, it calculates a hash value M_v of the service based on the service name. It is important to note that the services need to be given the so called "strong" names, and these names will uniquely identify a service which will aid the service discovery mechanism. Next, A determines whether M_v is within the range of its leaf set. If so, it forwards the message to the node in its leaf set L, which is numerically closest to M_v. L in turn replies with the node H's IP address, which hosts the service requested by A. Otherwise A looks in its routing table and tries to find an entry in its routing table with the longest shared prefix between a routing table entry and the calculated service hash value M_v. There are multiple node IDs stored at a particular entry in A's routing table. Node A forwards M_v to the node B whose ID shares the longest common prefix with M_v and is closest to A in terms of IP hops. This procedure is recursive and node B will repeat it. In each step, the longest common prefix between the current node C and M_v will increase with at least one digit until it reaches a node where M_v is within the range of its leaf set.

Adapter Back-up/Restore Component

This component ensures that an adapter is available in DMLSAD with a probability close to 100%. When an adapter D is loaded in DMLSAD via node A, this component takes care of storing a replica of the adapter in N other nodes. Once again, N is a configuration parameter determining the replication factor and we have given our supportive arguments for using such a parameter in 4.6.
It is part of our requirements, that an adapter is named uniquely. The algorithm for storing an adapter is the same as storing service metadata. The only difference is that the key-value pair is formed from the adapters name's hash value, which in fact is the key and the adapter name itself as the value in the pair. When the numerically closest node R, receives the key-value pair, it first establishes a communication session with the node A, advertising the adapter and downloads the adapter. R in turn stores the <adapter name, IP address and port> key-value pair, which can be queried by other nodes that need to load the adapter. Furthermore, for each adapter A stores a hash value of adapter's name as a key and IP address of the node storing backed-up adapter's binary as its value. The adapter restoration is taking place when a node A is announced dead by its neighbourhood. Each node is pinged by the nodes in its neighbourhood set. If a node dies out, the nodes within its neighbourhood set will detect this. Then the nodes will hash the dead node's node ID and will query the overlay network. They will get response from Chimera, containing all the adapters that the dead node had hosted. Then all of the neighbourhood set nodes will try to participate the free device. The nodes capable of attaching a device interfaced by the same adapter will negotiate which one will finally participate it by using a load balancing algorithm.

Load Balancing Algorithm Sometimes a DMLSAD instance could be deployed in an environment with very intensive service requests. Some nodes in particular could experience a relatively high load because of a high number of requests for the services they offer. In this case, a mechanism for ensuring a service availability needs to be applied. The distribution of the middleware to multiple locations does not guarantee that all nodes will be available and that the system will be strong enough to take incoming requests' load. It needs to find a way to balance the incoming requests. This can be achieved in the following way:

1. Initially all devices are assigned a weight of 1 and attach themselves to the nodes having the least weight sum.

2. A node weight is simply a sum of all the weights of the devices attached to it.

3. At a certain point, e.g. once a day, the weight of each device is recalcu-

lated based on statistics such as performed device requests, used CPU at the node side, etc. the calculated total sum of all devices' weights is divided by the number of nodes participating in the middleware and this equals the node average weight of X. Each node must have a weight in the range of $RANGE = X - MAX$ *(collection of devices' weights)* to $X +$ *MAX (collection of devices' weights)*. This means that the nodes must agree on $RANGE$ numbers and all of them need to be aware of this. Secondly, the nodes having a weight larger than $RANGE$ need to release devices and nodes with less weight than $RANGE$ need to set an adoption mechanism on.

4. From now on all new devices participating in the network are assigned a weight of X on join.

5. If a node fails, perform step 3.

Fault-Tolerance Composition Logic Component

As we have mentioned in 2, some existing home automation systems have the so called "Composition Logic" component, which executes a set of rules or makes transition between system states upon service state change of a subsystem device. In 4.2, we have analysed existing fault-tolerance architectures which we could apply for guaranteed delivery of correct composition logic execution. In 4.6, we have given our arguments for choosing the Triple Modular Redundancy fault-tolerance architecture. In this subsection, we will clarify how exactly this is going to be accomplished.
There are three phases in accomplishing this:

- Choosing composition logic nodes.
- Subscription to service's state change events.
- Result voting and election.

Choosing composition logic nodes Because of the nature of such kind of networks, the nodes chosen to have a composition logic components are chosen dynamically. Initially, there are no nodes in the network. The first three nodes that have joined a DMLSAD instance are chosen to be the composition logic nodes. Each of them is aware that the other 2 logic nodes exist and they know how to find each other. They exchange heartbeat messages on a regular basis. Thus each of them shall verify that the other two nodes are still available. In case that one of them does not respond to the heartbeat messages anymore, the other nodes agree on the absence of the 3rd node and each of them query its neighbourhood set to

find the node that has been on the network for the longest period and then they decide on which one to participate. Querying the neighbourhood set and not the leaf set results in participating a node that is proximally close to one of the other two nodes in term of IP hops. After a certain period of time, this procedure will result in that nodes in the same neighbourhood set become the composition logic nodes in the network. This is beneficial, because first of all, the nodes will receive the event notifications at pretty much the same time, and secondly, the probability for faster algorithm convergence will become higher than in the scenario in which random nodes are chosen.

Subscription to service's state change events Each composition logic component subscribes to all the nodes in the system to receive notification upon service state change. The event subscription/notifications component was discussed in 5.2. The composition event notification delivery "picture" is illustrated in 12

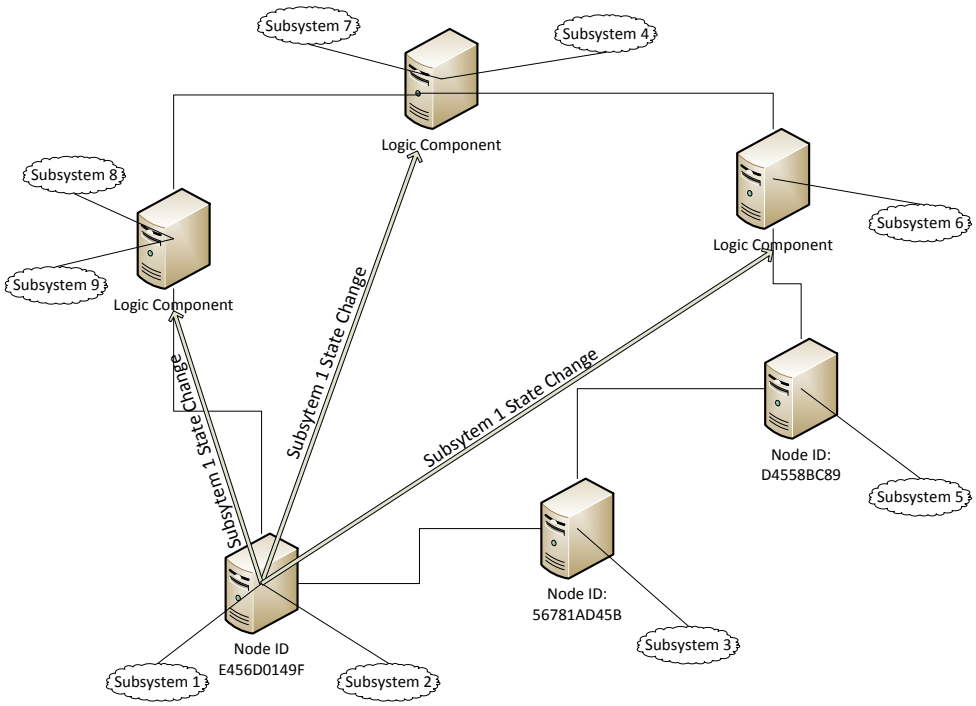

Figure 12: Event Notifications Delivery

Result voting and election The composition logic nodes receive service change states notifications and execute the composition logic. After executing the logic, the three logic execution nodes vote and elect the correct result. Thus service

states are changed only once and the result's correctness is guaranteed with a very high probability. Moreover, the risks of an incorrect result and of potentially triggering wrong actuators are negligibly small.
The algorithm for voting and electing the correct result is based on majority voting. Now let assume that the three composition logic nodes are A, B, C. Upon a service state change, a notification is delivered to the three nodes and their composition logic component execute the composition logic. Let's assume that node A finishes its execution first. In this case, A calculates a hash value based on all of the services' states. A sends the hash value to B and C. B and C receive A's hash value. When they output a result from the composition logic execution, they calculate a hash value over their output as well. Then B and C compare their services' hash values with A's hash value. If the values are equal, they return to A an acknowledgement that the hash values are identical. If both B and C, are different from A's result, A inform B and C that they need to decide on the correctness of the result on their own. Then B sends its hash value to C and asks it whether its result is the same as C's result. If so, B acknowledges C and it will take care of delivering the service updates to the appropriate nodes.
If A, B and C results are all different, new nodes need to be elected as execution engine nodes. This is accomplished by having A, B and C pick a node in their neighbourhood that has been part of the network for longest period.
This component makes the DMLSAD not very scalable, since it needs to keep track of the states of all services. This is why this component is for optional use. Furthermore, it can even be exported and could be run on several client applications.

Event Subscription and Notifications Component

A node X in DMLSAD can subscribe for other nodes' services state change and receive notifications by them upon a change of state. Using 5.2, a node Y's IP address can be discovered having a particular service . Having the IP address of node Y, X can subscribe at Y for a service S state change. Node Y will store X's IP address and it will notify X upon S state change. If the number of nodes subscribed for a particular service become large and DMLSAD is deployed on the same local network, we can use multicast 4.4 to disseminate the change of state to the subscribed nodes. The node hosting a service of interest notifies the nodes subscribed for a service, to listen to a specific multicast address.

5.3 Component Interaction

In this subsection, we are going to describe the component dependencies and interaction. We are going to clarify the interaction, using four scenarios which are

found to be the essential in DMLSAD.

Use of Authentication and end-to-end security All the interactions between components are encrypted and we are not going to discuss the authentication and encryption part in the component interaction here, as the authentication process was discussed in details in 4.6 and 5.2. We are not going to describe the process of authenticating a signed request which was also described in 5.2.

Node Join When a Node *A* joins the network, it first contacts the *Authentication* component, and is asked for its credentials. If *A* provides a valid network name and network password, the *Authentication Component* 5.2 allows newly joined nodes to become a part of the network. After that, *A* can use its service advertisement component to make its services available to the other nodes in the DHT network, using the procedure described in 5.2. Then the *Adapter Backup* component is used as well. The latter component will distributively store the adapter on multiple nodes in the network. Node join use case scenario is illustrated on figure 13

Service Request A client application can request a service in DMLSAD. In order to so, it "contacts" the *Front End* 5.2 component. The *Front End* in turn redirects the client request to the *Service Lookup* component . Using the procedure described in 5.2, the requested service is discovered, the signed request is first verified and then performed over the service. If the verification procedure fails, the request is ignored. Finally, the result is returned back to the client application using its public key and its IP address included in the certificate. Service request use case scenario is illustrated on figure 14

Node Fail/Leave When a node *F* fails or leaves, the *Adapter Back-up/Restore* 5.2 component takes over in order to participate the devices orphaned by *F*. In order to do so, a statistics by the *Load Balancing* component 5.2 is used. This will ensure that the freed devices will be participated to the nodes having the least service requests.

Service State Change For deployments of DMLSAD with *Composition Logic Fault-Tolerance Component* 5.2, it should be clear to the reader that composition nodes need to subscribe for service change events using the *Event-Subscription and Notifications* component 5.2.

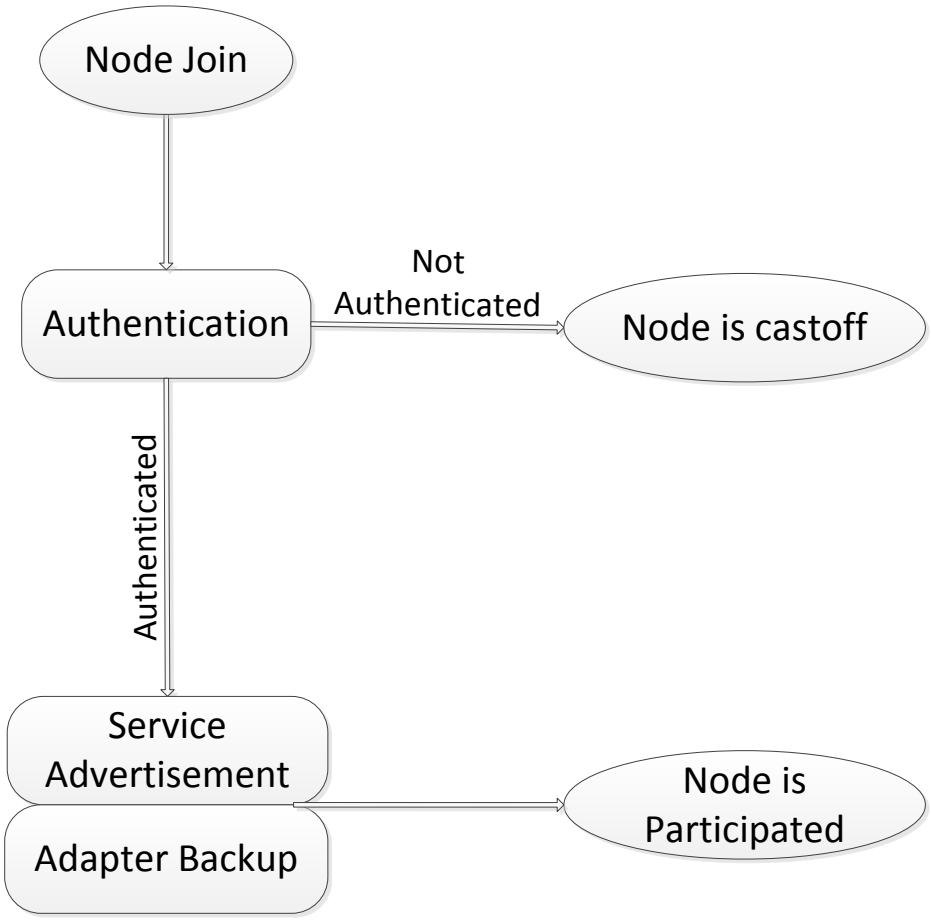

Figure 13: Node Join Scenario Components Interaction

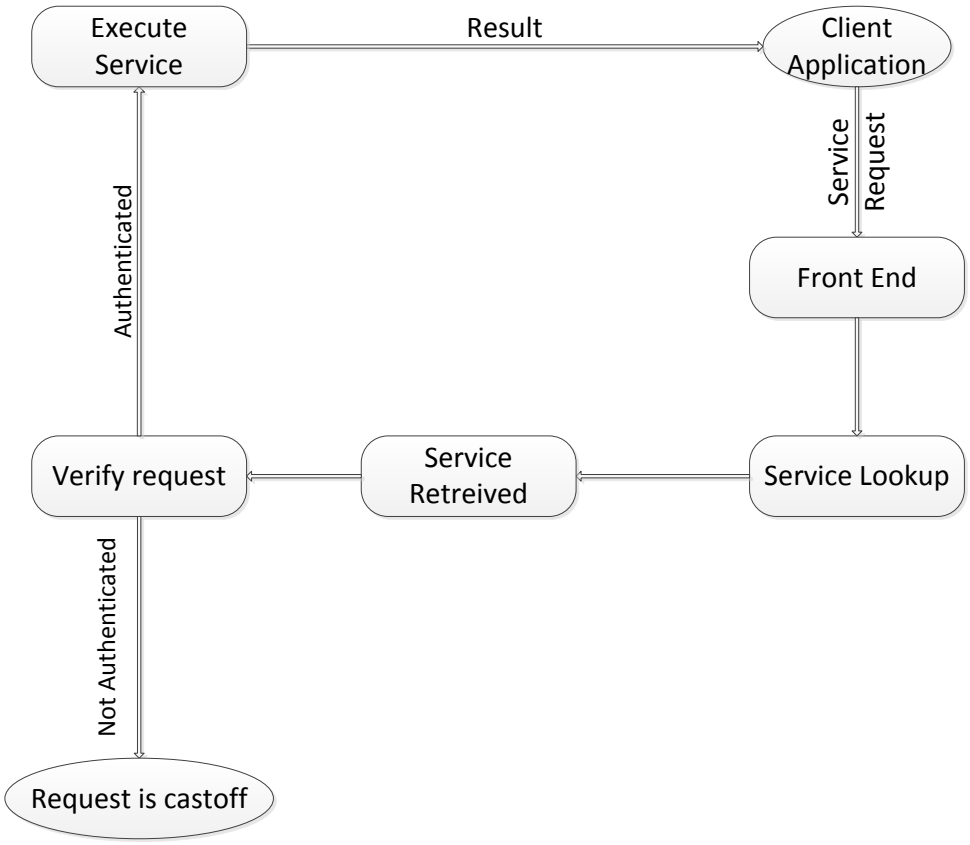

Figure 14: Service Request Scenario Components Interaction

6 Implementation

In this section, we are going to discuss about the implementation that we have conducted in order to prove our design 5. As our underlying platform, we are using the already existing home automation interoperability system HomePort 2.8 and in particular - its latest design. Moreover, we add the extra layers in order to distribute it. Figure 15 shows how the DMLSAD's architecture is incorporated into the HomePort's architecture. The greyed components and layers have not been implemented yet. In this chapter we will discuss about the components and the layers in DMLSAD, that have already been implemented and how it is incorporated into HomePort.

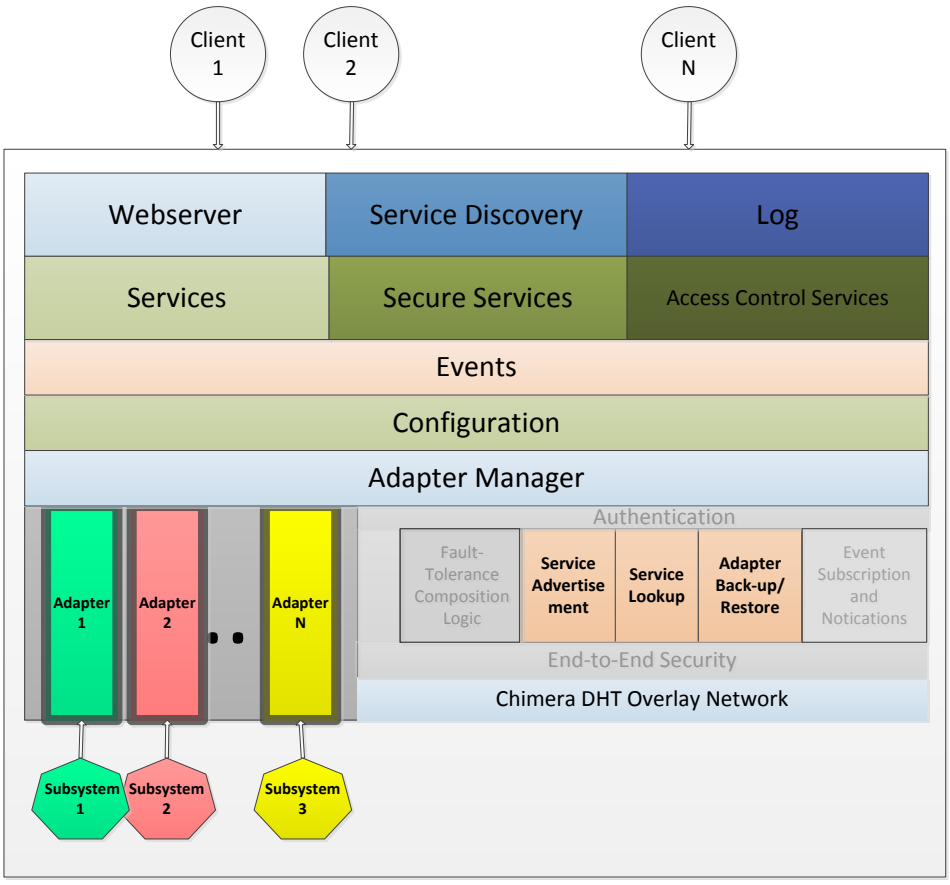

Figure 15: DMLSAD embedded into HomePort's architecture

Firstly, we will start by describing the Chimera DHT library 6.1. Secondly, we

will proceed by describing the HomePort functions 6.2 that are directly related to DMLSAD. Finally, we are going to describe the implementation of DMLSAD in the context of HomePort 6.3.

6.1 Chimera DHT library

As we previously mentioned, our implementation will make use of the Chimera's distributed hash table. The Chimera library exists implemented in the C language and this is the library used in the implementation of our project. This library gives a basis to create and work with a distributed hash table. In this part, we will provide more details about the library itself to give a better understanding of the way it is working.
The entry point of the library is the function:

- *ChimeraState chimera_ init(int port)*

This function is used to start an overlay network. It also starts the routing system of Chimera, its job queue, message layer, etc. After the network is started and is running, it becomes possible for nodes to join it. This is done with the function:

- *void chimera_ join(ChimeraState state, ChimeraHost bootstrap)*

The first node has to initialize the network, so no bootstrapping is needed, however the other joining nodes have to know at least one node in the network and use it as a bootstrap host by specifying its IP address and port.
Chimera also allows to set a custom node ID if needed.

- *void chimera_ setkey(ChimeraState state, Key key)*

For it Chimera uses a 160 bit key which corresponds to the SHA1 hash of the string "name:port" (computer name and port of the running Chimera node).
We have to use following function to register every custom message type that is necessary for our application purposes.

- *void chimera_ register(ChimeraState* state, int type, int ack)*

Only registered message types will be recognized and routed appropriately by the messaging system of Chimera.

- *void chimera_ send(ChimeraState state, Key key, int type, int size, char × data)*

This function routes a message through the Chimera network containing *size* bytes of data. Data will be send via the Chimera network to be delivered to the host closest to the key. When there is necessity to find a host within the network, the following function is used:

- *ChimeraHost host_ get(ChimeraState state, char hn, int port)*

It will return a ChimeraHost structure for the name which was searched. Chimera also allows for ways to add custom actions when certain event occurs. The first one occurs when a host leaves or joins the leaf set of a local node :

- *void chimera update(Key key, ChimeraHost host, int joined)*

The second is called before the routing layer forwards a message toward a destination key through some intermediate host:

- *void chimera forward(Key key, Message msg, ChimeraHost host)*

This allows the application to intercept the message and change the parameters to override the routing choices done by the Chimera routing layer or to modify the content of the message on the way.
The last one is triggered when the current node receives a message destined for a key that is within its leaf set:

- *void chimera deliver(Key key, Message msg)*

This indicates that the message has arrived at its final destination.

It might also be important to know about the neighbours of a node. This can be checked with the function:

- *ChimeraHost route_ neighbors(ChimeraState state, int count)*

This will return an array containing the nodes that are closest to the current node and in its leaf set.

6.2 HomePort API

The HomePort is a big project, containing a lot of elements. That is why we will concentrate here on what is the most interesting part for us: the adapter loading API, which was built in our previous work [18]. Currently, the HomePort adapter loading sequence is working this way:

- First, the function needs to know the path of the adapter that needs to be loaded. In our case, this can be provided manually or automatically in case of recuperation after node crash.
- Secondly, it uses this path to find and execute the adapter's binary file.
- This creates a new process that is placed into a so called sandbox to prevent it from doing any harm to the system or overusing its resources.

- After this, three communication pipes between the HomePort and the adapter are created. Two of them are used for communication between HomePort and the adapter. The last pipe is used to send the messages about new services that has to be registered with the HompePort.

- At this point adapter binary is loaded and all available services can be registered with HomePort.

It can be inferred from figure 15, that DMLSAD is used by the Adapter Manager layer which is not presented on latest HomePort architecture 6, since it was developed in our previous project and it has not been reflected on the latest HomePort architecture [18]. In this subsection, we are going to describe the main functions of the Adapter Manager component in HomePort, since it is the only layer that directly interacts with DMLSAD. On one hand, the Adapter manager loads and unloads adapters from the local machine. On the other hand, it directly interacts with the components available at DMLSAD. For instance, when an adapter is loaded by the Adapter Manager, it also stores the adapter binary at other remote nodes using the Adapter Back-up/Restore component available at DMLSAD. Complete description of all functions available at the Adapter Manager can be found in our previous project and here we are going to provide only a brief description about some of the Adapter Manager's essential functions. Firstly, we will describe the function involved in dynamically loading of an adapter 6.2. Secondly, we will proceed by providing a description of the functions involved in service registration 6.2. Finally, we shall describe the functions that are responsible for forwarding a GET/PUT request message from the HomePort daemon to a particular adapter 6.2.

Loading Adapter

- *void load(char)* - It loads an adapter, which is specified by the input formal parameter of this function. It creates a separate process for the adapter, which is also sandboxed to run with restricted system permissions and restricted set of resources. Three communication pipes are created between the adapter and the Adapter Manager processes: two of them for communication between these two and one for registering new services by the adapter.

Service Registration

- *ServiceMsg crServMsg(char description, char ID, char type, char unit,char device, char get_function, char put_function, char parameter)* - Creates a data structure which describes a service. The essential formal parameters are the names of the put_function and the get_function. These function names

are eventually used by the HomePort daemon to find and call the actual put and get functions in a particular adapter upon a HomePort service request. Device and Parameter structures are created analogically. They are also describing some important parameters needed to register a service with the HomePort daemon.

- *void wait()* - It is called at the Adapter Manager process and runs in an infinite loop in the separate thread waiting for an adapter to send messages containing the service information for registration. These services are further registered within HomePort.

- *void regService(ServiceMsg s, DeviceMsg d, ParameterMsg p, char (get)(ServiceMsg), char (put)(ServiceMsg))* - Is called when a service registration message arrives from the adapter. It registers the newly created service within the HomePort.

PUT/GET requests processing HTTP PUT and GET requests received from the client application on a service are processed in pretty much the same way, so we are going to describe only the GET request and mention the differences between PUT and GET requests processing. The process stars when HomePort daemon receives a PUT/GET request, then the corresponding adapter is found and informed about this request. After that the appropriate PUT/GET function is executed in the adapter process. The result is then returned back to the HomePort daemon. Here are the functions that are executed after receiving the request.

- *get (Service service, char buffer, size_t max_buffer_size)* - Main function for handling all GET requests locally (at HomePort daemon process) and forwarding them to the adapter process.

- *char findGet(char ID)* - Finds the name of the get function to be executed in the adapter process.

- *int findPID(char ID)* - Finds the process ID of the adapter that has to execute the requested "get" function.

- *void send_service(ServiceMsg s, int pipe)* - Sends the processed request to the adapter process using the appropriate communication pipe and waits for a response from the adapter. As soon as it gets the response, it returns the result to the HomePort daemon which in turn forwards it to the client application where the request originated.

Here are the functions executed on the adapter's side.

- *void waitForMsg(int pipes[2])* - Waits for incoming PUT/GET requests from the HomePort deamon.
- *ServiceMsg receive_sr()* - Receives service's (on which the request was performed) description from the HomePort daemon and initialises local ServiceMsg structure.
- *void findFun(char name)* - Finds the local function that has to be executed to satisfy the request.
- Executes function returned by *void findFun(char name)* and returns the result to the HomePort process.

The only difference between a PUT and a GET request processing is that a PUT request requires the client application to send one more parameter that defines the value, that has to be set on the service, to the adapter process and the adapter process in turn executes a PUT function on its side.

6.3 Distributed and Fault Tolerant HomePort

In this part, we will explain our actual implementation of a distributed and fault tolerant home automation system using HomePort as a basis for the home automation part and Chimera for the network part. First, we will describe the essential functions of the distributed HomePort and then we will illustrate and explain the event flow in our system in the actual order of happening to be as clear as possible.

Here is an explanation of the most essential functions used in the implementation of the distributed HomePort.

- *int message_send (void chstate, ChimeraHost host, Message message, Bool retry)*
- *void chimera_send (ChimeraState state, Key key, int type, int len, char data)*

These two functions are used to send messages through Chimera. The first one delivers a message directly to the destination using the IP address of the destination node. The second one routes it through Chimera using only the key.

- *void delivery_handler(Key key, Message msg)*

This is the main message handling function of Chimera messaging system. It is called each time when the message reaches its destination. It allows to take appropriate actions to handle each message type.

- *void startPing()*

This is the function that starts the thread responsible for keeping track of the other nodes. It helps to find out when the node in the network dies and needs to be replaced. This is done by sending periodical "pings" to a few random nodes. If the counter exceed the predetermined allowed number, the node is considered down and the program will start looking for a replacement for that node, which should host its devices by taking over and running appropriate adapters. This is done in the function:

- *void lookForReplacement(char deadHost)*

This function, which is called after a node goes down, starts the procedure to reconnect adapters of the dead node to some other suitable host.

- *void ReceiveFile(char ip,int portno, char ServerFilename[],char ClientCopyFileName[])*

This function is responsible for the transfer of an adapter's binaries. It is used when backing up or restoring an adapter after the crash.

We have provided an explanation of a few essential functions implemented and used in the distributed HomePort middleware. Now we are going to describe the execution workflow of the main tasks performed by the distributed HomePort, sequentially. First, any node boots-up and this is explained in the start-up section 6.4. The startup process creates a pinging thread and the pinging workflow itself is explained in 6.5. A service component is responsible for advertising new services and this is explained in 6.6. Finally, we provide a description of the adapter restoration procedure in 6.7.

6.4 Start-up

This procedure initialises a node that is joining the distributed HomePort system. Figure 16 illustrates the execution flow.

Now let describe the set of functions executed upon start-up.
First of all, the HomePort daemon is launched using the function:

- *HPD_start(HPD_USE_CFG_FILE, "HomePort", HPD_OPTION_CFG_PATH, "./hpd.cfg")*

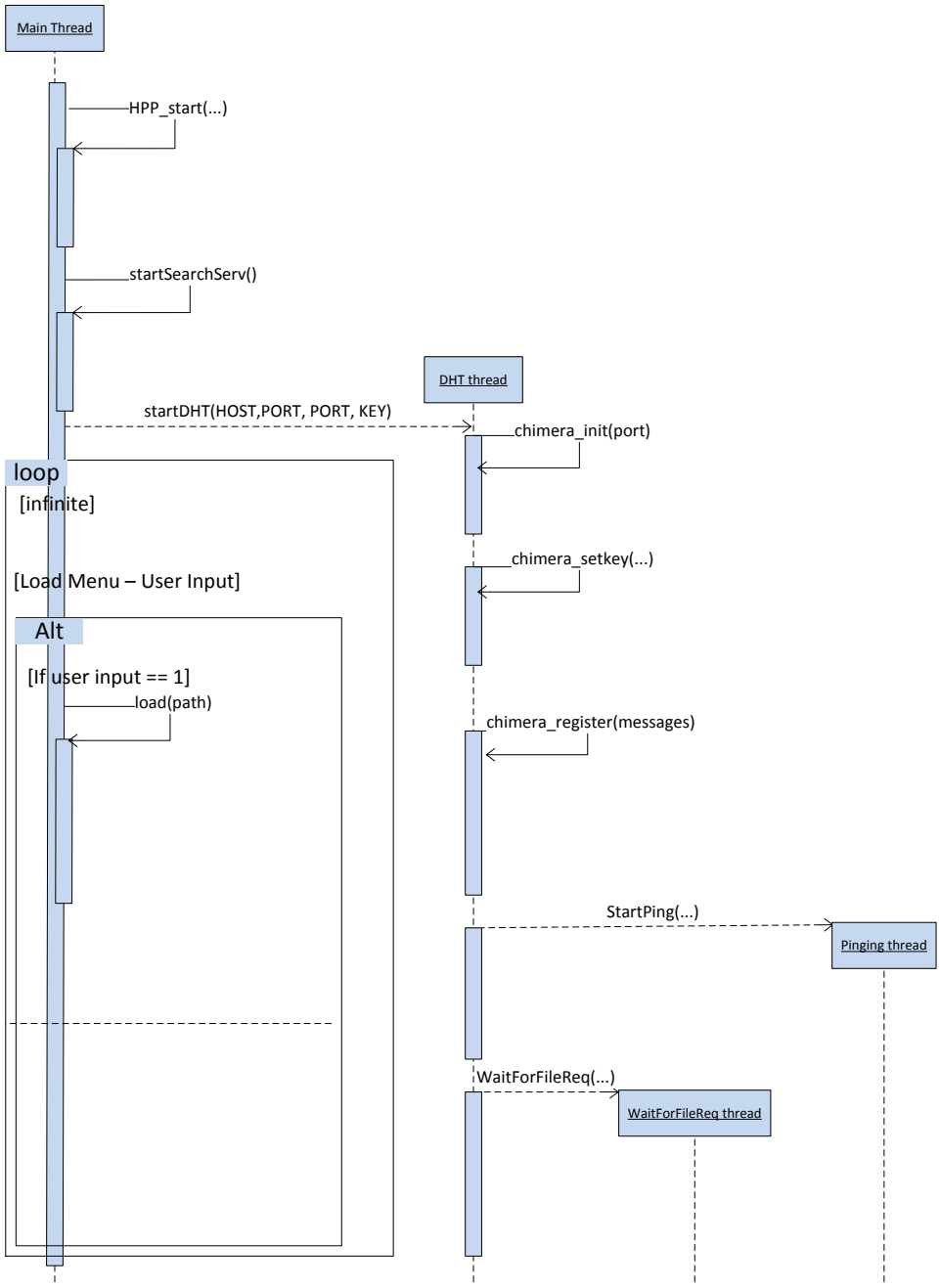

Figure 16: Startup Message Sequence Diagram

Then we are launching the so called *search service adapter* described in the design chapter 5.2. It is a very simple HomePort adapter where a GET function returns

the status of the node and a PUT function allows to look-up for the address (metadata) where a specific service is hosted. (The service metadata advertisement procedure is described in our design 5.2). Finally, the *search service* is registered within HomePort.

From this point in, the execution flow is divided into two threads. The main thread displays a menu that allows users to load adapters on the local node and the other thread will start executing functions necessary for joining and participating in Chimera P2P network (DHT thread).

Distributed Hash Table (DHT) thread This thread executes a function called *startDHT(Chimera_IP, Chimera_Port, local_port, nodeID)* . Here the first two arguments are an IP address and a port of an existing Chimera node (used for bootstrapping), and local_port is the port that the newly joining node will listen to for incoming subsequent requests from other HomePort nodes. The nodeID specifies the key which will identify the node in the peer-to-peer network. Here is a chronological explanation of the set of actions executed by the *startDHT(...)* function:

- A listening port is opened and Chimera host structure is initialised by using the function *chimera_init (port)*.
- The node ID is set in Chimera using the function *chimera_setkey (state, key)*.
- All the message types needed for identifying messages used by our application are registered within Chimera. Such messages are PUT_COMMAND, GET_COMMAND, ADAPTER_RESTORE, etc.
- Pinging thread is started. It will be explained 6.5
- An adapter binary transfer (WaitForFileReq) thread is started. It is used for sending an adapter, upon request, to a remote location. Further details about the adapter transfer procedure will be discussed in 6.7.

6.5 The Pinging process

In our project, "friends" is the term to call the nodes that the current node keeps track of. In our implementation each node has three friends that can ping it and up to four that it can ping (of course these numbers can be fine tuned if necessary). This difference is made to always leave some room for new nodes to join the network. These friends are established through the pinging procedure. The sequence diagram of the pinging procedure can be found in figure 17. Let us start with the main thread.

Main Pinging thread

The execution of this thread goes in the infinite loop which starts by checking if we still need more nodes to ping the current node. If we still need some (MyPingers list is not full), then the steps to add a new node, that will ping the current one, are started. First, we will call this function:

- *void searchForPinger()*

It corresponds to the main function which will be used to search for pinging nodes (pingers of the current one). We will then generate a random key. Next, we will create and send a ping request message to the node responsible for the random key we just generated:

- *void chimera_send (ChimeraState state, Key key, int type, int size, char _*data)*

After this, we will send pings to all the nodes which are already in the current node's pinging list. The main function for this is:

- *void pingFriends()*

As we know the direct addresses of the nodes that we are pinging, we can use the following function to send pings.

- *int message_send (void chstate, ChimeraHost host, Message message, Bool retry)*

Each time we ping someone, we have to increase a counter for that friend. This counter is used to keep track of unanswered pings. So if it reaches a certain limit (threshold), the friend is declared dead. The counter is reset only when we get a ping response from that friend.
Back in the main loop, the next step will involve checking if any counter reached the threshold. The function used is:

- *void check()*

We start by going through the list of friends to ping. We have two different cases here: if the counter value is more than zero (but less than the threshold), then we will just display a warning. But if the counter exceeds the defined threshold, then the node whose counter we checked should be declared dead and we have to start searching for a replacement for that node. This is done with the function:

- *void lookForReplacement(char deadHost)*

This part is explained in the restoration process 6.7.
Then we will sleep for a defined time (10 seconds in our case), and then start the next iteration of the main loop. Every Nth iteration of the main loop, the neighbourhood set will be refreshed with the function:

- *void storeNeighbours()*

The first step in *storeNeighbours()* is to get a fresh list of neighbours. To do this we are using the following function:

- *ChimeraHost route_ neighbors (ChimeraState state, int count)*

And calculate a hash of the localhost key + "neigh". Next, we remove the old neighbourhood set entries from the DHT using a DHT_RM message together with the hash value that we just calculated:

- *void chimera_ send(state, newKey, RM_ COMMAND, sizeof(HPMessage), (char)new)*

After that we can store the new neighbourhood set entries using a DHT_PUT message. This finishes the tasks of the main pinging thread. Further tasks are done on the node that actually receives the ping message.

Message handling DHT thread

This is the thread responsible to receive and handle incoming messages and pings. The main function in this thread is:

- *void delivery_ handler(Key key, Message msg)*

This function handles all the message types that are received by a node. Now we will analyse the part responsible for handling DHT_PING messages as we want to show what happens when a "ping" arrives at the node. This part of the mentioned function consists in two different cases. If the value of the message is equal to "REQ" (the request to become a pinging friend), then we will try to add the source of the message in the friends to ping list with the following function:

- *int addFriend(char name, char address, int port)*

To do this, we check if the source is suitable as a friend to ping (it could be not suitable if, for example, it is already within the list). We then check if we have any more room in the friends to ping list. If indeed there is some more space, then we will add data about the source of the message to the list of friends to ping. So during the next iteration of the main loop in the pinging thread, this node will be pinged during the execution of the *PingFriend()* function. Otherwise,

if the *addFriend(..)* function was not successful, we will forward the message to a random host using *getRandomKey()* and *chimera_send()*.

The second case is if the message value is equal to "PING" (indication that the friend is checking if this node is still alive). In this case we are calling:

- *void gotPing(char name, char address, int port)*

In this function, firstly we will check if the source of the message is not already in MyPingers list and if the MyPingers list has any room left. If the conditions are satisfied, the source of the message will be added to MyPingers list. After this, we will just send a DHT_PING_RESPONSE message directly back to the source.

The other part of the *void delivery_handler(Key key, Message msg)* function is responsible for handling DHT_PING_RESPONSE messages. Here we will show what happens when a message is received. We will try to find the source of the message in the friends to ping list and after finding it, we will only have to reset its counter to zero. This indicates that the friend is alive and no further action should be taken.

6.6 The Registration procedure

This is the procedure which is initiated when a new service or adapter needs to be registered. It starts when a service registration request is received from the adapter process. Figure 18 shows the sequence diagram of this procedure. Let us start with an explanation of the first part of a procedure that is carried out in the thread that is waiting for messages coming from adapter.

Communication thread

This thread is the one which is waiting for some message coming from the adapter. There are two different cases but both are part of the following function:

- *void wait()*

In the first case, the received message will correspond to a new service registration. In that case, we will first calculate the hash of the service name with the function:

- *char sha1_keygen1 (ID, digest, KEY_SIZE/BASE_B, power(2,BASE_B))*

Then we will put this hash in the DHT together with the IP address and the port of the local node by sending a DHT_PUT message. This way the service's metadata is stored in DHT. We are also using a function *HPD_register_service(service)* to

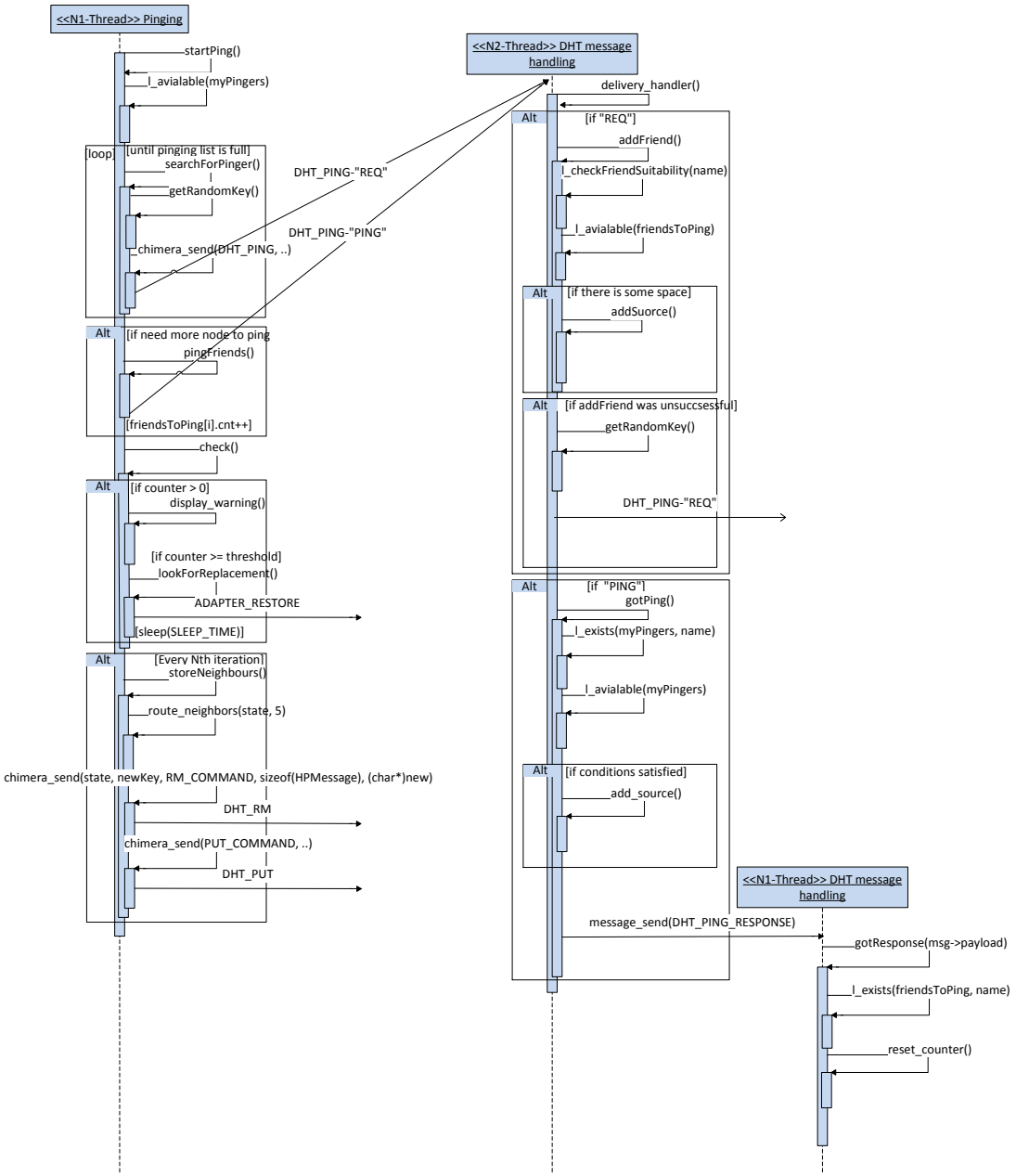

Figure 17: The Pinging Process

register the same service with HomePort daemon.
The second case is when we receive a new adapter ID. In that case, we will first calculate a hash value of the local node's key with the function:

- *char sha1_keygen1 (MY_KEY, digest, KEY_SIZE/BASE_B, power(2,BASE_B))*

This hash value will then be put into the DHT together with the name of the adapter by sending a DHT_PUT message again. The next task is to store the adapter's binary and DHT entry about the binary's location in the same node. This is done in:

- *void storeAdapter(char ID)*

In this function, we are calculating a hash value of the adapter's name:

- *char sha1_keygen1 (ID, digest, KEY_SIZE/BASE_B, power(2,BASE_B))*

And this hash value is placed into the DHT with an indication that this is an entry about an adapter's location. This step marks the end of the tasks in this thread, but they continue on the node N that stores the mentioned DHT entry and the binary file of the adapter.

Message handling DHT thread on N

This thread is executed on the node that is storing the key value pair sent in the previous message. It executes a part of message handling the function responsible for DHT_PUT messages:

- *void delivery_handler(Key key, Message msg)*

In this step, we first test if the message received is an entry about an adapter's location. Once this is confirmed, we will download the adapter binary from the source of the message with the function:

- *void ReceiveFile(dhtmsg->OrgAddress, dhtmsg->OrgPort, loc, loc)*

This part is explained in more details in the file transfer section 6.7. After this, we can finally put the entry about the binary location to the local hash table. The key of this entry is the one received in the message and the value is the local IP and port number. The following function *void dht_put(char block, char value)* is used to store the entry (key, value pair).

6.7 Restore Component

We have already described in 6.4 how a pinging thread is started and we have described the execution workflow of the pinging thread in 6.5. The set of actions in the restore component are initiated by the "pinging" thread in its *check()* function after a node is declared dead. The *check()* function in turn calls the function *lookForReplacement()*, and this is how the adapter restore procedure is initiated. A digest of the dead host key is calculated and a message structure is initialised,

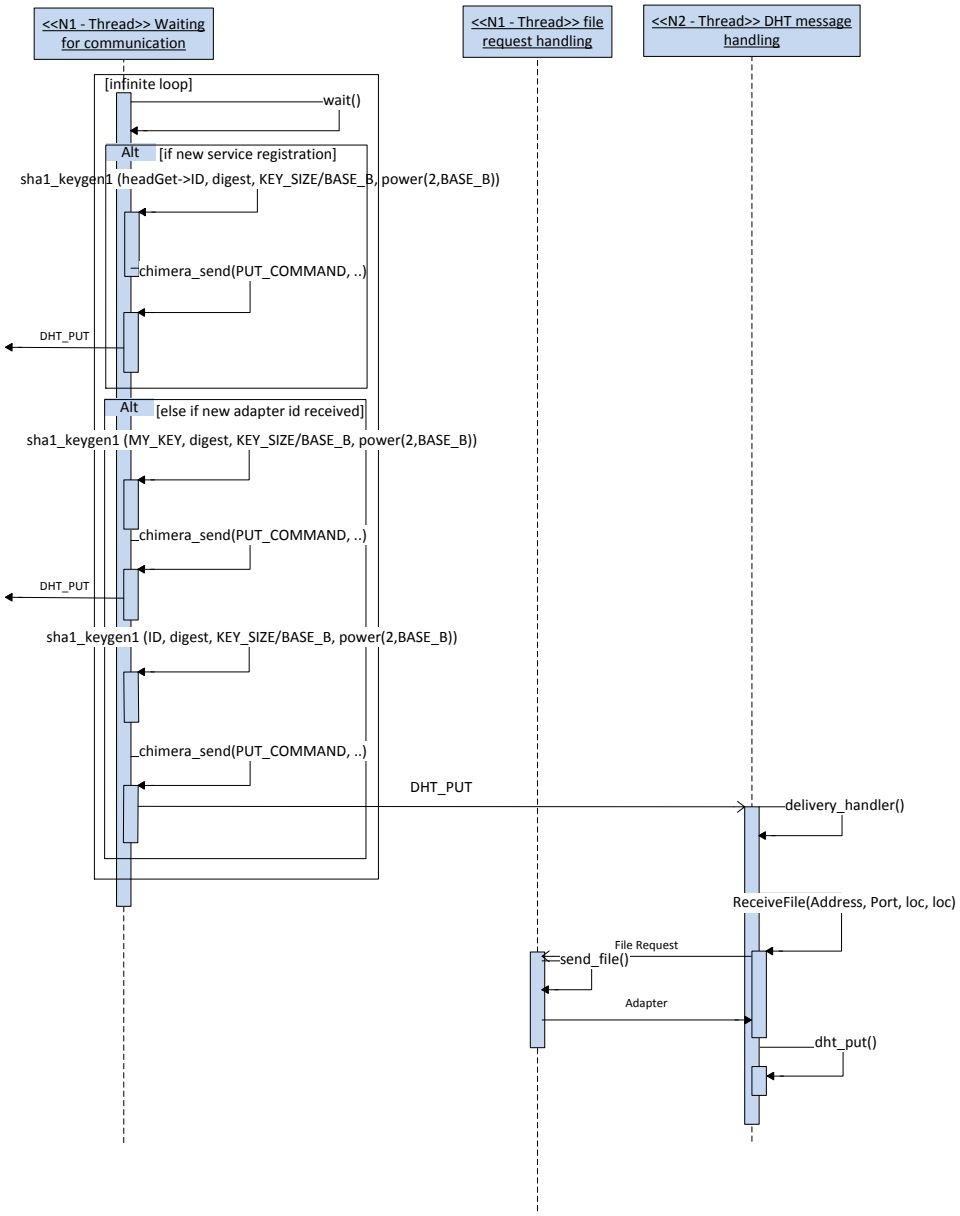

Figure 18: The Registration Process

which contains the newly calculated hash value, the key of the dead node and some additional parameters used for backward communication with the pinging node.

This message is propagated via Chimera to find meta-information about a dead node. The meta information consists of a list of devices that used to be connected to the dead node.
Then a node that has the relevant meta-information will receive a request and the "pinging" node will wait for a response in the blocking function *waitForRespList(device_List, random_nodeID.* This response will contain all descriptions of all devices (actually the names of device's adapters) that were connected to the dead host just before dying. Then the restoration procedure should be started for each device in the list. The restoration procedure is started by sending an ADAPTER_RESTORE message. This message is used to notify a node in Chimera that an adapter restoration must take place.
We have described what actions a pinging node will take in order to start the restoration procedure. Now let's describe the set of actions taken by the nodes receiving an ADAPTER_RESTORE request. When a message arrives at any node, a delivery hander is called which handles the message arrive event in Chimera. These events are handled on a separate thread that we call the DHT message thread. Some of the important types of messages that we use in Chimera are:

- PUT_COMMAND
- GET_COMMAND
- DHT_PING
- DHT_PING_RESPONSE
- ADAPTER_RESTORE
- ADAPTER_RESTORE_FWD
- ...

Message ADAPTER_RESTORE As we are describing the adapter restoration execution workflow, we are going to discuss about what happens when ADAPTER_RESTORE and ADAPTER_RESTORE_FWD messages are received. First of all, when an ADAPTER_RESTORE message is received, it is decoded. This will be beneficial as the receiving node will be capable to get information about a failed node such as its neighbours. A concatenated string is created from the dead node's key and the string "neigh". Then *chimera_send(GET_COMMAND, digest(deadNode+"neigh"),...)* is used in order to get the list of dead node's neighbours using the function *waitForRespList(...)*. The next step taken here is to send an ADAPTER_RESTORE_FWD message to the first (closest) neighbour in the list using the *chimera_send(...)* function.

Message ADAPTER_RESTORE_FWD Such a message will be received by a neighbour of a dead node. First of all, an acknowledgement is sent to the originator of the message. Secondly, if the node is capable of running the adapter, it parses the message content and gets the IP and port of the node hosting the binary file of an adapter. Then a procedure for adapter restoration is continuing with the function *ReceiveFile(...)*. This is described in details in 6.7. Once the file is copied locally, it is loaded by the HomePort plug-in manager with the function *load(adapterPath)*. If the node is not capable of running the adapter and the message hops are less than 5 (this means that we still have some unvisited neighbours in the list), the node executes the same set of actions as in the ADAPTER_RESTORE handler to forward the message further. If hop count is more than 5 but less than 70 (the limit that we set for a maximum length of restoration procedure), the request is propagated to a random node which is determined by a function which generates random nodeIDs *getRandomKEY()*. Then the function goes in a loop and waits for an acknowledgement. However, if an acknowledgement is not received, an ADAPTER_RESTORE_FWD message is sent to a different node. If an acknowledgement is received, the function returns. In case when the hop count is more than 70, the message is simply discarded and the effort to restore the adapter is abandoned. The whole procedure illustrating the interaction between end point nodes is illustrated on figure 19.

Adapter Transfer

In this subsection, we are going to describe the implementation of the Adapter Back-up/Restore Component. Note that this in fact is only one part of the adapter back-up/restore component and namely, that is the part waiting for incoming file requests.

Adapter Request Listener As we have mentioned in 6.4, this component runs in a separate thread which we call the *WaitForFileReq* thread. In order to set-up a procedure for file transfer, the following set of functions are executed in the *WaitForFileReq(port)* function running in the component waiting for adapter transfer requests:

- First a function *initSoc()* is called which sets-up and binds the process to a socket where the thread will listen for incoming messages from nodes requesting a file.
- Then the function executes an infinite loop. In each iteration of this loop, it waits for a new connection from the client that intends to receive a file. A function *SendFile()* is called for this reason. This function blocks by

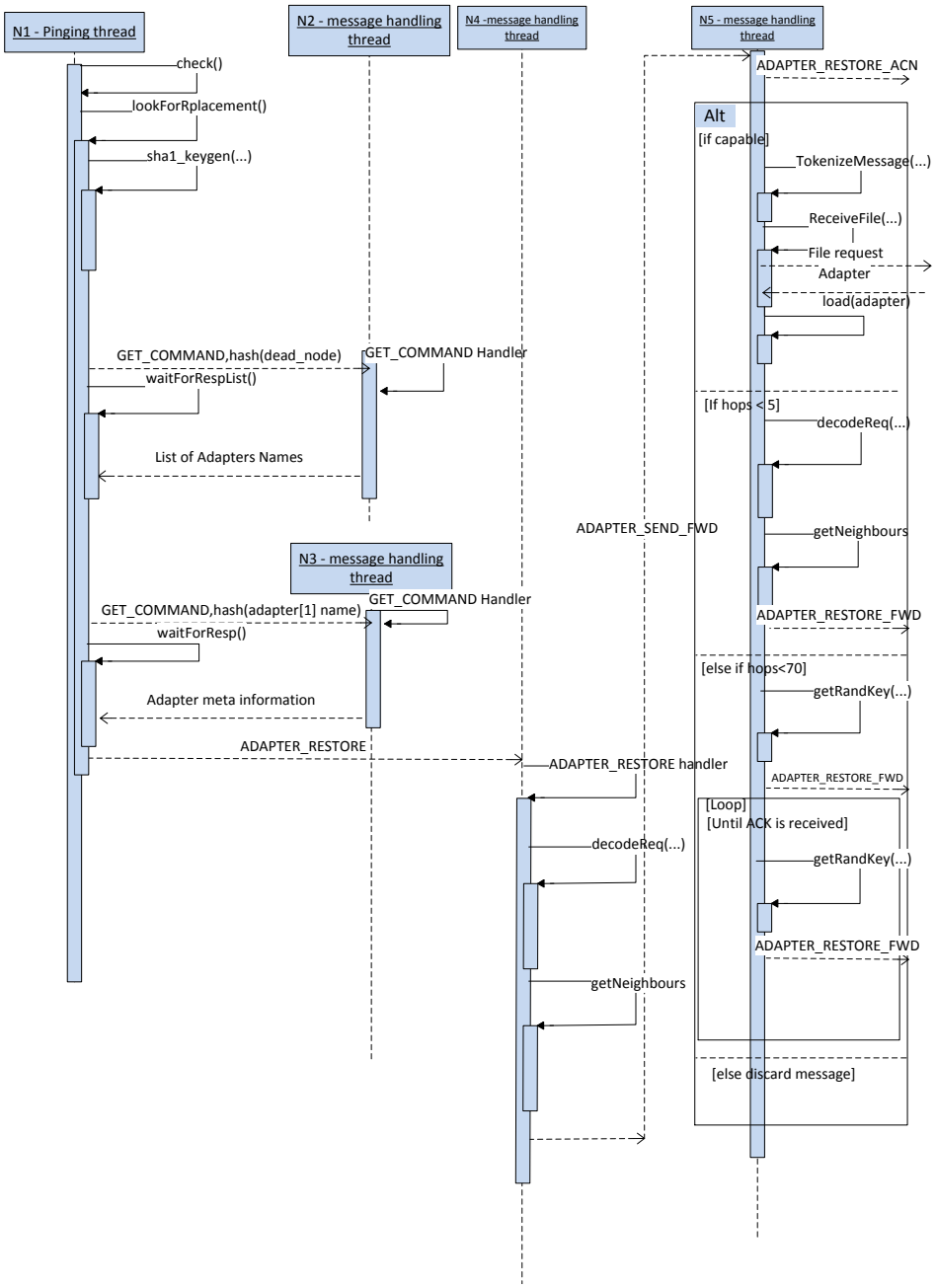

Figure 19: Adapter Restoration Message Sequence Diagram

executing the *accept()* function which makes the node to start listening for incoming messages from clients. If an incoming connection is initiated, it

receives the name of the file to upload, calculates its size and starts to send this file, cut in chunks, to the client.

The description of this component so far has dealt with procedure for opening a socket which will listen for an incoming adapter requests from clients. Now let see how a client initiates a sent adapter request.

Adapter Requester This is the component that requests a file and this is called when the node needs to get an adapter. Determining when an adapter is needed is discussed in 6.7. The function is called *ReceiveFile(char serverIP, int server_port, char filepath[], char localCopy[])*. This function uses the serverIP and server_port in order to connect to a server that listens for file requests. Indeed, this server is the one having an adapter that is needed by the process calling this function. Then an adapter that is described by the *filepath[]* parameter is transferred and copied into a *localCopy[]* file on chunk by chunk basis.

7 Experiments

In this chapter, we are going to describe the experiments that we have performed, in order to evaluate the implementation of DMLSAD. First of all, we will evaluate the service discovery overhead, caused by chimera DHT and messaging system. Secondly, we will evaluate the performance delay upon adapter restoration process.

7.1 Service Discovery Experiment

This experiment shows the average time taken for a node to retrieve information about a service metadata stored in the DHT with respect to the nodes participated in the network. Theoretically, the complexity of retrieving data stored in Chimera based on key is $\log_{2^b} N$ hops. **b** is a configuration parameter which determines the digit encoding and it usually is 4, which means that the digit encoding is $2^b = hexadecimal$. If there are 1,000,000 nodes in the network, the hops needed for data to be retrieved based on its key is $log_{16}(1,000,000)$, which equals around 5 hops. We are going to show the realistic delay caused by the routing overlay expressed in seconds. Figure 20 shows the delay caused by the network for 5, 10, 25, 50 nodes respectively.

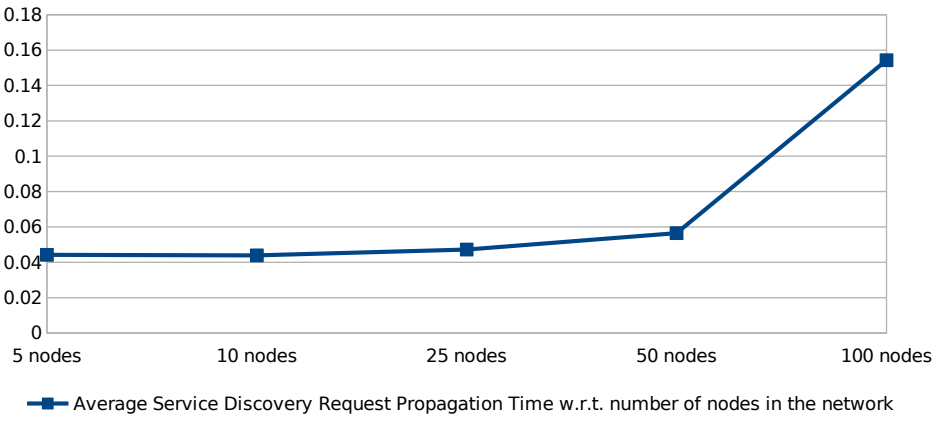

Figure 20

From this figure we infer that the delay caused by the network for 50 nodes is not significant. So a service can be discovered and executed in the same way as it was locally available with a relatively small overhead of about 0,15 seconds.

7.2 Restoration Experiment

In this experiment, we tried to measure the time between the different events happening after a node goes down. Also we checked the influence of the network

size to the process of finding a suitable replacement node. The network sizes for this experiment were again 5, 10, 25, 50 nodes. First of all, we will take the experiment with 25 nodes as an example in figure 21 and point out different events during this experiment. Here E_1 corresponds to the moment where the dead node

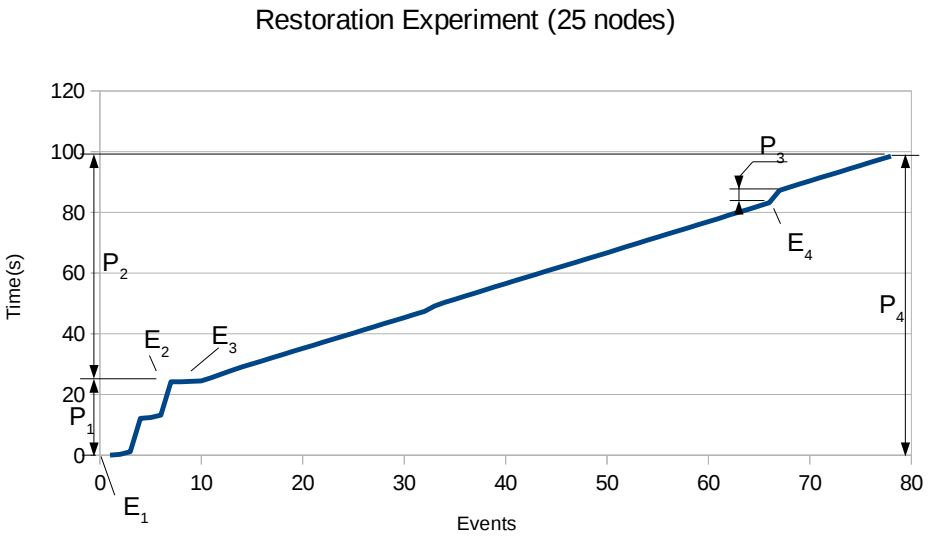

Figure 21: Restoration Experiment with 25 nodes

failed to respond to the ping for the first time. From here until the point E_2 that corresponds to the third time the node fails to respond, we have period P_1. This period is dedicated to find out if the node in question is actually dead. In our implementation, three pings which didn't receive any response to one node are the maximum allowed amount. And since we set the time between pings to 10 seconds, P_1 is a little bit over 20 seconds (time for two more pings). So from this point E_2 on, the node in question is considered as dead in the system and the system will start the restoration process at E_3. The configuration of the nodes participating in this experiment is that no node can actually restore the adapter so in theory restoration process would be infinite (but we have set a limit after which the process is terminated). The main reason for this is to see how long it takes to check each additional node for suitability and forward the message. Results show that this time linearly increases with the number of "visited" nodes. The period P_2 is the time spent for searching for a suitable node . To keep the restoration process more reliable, we introduced acknowledgements that are sent back to the origin of restoration request. E_4 marks a point where we failed to get such acknowledgement for some time P_3 (3 seconds), which implies restoration request was not received or processed correctly. In this case after P_3 the restoration request is resent to another node. Finally, P_4 is the total time of the restoration

process in this experiment.
Now we can take a look at the comparison between the same experiment with different network sizes to see if the number of nodes in the network is a relevant factor to the time taken within the restoration process. The result from this comparison can be seen in the following graph 22. We can see from the graph that

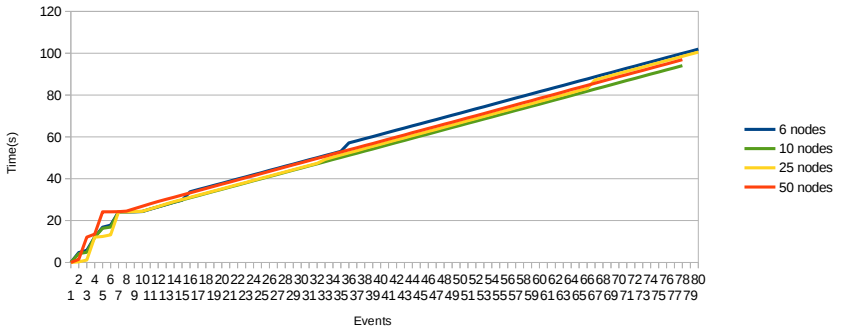

Figure 22: Restoration Experiment with different number of nodes

the number of nodes used in the network doesn't have a lot of influence on the time taken during the restoration process and should not be considered as a major factor. This is a good point because it proves that the system is scalable.

7.3 From Node Down to Adapter Restored

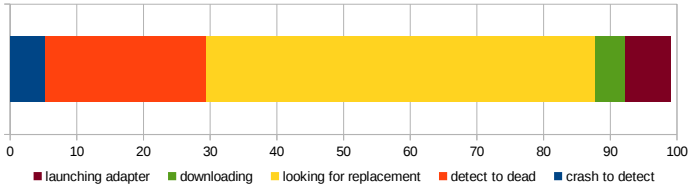

Figure 23: From Node Down to Adapter Restored 1

Figure 23 and figure 24 show the time taken for certain operations to be performed from a node crash to complete adapter restoration. For both of the experiments we use a network of 50 nodes and only one node was capable of hosting an adapter from crashed host. Furthermore, in both of the experiments, the node capable of hosting an adapter is not among the nodes in the neighbourhood set. Checking whether a node is capable of hosting an adapter takes around 1 second. If the neighbourhood set consists of about 30 nodes, then the procedure will take

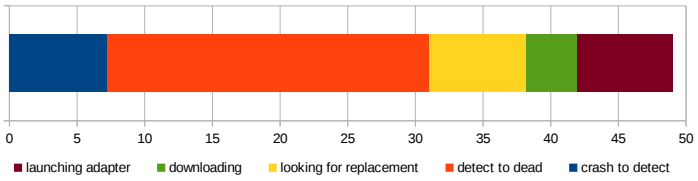

Figure 24: From Node Down to Adapter Restored 2

about 80 seconds in the worst case, with a not-so-big adapter binary (e.g. 1MB) and includes the time taken for announcing the death of a node, looking for a replacement node that can host the adapter, downloading the adapter itself, launching an adapter and waiting for it to register its services with HomePort.
In both cases we use a small neighbourhood set consisting of 5 nodes. This means that around 5 first seconds in the "looking for replacement" part of the diagram are spend to traverse the list of the nodes in the neighbourhood set. We are using a heuristics where after we do not find a suitable node among the nodes in the neighbourhood set of the dead node, we start to "shoot" at random and we choose the next node to be checked at a random basis. Indeed this is the reason why our results does not look as what they would look if a node capable of hosting an adapter will be within the neighbourhood set of a dead node. But as mentioned before we can consider only the first five seconds of "looking for replacement part, if we are sure that a suitable replacement node is in the neighbourhood set of a dead node.
Other time slices can also vary. "Crash to detect" should be no more than the pinging period. "Detect to dead" should be a little bit more than two pinging periods, because we have to count three unanswered pings in the row from the same node, to declare it dead. "Downloading" time strongly depends on adapter‘s binary size and should linearly increase with it. "Launching adapter" time actually depends on the implementation and it is hard to speculate. But in the adapter that is registering its services as soon as it is ready to do that this should not take more than 10 seconds.
Finally, we can see that the complete restoration time is very dependant on various factors, but we can state that the service outage period starts somewhere at 30 seconds and can be as high as 2 minutes or more depending on the number of suitable nodes in the network.

8 Conclusion and Further Work

8.1 Conclusion

In this project, we worked on providing fault tolerance feature and increasing the scalability of home automation systems. We first made a home automation system distributed in order to make it more scalable. The distribution of home automation nodes have become involved in rendering it fault-tolerant. We also consider some important aspects linked to security in order to provide a secure system which can be used in critical or business contexts.

Firstly, we decided to use a distributed hash table as the basis of our solution, in order to distribute the home automation services, remove single points of failure, add self-configurability and self-recoverability properties to the system, and most importantly providing a mechanism for modelling the entire set of nodes to work as a whole. Along with all of these benefits coming from the use of an overlay network, some shortcomings are evident as well. The biggest disadvantage of an overlay network is that it adds a delay to the time taken for an execution of a service request. This time delay stems from the service discovery procedure, evident in DMLSAD. Nevertheless, our service discovery experiment 7.1 shows that the delay caused by the service discovery is very small and a peer-to-peer solution seems to fit extremely well in our design.

Secondly, a distributed hash table is further used also to store the so called adapter, which aids the node fault-tolerance feature. The adapter restoration process used in DMLSAD adds some overhead 7.2 compared to a static restoration process, but indeed such an approach has its benefits such as self-configuration, scalability, low price and most importantly, easy to use.

Finally, DMLSAD considers using a fault tolerance composition logic component. Such an element can be applied in industrial or home automation systems only with lightweight set of relations between various services, since such a component having a lot of constraints could make DMLSAD not very scalable system. This is so, because, it needs to keep track of the states of all services. This is why this component is for optional use and can be even run outside DMLSAD as a client application. Even though not very scalable, composition logic component is sometimes mandatory in systems having dependencies between vendor specific subsystems. Adding a TMR 4.2 fault-tolerance over the execution of the composition logic is not very expensive in terms of computation complexity. No matter how modelled (with a timed automata or with set of constraints), solving such a task is a *NP* problem in its nature. In this sense, the TMR module adds

only linear overhead to the already existing computation complexity of solving set of rules. Therefore, a fault tolerance component would not reduce the scalability property of such a system. If a system, having a composition logic is scalable, it will remain scalable after adding a TMR component as well.

Our main requirements to build a working secure and fault-tolerant distributed home automation system, have been fulfilled in this project. Our prototype is proof that our solution can be applied not only from a theoretical point of view but also on a practical basis. Finally, we can state that all of the features available in DMLSAD makes the system, not just a solution that can be used in the home automation domain, but it is also a realistic alternative to the existing industrial SCADA systems, since it possess scalability, fault-tolerance, security and interoperability (essential for every home automation middleware system) between different vendor specific devices. Indeed, these features are essential for a system that is to be used in large buildings or real industrial environments.

8.2 Further Work

Our project is divided in a theoretical part of our solution and in its implementation. Both parts do not contain the same components, because part of the theoretical solutions has not been implemented yet. Such a project appears to be a massive one and if one is to make it to be used in reality, one needs to put a lot of effort on making it usable. Since the aim of the implementation was to prove the concept introduced in the design, we have implemented some of the concepts in a slightly different way than the one proposed in the design, which was forced by the deadlines that we had to deal with. That is why we divide the further work part in the theoretical and implementation work to be done.

Theoretical Work

Although our project implements some security elements, it does not possess any safety measure. This could be a good aspect to work on further works. While security is an important aspect in order to protect our system from external attacks, it would be good not to neglect its safety either. For example, our system could be responsible for fire monitoring. In this case, it would be interesting that if the node responsible for this were to fail, another node would automatically take the role of previous node, in order to keep fire monitoring up at any time. Indeed, DMLSAD provides such a functionality but as it can be inferred from our experiment 7, the delay caused by adapter discovery and transfer needs to be eliminated. This can be accomplished by, finding a back-up node that can host an adapter for each node existing in DMLSAD in advance, such that when a node crashes the back-up

nodes can attach the freed devices immediately. Thus the overall communication between a device and DMLSAD can work continuously.

A second consideration for future work could be development of an API, dealing with point to point connections as it was discussed in 4.6.

Implementation Work

While we have an authentication and an end-to-end security solution for securing our network, these solutions have not been implemented yet. Security is an important aspect of any system that is to be used in reality, especially in business or critical environments. Moreover, authentication and encryption are necessary steps to build a secure distributed home automation system that can be used in the industry as well. This is why this needs to be implemented before the system can be used in reality.
We currently have the service advertisement and service lookup implemented but we do not have any event subscriptions and notifications. Since our system is distributed, it should be possible for any node within the network to subscribe for any event and then receive notifications. Finally, one more theoretical component needs to be further developed and this is the adapter heterogeneity issue that we have theoretically addressed in 4.6.
We could conclude that the DMLSAD system could become widely adopted by households, offices and the industry, only after successful further development of the above mentioned features.

8.3 Final Words

This project was conducted at Aalborg University under the supervision of Associate Professor Dr. Arne Skou. We want to express our appreciation to him for his guidance. We are also grateful to Petur Olsen and Thibaut Le Guilly, PhD students at Computer Science Department, Aalborg University.

Bibliography

[1] T.R.Gopalakrishnan Nair A. Christy Persya. *Fault Tolerant Real Time Systems*. In *International Conference on Managing Next Generation Software Application*, 2008.

[2] P. Druschel A. Rowstron. *Tapestry: A Resilient Global-Scale Overlay for Service Deployment*. In *IEEE Journal on Selected Areas in Communications*, volume 22, pages 41–53.

[3] P. Druschel A. Rowstron. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems**. In *IFIP/ACM International Conference on Distributed Systems Platforms*, volume 18, November 2001.

[4] Ratul Mahajan-Sharad Agarwal Stefan Saroiu Colin Dixon A.J. Bernheim Brush, Bongshin Lee. *Home Automation in the Wild: Challenges and Opportunities*. In *ACM Conference on Computer-Human Interaction*, May 2011.

[5] Carl Landwehr Algirdas Avizienis, Brian Randell. *Basic Concepts and Taxonomy of Dependable and Secure Computing*. In *IEEE Transactions on Dependable and Secure Computing*, volume 1, pages 11–33, January-March 2004.

[6] OSGi Alliance. *Open Service Gateway initiative*. `http://www.osgi.org/Technology/WhatIsOSGi`.

[7] OSGi Alliance. *Open System Gateway Initiative*. In *OSGi Service Platform Core Specification, Release 4, Version 4.3*, pages 1 – 123, Bishop Ranch 6 2400 Camino Ramon, Suite 375, San Ramon, CA 94583 USA, April 2011.

[8] OSGi Alliance. *Open System Gateway Initiative Remote Services*. In *OSGi Service Platform Core Specification, Release 4, Version 4.3*, pages 125 – 135, Bishop Ranch 6 2400 Camino Ramon, Suite 375, San Ramon, CA 94583 USA, April 2011.

[9] Zigbee Alliance. *Zigbee wireless technology (ieee 802.15.4)*. `http://www.zigbee.org`.

[10] Zwave Alliance. *Z-wave protocol*. http://www.z-wave.com.

[11] Br. J. NELSON An. D. Birrell. *Implementing Remote Procedure Calls*. In *ACM Transactions on Computer Systems*, volume 2, pages 39–59, ACM New York, NY, USA, February 1984.

[12] Daniel D. Gajski Andreas Gerstlauer, Haobo Yu. *RTOS Modeling for System Level Design*. In *Design, Automation and Test in Europe Conference and Exhibition*, number 1530-1591, pages 130 – 135, 2003.

[13] U.C. Santa Barbara. *Chimera: Light-Weight and Efficient Implementation of a Structured Peer-to-Peer overlay Network*. http://current.cs.ucsb.edu/projects/chimera/.

[14] Global Caché. *GC-100 Network Adapter*. Number 032706-01, June 2008.

[15] Jiankun Hu Zahir Tari Xinghuo Yu Carlos Queiroz, Abdun Mahmood. *Building a SCADA Security Testbed*. In *Network and System Security*, number 978-0-7695-3838-9, pages 357 – 364, Oct 2009.

[16] Christian Rechberger Christophe De Cannière. *Finding SHA-1 Characteristics: General Results and Applications*. In *Lecture Notes in Computer Science, Advances in Cryptology – ASIACRYPT*, volume 4284, pages 1–20, 2006.

[17] Nikolaos Georgantas Valérie Issarny Jorge Parra Remco Poortinga Daniele Sacchetti, Yérom-David Bromberg. *The Amigo Interoperable Middleware for the Networked Home Environment*.

[18] Sami Zahran Donatas Poteliūnas, Tihomir Georgiev. *Seamless Integration of Devices into HomePort*.

[19] Kelvin Erickson. *Programmable logic controllers*. In *Potentials*, volume 15, pages 14 – 17, Feb 1996.

[20] Jacobson V. Liu C. McCanne S. Zhang L. Floyd, S. *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing*. In *IEEE/ACM Transactions on Networking*, volume 5, pages 784–803, December 1997.

[21] UPnP Forum. *UPnP Device Architecture 1.0*. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20080424.pdf.

[22] Kim G. Larsen Gerd Behrmann, Alexandre David. *A Tutorial on Uppaal 4.0*. http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

[23] IBM. *OSGi bundles*. http://publib.boulder.ibm.com/infocenter/radhelp/v8/index.jsp?topic=%2Fcom.ibm.osgi.common.doc%2Ftopics%2Fcbundles.html.

[24] A. Skou R. Torbesen J. Brønsted, P. Printz Madsen. *The HomePort System*. In *Consumer Communications and Networking Conference (CCNC)*, number 978-963-311-369-1, Las Vegas, NV, Jan 2010.

[25] E. F. Camacho J. M. Maestre. *Smart home interoperability: the DomoEsi project approach*. In *International Journal of Smart Home*, volume 3.

[26] Keith W. Ross Jian Liang, Rakesh Kumar. *Understanding KaZaA*.

[27] Dick Epema Henk Sips Johan Pouwelse, Paweł Garbacki. *The Bittorrent P2P File-Sharing System: Measurements and Analysis*. In *Lecture Notes in Computer Science, Peer-to-Peer Systems IV*, volume 3640, pages 205–216, 2005.

[28] Brian Jones Ed Price Elizabeth D. Mynatt Gregory D Julie A. Kientz, Shwetak N. Patel. *The Georgia Tech Aware Home*. In *Extended Abstracts on Human Factors in Computing Systems*, number 978-1-60558-012-8, New York, NY, USA, 2008.

[29] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. In *Computer Communications*, Amsterdam, 1992.

[30] Lehman E. Leighton T. Panigrahy R. Levine M. Lewin D. Karger, D. *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web*. In *STOC '97 Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, number 0-89791-888-6, pages 654–663, ACM New York, NY, USA, 1997.

[31] Anwitaman Datta Zoran Despotovic Manfred Hauswirth Magdalena Punceva Roman Schmidt Karl Aberer, Philippe Cudré-Mauroux. *P-Grid: a self-organizing structured P2P system*. In *ACM SIGMOD Record*, volume 32, pages 29 – 33, ACM New York, NY, USA, September 2003.

[32] Simon Rieche Klaus Wehrle, Stefan Götz. *Distributed Hash Tables*. In *Lecture Notes in Computer Science, Peer-to-Peer Systems and Applications*, volume 5, pages 79–93, 2005.

[33] A. Avizienis L. Chen. *N-Version Programming : a Fault-Tolerance Approach to Reliability of Software Operation*. In *Proceedings of FTCS-25*, volume 3, pages 3–9, 1996.

[34] W. Allen L. Nagy, R. Ford. *N-Version Programming for the Detection of Zero-day Exploits*. In *The 2006 IEEE Topical Conference on Cybersecurity*, April 2006.

[35] L. Mangeruca A. SangiovanniVincentelli M. Peri S. Pezzini M. Baleani, A. Ferrari. *FaultTolerant Platforms for Automotive SafetyCritical Applications*.

[36] David Parker Marta Kwiatkowska, Gethin Norman. *Probabilistic model checking in practice: Case studies with PRISM*. http://www.prismmodelchecker.org/papers/acmper_prism.pdf.

[37] Charles Morisset Anders P.Ravn Miaomiao Zhang, Zhirning Liu. *Design and Verification of Fault-Tolerant Components*.

[38] Victor S. Miller. *Use of Elliptic Curves in Cryptography*. http://link.springer.com/content/pdf/10.1007%2F3-540-39799-X_31.pdf, 1986.

[39] S. Misbahuddin. *Fault Tolerant Remote Terminal Units (RTUs) in SCADA Systems*. http://www.researchgate.net/publication/224143321_Fault_tolerant_remote_terminal_units_(RTUs)_in_SCADA_systems.

[40] R. Morrison. *Digital Certificate Vulnerabilities*. http://www.cs.ucsb.edu/~koc/ns/projects/02Reports/M.pdf.

[41] MSDN. *Windows Communication Foundation*. http://msdn.microsoft.com/en-us/library/dd456779.aspx.

[42] MSDN. *Windows Presentation Foundation*. http://msdn.microsoft.com/en-us/library/ms754130.aspx.

[43] Nintendo. *Wii Operations Manual*. 2009.

[44] University of California. *Fault-Tolerant Computing*. http://www.cs.ucla.edu/~rennels/article98.pdf.

[45] Oracle. *RMI*. http://docs.oracle.com/javase/tutorial/rmi/.

[46] M. Schroeder R. Needham. *Using encryption for authentication in large networks of computers*.

[47] Timothy M. Pinkston Rajeev Balasubramonian. *Buses and Crossbars*. 2011.

[48] L. Adleman R.L.Rivest, A. Shamir. *A Method for Obtainig Digital Signatures and Public-Key Cryptosystems*. http://people.csail.mit.edu/rivest/Rsapaper.pdf, 1978.

[49] H. Schulzrinne S. Baset. *An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol.*

[50] John C.-H. Lin Supratik Bhattacharyya Sanjoy Paul, Krishan K. Sabnani. *Reliable Multicast Transport Protocol (RMTP).* In *Selected Areas in Communications, IEEE Journal on*, volume 15, pages 407 – 421, April 1997.

[51] Charles Yiu-John Zimmerman Scott Davidoff, Min Kyung Lee and Anind K. Dey. *Principles of Smart Home Control.* In *Proceedings of the 8th international conference on Ubiquitous Computing*, number 3-540-39634-9 978-3-540-39634-5, Springer-Verlag Berlin, Heidelberg, 2006.

[52] Bruno Souza. *The Jini™ Architecture.* In *JavaOne, Sun's 1999 Worldwide Java Developer Conference*, San Francisco, California.

[53] St. D. Gribble St. Saroiu, Kr. P. Gummadi. *Measuring and analyzing the characteristics of Napster and Gnutella hosts.* In *Multimedia Systems*, volume 9, pages 170–184–59, Springler, 2003.

[54] Kenneth Lausdahl Augusto Ribeiro Thomas Skjødeberg Toftegaard Sune Wolff, Peter Gorm Larsen. *Facilitating Home Automation Through Wireless Protocol Interoperability.* In *Wireless Personal Communications: An International Journal*, volume 53, pages 465–479, May 2010.

[55] National Communications System. *Supervisory Control and Data Acquisition (SCADA) Systems.* `http://www.ncs.gov/library/tech_bulletins/2004/tib_04-1.pdf`.

[56] E. Rescorla T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2.* `http://tools.ietf.org/html/rfc5246`, 2008.

[57] A. P. Ravn J. Brix Rosenkilde A. Skou Th. Le Guilly, P. Olsen. *HomePort: Middleware for Heterogeneous Home Automation Networks.* In *Smart Environments and Ambient Intelligence*, number 0-89791-888-6, pages 654–663, "San Diego California, EEUU", 2013 1997.

[58] Aalborg Univeristy. *HomePort's Source Code Repository.* `github.com/home-port/HomePort`.

[59] Pulseworx UPB. . `http://pulseworx.com/UPB_.htm`.

[60] Maurizio Manca Gabriele Tolomei Vittorio Miori, Luca Tarrini. *An Open Standard Solution for Domotic Interoperability.* In *IEEE Transactions on Consumer Electronics*, volume 52, pages 97 – 103, 2006.

[61] W3C. *Server-Sent Events*. http://www.w3.org/TR/2009/WD-eventsource-20090423/.

[62] w3schools.com. *SOAP Tutorial*. http://www.w3schools.com/soap/default.asp.

[63] w3schools.com. *WSDL and UDDI*. http://www.w3schools.com/wsdl/wsdl_uddi.asp.

[64] D. Wagner. *Resilient aggregation in sensor networks*. In *Proceedings of the 2nd ACM workshop on Security of Ad Hoc Sensor Networks*, page 78–87, San Francisco, California, 2004.

[65] John Heidemann Wei Ye. *Enabling Interoperability and Extensibility of Future SCADA Systems*. Number SI-TR-625„ Oct 2006.

[66] X-10. . http://www.x10.com.

[67] Lee Breslau Nick Lanham Scott Shenker Yatin Chawathe, Sylvia Ratnasamy. *Making Gnutella-like P2P Systems Scalable*. In *SIGCOMM'03, Karlsruhe, Germany*, number 1-58113-735-4/03/0008, August 2003.

Appendix A

Taxonomy and Terminology

Note: the listed taxonomy uses source [5] :

System - Hardware, software, humans, etc. A system interacts with other systems.

Environment - A system and all systems that it interacts with.

System Boundary - The border between a system and its Environment.

Function - A system's intentional purpose.

Behaviour - The way that a system implements its Function. A behaviour characterizes itself with states.

Total State - A set of the computation, interconnection, networking ,etc states.

Structure - Makes it possible for a system to compose its behaviour (system of systems).

Service - System behaviour from service consumer's perspective.

User - Service's consumer.

Failure - Incorrect service.

Service Outage - Time interval of incorrect service delivery.

Service Restoration - The transition from incorrect to correct service.

Failure modes - Ranked failure severities.

Error - Incorrect total state of a system that may lead to a service failure.

Fault - Reason for a system error.

System's Degraded Mode - When a system offers reduced set of services to the service consumers. And the degraded mode can be further sub-classified into

- Slow service - When a service does not provide a result in a timely manner.
- Limited service - A service does not offer its full capabilities to the consumer.
- Emergency service - An alternative to the original version of a service. It is used only in emergency situations.

Dependability - Not allowing service failures more than an acceptable limit. A service is dependable if satisfies the following properties:

- Availability.
- Reliability.
- Safety.
- Integrity.
- Maintainability.

According to potential types of faults that can occur in a system, protective measures ensuring dependability subdivide into the following categories:

- *Fault Prevention* - General engineering activities take care of fault prevention.
- *Fault Tolerance* - Avoiding service failures when a fault occurs.
- *Fault Removal* - Decreasing severity of faults, when present.
- *Fault Forecasting* - Estimating the likelihood of fault occurrence and the probable fault consequences.

According to [5] and [1], the types of faults could be:

Development faults - Made during development phase and occur during use phase.

Operational Faults - Occur during the use phase.

Internal Faults - Caused by an internal error.

External Faults - Caused by external entities.

Natural Faults - Caused by a phenomenon. These faults can be eliminated if a system is distributed over a large geographic area.

Human-Made faults - Faults caused by human actions.

Hardware faults - Service is not accessible due to a hardware problem.

Malicious faults - Made during system development or during system use and occur during system use. These are intentional faults made to crash or alter the system's functionality at run-time.

Deliberate faults - Caused because of misplaced decisions.

Non-Deliberate faults - Developer's unintentional faults, mistakes.

Permanent faults - Once occurred, they persist until human intervention.

Transient faults - A fault that occur for an interval of time.

Intermittent - It is a type of fault specific to periodic tasks, where the error persists throughout the cycle of a periodic task.