Final thesis

Design and implementation of UPnP network functionality for a digital TV receiver

by

Dario Lentini and Gustav Salenby

LIU-IDA/LITH-EX-A--08/063--SE

2008-12-19
Final thesis

Design and implementation of UPnP network functionality for a digital TV receiver

by

Dario Lentini and Gustav Salenby

LIU-IDA/LITH-EX-A--08/063--SE

Supervisor : Simon Edlund
at Zenterio

Examiner : Prof. Zebo Peng
Dept. of Computer and Information Science
at Linköpings universitet
Abstract

Media extenders or digital media receivers are network devices that are used to retrieve digital media files (such as music, pictures, or video) from a media server and play or show them on a TV or home theater system. A technology that is often associated with these devices is the Universal Plug and Play (UPnP) technology. This technology enables network devices to be used without requiring the user to do network configuration on it.

This thesis demonstrates how a device that is normally used for receiving digital television broadcasts can be enhanced to support media extender functionality. The thesis describes the design and implementation of the technologies that are needed to accomplish this functionality. The main topics are centered around on how UPnP awareness and media rendering (decoding) are incorporated into the device.

Keywords: UPnP, media extender, digital media receiver, digital media adapter, digital TV receiver
Acknowledgements

We would like to begin with thanking Zenterio for giving us the opportunity of doing an interesting and challenging project. Especially we want to thank Thomas Abrahamsson and Simon Edlund who have been guiding us through the process. We also would like to thank Berth Axelsson, Erland Costyson and Carl-Eric Hedin at Zenterio for their technical support.

We would like to thank Prof. Zebo Peng at the Department of Computer and Information Science at Linköping University for his guidance, ideas and comments.

At last we want to thank Olof Friman and David Hamréén for helping us improve the thesis.
Contents

1 Introduction .................................................. 1
  1.1 Background ............................................. 1
  1.2 Purpose .................................................. 2
  1.3 Limitations of the report ............................... 2
  1.4 Intended audience ...................................... 2
  1.5 Methodology ............................................ 3
  1.6 Prerequisites ............................................ 3
    1.6.1 Digital TV receivers ............................... 3
    1.6.2 UPnP ................................................. 5

2 Problem definition ........................................... 7
  2.1 Vision ................................................... 7
  2.2 Constraints .............................................. 8
  2.3 Vision breakdown ....................................... 8
  2.4 Problem statement ...................................... 9
    2.4.1 Functional requirements .......................... 9
    2.4.2 Performance requirements ......................... 9

3 Theory ......................................................... 11
  3.1 UPnP ..................................................... 11
3.1.1 Step 1: Discovery ........................................... 12
3.1.2 Step 2: Description ........................................ 13
3.1.3 Step 3: Control ............................................. 14
3.1.4 Step 4: Eventing ............................................ 14
3.1.5 Step 5: Presentation ........................................ 14
3.2 UPnP AV ......................................................... 14
  3.2.1 Media Server .............................................. 15
  3.2.2 Media Renderer ............................................ 16
  3.2.3 Control Point .............................................. 16
3.3 SSDP ............................................................. 17
  3.3.1 SSDP Discovery Requests .................................. 17
  3.3.2 SSDP Presence Announcements ............................ 17
3.4 SOAP ............................................................ 18
3.5 ZiDS ............................................................. 18
  3.5.1 Architecture ............................................... 18
  3.5.2 Tasks ....................................................... 20
  3.5.3 HalfPipe .................................................... 20
  3.5.4 ControlIT .................................................. 21
  3.5.5 OS21 ....................................................... 22
  3.5.6 OSPlus ..................................................... 22
3.6 Hardware platform ............................................. 22
  3.6.1 The MB442 reference board ............................... 22
3.7 Architectural patterns ........................................ 23
  3.7.1 Event-driven architecture ................................. 23
  3.7.2 Layering ................................................... 24
  3.7.3 Client-Server ............................................. 24
4 Architecture ..................................................... 27
  4.1 System architecture .......................................... 27
  4.2 Design goals ................................................ 27
4.3 Proposed architecture ........................................... 28
  4.3.1 System interface ........................................... 30
  4.3.2 Control Point ............................................... 30
  4.3.3 Media Renderer .............................................. 32

5 Implementation ....................................................... 35
  5.1 Control point .................................................... 35
    5.1.1 UPnP ....................................................... 35
    5.1.2 Database ................................................... 40
    5.1.3 Content Browser ........................................... 41
    5.1.4 XML ........................................................ 43
    5.1.5 Navigator .................................................. 44
  5.2 Media Renderer .................................................. 45
    5.2.1 HTTP ....................................................... 46
    5.2.2 Injector .................................................... 46
    5.2.3 Playback .................................................... 48
    5.2.4 Viewer ....................................................... 48

6 Testing .............................................................. 49
  6.1 Method .......................................................... 49
  6.2 Module test ..................................................... 49
    6.2.1 UPnP ....................................................... 49
    6.2.2 XML ........................................................ 50
    6.2.3 Database .................................................... 51
    6.2.4 Result ....................................................... 51
    6.2.5 Content Browser ........................................... 51
    6.2.6 Navigator .................................................... 53
    6.2.7 HTTP ....................................................... 54
    6.2.8 Injector .................................................... 54
    6.2.9 Playback .................................................... 55
    6.2.10 Viewer ..................................................... 56
Chapter 1

Introduction

1.1 Background

Consumption of digital media is a part of everyday life for the majority of the western population. People listen to music through their portable mp3 player, watch movies in front of their living room TV and browse images on their personal computer. Because of this, there is a huge amount of media content spreading a diversity of devices. This also means that there is an increasing need for a uniform strategy for organizing and sharing media.

Almost every home is equipped with at least one personal computer. A computer that is most likely connected to a local home network. There exists an opportunity to use this fact and connect all media devices to this network. This would make it possible to use some strategy for organizing and sharing media at home.

Traditionally the TV and its surrounding equipment have been the natural center for media consumption. Here people watch their favorite TV program, enjoy DVD movies and often listen to music. This would be the ideal place to put a device that is able to reach media that spreads across other devices at home.

Zenterio AB is a Swedish company who is providing software for digital
TV receivers. They see a potential market where people use their existing digital TV receiver solution not only for watching digital TV broadcasts, but also as the media center device described above. Having software that implements this on a regular digital TV receiver platform would give their software a competitive edge. This thesis describes the design and implementation of such a digital TV receiver.

1.2 Purpose

The purpose of the project is to build a prototype for a media center device using an ordinary off-the-shelf digital TV receiver platform.

The prototype is required to incorporate UPnP technology and have basic functionality for viewing images, playing audio and video.

1.3 Limitations of the report

The project report is focused on describing the main features of the process that lead to a working prototype. There are no intentions to give a detailed recipe, as this would mean that the report rapidly would grow out of scope.

1.4 Intended audience

The report is primarily intended for people in the computer industry or people in the computer science academia. Large parts of the report are written so that anyone with a general technical background should comprehend the principal features of the report. In order to get the full picture, some knowledge about software development, computer hardware architecture and Internet technology is good to possess.
1.5 Methodology

The project was performed using an agile software development process called Scrum [1].

Scrum uses iterative development in order to achieve a product. Each iteration is called a sprint and consists of 30 days which are dedicated to implement the product’s backlog. The product backlog is a list of prioritized product requirements determined by the product owner. The product owner is the voice of the customer and tries to ensure that the right things, from a business perspective, are done. Multiple teams implement product increments in parallel. These teams are self-organizing and coordination is done at a brief daily status meeting called a scrum [2].

1.6 Prerequisites

The problem definition in Chapter 2 assumes that the reader is familiar with the concept of UPnP and the structure of a digital TV receiver. In order to help the reader comprehend this some material of importance will be presented in this section. If the reader is familiar with these technologies, he or she can choose to move directly to Chapter 2.

1.6.1 Digital TV receivers

In the past, television was broadcasted and received by analogues systems. The analogous receiver was usually a part of the actual TV. As the transition from analogue to digital television began, people needed to be able to receive and watch digital broadcasts on their old analogous TV. To make this possible external digital receivers where made available.

Even though most TVs today are capable of receiving digital broadcasts, the external receiver has for various reasons stayed.
Structure

A typical receiver consists of a special purpose computer with some specific peripherals attached. An overview of such a system is depicted in Figure 1.1. The digital TV specific parts are not covered, as they are not relevant for the project.

Figure 1.1: The structure of a digital TV receiver

**Processor:** The processor usually consists of a main controller and some additional decoders.

The main controller runs software that initializes and manages the on board hardware components. The decoders are special purpose CPUs for audio and video decoding.

**Memory:** The receiver is often equipped with flash memory for storing permanent data and RAM for run-time usage.

**External Interfaces:** Connection with other devices is made by a set of external interfaces. A standard receiver is equipped with different types of audio and video outputs. Modern receivers are also often equipped with an Ethernet connector.
1.6.2 UPnP

The Universal Plug and Play (UPnP) is a networking protocol made to enable simple connectivity between networked home appliances. It provides supports for zero-configuration, “invisible” networking and automatic discovery for a wide variety of devices [3].

Given an arbitrary UPnP compatible device, it should by connection to a network, without user interference, obtain an IP address, convey its capabilities and learn about the presence and capabilities of other devices.

UPnP uses the IP protocol for operation.

The UPnP protocols are organized and published by the UPnP Forum.

Structure and operation

There are three entities described in the UPnP architecture: the device, the service and the control point. Devices contain services and the control points interact with them. Using normal network jargon, the (controlled) device is equivalent to a server and the control point is equivalent to a client. The server hosts several services, e.g. a print spooler system, and the client uses (controls) these services.

In order for the devices and control points to find each other they multicast several types of messages over the network. The device sends advertising messages and the control point sends discovery messages. If a control point find an interesting device, it can get more information about it by acquiring the device description. The device description contains, among other things, a list of services that are available on the device. The control point can choose to get more information about one of these services by acquiring the service description, which includes a list of commands or actions that the service supports. By using the information in a service description, the control point can control the service and the device.

Chapter 3 gives a detailed explanation of the UPnP protocol.
1.6. Prerequisites

UPnP AV

The UPnP Audio and Video (UPnP AV) architecture contains additional requirements for UPnP devices and control points that are used in a media oriented environment. It defines the general interaction between control points and devices.

The architecture defines two types of special devices: the media server and the media renderer. The media server is used to locate content that is available via the network. These devices can be in the form of TV tuners, CD players, PCs, etc. The media renderer is used to render content obtained from the network [4]. Some examples of media renderers are TVs, stereos, speakers, etc.

The architecture also specifies a special UPnP AV control point whose purpose is to coordinate the functionality of the media server and the media renderer. It is typically used for browsing content and deciding on which content to render. The control point often features a graphical user interface.

In standard UPnP, only control points communicate with devices. The UPnP AV architecture introduces a protocol that can be used for communication between devices, without the involvement of the control point. This is supposed to be used for transferring content from the media server to the media renderer.

\footnote{display and/or listen to}
Chapter 2

Problem definition

2.1 Vision

A new type of media device has appeared on the market. They are called media centers or media extenders and come in a wide variety of different forms. One thing they all have in common, though, is that they are used for reaching digital media that is present somewhere else than in the actual device. These devices are often seen as a complement to more regular devices, like TV receivers, DVD players, etc.

Zenterio thinks that instead of having two separate devices, one for television and one for other types of digital media, it should be possible to combine these two systems. This could be accomplished by developing a completely new platform incorporating both sides. Another, more tempting idea, would be to take an existing platform; either television or media center and extend it with the missing functionality. Zenterio’s obvious choice would be to extend a familiar digital TV platform. They therefore have a vision where one of their digital TV platforms is extended with functionality so that it also can be used as a media center device.
2.2 Constraints

Zenterio have two fundamental constraints on how their vision should be realized:

- the prototype should be based on the STMicroelectronics MB442 reference board; and
- the prototype should use UPnP technology.

2.3 Vision breakdown

In order to define the problem that the vision and constraints give rise to an investigation needs to be done. The purpose of the investigation is to identify what is missing in the current software. By analyzing the informal vision parallel with having a continuous discussion with Zenterio the following missing features are found:

- the lack of network support in the software
- the lack of UPnP support
- the lack of media center capability
- the lack of rendering support

Because of the intimate relationship between UPnP and the IP protocol, it is essential to have an IP software stack. With an IP stack in place, there is still no implementation of the protocols that constitutes UPnP. This is an essential part.

The rendering of content is also an issue. The formats used in a home environment are in some ways different from the ones used in digital television. The relative low performing hardware platform further limits the possibility to decode high-resolution formats.

Summarized, to be able to realize the vision a number of technologies need to be implemented. To delimit the project the possibility of adapting
existing software, rather than developing new one, exists. The project time constraint also allows a narrowing of the list of formats to render. The next section in this chapter tries to define the project or problem.

2.4 Problem statement

The problem is to build a prototype that realizes the vision described in Section 2.1 with the constraints defined in Section 2.2. The remainder of this section defines the problem as nine functional and non-functional requirements. These requirements are a more formal description of the conclusions made in the previous section.

2.4.1 Functional requirements

The prototype should:

- be compatible with the UPnP Device Architecture (1)
- implement a UPnP AV control point (2)
- use the HTTP protocol for content transfer (3)
- be able to render the JPEG image format (4)
- be able to render the MP3 (MPEG-1 Layer III) audio format (5)
- be able to render the MPEG-2 video format (6)

2.4.2 Performance requirements

The following characteristics should apply to the prototype:

- have an average network throughput of 10 Mib/s (7)
- have a maximum of 1 MiB static memory consumption (8)
- use less than 10% of CPU time, on average, under normal usage (9)
2.4. Problem statement
Chapter 3

Theory

3.1 UPnP

The core of UPnP is defined in the UPnP Device Architecture [5] (UDA). It defines the participating devices and the protocols used for communication between these.

Two general classifications of devices are defined by the UDA:

**Devices:** A UPnP device is a container of services [3]

**Control Points:** A UPnP control point is a controller capable of discovering and controlling other devices [3]

A service is the smallest unit of control in a UPnP network.

The different features or functionalities that constitute the UDA are divided into five different steps: discovery, description, control, eventing and presentation. For operation, these steps use the protocol stack illustrated in Figure 3.1. The remainder of this section describes the structure and operation of these steps.
3.1. UPnP

3.1.1 Step 1: Discovery

Through discovery, control points find interesting devices. In order to accomplish this, the following behavior applies to all UPnP devices:

1. When the device is added to the network it should multicast a number of discovery messages advertising itself and its services.

2. The device should listen to the standard multicast address for discovery messages. Upon receiving a discovery message, the device may send a response to the sender if it or its services match the search criteria.

3. When the device is removed from the network it should send a discovery message to the standard multicast address announcing that its services will no longer be available.
The following should apply to all control points:

1. When the control point is added to the network, it should multicast a discovery message searching for interesting devices and services.

There are two types of discovery messages: Advertisement and Search. The search message is sent by control points and the advertisement message is sent by both devices and control points. The discovery messages are sent using the SSDP protocol.

There are two types of advertisement messages: Device available and Device unavailable. The first corresponds to the message sent when a device is entering a network and the last when leaving a network. Similar, there are two types of search messages: Request and Response. The first is for the actual search and the other for the answer of that search.

3.1.2 Step 2: Description

The advertisement and search response message contains a URL to the UPnP device description document. With this document, a control point can learn more about the capabilities of a device.

The UPnP description for devices is divided into two parts:

Device description: The device description contains

- vendor-specific, manufacturer information (serial number, manufacturer name etc.), and
- a list of each service that is included in the device. Each element in the list has a service type, a name, a URL to the service description, a URL for control (step 3) and a URL for eventing (step 4).

Service description: The service description contains

- a list of the different available actions and their arguments, and
- a list of the different state variables and their data type, range and event characteristics.

The description documents are encoded using XML.
3.1.3 Step 3: Control

In the control step, a control point can ask a service to invoke actions or variable queries. To accomplish this, a control point sends a suitable control message to the control URL for the service. In response, the service returns any results or errors from the action or query.

The Simple Object Access Protocol (SOAP) is used for invoking actions and queries.

3.1.4 Step 4: Eventing

Control points can listen to state changes in devices. This is done through eventing, step 4.

To subscribe to eventing, a control point (subscriber) sends a subscription message to a device (publisher). The publisher signifies changes to state variables by sending event messages. The event messages contain the names of one or more state variables and the current value of those variables. The messages are encoded using XML.

All subscribers are sent all event messages. The subscribers receive event messages for all evented variables.

3.1.5 Step 5: Presentation

As a complement to step 3 (control) and step 4 (eventing), presentation enables another way of controlling and viewing device status. It does so by exposing an HTML-based user interface.

3.2 UPnP AV

Residing on top of the UDA is the UPnP AV Architecture [4]. It defines three type of devices: media server, media renderer and control point.

The goal [4] of the UPnP AV Architecture is to:

- allow arbitrary protocols and content formats,
• enable content to be transferred between devices without intervention of control points, and

• allow devices with a wide variety of system resources.

A typical example of the interaction between the different devices can be seen in Figure 3.2. The control point is invoking the media server and the media renderer to setup the content transfer. It allows the media renderer to stream content from the media server “out-of-band”, i.e. without the control point being involved.

![Figure 3.2: UPnP AV Device Interaction Model](image)

### 3.2.1 Media Server

A media server’s main objective is to let control points browse and search for content that are available to render. It contains three services:

**Content directory:** This service let the control point enumerate the content provided to the home network, mainly with the `Browse()` action. It provides detailed information\(^1\) about each item that can be received. Information about supported transfer protocols and data formats (used with transcoding) that is available for the content item.

\(^1\)meta-data such as title, artist and size
That information is used to determine if an available media renderer is capable of rendering the content item [6].

**Connection manager** The connection manager service is used to control the connections associated with a particular device. It also provides the `GetProtocolInfo()` action to enumerate supported protocols and formats [7].

**AVTransport** This optional service is used to control the flow of associated content [8].

### 3.2.2 Media Renderer

The task of the media renderer is to render content obtained through the network.

**Rendering control:** This service provides a set of actions that allows a control point to change the way content is rendered, e.g. volume, contrast and brightness.

**Connection manager:** As described in Section 3.2.1.

**AVTransport:** As described in Section 3.2.1.

### 3.2.3 Control Point

The control point coordinates the functionality of the media server and media renderer. An example on how the control point interacts with media servers and media renders is described below:

1. Discover audio and video devices
2. Locate desired content using the `Browse()` action
3. Get renderer’s supported protocols and formats using the `GetProtocolInfo()` action

\(^2\text{display and/or play}\)
4. Match protocols/formats

5. **Configure the server and renderer** to be prepared for an incoming transfer

6. **Start content transfer**

7. **Cleanup server and renderer** Close the connection

### 3.3 SSDP

The Simple Service Discovery Protocol (SSDP) provides a technique for network clients to discover network services. This is accomplished in two ways; multicast discovery and server based notification [9].

The main SSDP requests are `ssdp:discover`, `ssdp:alive` and `ssdp:byebye`.

#### 3.3.1 SSDP Discovery Requests

As described above, SSDP is a mechanism for clients to find network services. It multicasts a search request (`ssdp:discover`) for services that match a certain criterion. Services that match this criterion may respond to the IP address where the `ssdp:discover` request came from. The multicast is done using the HTTP Multicast (HTTPMU).

#### 3.3.2 SSDP Presence Announcements

A service declares its presence on the network by multicasting a message (`ssdp:alive`) notifying its existence.

When a service decide to disappear from the network it multicasts a message (`ssdp:byebye`) that it is leaving the network.

The multicast is done using the HTTP Multicast (HTTPMU).
3.4 SOAP

Simple Object Access Protocol (SOAP) is used to exchange structured information\(^3\) between network services. It is typically used on top of HTTP and define how services can send control messages and how to return result and error response messages. [10]

3.5 ZiDS

The Zenterio integrated DVB System, ZiDS for short, is a bundled software solution for digital TV receiver hardware.

This section intends to give an overview of the ZiDS architecture and some more detailed explanations of parts that are important for this project.

3.5.1 Architecture

The ZiDS solution stack architecture can be described with the layer model in Figure 3.3.

The major part of ZiDS is developed in C++ using an object-oriented methodology.

\(^3\)in XML format
The remainder of this section gives a bottom-up description of each layer in the ZiDS architecture.

**Hardware platform**

See Section 1.6.1 and 3.6.1.

**Operating System**

ZiDS tries to minimize its dependency to the operating system (and hardware platform) by introducing an abstraction layer between the platform layer and the operating system API. The actual operating system details therefore depend on which system one chooses to use. Traditionally ZiDS has been running on hardware developed by STMicroelectronics. STMicroelectronics supply an operating system called ST OS21. This system runs unmodified on variety of hardware platforms from STMicroelectronics. Because of this ZiDS is often runned on top of ST OS21. Section 3.5.5 gives a brief description of OS21.

**Platform abstraction layer**

As described above the platform abstraction layer (PAL) is used to separate ZiDS from the operating system. The parts that depend on the operating system and hardware reside in the platform layer. Therefore, the actual implementation of that layer is done here.

**Platform layer**

The platform layer is the foundation of ZiDS. It consists of modules that handles the tuner, enables audio and video output, controls subtitling, etc. It also contains libraries for graphical user interfaces (GUI) and thread support. As some of these modules are platform dependent the layer only defines the interfaces, the implementation resides in the abstraction layer below.
An eventing system is used inside and above the platform layer. This system provides an asynchronous way of communication between modules in the system.

**Application layer**

The application layer contains software that is used by the user. Some applications that are typically placed here are zapper\(^4\), electronic program guide and user settings applications.

### 3.5.2 Tasks

A module in the platform layer provides ZiDS task\(^5\) management system. It supports creation, management and synchronization of tasks. The scheduling is done by a prioritized pre-emptive scheduler that uses round-robin for equally prioritized tasks.

**Synchronization**

To avoid inconsistency when doing concurrent programming, ZiDS has two synchronization mechanisms, namely the *semaphore* \(^[11]\) and *lock* constructions.

### 3.5.3 HalfPipe

The HalfPipe is a user interface framework developed by Zenterio. It is used for navigation, typically in a menu system. The idea is to divide the menu into different logical halfpipes and only present one of them at a time. Figure 3.4 shows an example on how navigation is done in HalfPipe. The circles represent the different elements that the user can select. Initially element 3 is selected. When the user presses the right arrow on the remote control, the HalfPipe rotates one step to the left and element 4 gets selected.

\(^4\) a software that changes between TV channels  
\(^5\) thread
Analogous, when the user presses the left arrow the HalfPipe rotates to the right and element 3 gets selected.

There are two types of elements in a HalfPipe: Folders and Content. A folder can contain other folders and content. A Content is an entity, e.g., an image or a video. Elements have an identifier describing their type. A specific folder controller, called the folder populator, handles folders with a certain type.

3.5.4 ControlIT

ControlIT is a user interface for controlling the playback of media. The concept is to have three different states, e.g., play, pause and stop, where one of them is always selected and the other two is one-step away. A simple scenario of how to use the concept is depicted in Figure 3.5. Assume that the initial state is 2. When the user presses the up arrow, the state changes from 2 to 3. If the user then presses the down arrow, the state goes back to 2. Note that there is always one key stroke from every state.
3.5.5 OS21

OS21 is a proprietary real-time operating system from STMicroelectronics. The system is often seen in digital TV receiver solutions from STMicroelectronics.

3.5.6 OSPlus

OSPlus is an extension package for OS21 that adds support for additional devices and file systems. It also contains an optional TCP/IP stack provided by NexGen.

3.6 Hardware platform

This section briefly introduces the platform that is used to implement the prototype.

3.6.1 The MB442 reference board

The STMicroelectronics MB442 is a developer board or reference board for the STx710x platform. It is used by customers to evaluate the platform and assist in porting and developing software for the platform. The reference board’s main components are listed below.

- STx710x processor
- 128 MiB system memory
- 64 MiB video memory
- 8 MiB flash memory

It is also equipped with two Ethernet connectors (connected to two Ethernet controllers), on board SATA, USB, ATAPI and RS232 interfaces. The board comes with two NIM ports that connect two optional tuners.
The STx7109 processor uses a 266 MHz ST40 CPU as main controller and two 400 MHz ST231 CPUs for audio and video decoding. The ST40 CPU is based on the SH-4 architecture, first introduced for the Sega Dreamcast by Hitachi. It features a superscalar 2-way instruction execution pipeline combined with a parallel vector floating point unit.

### 3.7 Architectural patterns

This section describes some architectural styles that are used or referenced to in the project or project description.

#### 3.7.1 Event-driven architecture

The main idea with the event-driven architecture (EDA) is that components should be able to signify a change in state to other interested (or listing) components. This should be done without coupling the actual components.

To accomplish this we have a number of fundamental entities: The participating components - the listeners and the generator, the actual event and the channel upon which this is sent.

**Event:** An event can be defined as a change in state, inside or outside a component. It can also be seen as the actual instance or occurrence of a change in state.

**Channel:** The channel is what connects the participants with each other.

**Generator:** The event generator is the component that notifies the interested participants that a state change has happened in the component. The component does this by sending an event on the channel.

**Listener:** Listens and acts on events heard on the channel.

A component, interested in events that occur in another component, registers itself as a listener to that component. This can be seen as connecting the two components to a common channel. The listener listens on the channel and the generator notifies that something has happened by sending an event on the channel.
3.7.2 Layering

The main purpose with the layered architecture is to encapsulate different key areas in different layers. A layer is only allowed to communicate with the layer closest over and under.

![Layer Diagram](image)

Figure 3.6: Layer

3.7.3 Client-Server

The client-server architecture divides the application into two distinct systems that communicate through a computer network, as illustrated in Figure 3.7. The general philosophy is to distribute functionality among different servers and make them available for the clients.

**Client:** A *client* is invoking the server and waits for replies.

**Server:** A *server* waits for request from the client and replies.
Figure 3.7: Client-server
3.7. Architectural patterns
Chapter 4

Architecture

4.1 System architecture

The starting point of any development should be the architectural model of a system [12]. An architectural model is an artifact that captures the design decisions that make up a system’s architecture [13]. The architecture for a system is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [13].

4.2 Design goals

Besides the obvious, to solve the functional requirements, the architecture needs to fulfill the performance requirements.

The prototype is a consumer product because of this user interaction is prioritized. In systems where synchronous function calls are heavily employed, resource blockage often occurs. This is especially bad on embedded systems that lack proper resource management. These systems usually suffer from slow responsiveness. The major reason for this is that the CPU stalls on I/O operations. This is a highly unwanted behavior.
In embedded systems, the resources are scarce. Therefore, all modules in the system need to take a collective responsibility for retaining the available system resources.

As time goes, the system will need to be modified to meet new requirements and cope with changes in the environment. This need to be taken into consideration so that further improvements can be done without any major changes to the system.

These considerations can be summarized into three key design goals:

**High responsiveness:** To achieve high responsiveness the system needs a high level of asynchronism. This implies the use of an event-driven architecture (see Section 3.7.1).

**Resource constraint:** Extra concern needs to be taken when allocating resources to the different modules in the system.

**Maintainability:** Prefer solutions with high coherence and loose coupling.

### 4.3 Proposed architecture

The system can be seen as a single UPnP device that includes both control point and media renderer functionality. This device type is defined as a “Render Combo Device” in the UPnP AV architecture [4]. The UPnP AV architecture distinguishes between two types of combo devices, one using isochronous-push and one using asynchronous-push for content transfer. The prototype is required to use the HTTP protocol for content transfer (see Section 2.4). Hence, it can be classified as a “Render Combo Device using Asynchronous-Push” [4].

Due to the apparent difference between control point and media renderer functionality the system is divided into two separate components. These components are divided into several modules that are placed into the three layers of the ZiDS architecture. The communication between the modules is done in a request-response manner. Modules in a layer can directly call functions found in the layer directly below itself. To communicate with the layer above an event must be sent. In short, function calls down and events go up. See Figure 4.1 for a visual explanation.
A render combo device is used in collaboration with a media server device. The media server is not considered as being part of the actual architecture. Only the interface to it is of interest. In Figure 4.2, the system is depicted. In the sequence diagram in Figure 4.3 an example of the communication between the different components are depicted.
4.3. Proposed architecture

The communication between the render combo device and the media server device is defined by the different UPnP protocols described in the UPnP device and AV architectures [4][5]. These are described in Chapter 3. Besides the UPnP specific protocols, the HTTP protocol is used for transferring content between the media server device and the render combo device.

4.3.1 System interface

The communication between the render combo device and the media server device is defined by the different UPnP protocols described in the UPnP device and AV architectures [4][5]. These are described in Chapter 3. Besides the UPnP specific protocols, the HTTP protocol is used for transferring content between the media server device and the render combo device.

4.3.2 Control Point

The Control Point component is responsible for the UPnP specific parts of the system, i.e. the ability to behave and communicate as the UPnP standard prescribes. It is also responsible for the communication with media servers, in particular with content directory services, as described in Section 3.2.3. These two responsibilities divide the component into two integral parts; one that provides an implementation of the UDA and one that provides the ability to communicate with a content directory service, as defined in the UPnP AV Architecture. The control point provides the user interface that let the user interact with the content service.
The foundation of the Control Point component is the ability to conform with the UPnP Device Architecture. This core UPnP functionality is encapsulated in a module called \textit{UPnP}, placed in the PAL.

Looking closer at the part responsible for content directory communication, three important mechanisms are identified:

\textbf{Browsing content directories:} Browsing a content directory is done by invoking a \textit{Browse()} action request and handling the response of that action. The \textit{Content Browser} module provides this functionality.

\textbf{Parsing XML:} The Content Browser module need to parse XML documents and transform the content information to an internal representation. This parsing functionality is placed in a module called \textit{XML}.

\textbf{Storing content information:} The state of the content directory service is considered to change very seldom. To avoid unnecessary network traffic a content directory is cached or stored in a local database. The \textit{Database} module provides this local storage.

These three modules are placed in the platform layer because they are platform independent and do not contain any direct user interaction. The user interaction is instead encapsulated in the \textit{Navigator} module, placed in the application layer.

Figure 4.4 gives an overview of the component.

![Diagram](image)

Figure 4.4: The Control Point component
Module interface

The Navigator provides a graphical interface that enables the user to navigate through a content structure. The user can open, close and select different content. To the Navigator module, the Content Browser module provides the UPnP `Browse()` and `Discover()` functions. These are used for finding and browsing media services. Both the Content Browser module and the Navigator module use the database module. Its interface consists of add, delete and search functions. At last, the UPnP module provides an interface to the different UPnP devices that may exist on the network. As described in Section 3.1 this interface uses XML encoded documents extensively. The XML module implements an interface for parsing these documents.

4.3.3 Media Renderer

The purpose of the Media Renderer component is to render content, i.e. view images, and play audio and video.

Most content are encoded using some format, e.g. JPEG, MPEG, etc. In order to be able to render such content it needs to be decoded to some raw format. This can be done either by using a software decoder or a hardware decoder. The essential part in the Media Renderer component is to achieve a sufficient flow of data between the content source and these decoding facilities.

The content data are fetched from a media server using an arbitrary protocol. UPnP messages are sent on top of the HTTP protocol; because of this, content data are often transferred using this protocol. The `HTTP` module is responsible for receiving data from the media server. Data received by the HTTP module need to be moved, injected, to the appropriate decoding facility. The `Injector` module is responsible for controlling this flow of data between the source and the decoder. These two modules are placed in the PAL.

To control the decoding, e.g. play, pause and stop, the `Playback` module is introduced in the platform layer.

The `Viewer` module is responsible for providing a user interface, for
which the user can play, pause and stop.

Figure 4.5 gives an overview of the component.

**Figure 4.5: The Media Renderer component**

**Module interface**

The Viewer module interface can be used for stopping, playing and pausing audio and video content. It is also used when viewing images. The playback module provides a hardware independent interface to the decoding facilities. It is used directly by the Viewer module. From the Viewer module calls go directly down to the Injector module. Its interface provides the connection between the content source and the hardware. The actual content data fetching is triggered by using the HTTP module interface. This is done using the HTTP GET function.
4.3. Proposed architecture
Chapter 5

Implementation

5.1 Control point

The most prominent part of the control point component is the UPnP layer. This layer is supposed to provide the basic UPnP functionality described in the UPnP Device Architecture. One of the key decisions to make is whether to choose to implement this or use an existing software package. Doing an own implementation has some advantages; no licensing issues, software flexibility and a platform optimized solution. The downside is that the software would need to be maintained and kept up-to-date with newer versions of the UPnP protocol.

When looking at available solutions, three especially important criteria need to be met. The solution would need to be compliant with the UPnP Device Architecture specification, use a free, non-viral license and without major changes be portable to the ZiDS platform.

5.1.1 UPnP

The UPnP layer is built using the Portable SDK for UPnP Devices (libupnp), an open source library that provides an API for building UPnP control
points and devices. The library matches well with the stipulated criteria; it complies with the UDA[5], it is using a free license \(^1\) and it is written in standard C.

The library is said to focus on being portable [14], but this only extends to being compatible with Windows and different UNIX derivatives. The main problem that prevents libupnp from being able to run on the ZiDS platform is that it uses POSIX threads for creation, management and synchronization of threads. To remedy this, a POSIX threads adapter is created.

**POSIX threads adapter**

The idea behind the adapter is to implement a POSIX threads interface that translates calls to its interface into calls to the ZiDS task system. This is depicted in Figure 5.1.

![Figure 5.1: POSIX adapter](image)

An implementation of this is quite straightforward; it consists of mapping one function call to another. Unfortunately, ZiDS and POSIX threads use two different programming paradigms. ZiDS is object oriented and the POSIX threads use a procedural C interface. This is a problem because the adapter needs to pass around C++ objects in a C environment. The main issue is that C functions cannot take C++ declared objects as parameters.

\(^1\)BSD license
One way of solving this is by declaring a container structure in C that contains a void declared pointer that references a memory address where a C++ object resides. This makes it possible to pass around references to an object in a C environment. A demonstration of this can be seen in Code 5.1 and Code 5.2. It shows the function `pthread_create` that is used for creating new threads. The container structure, `ThreadContainer`, allows a C function call to `pthread_create` implicitly using a (C++) `Thread` object as a parameter. This container method is used in the implementation.

**Code 5.1** posix_adapter.h

```c
struct ThreadContainer { void *p; }

typedef struct ThreadContainer pthread_t;

#ifdef __cplusplus

class Thread : public os::Task {
    public:
        Thread(void *(*func_ptr)(void *));
    
#endif // __cplusplus

extern int pthread_create(pthread_t *thread,
    void *(*start_routine)(void *), ...);
```
The implementation process is divided into three parts:

1. Declaring the POSIX threads API
2. Creating mapping classes
3. Implement 1 using 2

At first, the POSIX threads API is declared. This “dummy” interface is needed for linking the program (libupnp). Remember, the program still thinks it is using POSIX threads. Next, the POSIX interface is implemented using the ZiDS task system. This is achieved by using objects constructed from a set of mapping classes. These classes are used to map functionality between POSIX threads and ZiDS tasks. Most of these classes contain a direct mapping, but some functionality in POSIX threads has no equivalent in ZiDS. This is no major issue, because there is enough functionality in ZiDS to create composite mapping classes with sufficient functionality, e.g. a condition class can be constructed by using semaphores and locks, as seen in Code 5.3.
void Condition::Wait(Mutex *mutex)
{
    // Release mutex lock
    mutex->Unlock();

    // One more waiting task
    waitingTasks++;

    // Semaphore wait
    semaphore.Wait();

    // Acquire lock
    mutex->Lock();
}

// POSIX API functions
int pthread_cond_wait(pthread_cond_t *cond,
                        pthread_mutex_t *mutex)
{
    Condition *cond_temp = static_cast<Condition*>(cond->p);
    Mutex *mutex_temp = static_cast<Mutex*>(mutex->p);

    cond_temp->Wait(mutex_temp);

    return 0;
}
5.1.2 Database

The database module consists of three parts: an interface, transaction logic and a data structure.

The database interface will be used for adding and listing data received from different content directory services. The structure of this data resembles the one seen in file systems. Because of this, the interface is designed to be used in a way similar to a typical command shell prompt.

Primarily two different modules of the system will use the database, the Content Browser and Navigator module. The Content Browser uses the database for synchronizing the local instance of the content tree with the content directory service. The Navigator module uses the database for displaying the local copy to the user.

To accomplish this a session management object is used. This object is used for saving the current working directory in the database and for database interaction. The session objects are also responsible for ensuring consistency in the database. In Figure 5.2 the session object constitutes of the AbstractDBSession and the implementation ConcreteDBSession.

The Database class does the transition logic. It uses the singleton pattern [15] and all requests to it are done by the session objects.

Figure 5.2: The database module
A special purpose tree structure, a k-ary tree, is used for storing service and content data. The structure looks conceptually much the same as a file system tree view. The root node represents the local network and its children represent the available content directory services. Each service node has a sub-tree that contains the available content. The tree is implemented using a leftmost child-right sibling representation, see [16]. In Figure 5.3 an example of the tree structure is depicted.

Figure 5.3: The tree data structure

5.1.3 Content Browser

The Content Browser module should implement the Browse() and Discover() functions. To implement these functions the module uses the underlying UPnP library (libupnp). Because the UPnP library eventually will communicate through the network, it is important that the calls to the library do not stall the CPU when waiting for response. Therefore, the communication with the UPnP module is done in an event-response manner, i.e. a request is done using a normal function call and the response is triggered through an event callback. Event callbacks that are associated with the Browse action are handled by the ControlEventHandler(). The DiscoveryEventHandler() handles event callbacks belonging to the dis-
cover request.

From an UPnP perspective, the component is a single control point in the network and therefore all UPnP calls from other modules needs to be coordinated. To achieve this coordination the Singleton design pattern [15] is applied to the module.

In Figure 5.4 the Content Browser module is described in a class diagram.

![Class Diagram](image)

**Figure 5.4: Content Browser**

**Discover()**

When the Discover() function is called it does an asynchronous device search using a function provided by libupnp.

Every device response is handled by the DiscoveryEventHandler(). The DiscoveryEventHandler() starts by retrieving the device description document from the answering device. Using this document the function then creates a service object representing a particular service in the device. This object is inserted at the root level directory in the database. The function ends with sending an event to Navigator module, telling it to synchronize itself with the database.
Browse()

When the `Browse()` function is called it triggers an asynchronous browse action directed to a specified service. The facilities for sending actions are located in `libupnp`.

The response of this action is handled by the `ControlEventHandler()`. The response contains a document describing a subset of a content directory tree. Using this document the function then creates and adds objects corresponding to the subset tree into the database. The function ends with sending an event to Navigator module, telling it to synchronize itself with the database.

5.1.4 XML

The UPnP architecture uses XML extensively. All information shared between devices is sent through XML encoded documents. The XML module is used to parse this information. Put in other words, the XML module should find certain elements in an XML document and converts these to the representation used in the control point component. A typical usage of this module is when the Content Browser module wants to create a content object for each element in an XML document describing, for instance, a content directory of some media service.

To accomplish this behavior a special class, using the Iterator pattern[15], is created. The provided iterator is used to sequentially create objects of certain elements in an XML document. The class uses an existing library called Linux DOM2 XML Parser (`IXML`), which contains the primitive operations that are used for manipulating XML structures. Figure 5.5 shows an UML diagram of this class.
5.1.5 Navigator

The Navigator component uses the HalfPipe framework. A special purpose folder populator for content directory browsing is implemented. It is responsible for handling user interaction and synchronization with the database. The folder populator is briefly described in Section 3.5.3.

When the UPnP folder populator is created, it initializes the Content Browser module and tells it to start the discovery step.

There are two types of cases that the folder populator handles:

User opens a folder: If the folder has not been synchronized with the database a Browse request is sent to the Content Browser module.

User idles in a folder: The populator waits for an event that indicates the database has new content. When that event is received the populator synchronizes the user interface with the database.
In Figure 5.6, an example of the interaction between the Navigator module and the other modules in the Content Browser component is depicted.

![Diagram](image.png)

**Figure 5.6: Interaction between the modules in the Content Browser component**

### 5.2 Media Renderer

The Media Renderer is divided in three different stacks, one for each content type: image, audio and video (see Figure 5.7). The idea behind this is that the different types of content are used in different ways.
5.2. Media Renderer

The modules in the top layer provide the user interface. The middle layer contains the functionality that is used to control the different decoder facilities. In the bottom layer, the different decoders are located. The main problem is the implementation of the functionality to decode various formats. With the decoding functionality in place, the other is merely a question of providing an abstraction from the platform specific implementation to the user interface.

5.2.1 HTTP

The HTTP module implements the HTTP GET data receive part of the HTTP communication protocol. It is constructed using two parts, one that handles the connection, i.e. create the socket, and one the other receives data from the created socket.

This functionality is available in the libupnp package.

5.2.2 Injector

The Injector module consists of three parts, one for each content type.

The ImageInjector implements support for decoding JPEG images by
injected data to the libjpeg software decoder.

The AudioInjector and the VideoInjector are both implemented in a similar way and therefore the technical details are described together. They both inject data into on-chip decoding facilities. The difference lies in the initializing of the different codec facilities. They use the concept of two independent tasks, one for fetching data over HTTP (the producer) and the other for data injection to the decoder (the consumer). The producer makes data available to the consumer using a shared memory space. The naive solution is to have simple buffer with some sort of mutual exclusion, e.g. by using a semaphore. A more elegant solution is to have a circular buffer, which requires no locking mechanism on the memory space.

Circular buffer

The idea of the circular buffer is that there is a common memory space of a given size with independent pointers that references places in the common memory space. The buffer is called circular because when a pointer reaches the top address of the memory space it is moved to the base address. In the typical case, there are two pointers, a read pointer (consumer) and a write pointer (producer), as seen in Figure 5.8.

![Circular buffer example](image_url)

Figure 5.8: Circular buffer example

In the implementation there is an additional variable representing the free space in the circular buffer. A single semaphore is used to synchronize that variable.
5.2.3  Playback

The Playback module implements an abstraction to the Injector module. This abstraction creates a common interface to the decoding facilities.

The Singleton design pattern [15] is used because there is only one decoder available or at least only one stream decoded at the time.

JpegDecoder

The JpegDecoder encapsulate the libjpeg library functions and provide the possibility to decode an image in the JPEG format.

UpnpAudio and UpnpVideo

The UpnpAudio and the UpnpVideo module are constructed using an abstract class and a platform specific class for implementation.

5.2.4  Viewer

The Viewer module provides functionality for rendering control. The Audio and Video Viewer use the ControlIT concept for playback control. The Image Viewer has only the functionality to display an arbitrary image that is coded in a supported format.
Chapter 6

Testing

6.1 Method

In order to verify that the prototype fulfills the requirements and thus solves the problem a series of tests was performed. The tests do not intend to give a complete coverage, but instead confirm using objective evidence that the specified requirements are fulfilled.

6.2 Module test

The module tests are performed using a type of black box testing. In some parts, the tests resemble the idea of smoke testing, i.e. verifying basic functionality for each feature [17].

6.2.1 UPnP

The UPnP module consists of libupnp and the POSIX threads adapter. libupnp is considered tested by its authors; therefore, no test on that particular part is performed.
6.2. Module test

Goal
The goal of the test is to verify that the POSIX threads adapter is able to provide a sufficient subset of the POSIX threads interface. Sufficient subset is defined as the functionality used by libupnp.

Method
The module is tested by using the adapter with libupnp in debug mode and check for abnormal behavior and crashes.

Result
Three months of extensive usage of the UPnP module have not revealed any defects. The conclusion is that the POSIX adapter works sufficiently well.

6.2.2 XML
The XML module consists of two parts: the IXML library and the special iterator class NodeXML. IXML is not tested because it is considered tested by its authors.

Goal
The goal of the test is to confirm that NodeXML can produce objects that correspond to the information in the input XML documents.

Method
The test is done by creating a test program that generates an XML document, describing a content directory, and using this document as the input to the NodeXML iterator. The output objects of the iterator are verified against the input XML document. The test confirms that:

- The correct number of objects is created
• Each created object is of the correct type
• Each created object contains the correct information

Result
The result is presented in Appendix A.

6.2.3 Database
The database module relies heavily on the tree data structure.

Goal
The goal of the database test is to verify that the internal tree data structure behaves as defined.

Method
In order to test the functionality of the tree data structure a simple test application is written to perform the three basic operations that the tree have. The test program uses a pre-defined sequence of operations with an expected outcome.

6.2.4 Result
All created trees conform with the pre-defined sequence of operations.

6.2.5 Content Browser
Goal
The purpose of this test is to show certain key features of the module.
6.2. Module test

Method

To test the Content Browser module three test cases are composed. The module should (in collaboration with the UPnP module) be able to:

- find available UPnP devices upon connection to a network
- find a new UPnP devices that connects to the network
- send a browse action to one available UPnP service in the network
- to receive the response of the above action

The test is performed in an isolated network with the following devices:

- three PCs running UPnP media servers
- the prototype

Each device on the network is assigned a unique IP address. The PCs are running different UPnP software \(^1\). On top of the Content Browser module is a simple test program that can trigger the Discover() request and Browse() action.

Three tests are performed:

**Test case 1:**

**Step 1:** Connect the UPnP devices to the network

**Step 2:** Connect the prototype to the network

**Test case 2:**

**Step 1:** Connect the prototype to the network

**Step 2:** Connect the UPnP devices to the network

**Test case 3:**

\(^1\)Google Desktop, TVersity and MediaTomb
Step 1: Send a browse action to all devices
Step 2: Receive the action responses

The expected results of these tests are:

Test case 1: The Discover() request find the three connected UPnP devices.

Test case 2: The module discovers the three connected UPnP devices.

Test case 3: The module receives an action response from all devices

Result
The module succeeded on all three test cases. A detailed test protocol is presented in Appendix A.

6.2.6 Navigator

Goal
The goal with the test is to verify that the Navigator module enables the user to navigate through a content structure.

Method
To verify that the user can navigate through a content structure the test need to verify that the module can:

- synchronize itself with database
- invoke the browse action when opening a folder

In order to accomplish this a user navigates through a pre-defined content structure.
Result

The user can navigate through a content structure. This implies that the module can synchronize itself with the database and invoke the browse action.

6.2.7 HTTP

The HTTP module is part of libupnp, which is considered tested by its authors; therefore, no test is performed on this module.

6.2.8 Injector

Goal

The purpose with the Injector module test is to verify that the module can inject content to the different decoders.

Method

The Injector module is troublesome to test. It has dependency on both the actual hardware and the data fetching mechanism. One naive way to confirm the required functionality is to take a small subset of available content and render it using the Injector module and a reference renderer.

By manually comparing the result of the Injector module with the result from the reference, it is possible to decide whether the module correctly renders the input. This assumes that both the hardware and HTTP module works correctly.

The reference renderer used is the VLC media player [18].

The following test content is used:

- Three JPEG images
- Three MPEG-2 videos
- Three MPEG-1 layer 2 phonograms
A driver program replaces the software that normally would invoke the Injector module. The program uses the module interface with a list of HTTP URLs pointing to the set of test content.

Result

No apparent difference between material rendered by the reference and material rendered using the Injector module can be detected.

6.2.9 Playback

As described in Section 5.2.3 the module does not contain any functionality, it only provides an abstraction to the other modules in the rendering system.

Goal

The goal of the test is to verify that the Playback interface complies with the predescribed behavior.

Method

In order to test this a stub replaces the Injector module. The stub mimics the Injector module interface, but instead of invoking an action, the stub indicates that a call to its interface has been done.

The test is carried out by performing the following function calls to the Playback module:

- Play
- Stop
- Pause
- Resume
6.2. Module test

Result

The Playback module interface complies with the predescribed behavior. The result is available in Appendix A.

6.2.10 Viewer

The Viewer module controls and shows the state of the Playback module to the user.

Goal

The goal of the test is to verify that the Viewer module can call the correct functions of the Playback module.

Method

The test is performed by replacing the Playback module with a stub that mimics Playback modules interface. The stub indicates that a call to its interface has been done.

The test is carried out by triggering the following actions when using the Viewer module:

- Play
- Stop
- Pause
- Resume

Result

The Viewer module is able to call the Playback module correctly. The result is available in Appendix A.
6.3 Performance test

The performance tests are performed to verify that the performance requirements, defined in Section 2.4, are fulfilled.

6.3.1 Network throughput

Goal

The goal is to measure the average throughput using the HTTP protocol.

Method

The test is performed using a test program that receives a set of differently sized data files. The test is performed on a network where the maximum transfer rate is 100 Mib/s.

Result

The result is presented in Figure 6.1 and in Appendix A. The figure shows the transfer rate (Mib/s) for different sized messages. If the message size is larger than 512 kiB, the speed is greater than 10 Mib/s.
6.3. Performance test

6.3.2 Memory consumption

Goal

The goal is to measure the total number of bytes of the running software.

Method

Using an available profiling tool from STMicroelectronics the memory consumption of the project specific components are summarized.

Result

The total static memory consumption is 345152 B.
6.3.3 CPU usage

Goal

The goal is to measure the amount of CPU time the running software consumes.

Method

Using an available profiling tool from STMicroelectronics the CPU time of the project specific components are summarized.

Result

On average 6.83% CPU time is consumed by the project specific tasks.

6.4 Verification

This section verifies that the requirements in Section 2.4 are fulfilled. This is accomplished using the results of the tests presented in this chapter.

Requirement (1) and (2): Fulfilled according to the test in Section 6.2.5.

Requirement (3): The test in Section 6.2.8 uses the HTTP protocol for transferring the set of test content, requirement (3) is therefore considered fulfilled.

Requirement (4), (5) and (6): According to the test in Section 6.2.8 it is possible to render JPEG images, MP3 audio and MPEG-2 video content. The requirements (4), (5) and (6) are considered fulfilled.

Requirement (7): The test in Section 6.3.1 shows that the network throughput is converging towards 17 Mib/s (when the message size is sufficiently large). This throughput is sufficient in order to fulfill requirement (7), which stipulates an average throughput of 10 Mib/s.
Requirement (8): Section 6.3.2 shows static memory consumption of 345152 B, which implies that requirement (8) is fulfilled.

Requirement (9): According to Section 6.3.3 the average consumed CPU time is 6.83 %. This is sufficient in order to fulfill requirement (9).
Chapter 7

Discussion and conclusion

7.1 Implementation

This section discusses the pros and cons of the chosen implementation.

7.1.1 Control Point

UPnP

The decision to use libupnp for the UDA implementation has several advantages. The use of a proven solution has some implications; the software can, in some degree, be considered tested. The authors are also responsible for maintenance, which means that they will be responsible for keeping it up-to-date with further versions of the UDA.

A platform specific solution would not have the overhead that the POSIX threads adapter introduce, i.e. it would have a smaller number function calls. The intended target platform for libupnp is not embedded systems; therefore, it is more general than necessary. The target platform not being embedded systems, also means that the memory footprint is significant larger than what could have been achieved with a platform-optimized solution.
7.1. Implementation

Implementing a platform-optimized solution would be too extensive for the project’s limited period. From a technical viewpoint though, where there are no limiting time constraints, this would be the ideal solution.

Database

The database implementation has two design decisions of importance: the special interface and the chosen data structure.

There are several ways to implement a data structure for a database. The data structure that is used is an ordinary k-ary tree where the nodes lack unique identification. This makes it impossible to organize the nodes to any scheme. In the case where the find operation is used often, the data structure would be horrible, but in the way the database is supposed to be used, it is sufficient.

Another approach is to generate unique identifications for each node and organize these using some scheme, e.g. d-ary heaps, B+ trees, etc.

The idea behind the interface is to provide the programmer with an intuitive way of moving around in the database. In some cases, this interface has proven to be of more annoyance than of use. The database could use another interface, using some query language like SQL or a standard tree interface.

Content Browser

The Content Browser module coordinates all requests to the underlying UPnP module. This design decision makes it easy to change the implementation of UPnP module without affecting other modules.

Navigator

The implementation of the Navigator module uses the HalfPipe framework, and therefore, any major design decisions cannot be made. The implementation has a lot of logic that can be moved to another module dedicated to provide this functionality.
XML

The use of an iterator that creates objects can be considered as the major decision in this module. Another approach is to let another module be responsible for creating these objects.

7.1.2 Media Renderer

The media renderer component is dependent on the hardware platform. The prototype is implemented on one platform, but several design decisions tries to minimize the dependencies to the hardware.

HTTP

The HTTP module implementation uses synchronous calls, which can have some negative implications on the performance. In order to solve this, a solution would be to rewrite it so that it uses asynchronous calls.

Injector

The Injector module is highly coupled with the hardware decoding facilities. This is an inevitable consequence of the fact that the module is the final step before the actual decoder.

Playback

In order to provide a uniform interface that is not dependent on the platform the Playback module is introduced. This relaxes the hardware dependency.

Viewer

The Viewer only takes care of user input and relays it to the Playback module.
7.2 Architecture

The decomposition of render and UPnP functionality is a natural design decision. This makes it possible to use either of these components individually.

Each component has its modules divided into different layers. These layers are used to enforce platform independence. In some occasions it is difficult to find proper functionality for some layers; this sometimes introduces poltergeists [19]. The Playback module can be seen as poltergeists. It could be replaced by defining a unified interface for the Injector module implementations.

7.3 Testing

The tests that are performed on the system are relatively naive. They would not be sufficient when testing a complex system. One of the reasons that explain the bad test coverage is that system testing was not taken into consideration when planning and designing the system.

7.4 Conclusion

The prototype fulfills the list of stipulated requirements. In order to use the prototype in a real environment, with real users, further extensions would need to be done. The applications at the highest level, i.e. the user interfaces, need to be enhanced with additional functionality, but the architectural foundation can be considered complete and robust.
Bibliography


Appendix A

Test results

A.1 XML module

The input, presented in Section A.1.1, of the XML module test is a file describing a UPnP content structure. Using this file the XML module is able to produce objects with the right properties. These objects are listed in Section A.1.2.

A.1.1 Input

```xml
<DIDL-Lite xmlns:dc="http://purl.org/dc/elements/1.1/
xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/
xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/>

<item id="1" parentID="1" restricted="false">
<dc:title>A Thousand Years1</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
```
http://pc/getcontent?contentID="1"
</res>

<item id="2" parentID="1" restricted="false">
<dc:title>A Thousand Years2</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="2"
</res>

<item id="3" parentID="1" restricted="false">
<dc:title>A Thousand Years3</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="3"
</res>

<item id="4" parentID="1" restricted="false">
<dc:title>A Thousand Years4</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="4"
</res>

<item id="5" parentID="1" restricted="false">
<dc:title>A Thousand Years5</dc:title>
<dc:creator>Sting</dc:creator>
Test results

<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="5"
</res>

<item id="6" parentID="1" restricted="false">
<dc:title>A Thousand Years6</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="6"
</res>

<item id="7" parentID="1" restricted="false">
<dc:title>A Thousand Years7</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="7"
</res>

<item id="8" parentID="1" restricted="false">
<dc:title>A Thousand Years8</dc:title>
<dc:creator>Sting</dc:creator>
<upnp:class>
object.item.audioItem.musicTrack
</upnp:class>
<res protocolInfo="http-get:*:audio/x-ms-wma:*">
http://pc/getcontent?contentID="8"
</res>
<item id="9" parentID="1" restricted="false">
  <dc:title>A Thousand Years9</dc:title>
  <dc:creator>Sting</dc:creator>
  <upnp:class)object.item.audioItem.musicTrack</upnp:class>
  <res protocolInfo="http-get:*:audio/x-ms-wma:*">
    http://pc/getcontent?contentID="9"
  </res>
</DIDL-Lite>

## A.1.2 Result

item id = 1
dc:title = A Thousand Years1
res = http://pc/getcontent?contentID="1"

item id = 2
dc:title = A Thousand Years2
res = http://pc/getcontent?contentID="2"

item id = 3
dc:title = A Thousand Years3
res = http://pc/getcontent?contentID="3"

item id = 4
dc:title = A Thousand Years4
res = http://pc/getcontent?contentID="4"

item id = 5
dc:title = A Thousand Years5
res = http://pc/getcontent?contentID="5"
A.2 Content Browser module

This section lists the result of the Content Browser module test. Four types of actions are described:

**Connect**: Indicates that a UPnP device is connected to the network

**Discover**: Indicates that the prototype discovers a UPnP device

**Browse**: Indicates that browse request is sent

**Response**: Indicates that a browse response is received

A.2.1 Test case 1

<table>
<thead>
<tr>
<th>Time(s)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Connect TVersity</td>
</tr>
<tr>
<td>1</td>
<td>Connect Google Desktop</td>
</tr>
</tbody>
</table>
A.3. Playback module

This section lists the result of the Playback module test. The listing show the corresponding stub result for different types of calls to the Playback module interface.

<table>
<thead>
<tr>
<th>Time(s)</th>
<th>Interface call</th>
<th>Stub result</th>
</tr>
</thead>
</table>

## A.4 Viewer module

This section lists the result of the Viewer module test. The listing show the corresponding stub result for different types of actions initiated by the Viewer module.

<table>
<thead>
<tr>
<th>Time(s)</th>
<th>Action</th>
<th>Stub result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Play</td>
<td>Play()</td>
</tr>
<tr>
<td>2</td>
<td>Stop</td>
<td>Stop()</td>
</tr>
<tr>
<td>3</td>
<td>Pause</td>
<td>Pause()</td>
</tr>
<tr>
<td>4</td>
<td>Resume</td>
<td>Resume()</td>
</tr>
<tr>
<td>5</td>
<td>Pause</td>
<td>Pause()</td>
</tr>
<tr>
<td>6</td>
<td>Play</td>
<td>Play()</td>
</tr>
<tr>
<td>7</td>
<td>Resume</td>
<td>Resume()</td>
</tr>
<tr>
<td>8</td>
<td>Stop</td>
<td>Stop()</td>
</tr>
<tr>
<td>9</td>
<td>Stop</td>
<td>Stop()</td>
</tr>
<tr>
<td>10</td>
<td>Play</td>
<td>Play()</td>
</tr>
<tr>
<td>11</td>
<td>Pause</td>
<td>Pause()</td>
</tr>
</tbody>
</table>
A.5 Network performance

This section describes the average transfer rate in relation to messages of different size.

Size: 1024 (B)
15,04106
15,042029
15,070201
15,050761
15,008168
15,084815
15,522318
14,991776
15,04882
15,004308
Average speed: 15,0864256 (kiB/s), 0,1178627 (Mib/s)

Size: 65536 (B)
135,278525
162,898493
162,806148
162,829225
147,450293
162,733413
162,878949
162,999836
147,332465
162,728093
Average speed: 156,993544 (kiB/s), 1,226512063 (Mib/s)

Size: 131072 (B)
294,886028
Test results

326,120694
325,676204
295,066643
325,622946
325,509389
325,768558
293,928399
325,690409
325,509389

Average speed: 316,3778659 (kiB/s), 2,471702077 (Mib/s)

Size: 524288 (B)
919,848973
1302,364001
1302,449188
1184,126863
1303,386979
1301,682908
1302,562788
1180,663135
1301,48439
1302,846874

Average speed: 1240,14161 (kiB/s), 9,688606327 (Mib/s)

Size: 1048576 (B)
1603,684105
2155,144427
2166,104576
2186,700539
2160,913463
2156,739519
2157,226297
1604,330149
1423,415449
2155,766621

Average speed: 1977,002515 (kiB/s), 15,44533214 (Mib/s)
A.5. Network performance

Size: 8388608 (B)
2315,137121
2142,597804
2164,927047
1898,756323
1898,865718
1749,53815
2165,810074
2341,587809
2398,010196
2188,50054
Average speed: 2126,373078 (kiB/s), 16,61228967 (Mib/s)

Size: 67108864 (B)
2223,200465
2267,173106
2184,205098
2195,843461
1589,195597
1998,918625
2336,607178
2180,9406
2280,117763
2232,561856
Average speed: 2148,876375 (kiB/s), 16,78809668 (Mib/s)
Media extenders or digital media receivers are network devices that are used to retrieve digital media files (such as music, pictures, or video) from a media server and play or show them on a TV or home theater system. A technology that is often associated with these devices is the Universal Plug and Play (UPnP) technology. This technology enables network devices to be used without requiring the user to do network configuration on it.

This thesis demonstrates how a device that is normally used for receiving digital television broadcasts can be enhanced to support media extender functionality. The thesis describes the design and implementation of the technologies that are needed to accomplish this functionality. The main topics are centered around how UPnP awareness and media rendering (decoding) are incorporated into the device.
Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oändrande för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera aktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Dario Lentini and Gustav Salenby
Linköping, January 27, 2009