MASTER’S THESIS

Light Weight Collaborating Smarties

M.A. Nazir

Supervisors

Prof. Dr. Johan Lukkien, TU/e
Dr. Aly A. Syed, NXP Semiconductors

Eindhoven, December 2008
Abstract

Ambient Intelligence is a well known concept these days. In Ambient Intelligence environment many devices collaborate to perform some functions for a user, taking into account the user’s context. This can be achieved by ad hoc networking between such devices.

At NXP Semiconductors a concept of ad hoc network has been developed, called the Smarty Architecture, where knowledgeable devices, named Smarties, are capable of forming an ad hoc network and collaborating with each other to satisfy a user’s request depending on the capabilities and resources available in the environment.

The initial development goal of Smarties was to equip backward compatible UPnP devices with extra functionality defined by the Smarty Architecture. UPnP is a prevalent service discovery and usage framework and it was being used at NXP Semiconductors for a number of applications, therefore a decision was made to build Smarties using UPnP. However, for many systems (e.g. Lighting Systems), UPnP is seen as a functionality overkill. The goal of this project is to explore the possibilities of replacing the UPnP with a leaner framework for the Smarty Architecture.

In this project we have determined the requirements that the Smarty Architecture has for the frameworks like UPnP. We have found that the Smarty Architecture requires a mechanism to discover and access functionality distributed in the environment, in addition to this it also requires communication support for Smarty Architecture dependent messages.

In order to find a UPnP-like framework in the domain of resource constrained embedded devices (e.g. a Light Switch), a number of frameworks were examined. Based on the investigations, we have made a decision to extend the Smarty Architecture itself to provide the required support of service discovery and usage. This enables a Smarty to expose its functionality to other Smarties and exchange messages between them. A Smarty Message format has been defined in order to support interactions between Smarties. These messages are independent of the underlying Communication carrier; it can be any network capable of broadcasting messages such as UDP/IP, Zigbee etc.

In order to prove the extended Smarty Architecture concept, a prototype has been developed using development boards based on NXP’s ARM7TDMI LPC2468 microcontroller family. A use case demonstrates a number of Smarties collaborating and performing the required functionality. Furthermore, the developed prototype has a code size of 60 KB and it is within the defined goal of 100KB, as compared to the previously built prototype based on UPnP, which had a code size of 1.6 MB.
Table of Contents

1 Introduction ................................................................................................................. 9
  1.1 Context................................................................................................................ 9
  1.2 Problem Statement: Light Weight Collaborating Smarties .............................. 13
  1.3 Outline............................................................................................................... 13
2 Problem Analysis ...................................................................................................... 15
  2.1 Research Questions ........................................................................................... 15
  2.2 Initial Investigations.......................................................................................... 15
      2.2.1 Smarty Architecture and UPnP ................................................................. 15
      2.2.2 UPnP, current and target Application Domains........................................ 18
  2.3 Conclusion ........................................................................................................ 18
3 Search for a Service Discovery and Usage Framework for Smarties ....................... 19
  3.1 Functional Requirements .................................................................................. 19
  3.2 Non Functional Requirements .......................................................................... 20
  3.3 Findings and Decision ....................................................................................... 21
4 System Architecture & Design ................................................................................. 23
  4.1 General Smarty Architecture Overview............................................................ 23
      4.1.1 General Smarty Architecture .................................................................... 23
      4.1.2 General Smarty Architecture Extensions .................................................. 24
  4.2 Smarty Device Architecture .............................................................................. 26
  4.3 Smarty Interface Components and Interactions ................................................ 27
  4.4 Smarties’ Interactions and Configuration Management ................................... 30
      4.4.1 Configuration Setup State ......................................................................... 31
      4.4.2 Configuration Execution State ................................................................. 32
      4.4.3 Configuration Control State ...................................................................... 32
      4.4.4 Configuration Inactive State ..................................................................... 32
      4.4.5 Destroy Configuration State ..................................................................... 32
  4.5 Smarty Message Design .................................................................................... 33
      4.5.1 Smarty Message Headers .......................................................................... 33
      4.5.2 Smarty Message Payload .......................................................................... 34
      4.5.2.1 Configuration Setup State Payloads...................................................... 34
      4.5.2.2 Execution, Inactive and Control States Payloads ................................. 40
      4.5.2.3 Destroy Configuration State Message .................................................. 41
  4.6 Knowledge Base (KB) Design .......................................................................... 42
  4.7 Software Design ................................................................................................ 42
5 Implementation ......................................................................................................... 44
  5.1 Use case Scenario and Analysis........................................................................ 44
      5.1.1 Use Case1: Switch on/off Lights .............................................................. 44
      5.1.2 Use Case2: Light Dimming ...................................................................... 44
  5.2 Demonstrator Scenarios .................................................................................... 44
  5.3 Smarty Devices’ Hardware Implementation ..................................................... 47
      5.3.1 Light Controller Smarty ............................................................................ 48
      5.3.2 Light Sensor Smarty ................................................................................. 48
      5.3.3 Light Switch Smarty .................................................................................. 49
5.4 Software Implementation of Smarties .............................................................. 50
6 Conclusion ................................................................................................................ 52
  6.1 Results ............................................................................................................... 52
  6.2 Future Work ...................................................................................................... 53
Appendix B: Knowledge Base Design [7] ........................................................................ 58
Appendix C: Sequence and Class Diagrams .................................................................. 59
Appendix D: Knowledge base of a Lighting Switch Smarty ............................................ 63
Bibliography ..................................................................................................................... 64
### List of Figures

| Figure 1 – Smarty Architecture Example I [6]  | 10 |
| Figure 2 – Virtual Device [6]  | 10 |
| Figure 3 – Smarty Architecture Example II  | 11 |
| Figure 4 – Smarty  | 12 |
| Figure 6 – Smarty-based ad-hoc network [6]  | 16 |
| Figure 7 – Knowledge base of Smarty S0 [6]  | 16 |
| Figure 5 – UPnP Implementation of Smarties [6]  | 17 |
| Figure 8 – Smarty (revised)  | 27 |
| Figure 9 – Smarty Interface Components  | 28 |
| Figure 10 – State diagram of a Configuration  | 31 |
| Figure 11 – Format of Smarty Message Headers  | 33 |
| Figure 12 – Search request Payload  | 35 |
| Figure 13 – Search response Payload  | 35 |
| Figure 14 – Join Configuration request Payload  | 36 |
| Figure 15 – Join response Payload  | 37 |
| Figure 16 – Configuration Complete notification Payload  | 37 |
| Figure 17 – Transition notification Payload  | 38 |
| Figure 18 – Search Configuration request Payload  | 39 |
| Figure 19 – Search Configuration response Payload  | 39 |
| Figure 20 – Action Call Payload  | 40 |
| Figure 21 – Destroy Configuration State Message  | 41 |
| Figure 22 – Scenario 1  | 46 |
| Figure 23 – Scenario 2  | 47 |
| Figure 24 – Light Controller Smarty  | 49 |
| Figure 25 – Light Sensor Smarty  | 50 |
| Figure 26 – UPnP Stack [6]  | 56 |
| Figure 27 – Knowledge Base Design  | 58 |
| Figure 28 – Sequence Diagram (Configuration Setup 1)  | 59 |
| Figure 29 – Sequence Diagram (Configuration Setup 2)  | 60 |
| Figure 30 – Sequence Diagram (Configuration Execution, Inactive, Control, Destroy)  | 61 |
| Figure 31 – Class Diagram  | 62 |
List of Tables

Table 1 – Comparative Table on Current and Target Application Domains .......... 18
Table 2 – Functional Requirements of a Service Discovery & Usage Framework for the Smarty Architecture ................................................................. 19
Table 3 – Non-Functional Requirements of a Service Discovery & Usage Framework for the Smarty Architecture .......................................................... 20
Table 4 – Smarty Message Header for a Configuration ...................................... 34
Table 5 – Configuration Setup State: Search request Payload Fields ................ 35
Table 6 – Configuration Setup State: Search response Payload Fields .............. 35
Table 7 – Configuration Setup State: Join Configuration request Payload Fields .... 36
Table 8 – Configuration Setup State: Join response Payload Fields ............... 37
Table 9 – Configuration Setup State: Configuration Complete notification Payload Fields ................................................................. 38
Table 10 – Configuration Setup State: Transition notification Payload Fields .... 38
Table 11 – Configuration Setup State: Search Configuration request Payload Fields .... 39
Table 12 – Configuration Setup, Inactive, Execution and Control State: Search Configuration response Payload ......................................................... 40
Table 13 – Execution, Inactive and Control State Payload Format ......................... 41
Table 14 – Destroy Configuration Smarty Message ........................................... 41
Table 15 – Prototype Functional Requirements ............................................... 45
1 Introduction

This chapter briefly introduces the context in which the project has taken place and the problem statement for this project. Furthermore, an overview of this document’s structure is also provided.

1.1 Context

Digital technology has enabled exchange of data between different devices. We see people connecting digital cameras to watch photos on TV screen. This inter-operability of devices is also seen as a first step towards Ambient Intelligence. [1] A next step in the march towards Ambient Intelligence would be to make devices collaborate with each other to provide services to the user by forming an ad hoc network e.g. playing music stored on your portable media device using your mobile phone to decode it, your TV’s audio system for post processing it and your home audio system to render it on the big speakers.

At NXP Semiconductors [5] a concept of ad hoc network has been developed, called the Smarty Architecture, where knowledgeable devices named Smarties are capable of forming an ad hoc network and collaborating with each other to satisfy a user’s request, depending on the capabilities and resources available in the environment. In order to realize this concept a project “Collaborative Smarties” was initiated at NXP Semiconductor.

Collaborative Smarties

The basic idea of the Collaborative Smarties project was to define an architecture for building an ad-hoc network of functionalities that are distributed over more physical devices. A device is called a Smarty, if it can perform a set of functionalities and participate in forming an ad hoc network through discovering and interacting with more physical devices in an ad-hoc way. The architecture of building such collaborations has been given a name the Smarty Architecture and it is illustrated in figure 1.

An example from the multimedia application domain is shown. A User request is received at the Mobile Phone device through the Request Center device. This Request Center device is a user interaction point. The user request is to listen to a song on the Audio System, whereas this song is actually stored on the Mobile Phone device. Furthermore, this song can not be decoded by Mobile phone device; therefore it needs to discover a device which can decode such formats. In this case a Laptop is discovered, which decodes the song and sends the decoded data to the Audio System in order to render it. In this way a virtual device is formed by the collaboration of these devices, where each device is a Smarty device. The formation of virtual device is illustrated in figure 2.
In figure 2, there are two virtual devices; each virtual device is formed by a number of physical devices, where each physical device provides one or more functionalities. The initiation of this process is based on the request generated by the users using a Request Center or triggered automatically in the environment. This results in forming a virtual device which is transparent to the user and from a user’s point of view; he interacts with a device which is actually a smarty-based ad hoc network of devices. According to the Smarty Architecture this process of building a virtual device has been given a name Configuration.
Similarly, the concept of creating virtual devices can be used in the domain of Lighting Systems. In figure 3, it is illustrated that a number of Smarties of type Light Bulb, Light Sensor and Light Dimmer collaboratively provide user required functionality of maintaining light level.

![Figure 3 – Smarty Architecture Example II](image)

**Smarty Device**

A Smarty is a physical device and a basic unit in a Smarty based ad hoc network. Figure 4 shows a logical view of a Smarty device. Each Smarty has hardware resources (processing unit, memory, etc.) and a network interface in order to communicate with other devices. In addition to its basic functionality a Smarty device has some Smarty Architecture defined Smarty Functions comprising of Knowledge Base and Smarty Control.

- In the Knowledge Base information is stored in the form of “Recipes”, “Rules”, and “Templates” etc. This information is necessary to participate and play a role in the smarty-based ad hoc network. The information in the Knowledge Base is organized in a way that facilitates its interpretation as well as adaptation to the increasing needs.

Knowledge Base is based on Ontology [4], which defines explicit specifications of the terms in a domain and relations among them. Ontology makes the Knowledge Base well structured, understandable and readable.
Recipes prescribe what a Smarty device should do to perform a role in a network of Smarty devices. Rules allow the Smarty device to make decisions and Templates let it adapt to the introduction of new Smarties.

- A Smarty has a Smarty Control Component, which interprets the Knowledge Base and provides an interface to the device’s internal functionality and to other Smarties. In the example shown in figure 1, it is the Smarty Control component of Mobile phone Smarty which interprets the user request received by Request Center. It further interprets the Knowledge Base and finds if it can fulfill this request and in case of success, it initiates discovery of a Smarty with decoding capability. Similarly Smarty Control present at each Smarty interprets its Knowledge Base and takes required actions.

- A Smarty can expose functionality in the form of services using a framework, such that this framework enables discovery and usage of these services by other Smarties. For example, in figure 1, a mobile phone device needs to discover and use functionality of media decoding. In order to achieve this each Smarty needs to have a mechanism to send and receive messages for discovering the required functionality and later on accessing it.

![Smarty Diagram](image)

**Figure 4 – Smarty**

### Previous Development of Smarties

Initially Smarties were developed for multimedia application domain for the example application shown in figure 1. This development was based on the Universal Plug and Play (UPnP) [3] framework, which provided support to expose Smarty device’s functionality in the form of services and enabled discovery and usage of these services. In
addition to this, Smarty functions were also built using UPnP. (See UPnP introduction in Appendix A).

A prototype was implemented using UPnP SDK and NXP Semiconductor’s SoC PNX0106. [6]

1.2 Problem Statement: Light Weight Collaborating Smarties

NXP Semiconductors is interested to use concept of the Smarty Architecture for resource constrained embedded devices, we find in Lighting Systems for example. Figure 3 illustrates that a number of Smarties of type Light Dimmer, Light Sensor and Light Bulb collaboratively provide user required functionality.

The initial development goal of Smarties was to equip backward compatible UPnP devices with extra functionality defined by the Smarty Architecture. UPnP is a prevalent service discovery and usage framework and it was being used at NXP Semiconductors for a number of applications, therefore a decision was made to build Smarties using UPnP. However, for many systems (e.g. Lighting Systems), UPnP is seen as a functionality overkill.

The goal of this project is to study the UPnP based implementation of the Smarty Architecture in [6], [7] and identify the needs of the Smarty Architecture for such a framework, and then find a framework for the target application domain that fulfills these needs. If such a framework is not found then a customized framework should be built to fulfill these requirements. In order to prove the purposed solution, a prototype should be developed by building a demonstrator for the device types shown in figure 3, for example.

1.3 Outline

Chapter Two In this chapter, we discuss the initial investigations in order to understand the Smarty Architecture’ requirements for UPnP and prospects of using UPnP for target application domain.

Chapter Three In this chapter, we formulate the functional and non-functional requirements of the Smarty Architecture for a Services Discovery and Usage Framework. We perform a survey to find such a framework and based on our findings we make decisions regarding the solution.

Chapter Four In this chapter, a solution is provided such that the requirements identified in chapter 3 can be accomplished. Important concepts and definitions of the Smarty Architecture are briefly described; a number of modified and new concepts are described. The software design is also included in this chapter.

Chapter Five In this chapter, use case scenarios of the prototype are defined, furthermore hardware and software components of the demonstrator are described.
Chapter Six In this chapter, the results of the project and the contributions to the Smarty Architecture are described. Suggestions for future work are also provided.
2 Problem Analysis

In this chapter, we discuss the investigations performed in order to understand dependencies between the Smarty Architecture and UPnP. Furthermore, we analyzed the use of UPnP for target application domain.

2.1 Research Questions

Before investigating the use of the Smarty Architecture in a different application domain and its dependencies on certain technologies, we encountered a number of questions:

- What are the Smarty Architecture’s dependencies on the technologies like UPnP?
- How does the change of application domain influence the Smarty Architecture?
- How different is the target application domain from the domain for which Smarties were initially developed (current application domain) and in what respect?
- Is UPnP suitable for the target application domain?

2.2 Initial Investigations

We did an investigation in order to find answers to the questions raised above; the details of the investigation are given in following sections.

2.2.1 Smarty Architecture and UPnP

In order to understand, the concept of the Smarty Architecture and the ad hoc network of functionalities between Smarties, we use a generic example given in [6].

Consider four devices I, II, III, and IV shown in figure 6. These devices offer different services from the group of S1-S9. Let us consider that a user request is received at the device IV and another user request at the device I. In order to respond to these requests a network is configured, where services are linked in a logical order as follows:

- Request1 = {S5<>S3, S3<>S6, S3<>S0, S6<>S9, S0<>S2}
- Request2 = {S0<>S2, S2<>S7, S7<>S3, S3<>S1}

According to the Smarty Architecture, it is possible to build an ad hoc network of functionalities between these devices, for example the following steps are performed for Request 2:

- User Request2 is received by the smarty S0. This Smarty is part of the device IV (smarty-in-the-box, see 4.1.1). S0 interprets its Knowledge Base and analyzes the request.
- S0 finds that it has to perform a local scenario B, which is defined in the Knowledge Base as shown in figure 7.
to S3, which further forwards it to S7. This message is finally received at S0, which starts propagating data through the created network.

We observe that no matter where the Smarties are, in the same device or in different appliances, the way the ad-hoc network is forming follows the same rules.

**UPnP Based Development of Smarties**

In [6], a decision was made to build Smarties using the UPnP, as the UPnP is a prevalent service discovery and usage framework and it was being used at NXP Semiconductors for a number of applications. The purposed demonstrator in [6] and Smarty Architecture components were developed using UPnP primitives and device types. A detailed discussion on this choice can be found in [6].

We have studied the Smarty Architecture’s implementation and identified the functionalities of UPnP that are used by the Smarty Architecture. In figure 5, the UPnP based implementation of Smarties is shown. According to the UPnP architecture two devices work together once a Control Point (See Appendix A) establishes a connection between them and it is the Control Point, which invokes actions on the exposed services of a device. In order to build UPnP compatible Smarties, it was required to introduce a new service called the Smarty Service. This service could provide a set of actions, which could allow an external Control Point to invoke smarty-like behavior of the device. Internally a Smarty Service interacts with the Knowledge base and in case it needs to trigger an action on other Smarty device, it passes this responsibility to the control point. It is further illustrated in figure 5 that the Smarty Control component previously shown in figure 4 is actually implemented as UPnP Control Point (Smarty Control) and UPnP Service (Smarty Service).
that the Smarty Architecture can be developed independent of UPnP if the
requirements it has for UPnP can be fulfilled by some other framework.

- The three basic interactions between the Smarty Architecture and UPnP have been identified:
  - UPnP handled communication needs of the Smarty Architecture.
  - Smarty Architecture used the Service Discovery Protocol provided by UPnP to find other Smarties in the network.
  - In order to interact with other Smarties to use their functionality a mechanism provided by UPnP was used.

### 2.2.2 UPnP, current and target Application Domains

While investigating the relationship of UPnP with the Smarty Architecture, we predicted some potential issues about the use of UPnP in the target application domain. It is important to remember that the current application domain refers to the domain for which Smarties were initially developed, where resource rich devices for multimedia applications were used, whereas the target application domain refers to the domain of very resource constrained devices, which we find in lighting systems for example.

In order to investigate the feasibility of using UPnP in the target application domain, we tried to understand the differences between current and target application domains. It was found that the UPnP and the current application domain differed substantially from the target application domain on number of issues. These differences have been formulated by consulting the literature [2] and domain experts, as well as the examples from industry [8]. This is illustrated in the following table.

<table>
<thead>
<tr>
<th>Type of Devices</th>
<th>UPnP &amp; Current Application Domain</th>
<th>Target Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Resource rich devices to home appliances (e.g. TV)</td>
<td>Resource constrained devices (e.g. Light Sensor)</td>
</tr>
<tr>
<td>Network type</td>
<td>Enterprise (LAN)</td>
<td>Ad hoc</td>
</tr>
<tr>
<td>Network Speed</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Size of the Network</td>
<td>Medium (hundreds to thousands of devices)</td>
<td>Small to Medium</td>
</tr>
</tbody>
</table>

### 2.3 Conclusion

Based on the results of the investigations in above section, we decided to find a service discovery and usage framework for the Smarty Architecture, which could be used in the target application domain. We formulated the requirements in chapter 3 and performed a survey in order to find such a framework.
3 Search for a Service Discovery and Usage Framework for Smarties

In this chapter we specify some basic functional and non-functional requirements of the Framework for the Smarty Architecture and target application domain, as identified in the Initial Investigations in section 2.2. Furthermore, these requirements have been formulated keeping in mind the time and effort that can be spent reasonably on such a framework as a proof of concept. A number of frameworks are identified and a decision is made based on the findings and suitability of these frameworks as supporting infrastructure for the Smarty Architecture.

3.1 Functional Requirements

In the following table the functional requirements of a Service Discovery and Usage Framework for the Smarty Architecture are presented, these requirements are based on the findings of the investigations performed in section 2.2.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM-F-01</td>
<td>A Framework should have a Service Discovery Protocol.</td>
<td>Service Discovery is required by a Smarty to find other Smarties in the environment, so that they can collaborate to fulfill user request.</td>
</tr>
<tr>
<td>SM-F-02</td>
<td>A Framework should provide a mechanism to access functionality on other devices.</td>
<td>This is required so that a Smarty can access functionality provided by other Smarties to fulfill required tasks.</td>
</tr>
<tr>
<td>SM-F-03</td>
<td>A Framework should have interfaces to fulfill Communication needs of the Smarty Architecture</td>
<td>Smarties need to send Smarty Architecture dependent messages in the network, the capability to do so must be provided by the selected framework.</td>
</tr>
</tbody>
</table>
3.2 Non Functional Requirements

In table 3, non-functional requirements of a Resource Discovery and Usage Framework for the Smarty Architecture are presented; these requirements are based on the findings of the investigations performed in section 2.2.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM-NF-01</td>
<td>The Framework should have an open source SDK available.</td>
<td>It is required so that software applications can be developed for Smarties to fit our research needs.</td>
</tr>
<tr>
<td>SM-NF-02</td>
<td>Software executable size should be less than 100 KB</td>
<td>Software size in this range can meet the available resources in the Smarties ranging from resource poor to rich devices. For resource constrained devices, we have put an arbitrary limit of 100 KB.</td>
</tr>
<tr>
<td>SM-NF-03</td>
<td>Framework’s dependencies on Network and Physical Layers.</td>
<td>The Framework shouldn’t be bound to a particular Communication standard.</td>
</tr>
<tr>
<td>SM-NF-04</td>
<td>Network Scalability is important and should be scalable to thousands of devices</td>
<td>Network should be scalable to adjust with a variable number of devices. For home environment this could be up to 100 devices and in public places could reach 1000s</td>
</tr>
<tr>
<td>SM-NF-05</td>
<td>A Framework should be able to operate in ad-hoc network of devices</td>
<td>The devices in the target domain need to build ad hoc network in order to collaborate.</td>
</tr>
<tr>
<td>SM-NF-06</td>
<td>Type of Devices</td>
<td>Very resource-poor devices</td>
</tr>
</tbody>
</table>
3.3 Findings and Decision

A survey was performed in order to find a matching framework based on the requirements in section 3.1 and 3.2; our goal was to find and not to compare or identify the possible improvements in the framework.

We explored a number of available frameworks while having above mentioned requirements as guide line. The paper on “Classification of the Service Discovery Protocols” [2] gave an insight into the existing frameworks in the field of Service Discovery and Usage. We narrowed down our investigations by categorizing the frameworks on the basis of SM-F-01, SM-F-02, SM-NF-05 and SM-NF-06 defined in section 3.1 and 3.2. We further investigated based on other non functional requirements mentioned in section 3.2. The frameworks we found and the results of our explorations are shown in table 4.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>BlueTooth Service Discovery &amp; Usage [9]</td>
<td>Yes</td>
<td>No (Bluetooth)</td>
</tr>
<tr>
<td>2.</td>
<td>Salutation/Salutation Lite† [2, 15]</td>
<td>No</td>
<td>Yes (No preconditions)</td>
</tr>
<tr>
<td>3.</td>
<td>CDS [10]</td>
<td>No</td>
<td>No (ODRMP)</td>
</tr>
<tr>
<td>5.</td>
<td>Cheng and Marsic[12]</td>
<td>No</td>
<td>Not Available</td>
</tr>
</tbody>
</table>

† Salutation consortium was disbanded on June 30, 2005 [20]
‡ Software doesn’t implement complete Zigbee Application and Network stack.[16] Furthermore it is built for TinyOS Operating System only.
**Decision**

The findings in above table show that none of the frameworks fulfill the requirements mentioned in 3.1 and 3.2. It was therefore decided to design a customized framework by extending the Smarty Architecture in order to fulfill these requirements.
4 System Architecture & Design

In this chapter we describe the important concepts of the Smarty Architecture and extensions made in the Smarty Architecture to fulfill the requirements mentioned in chapter 3.

4.1 General Smarty Architecture Overview

The previous projects in the series of “Collaborative Smarties” [6], [7] have elaborated the architectural vision and basic concepts of the Smarty Architecture; in this section we will summarize some of the important concepts as well as the new ones.

4.1.1 General Smarty Architecture

The basic concepts of the Smarty Architecture are briefly described here, whereas some of these concepts have already been introduced in section 1.1.

1. Smarties in a Box

This concept refers to a Physical device, where two or more Smarties are embedded in it. They connect and collaborate with each other within the box using the Smarty Architecture.

2. Configuration

A configuration represents ad-hocly networked Smarties, which are collaborating to realize a particular user request at a certain time. A smarty device can be part of one or more than one Configurations at a time depending on the resources available, and this decision is made by a Smarty itself. During the life time of Configuration, a Configuration can be in different states. It is further described in section 4.4.

3. Smarty Messages

Smarty Message is the basic unit of communication between Smarties.

4. Request Center / Request Center Smarty

In order to involve a number of Smarties in collaboration or in other words build a Configuration, a Smarty device needs some sort of indication or user input. If a user initiates a request then we need to have an interaction point called “Request Center (RC)”, which can receive user request and make necessary translations. An RC can be part of a Smarty device or an independent device, in case it is part of a Smarty, this Smarty is called Request Center Smarty (RCS).
5. Introduction Protocol

In [7] a functionality of self learning was built that allows Smarties to learn from other Smarties in the network and also teach them.

4.1.2 General Smarty Architecture Extensions

In this section any revisions of the basic concepts or introduction of new concepts of the Smarty Architecture are described. The revisions are required to fulfill the requirements formulated in chapter 3. The introduction of new concepts is just to extend general Smarty Architecture.

1. Smarty Interface

Smarty Interface is the revised and extended version of the Smarty Control, where internal and external interfaces are explicitly defined. The key components of Smarty Interface are also introduced, such that the goals of the Smarty Control can be accomplished. In addition to this, Smarty Interface has been designed in such a way that the needs of service discovery and usage can also be fulfilled. It generates Smarty Messages for such needs, whenever required. The reason to revise and extend Smarty Control is discussed in section 4.2 and the details of Smarty Interface and its components are discussed in section 4.3.

2. Smarty Messages

Smarties can only communicate through exchanging Smarty Messages; therefore it is required to define the format and contents of these messages. The contents of Smarty Message are defined such that the Smarty Architecture’s requirements described in chapter 3 can be fulfilled. In section 4.5, the format and contents of Smarty Messages are described in detail.

3. Configuration States

During the life time of a Configuration, it can be in different states. These states are named as “Setup”, “Execution”, “Control” and “Destroy”, a new state “Inactive” has been introduced in this project. A Configuration moves to Inactive state from an Execution state. In the execution state of a Configuration, normal device functions can be accessed, but in Inactive state of a Configuration a device can make its own decision about the use of its device functions. A detailed discussion on Configuration states can be found in section 4.4.

4. Integration of New Smarties in a Configuration

This novel ability allows the integration of new Smarties in a Configuration. Any device which fulfills a certain criteria can join a Configuration without breaking or suspending
already running Configuration. In this way we achieve an easy integration of Smarties in Smarty-based ad-hoc network.

The following two scenarios may occur, where we need to integrate new devices in a running Configuration:

- A Request Center Smarty joins a Configuration by fulfilling necessary and sufficient conditions.
- A configuration is built such that it only fulfilled the necessary conditions. It may be the case that a new Smarty device brings some auxiliary functions and can join this Configuration, therefore adding more features to a Configuration.

5. Configuration Initiator Smarty (CIS)

The Request Center Smarties can receive user requests and initiate the process of building a Configuration; in other cases some automatic interrupt may also initiate this process of building a Configuration. The Smarty device, which initiates this process, is called a Configuration Initiator Smarty.
4.2 Smarty Device Architecture

In this section, we will discuss Smarty Device Architecture, which provides support of Service discovery and usage to Smarties. We will also provide a logical overview of a Smarty device in order to illustrate its revised and new components.

We have found in chapter 3, that in addition to other requirements, the Smarty Architecture needs a mechanism to discover and use services available on other Smarties. It is concluded in chapter 3, that a customized solution is required to fulfill these needs. In order to achieve this, we have extended the Smarty Architecture and introduced a simple service discovery and usage mechanism within the Smarty Architecture and introduced a Smarty Message format.

Whenever a Smarty needs to discover a service, it sends Smarty Messages in the network, where Smarties providing this particular service respond to this message. The details of the Smarty Messages for discovery are provided in section 4.5. Similarly whenever a Smarty device needs an action to be performed by other Smarties, it builds a Smarty Message filled with actions and sends to other Smarties. We will discuss in section 4.4 and 4.5, how this service discovery and usage mechanism is provided and its relation with Smarty Architecture’s concept of Configuration.

A brief overview of a Smarty device has been discussed in section 1.1, it is illustrated that each Smarty device has a Smarty Control and a Knowledge Base component in addition to the basic functionality of the device. We find that the interactions between Smarty device components have been stated but not elaborated enough, such that we can identify them clearly at architectural level, which is also required in case we need to introduce new components or replace and modify the existing ones.

In this project, we have revised and extended Smarty Control of a Smarty device and renamed it to Smarty Interface, we have also embedded Knowledge Base within Smarty Interface. Since the design and implementation of the Smarty Architecture in [6], [7] was to equip UPnP devices, therefore Smarty Control and Knowledge Base components were specified separately. Smarty Interface generates Smarty Messages, which are communicated in the network by a newly introduced component called Smarty Infra.

In figure 8, we have shown a logical overview of a Smarty device, where Smarty Interface is shown as a black box. Its internals are discussed in section 4.3. Here, we describe the purpose of interactions between different Smarty device components as follows:

1. This is the interaction between Smarty Interface and Local Device Application and its Middleware. Here Local Device Application refers to the core functionality of a device, where a Middleware may or may not be present. Using this interaction the Smarty Interface can access a device’s functionality and expose to other devices in the form of services. We’ll see in section 4.3 that a Smarty Service component of the Smarty Interface interacts with device’s functions, it enables the use of these
functions based on the interpretation of the Knowledge base by Smarty Interface components. Smarty messages are built to discover and build ad hoc network of functionalities distributed on a number of Smarties, and later inform Smarties to perform some actions or respond to requests initiated by other Smarties.

2. This is the interaction between Smarty Interface and Smarty Infra; Smarty Infra provides the communication support to Smarty Interface.

3. Interactions 3, 4 and 5 correspond to the interactions of Smarty Interface, local Device Application and Smarty Infra to the device’s hardware and software resources like operating system, memory etc.

4.3 Smarty Interface Components and Interactions

In the previous section, we have provided a black box overview of the Smarty Interface along with other components of a Smarty device, whereas in this section, we show the internal components of the Smarty Interface and describe the interactions between these components.

Figure 9 shows the internals of the Smarty Interface.
Description of the Interactions

The interactions between different components of the Smarty Interface are described below:

1. **Request Center (RC) and Smarty Controller (SC)**
   a. RC receives user requests and forwards to SC.
   b. SC may need to inform the user about the response to user’s request through RC.

2. **SC and Smarty Infra**
   a. Smarty Infra forwards the messages received from other Smarty devices to SC.
      SC identifies type of the messages and does the following:
      i. If it is for introduction protocol (see definition in section 4.1) then sends to Ontology Update Engine.
      ii. If it is for the Configuration sends it to Interpreting Engine.
   b. SC sends Smarty Messages to other Smarty devices using Smarty Infra, and these messages are either generated by the Interpreting Engine or Ontology Update Engine.
3. **Ontology Update Engine (OUE) and SC**  
   a. SC contacts OUE for following reasons:
      i. If theSmarty device has been introduced in the network 1st time, it may need to go through Introduction Protocol. This indication is received by SC from RC and it forwards it to OUE.  
      ii. SC has received Smarty Message which is for Introduction Protocol then it forwards to OUE.
   b. OUE generates packets for Introduction Protocol and forwards to the SC for further action.

4. **SC and Interpreting Engine (IE)**  
   a. SC receives a user request from RC and forwards it to IE, which takes necessary steps to complete this request.
   b. SC receives Smarty Messages from some other Smarty and if messages are for some Configuration, it forwards to IE. IE interprets the messages and performs actions accordingly.
   c. In case IE needs to send a response to the request received from RC or send a Smarty Message, it forwards such information to SC.

5. **IE and Long Term KB (LTKB)**

IE needs to consult LTKB in order to retrieve and interpret the information stored in LTKB. In some cases IE may need to update the information stored in the LTKB.

6. **IE and Short Term KB (STKB)**

IE needs to store some information in the STKB, like the status of the running Configuration(s) and the decisions it makes in order to build a Configuration. It may also store its experience of a building/maintaining a Configuration, which can be stored in LTKB, later on.

7. **OUE and LTKB**

OUE consults the LTKB for the availability of the Update Template\(^1\). It may need to update information in LTKB in response to the completion of Introduction Protocol.

8. **OUE and Short Term KB**

OUE may need to store some temporary information in STKB related to the Introduction Protocol.

9. **Smarty Service (SS) and IE**
   a. IE interacts with SS in order to perform some device specific actions. IE may receive action requests from some other Smarty, encapsulated in a Smarty Message or based on the actions stored in a Recipe.

---

\(^1\) Update Template is used in Introduction Protocol in order to exchange the necessary information with other Smarties.
b. SS may contact IE to inform about the device status or to generate Smarty Messages on behalf of the device, while being part of a Configuration.

10. SS and Local Device Application and Middleware

a. SS has been divided into two parts, generic and device specific. The generic part allows it to interact with IE, whereas the device specific part handles local device action calls and responses.

b. The local Device Application may require a function to be performed by some other Smarties or send data to other devices, therefore it interacts with SS.

4.4 Smarties’ Interactions and Configuration Management

Smarties collaborate in order to build ad hoc network of functionalities as described in section 1.1, which results in the formation of a Configuration. This collaboration requires exchange of a number of messages between Smarties to build and manage a Configuration. The nature of the messages varies from discovery of services, accessing the functionality provided by other Smarties or managing the configuration. In this project, it has been decided to design a message format, hence a protocol such that the needs of these messages can be fulfilled, such messages are generated by the Smarty Interface or Ontology Update Engine components of the Smarty Interface, which is part of each Smarty.

It is described in section 1.1, that a Smarty has Recipes stored in its Knowledge Base and these Recipes provide the information a Smarty needs for collaboration with other Smarties. In a Recipe, it is defined which actions a Smarty needs to perform in order to build a configuration. This results in generation of Smarty Messages for Configuration management including messages for discovery and negotiation as well as message for application specific actions. The application specific actions refer to the actions which are performed by different Smarties in order to realize the goal of an overall application. These actions can be defined in the Recipe, whereas the need of data transfer between Smarties can be handled by generating and exchanging Smarty Messages, or it can be achieved by using some out of band protocols.

In this project the definition of a Smarty device’s services is defined in the Recipes, where it is also mentioned explicitly the number and sequence of actions that a Smarty has to perform in order to collaborate with other Smarties. The information of these application specific actions distributed on different Smarties can be defined at the time of Recipe development or can be shared during Introduction Protocol.

We have already described in section 4.1.2 that a Configuration can move between different states and each Smarty device maintains the same state of the Configuration once a Configuration is in place. It is important to remember here that a Smarty device may engage in a number of Configurations depending on its resources. In order to understand better the concept of Configuration states, we show the state diagram of a Configuration in figure 10. The labels on the edges in this diagram may result in internal
actions within a Smarty or generation and exchange of Smarty Messages between Smarties.

In the following, we describe the behavior of a Configuration in detail and which important steps are taken in different states of a Configuration.

4.4.1 Configuration Setup State

In the configuration setup state, a CIS starts building a Configuration. In this state all the required Smarties are discovered, selected and requested to join a Configuration. This process can be initiated by a RC or automatically triggered at CIS.

In order to mark the completion of a Configuration Setup, CIS waits until all the selected Smarties have joined, and then it propagates a transition message, such that all the joined Smarties move this particular Configuration to next State. It may be the case that CIS has to wait for a Configuration setup completion message from other Smarties, before it can propagate a transition message.
4.4.2 Configuration Execution State

Once a Configuration is successfully built, all the Smarties involved in the Configuration move this particular Configuration to Execution state, where application specific actions can be executed to realize the goal of building a Configuration. While moving a Configuration to this state, Smarties update the list of Actions to be performed in Configuration Execution. These actions are retrieved from LTKB and stored in STKB. Smarties perform these actions either locally or send to other Smarties based on the nature of actions defined in the Recipe. The number and sequence of actions is also defined in the Recipe.

Based on the Configuration completion criteria defined in LTKB, a Configuration may move to Destroy Configuration, Inactive or Control states.

4.4.3 Configuration Control State

In this state control actions may interrupt a running Configuration or the application being performed in the Configuration. In order to control a Configuration, it can be moved to Control state and later back to Execution state. The control of a Configuration may refer to tuning of the Configuration. In case Control actions target the running application, it is defined by the application designer itself.

Request for Configuration control may be initiated at CIS, which can respond accordingly, whereas requests for application control can be initiated at any Smarty device and these are application dependent.

4.4.4 Configuration Inactive State

In this state the status of an already running Configuration can be stored and retrieved whenever required. In order to move a Configuration to this state, Smarties update their STKB and if necessary LTKB.

4.4.5 Destroy Configuration State

Once the goal of building a Configuration is achieved or Configuration setup fails, all the necessary information regarding a Configuration is destroyed. This process is initiated by CIS.

Once a Destroy Configuration Message is initiated, Smarties clear Configuration related information from STKB and may Update LTKB in order to store experiences of this Configuration.
4.5 Smarty Message Design

Smarty devices collaborate through exchanging Smarty Messages for introduction or Configuration management. These messages can be of variable length and have a specific format. A smarty Message is divided in two parts, a Header and a Payload. The interpretation of the message Header enables a Smarty if it needs to interpret the payload. By Interpreting the payload a Smarty can identify if it can perform the action(s) stored in the payload.

Smarty Messages designed in this project are broadcast messages and can be bound to any transport protocol like UDP. Additional features like Quality of Service (QOS) and Security can also be defined either by extending the Header itself or defining an additional header.

In this section, we describe the format of Smarty message and the contents of its Header(s) and Payload. The contents of the Smarty Messages for Service Discovery and Usage are defined in the messages for Configuration Setup and Execution states respectively.

4.5.1 Smarty Message Headers

The following figure shows the format of the Smarty Message Headers for a Configuration and Introduction Protocol.

![Configuration Header](image)

### Configuration Header

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Configuration ID</th>
<th>State ID</th>
</tr>
</thead>
</table>

### Introduction Protocol Header

<table>
<thead>
<tr>
<th>Message Type</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

Figure 11 – Format of Smarty Message Headers
In the following table the fields of a Smarty Message Header for a Configuration are described. The fields for Introduction Protocol will be defined in future work.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Type</td>
<td>It identifies that this message is for a Configuration</td>
</tr>
<tr>
<td>Configuration ID (CID)</td>
<td>It identifies a unique Configuration and is generated by a CIS. It enables a receiver to decide if it is part of such a Configuration</td>
</tr>
<tr>
<td>State ID</td>
<td>It identifies the state of a Configuration for which this Smarty Message is generated</td>
</tr>
</tbody>
</table>

Once a Smarty device interprets this Header, it finds that if it is part of a Configuration mentioned in the Header. If it is part of such a Configuration and holds same Configuration state, it starts interpreting the Payload otherwise it ignores the payload. In case the Configuration ID is not matched then a Smarty interprets the State field, because this header may represent a request for Configuration Setup, in this case Smarty can start interpreting the payload of the Smarty Message. If the Configuration ID is not matched and the state field represents a state other than Configuration Setup, a Smarty can simply ignore this message without interpreting the Payload.

### 4.5.2 Smarty Message Payload

A Smarty Message Payload is used to inform other Smarty devices in a Configuration that which actions have to be performed. The contents of the payload depend on the Configuration state as discussed in the following sections

#### 4.5.2.1 Configuration Setup State Payloads

For Configuration Setup state, a CIS starts building a Configuration and sends messages with payloads embedding requests, responses and notifications for service discovery and Configuration Setup negotiation.

**Search request Payload**

A CIS sends a Smarty Message with this payload in order to start discovering services exposed by other Smarties. In the basic form, this discovery mechanism is based on search of services request and selection of the Smarties responding to this request. The search and selection criterion is based on the Rules defined in the Knowledgebase. The following figure shows the contents of this payload.
In the following table contents of the Payload for Search Request are described.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Name</td>
<td>Search request</td>
</tr>
<tr>
<td>Request ID</td>
<td>Identification of this particular Search request. This is required because a</td>
</tr>
<tr>
<td></td>
<td>number of Search requests may be generated by a CIS and this field is useful</td>
</tr>
<tr>
<td></td>
<td>in identifying a particular Search request and Response.</td>
</tr>
<tr>
<td>Service 1,2</td>
<td>Actual search criteria (the required service on a Smarty)</td>
</tr>
</tbody>
</table>

**Search response Payload**

A Smarty device responds to a Search request with the payload contents as shown in the following figure:

In the following table, contents of the Payload for Search response are described.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Name</td>
<td>Response to a Search request</td>
</tr>
<tr>
<td>Response ID</td>
<td>It refers to the Request ID in the Search request and</td>
</tr>
</tbody>
</table>
it helps a CIS in identification of a particular search response

| Service | It is the name of the service that was asked for in the Search request. This field is sent to keep discovery process simple, such that it enables the initiator of the search request to identify that one of its requested services has been found |

Join request Payload

A CIS sends a Smarty Message payload with the Join request. The contents of this payload provide enough information such that the receiver can find a corresponding Recipe in its LTKB and decide if it can join this Configuration. The following figure shows some possible contents of Join Configuration request Payload.

<table>
<thead>
<tr>
<th>Request Name</th>
<th>Configuration request Name</th>
<th>Configuration request Type</th>
<th>Configuration request Context</th>
<th>CIS ID</th>
<th>IDs</th>
</tr>
</thead>
</table>

Figure 14 – Join Configuration request Payload

In the following table contents of the Payload for Join request are discussed.

Table 7 – Configuration Setup State: Join Configuration request Payload Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Name</td>
<td>Join Configuration request</td>
</tr>
<tr>
<td>Configuration request Name</td>
<td>The name of the request for which a configuration should be built</td>
</tr>
<tr>
<td>Configuration request Type</td>
<td>The type of the request for which a configuration should be built</td>
</tr>
<tr>
<td>Configuration request Context</td>
<td>The context of the request for which a configuration should be built</td>
</tr>
<tr>
<td>CIS ID</td>
<td>ID of the Configuration Initiator Smarty</td>
</tr>
<tr>
<td>IDs</td>
<td>It is the list of IDs of the Smarties, which are selected in response to Search request. Only these Smarties can respond to this request</td>
</tr>
</tbody>
</table>
Join response Payload

A Smarty device responds to a Join Configuration request with the payload contents as shown in the following figure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Name</td>
<td>Join Response to a Join Configuration request</td>
</tr>
<tr>
<td>Accepted</td>
<td>It is the value of a Smarty’s response that if it has accepted the Join request</td>
</tr>
</tbody>
</table>

Figure 15 – Join response Payload

In the following table contents of the Payload for Join response are described.

Configuration Completion Payload

A Smarty device may send a Configuration Completion notification Payload. It is required in case there are some Smarties that have replied to “Join” configuration request and these devices may also need to look for other Smarties in order to involve them in the same Configuration. The following figure shows contents of such a notification:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification Name</td>
<td>CIS ID</td>
</tr>
</tbody>
</table>

Figure 16 – Configuration Complete notification Payload

In the following table contents of the Payload for Configuration Complete notification are described.
Table 9 – Configuration Setup State: Configuration Complete notification Payload Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification Name</td>
<td>Configuration Complete notification</td>
</tr>
<tr>
<td>CIS ID</td>
<td>This field identifies that this notification is for CIS, because, it is CIS which keeps track of a Configuration if all the requirements have been fulfilled and Configuration is complete</td>
</tr>
<tr>
<td>Complete</td>
<td>Boolean describing that if this device has completed its part of building Configuration successfully</td>
</tr>
</tbody>
</table>

Transition notification Payload §

A CIS sends a Transition notification to the Smarty devices, which have joined a Configuration. The receivers of this notification move the particular Configuration to the state mentioned in the Payload. The following figure shows its contents:

![Figure 17 – Transition notification Payload](image)

In the following table contents of the Payload for Transition notification are described.

Table 10 – Configuration Setup State: Transition notification Payload Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification Name</td>
<td>Name of the notification from CIS such that receivers can move the Configuration to next state mentioned in the following field</td>
</tr>
<tr>
<td>Configuration State</td>
<td>It identifies the name of the state a Configuration should be moved to</td>
</tr>
</tbody>
</table>

Search Configuration request Payload

A Smarty device may join some already running configuration depending on the criteria defined in the Knowledge Base. This concept has been introduced in section 4.1.2.

§ This payload can be sent during any Configuration States.
A Smarty device may send a Smarty Message with Search Configuration request Payload to look for some already running configuration, which it can join.

<table>
<thead>
<tr>
<th>Request Name</th>
<th>Device Type</th>
<th>Necessary Condition</th>
<th>Sufficient Condition</th>
</tr>
</thead>
</table>

Figure 18 – Search Configuration request Payload

In the following table contents of the Payload for Search Configuration request are described.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Name</td>
<td>Name of the Search Configuration Request</td>
</tr>
<tr>
<td>Device Type</td>
<td>It identifies the type of Smarties that can listen to this request, for example RCS/CIS or other type of Smarties</td>
</tr>
<tr>
<td>Necessary Condition</td>
<td>Any Smarty device willing to join some running Configuration must satisfy this Condition</td>
</tr>
<tr>
<td>Sufficient Condition</td>
<td>Any Smarty device willing to join some running Configuration may satisfy this Condition</td>
</tr>
</tbody>
</table>

Table 11 – Configuration Setup State: Search Configuration request Payload Fields

Search Configuration response Payload

A Smarty device responds to a Search Configuration Request with the payload contents as shown in the following figure.

<table>
<thead>
<tr>
<th>Response Name</th>
<th>Device ID</th>
<th>Configuration ID</th>
<th>Configuration State</th>
</tr>
</thead>
</table>

Figure 19 – Search Configuration response Payload

In the following table contents of the Payload for Search Configuration response are described.
Table 12 – Configuration Setup, Inactive, Execution and Control State: Search Configuration response Payload

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Name</td>
<td>Name of the Search Configuration Response</td>
</tr>
<tr>
<td>Device ID</td>
<td>The ID of the device which initiated a Search Configuration Request</td>
</tr>
<tr>
<td>Configuration ID</td>
<td>If the Conditions in the Search Configuration Request are matched against a Configuration then this Configuration’s ID is sent in this Response</td>
</tr>
<tr>
<td>Configuration State</td>
<td>It is important to notify the receiver of this Payload about the State of the Configuration, so that it can perform actions accordingly</td>
</tr>
</tbody>
</table>

4.5.2.2 Execution, Inactive and Control States Payloads

The format of the Payload for Execution, Inactive and control States is defined in this section, whereas the actual contents of the Payload are read from the Recipe. A Smarty can send a payload with more than one action and the receiving Smarties can perform an action based on the information in their Knowledge Base.

It is important to remember here that the Payloads in these states also correspond to the Execution and Control of device specific functions as well as a Configuration. An example of Configuration related actions are Transition and Search Configuration Response. The device specific actions depend on the application and are different for different device types.

Execution, Inactive and Control State Action(s) Payload

The following figure shows the format of Payload for Execution, Inactive and Control States.

In the following table contents of the Payload for Execution, Inactive and Control states are described.
### Table 13 – Execution, Inactive and Control State Payload Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Actions</td>
<td>It identifies the number of actions in the Payload</td>
</tr>
<tr>
<td>Action ID</td>
<td>It identifies the Action to be performed</td>
</tr>
<tr>
<td>No. of Args.</td>
<td>The number of arguments for an Action</td>
</tr>
<tr>
<td>Args</td>
<td>Argument’s name and value</td>
</tr>
</tbody>
</table>

### 4.5.2.3 Destroy Configuration State Message

In this state a Smarty Message Payload is optional and a Smarty Message itself is sufficient to inform other Smarties to destroy a Configuration.

The following is a Smarty Message for Destroy Configuration State.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Configuration ID</th>
<th>Phase ID</th>
<th>Payload (optional)</th>
</tr>
</thead>
</table>

![Figure 21 – Destroy Configuration State Message](image)

In the following table contents of the Smarty Message for Destroy Configuration state are described.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Type</td>
<td>This message is for a Configuration</td>
</tr>
<tr>
<td>Configuration ID</td>
<td>The ID of the Configuration</td>
</tr>
<tr>
<td>State ID</td>
<td>State of the Configuration</td>
</tr>
<tr>
<td>Payload</td>
<td>optional</td>
</tr>
</tbody>
</table>
4.6 Knowledge Base (KB) Design

In this section, we talk about the design of Long Term Knowledge Base (LTKB), where information is stored in the form of “Recipes”, “Rules”, and “Templates” etc. This information is necessary to participate and play a role in smarty-based ad hoc network. The information in the Knowledge Base is organized in a way that facilitates its interpretation as well as adaptation to the increasing needs.

Recipes prescribe what a Smarty device should do to perform a role in a network of Smarty devices. Rules allow the Smarty device to make decisions and Templates let it adapt to the introduction of new devices.

Knowledge Base is designed based on the concept of Ontology, which makes it well structured, readable and understandable. Knowledge Base design is inherited from [7] and can also be found in Appendix B.

4.7 Software Design

We have defined following three main components of the Smarty Software:

- Smarty Interface
- Smarty Infra
- Device Specific

The sequence of the interactions between internals of some of these components has been highlighted in the Sequence diagrams A, B, C. in Appendix C, whereas the structure of these components and their relationship has been depicted in the Class diagram in Appendix C.

Smarty Interface

Smarty Interface is a generic software component which is same for all type of Smarties, but part of a Smarty Interface component is device specific to take care of the interactions of Smarty Interface with the Device functions.

Smarty Infra

This component handles the communication needs of the Smarty Interface; it is a semi-generic component for all Smarty devices. This component interacts with Smarty Interface though provided Interfaces.
Device Specific

This component is independent of the Smarty Interface and Smarty Infra and only interacts with Smarty Interface using the provided interfaces as shown in the diagrams in Appendix C.
5 Implementation

A prototype is developed as a proof of concept for the Smarty Architecture for resource constrained embedded devices in the lighting systems domain. An example application of this domain has already been shown in figure 3. It is intended to provide an implementation of the Smarty Architecture satisfying the requirements described in chapter 3.

In this chapter scenarios for the prototype are described. Furthermore details of the hardware and software implementations of the Smarties have also been provided.

5.1 Use case Scenario and Analysis

A number of use case scenarios are identified and different types of Smarties are defined. The requirements and scenarios for the demonstrator are also included in this section.

5.1.1 Use Case1: Switch on/off Lights

A user at home has the capability to switch on/off lights using a switch installed on the wall. The lights and switch could build a network to carry out this functionality.

5.1.2 Use Case2: Light Dimming

User bought a light dimmer with capabilities of sensing and controlling light bulbs. This dimmer can be tuned to perform dimming and day light harvesting. Once installed, the dimmer may join the network which already exists between light bulbs and light switch. In case no network exists, dimmer can build its own network.

This enables user to control lights from the switch as well as the interface provided by the dimmer e.g. which can be on the user’s mobile phone. Furthermore, when user leaves and returns back home, he finds that the network between light bulbs, light dimmer and light switch is behaving as required and he can still control light bulbs either using switch or dimmer.

5.2 Demonstrator Scenarios

In order to build the prototype, we have formulated a number of requirements for the prototype as shown in table 16. In order to realize use case scenarios, we have defined the following types of Smarties:

- Light Switch (Lsw) Smarty
- Light Controller (LC) Smarty
- Light Sensor (LS) Smarty
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMP-F-01</td>
<td>The Lsw Smarty should be able to communicate and form a Configuration with Smarties of type LC and LS.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-02</td>
<td>The Lsw and LS Smarties must provide a user interface or in other words these Smarties should have a Request Center.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-03</td>
<td>The Request Center should provide a choice to the user to start/stop or give some other input to the device. The devices without a Request Center are started at installation time and can be stopped using a hardware switch available on the device.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-04</td>
<td>All the Smarties involved in a Configuration must have a knowledge base file, with Recipes and Rules included.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-05</td>
<td>All the Smarties should be able to participate in building a network in order to achieve the functionality defined in use case scenarios.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-06</td>
<td>The different states of the Configuration should be displayed on the screen of each Smarty, so that user can appreciate the behavior of the Configuration.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-07</td>
<td>It should be possible to integrate LS Smarty in a Configuration (which is built between LCs and Lsw) without affecting the status of the Configuration.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-08</td>
<td>If LS Smarty joins an already existing Configuration, it should inherit the current state of the Configuration and starts providing functions required in this particular state.</td>
<td>M</td>
</tr>
<tr>
<td>SMP-F-10</td>
<td>An action (like Switch off) at Lsw may require other Smarties (LCs or LS) involved in the Configuration to move to Inactive state. These Smarties should move to Inactive state and if later required by Lsw move back to the Execution State.</td>
<td>M</td>
</tr>
</tbody>
</table>
The following two scenarios demonstrate Smarties participating in order to fulfill user needs defined in above mentioned use cases and in table 16.

**Scenario 1:**

Smarty devices of type LC and Lsw are available in a network and ready to serve user request of switching on/off lights.

1. User interacts with the Request Center at Lsw and gives the command to switch on lights.
   a. If Lsw is turned on 1st time, it makes necessary translations of user commands to select necessary actions from the Knowledge Base in order to build a Configuration with Smarties of type LC.
2. Lsw searches, selects and informs LCs as shown by the dotted lines in the figure 22 and builds a Configuration with these LCs. Lsw and LCs interprets their Knowledge base files in order to select actions, build Smarty Messages and make decisions.
3. Lsw starts communicating with the LCs, which are part of the Configuration and performs the user defined request of switching on/off.

In case user switches off lights using Lsw and switches on after some time, the network exists between devices and they don’t need to make a new Configuration as the previously built Configuration is restored.

![Figure 22 – Scenario 1](image)

**Scenario 2:**

In the existing network of Smarties a new type of Smarty device named Light Sensor (LS) is introduced as shown in figure 23.

1. User interacts with the Request Center at LS and wants to tune the device to maintain a particular light level.
2. As LS has been switched on 1st time, it needs to build a Configuration with devices of type LC and LSw. Before initiating a new Configuration request it finds if there is a Configuration already present between surrounding LCs.
   a. LS interprets its Knowledge base and searches for a Configuration, which involves LCs and Lsw. This request of LS is responded by Lsw and LS joins this Configuration between LCs and Lsw. This search request is responded by the Lsw as it is the initiator of the Configuration between Lsw and LCs. (see details of this Search Configuration message in section 4.5)
3. LS starts communicating with LCs to perform user defined request of maintaining a particular light level.

LS keeps on performing the required function, until user explicitly turns off the LS or Lsw. In case Lsw is turned off, all the Smarties involved in the Configuration stop performing the current function, in other words they place the Configuration in Inactive state. If user interacts again and switches on the LCs, LS may also starts performing its function again.

![Diagram](image)

**Figure 23 – Scenario 2**

### 5.3 Smarty Devices’ Hardware Implementation

We used the following hardware for building the prototype.

- Embedded Artists’[14] LPC2468 OEM (based on NXP’s ARM7TDMI LPC24xx microcontroller family)
- NXP UBA 3070 Driver Board
• LEDs
• Photo Transistor

LPC2468 OEM Board Capabilities

The development board comes with the capabilities mention below:

• Processor (NXP’s ARM7TDMI LPC2468 MC)
• Program Flash (128 MB NAND, 4MB NOR, 512 KB Internal)
• Data Memory (32 MB SDRAM, 96 KB Internal)
• Ethernet (100/10M Ethernet Interface)
• Clock Crystals (12 MHz)

The power supply required by the development board is 9-15 V DC. In the current project it is being used for demonstration purposes only and can be replaced by some battery power devices.

NXP UBA 3070 Driver Board

This driver board gets Pulse Width Modulation (PWM) signals from the LPC 2468 development board and controls the intensity of the light based on PWM signals. This driver board needs continuous power supply though power lines and supplies to LEDs, based on the PWM signals it receives from the development board.

LEDs

LUXEON K2 [17] LEDs are used.

Photo Transistor

A photo transistor is used to sense the light level.

5.3.1 Light Controller Smarty

A Light Controller device was built using the LPC2468 development board, such that the light intensity of the LEDs was controlled by the NXP UBA 3070 Driver board, which receives PWM signals from the LPC2468 development board. Figure 24 shows a Light Controller Smarty.

5.3.2 Light Sensor Smarty

A Light Controller device was built using the LPC2468 development board, such that the photo transistor was connected to the ADC pin on the board. Figure 25 shows a Light Sensor Smarty.
5.3.3 Light Switch Smarty

A Light Controller device was built using the LPC2468 development board, such that ADC capability of the board was used. Switching on/off signals were provided using the knob on the board.
5.4 **Software Implementation of Smarties**

Generic software is implemented for the Smarties based on the Software design defined in section 4.7. In addition to this a simple utility is also built, which generates Knowledge Base files for the Smarties. The design of the Knowledge Base shown in Appendix B has been followed to generate a Knowledge Base file for each Smarty. The software running on each Smarty device interprets its Knowledge Base file and performs actions and generates Smarty Message accordingly. A sample knowledge base file for a Light Switch Smarty device has been included in Appendix D.

This software enables Smarties to build and exchange Smarty Messages for building and managing Configuration. All the messages are broadcast messages, implemented on top of UDP/IP. This implementation has been done from scratch and it is single threaded, whereas following platform and technologies have been used to build Smarties software.

- uClinux Operating System
- C Programming Language
- Internet Sockets for communication
6 Conclusion

In this chapter, the results of the project are described and some suggestions for the future work are provided.

6.1 Results

This is a continuation project in the series of the Collaborative Smarties project conducted at NXP Semiconductors. In the initial development phases of Collaborative Smarties project a concept of ad hoc networking of devices named Smarties was defined and a number of features were added, this concept was called Smarty Architecture. The initial development goal of Smarties in previous projects was to enable backward compatible UPnP [3] devices for Multimedia applications domain. Therefore, a decision was made to develop Smarties based on UPnP. UPnP defines a set of protocols to expose a device’s functionality in the form of services and enable discovery and usage of these services. (See Appendix A for UPnP overview)

NXP Semiconductor is intended to use the concept of Smarty Architecture in the domain of Lighting Systems and it is perceived that UPnP is functionality overkill for such systems. The goal of this project was to identify the exact requirements of Smarty Architecture for UPnP and find a similar framework for very resource constrained Smarties.

We performed investigations in order to understand the concepts introduced in [6], [7], and identified the relationship between Smarty Architecture and the use of UPnP. It was identified that Smarty Architecture required a mechanism, which could fulfill the Smarty Architecture’s needs of discovering and using functionality distributed in the environment as well as the needs of exchanging Smarty Architecture dependent Messages. These early investigations are presented in the Problem Analysis chapter.

Based on these investigations, we formulated the requirements of Smarty Architecture for technologies like UPnP that could be used for resource constrained Smarties. A survey was performed in order to find UPnP replacement and a decision was made based on the findings. It was found that a customized framework is needed, because of the inherent constraints of the investigated frameworks and conflicts with the requirements in chapter 3. These findings and decision has been presented in chapter 3.

In order to fulfill the requirements of Smarty Architecture, we have extended the Smarty Architecture itself, such that the needs of finding services available on other Smarties and using these services can be fulfilled. The Smarty Control which is part of each Smarty has been renamed to Smarty Interface, and extended such that it generates Smarty Messages for service discovery and Usage. The Smarty Message format has been defined and the contents of these Messages are filled based on the information stored in the knowledge base of each Smarty.
The design of knowledge base is defined in [7], where the information about the services of a Smarty and the actions it has to perform to fulfill some functionality is described. This information is interpreted by the Smarty Interface. We have adapted this knowledge base design. The interpretation of the information stored in the Knowledgebase enables building Smarty messages for the requirements of discovering services available on other Smarties, informing Smarties about an action to be performed or exchanging Smarty messages for Smarty Architecture related issues like management of Configurations.

This makes a discovery mechanism dynamic such that decisions to discover and select devices are defined in the Knowledgebase in the form of Rules, and these Rules can be easily modified by making changes in the Knowledgebase of a Smarty. Similarly in order to inform a Smarty to perform some action, a Smarty Message is built based on the information stored in the Recipes, which includes the number and sequence of actions to be filled in the Smarty message.

In addition to this message format a number of features have been included in Smarty Architecture like the addition of a new state in the Configuration and integration of Smarty devices in a running Configuration. Some of the Smarty Architecture components like Smarty device architecture have also been modified. Furthermore, generic software was designed, by modifying the exiting one. Such design issues have been presented in chapter 4.

In order to verify the introduced concepts of Smarty Architecture, a working prototype was developed. A number of use case scenarios were identified and three types of Smarty devices were defined including Light Controller Smarty, Light Switch Smarty and Light Sensor Smarty. The generic software design (see Appendix C) was implemented and broadcast messages were exchanged between these devices. A utility was developed in order to generate a knowledge base file for a device. This has been described in chapter 5.

The specified use cases were executed and the prototype exhibited the required behavior. Three different type of Smarty devices were implemented including Light Controller, Light Sensor and Light Switch smarty. The prototype was implemented with code size of 60KBs, whereas the code size of the previously developed Smarty in [6], [7] was 1.7 MB. The code size achieved in this project is less than the initially defined size of 100KBs.

6.2 Future Work

The results of the current project show the prospects of using Smarty Architecture in the domain of very resource constrained embedded devices. We find a number of issues on which it can be further extended and improved.

The design of service discovery protocol defined in this project can be extended such that the information regarding the location of the devices and context of the user can be included in the discovery and selection process. It can also be improved such that the continuous presence of the selected devices can be assured. This can be done either by
receiving availability messages from the selected devices or sending monitoring messages by the Smarty device which has selected these devices.

The discovery and usage mechanism can be improved to deal with the scalability of the network. Currently a broadcast mechanism is used to send Smarty Messages; a technique of peer to peer message transfer can be adopted in case the limited number of devices participate in forming an ad hoc network in a community of thousands of devices.

The Smarty Message format can be extended to include Quality of Service (QoS) and security needs.

The extensions in knowledgebase design are also required in order to make Interpreting Engine component of Smarty Architecture more generic, such that it becomes possible to extract most of the information from Knowledgebase, which is required by Interpreting Engine to make decisions. This information can be related to generate Smarty Messages, handling actions or making decisions in a context aware manner. This flexibility is required to realize the extendible concept of Smarty Architecture, where more functionality can be added in the network of Smarties without modifying the exiting Smarties in the network.
Appendix A: UPnP Overview [6]

UPnP – the Universal Plug and Play architecture is an industry initiative designed to enable simple and robust connectivity among stand-alone devices and PCs from many different vendors. It is a result of the fact that these days all sorts of devices, starting from mobile phones to handheld computers to DVD players, are increasingly connecting to networks and are using a multitude of connectivity methods to do so. This trend increases the need for a self-configuring network that allows devices to enter the network, automatically find each other and leave the network again. The Universal Plug and Play architecture tries to accomplish that. It is build upon TCP/IP, UDP/IP and HTTP (web) technologies to allow seamless proximity networking in addition to control and data transfer of networked devices. The UPnP Forum [3] maintains the UPnP architecture.

UPnP Device Architecture

The UPnP Device Architecture (UDA) [3] is designed to support zero-configuration, “invisible” networking and automatic discovery for a breadth of device categories from a wide range of vendors. This means a device can dynamically join a network, obtain an IP address, convey its capabilities and learn about the presence and capabilities of other devices. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind.

UPnP uses common protocols; hence no device drivers are needed. UPnP networking is media independent. UPnP devices can be implemented using any programming languages and on any operating system. The UPnP architecture doesn’t specify or constrain the design of an API for application.

UPnP basic elements

The basic entities in UPnP are: control points, devices, services, actions, and events.

Control point: is an entity in UPnP network (some device) that is capable of discovering and controlling other devices.

Device: is an entity that implements service that can be used by others (control points). Within a device there can be other (embedded) devices. An example of device nesting is a video recorder device, containing a tuner device and a recording device. The tuner device will implement one or more “tuning” services, while the recording device will implement “recording” and “playback” services.

Service: is characterized by a name and set of actions. It has a state, modeled by a set of state variables.

Actions: consist of a name and input or output arguments. After invocation of an action, output arguments are updated and the internal state of services may be modified.
The UPnP device architecture defines the protocols for communication between UPnP control points and devices. It specifies six major phases of interaction based on the UPnP stack. Following figure gives an overview of the stack.
name of service, a URL for the service, and the actions (functions) provided by the service.

Similar to plain class methods, actions are characterized by a name and an argument list. Furthermore, the service description specifies for each argument if it is an input or an output. Finally, the service description also lists the state variables that are associated with a service and a change of such a variable must trigger the sending of events to other devices/control points.

4. Control

A control point that has retrieved the device and service description of a device can control it by sending actions to it. An invocation of actions is similar with remote procedure calling. After the action completes the result is returned.

The control point identifies a particular device and service by means of a URL that it has received earlier during device discovery and service retrieval. Control messages are expressed in XML and sent via HTTP requests. Results and errors are returned via HTTP responses. A control message is a plain ASCII string that can be created directly or by functions provided by a typical UPnP utility library, such as Intel SDK for UPnP on Linux.

5. Eventing

Besides controlling a device, a control point can receive events from a device. To enable this, a control point must send subscription message to a URL that is specified in the device description of the device. Upon state changes, a device will send an event notification to the delivery URL of the control point that was specified by the control point during the subscription.

6. Presentation

A device may provide a presentation URL that can be accessed from a standard web browser. If such a URL is specified in the device description, a control point can use a HTTP GET request to retrieve a HTML page from this URL and load it into a browser. Dependent on the underlying implementation, via the web page, the device may be controlled and its status may be monitored.
Appendix B: Knowledge Base Design [7]

Figure 27 – Knowledge Base Design

Recipes
Recipe a:
If request name is Switch On
then execute sequence [search, select, negotiate, perform action A,
send data ]

Capabilities
I have following Services:
- Service Type A: Light Switch (Lsw)
I can Collaborate With:
- Service Type B: Light Controller (LC)
- Service Type C: Light Sensor (LS)

Update Template  Rules  Properties
Figure 28 – Sequence Diagram (Configuration Setup 1)
Figure 29 – Sequence Diagram (Configuration Setup 2)
Figure 30 – Sequence Diagram (Configuration Execution, Inactive, Control, Destroy)
Figure 31 – Class Diagram
Appendix D: Knowledge base of a Lighting Switch Smarty

**Recipes**
- **Recipe Id (Context name, request name)**
  
  Example: Context name: Request name: Switch ON/OFF lights

**Configuration States**
- **Configuration Setup**
  
  - **Actions and Arguments**
    - Action1: Search('LC')
    - Action2: SearchResponse()
    - Action3: JoinConfigurationRequest('switchlights', ...)
    - Action4: JoinConfigurationResponse()
    - Action5: CheckConfigurationComplete()
    - Action6: SendTransition('exe')

- **Configuration Execution**
  
  - **Actions and Arguments**
    - Action1: SendSwitchValue(value)
    - Action2: SendTransition('exe/inactive/destroy')

- **Configuration Inactive**
  
  - **Actions and Arguments**
    - Action1: SendTransition('exe')

- **Configuration Control**
  
  - **Actions and Arguments**
    - Action1: SendTransition('exe')

- **Configuration Destroy**
  
  - **Actions and Arguments**
    - Action1: SendTransition('destroy')
    - Action2: DestroyConfiguration

**Capabilities**
- **Services**
  - **Service Type**
    - Example: Light Switch

- **Device Type**
  - CIS/RCS

**Rules**
- **Search Selection Criteria**
  
  Category of device Type 1: LC
  Number of Devices of type 1: 2
  Number of Device Types: 1

- **Configuration Complete Criteria**
- **Configuration Destroy Criteria**

**Update Template**
## Bibliography

### References

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>