Bachelor's Thesis

Implementing a DLNA-compliant UPnP AV MediaServer with DVB and RTSP/RTP support

Martin Emrich

Submitted on	:	April 28th 2009
Supervisor	:	Manuel Gorius, M.Sc.
1st Reviewer	:	Prof. Dr.-Ing. Thorsten Herfet
2nd Reviewer	:	Prof. Dr.-Ing. Philipp Slusallek

Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Universität des Saarlandes

Postfach 15 11 50, 66041 Saarbrücken*artin Emrich*

UNIVERSITÄT DES SAARLANDES

Lehrstuhl für Nachrichtentechnik

FR Informatik

Prof. Dr. Th. Herfet

Universität des Saarlandes
Campus Saarbrücken
C6 3, 10. OG
66123 Saarbrücken

Telefon (0681) 302-6541
Telefax (0681) 302-6542

www.nt.uni-saarland.de

Bachelor-Arbeit für
Martin Emrich

Implementierung eines DLNA-konformen Streaming-Servers

Heutige Heimnetzwerke sind geprägt von der Konvergenz zwischen Personal Computern, Consumer Elektronik und Mobilgeräten. UPnP und DLNA definieren Richtlinien für die Interoperabilität solcher Geräte und ermöglichen dem Nutzer das komfortable Verteilen digitaler Medien im Netzwerk. Üblicherweise handelt es sich um gespeicherte Inhalte. Es gibt jedoch auch bereits erste Lösungen für die Verteilung von digitalem Fernsehen an UPnP-kompatible Endgeräte.
Die Verwendung von HTTP/TCP in den derzeit vorhandenen Ansätzen bedingt unberechenbare Verzögerungen in der Wiedergabe und lange Umschaltzeiten. Langfristiges Ziel ist es jedoch, die von herkömmlichen Übertragungswegen des digitalen Fernsehens gewohnte Qualitätserfahrung auch über das paketbasierte Netzwerk liefern zu können.
Im Rahmen der vorliegenden Bachelor-Arbeit soll ein unter den Vorgaben von DLNA operierender Streaming-Server für digitales Fernsehen entwickelt werden. Bei der Implementierung sollte besonders auf qualitätserhaltende Maßnahmen im obigen Sinne geachtet werden.

Im Einzelnen sind folgende Aufgaben zu lösen:

- Einführung in die Grundlagen der DLNA-basierten Heimvernetzung
- Konzeption eines DLNA-konformen Streaming-Servers unter Berücksichtigung echtzeitfähiger Transport-Protokolle.
- Implementierung und Demonstration des Lösungsansatzes auf dem Lehrstuhl-Netzwerk.

Arbeitsumgebung:
Die Entwicklung und Implementierung sollte in der Sprache C++ unter Linux stattfinden. Hierzu stehen Rechner mit diversen DVB-Quellen zur Verfügung, die sowohl drahtlos als auch kabelgebunden in ein Heimnetzwerk-Szenario eingebracht werden können. Hilfsmittel zur Validierung der Implementierung des Media-Servers sind vorhanden. Das zu verwendende Transport-Protokoll liegt bereits in Form einer C++-Bibliothek vor.

Betreuer: Gutachter:

Manuel Gorius *Prof. Dr.-Ing. Th. Herfet*

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement under Oath

I confirm under oath that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,..... .....

(Datum / Date) (Unterschrift / Signature)

Acknowledgment

The presented thesis was supported by Intel GmbH in the course of the “Digital Home Compliance Lab”.

Abstract
In the home multimedia network, the distribution of stored content (photos, music, video files) is currently separated from live broadcast TV and radio, even when the latter is transmitted via IP networks. This thesis describes the design and implementation of a prototype media server that publishes both stored content and DVB-received live broadcast to the home network. It adheres to the DLNA device interoperability guidelines for compatibility with many existing devices, and provides transmission of live broadcast to UPnP AV clients via the Real-time Transport Protocol (RTP) as well as HTTP.

Contents

Abbreviations

AHEC Advanced Hybrid Error Correction
BGD Bidirectional Gateway Device
CTT Compliance Test Tool
DHCP Dynamic Host Configuration Protocol
DLNA Digital Living Network Alliance
DMP Digital Media Player
DMS Digital Media Server
DNG Delivery Network Gateway
DUT Device Under Test
DVB Digital Video Broadcasting
DVBSTP DVB SD&S Transport Protocol
EPG Electronic Program Guide
FEC Forward Error Correction
GENA General Event Notification Architecture
HNED Home Network End Device
IGMP Internet Group Management Protocol
IPI Internet Protocol Infrastructure (in DVB context)
IPTV Internet Protocol Television
NPT Native Play Time
RTCP Real Time Control Protocol
RTP Real Time Protocol
RTSP Real Time Streaming Protocol
SDP Session Description Protocol
SSDP Simple Service Discovery Protocol
UCDAM Uniform Client Data Availability Model
UGD Unidirectional Gateway Device
UPnP Universal Plug'n'Play
UUID Universally Unique Identifier

1 Introduction

1.1 Motivation

In the last few years, digital broadcast has progressed very far in replacing legacy analog broadcasting schemes. DVB-based digital TV is currently (with a few exceptions) the standard in Germany. Despite that, the transmission of live broadcast content over IP networks in a standard compliant way is until now only available in a few corner cases[1].

At the same time, the multimedia home network is becoming more and more a reality. New standards such as UPnP AV and DLNA enable multimedia home devices to find each other without any configuration effort by the end user. This way, many people already use their home network to make their collection of photos, videos and music available to media client devices throughout their household. New initiatives such as the DLNA CERTIFIED™logo program ensure the compatibility between devices from different vendors not only from a protocol perspective, but also in terms of content formats.

1.2 Goals

Currently, these two worlds exist mostly separate. The typical home network user has a networked media player device standing next to a DVB receiver box (Even the DLNA admits[10] these “islands”). The goal is to combine these two applications by providing a streaming media server that is built using existing standards and delivers live broadcast received via DVB tuner hardware as well as stored content like music, videos and photos. Of course, for each type of content, the appropriate transport protocols (HTTP, RTSP/RTP unicast or multicast) will be chosen. The server will be compliant to the DLNA device guidelines[9] and thus be compatible with many devices on the market. Future work may include a corresponding media client design, including hardware, operating system configuration and software.

A prototype of a DLNA compliant media server that is capable of streaming live broadcast TV and radio using the RTP protocol will be developed in this thesis. As there is currently no UPnP or DLNA compliant receiver device available that can receive RTP transmissions, a suitable media renderer is being developed in parallel by Jochen Grün[13]. To prove the DLNA compliance, the resulting media server is tested using the DLNA Compliance Test Tool (CTT), and it should of course pass all relevant tests.

[1] As of 2008, only three German ISPs so far transmit the freely available IPTV streams of the public TV stations to their customer’s premises, all only together with their proprietary IPTV subscriptions.

2 Existing frameworks and standards

Several standards organizations as well as individual companies are working on various frameworks to bring broadcast and/or media sharing to the home network.

2.1 DVB-IP

The DVB-IPI[11] (DVB Internet Protocol Infrastructure, recently also referred to as DVB-IPTV[1]) describes a framework to bring DVB into the home network. A Delivery Network Gateway (DNG) connects one or more delivery networks (tuner-based like DVB-S/-C/-T as well as IP-based services) to the IP-based home network. Different segments of the home network are connected by a Home Network Node (HNN), e.g. a wireless access point connecting the cable-based 802.3 and the wireless 802.11 network. On the other end of the home network is a Home Network End Device (HNED) that receives the streams.

A HNED discovers available services via the *Service Discovery & Selection* protocol (SD&S). This can happen either in push mode, where the service provider sends SD&S data via DVBSTP multicast, or in pull mode, where the HNED fetches the current SD&S data via HTTP. Several bootstrap methods are specified, such as via DNS lookups, DHCP configuration options, a IANA-registered multicast channel or a user-provided HTTP URL).

After the discovery, the HNED can fetch the DVB-SI (Service Information) via SD&S as XML documents.

The only container for the media streams is MPEG Transport Stream (TS), transported either via RTP or UDP (ITU H.610). Session management happens either via IGMP (for multicast) or via RTSP (also limited, no trick play operations).

Currently, there are only a few end-user devices clearly compatible with DVB-IPTV[2], although it is possible that some of the proprietary IPTV offerings at least on the German market incorporate portions of DVB-IPI. To confirm that, more reverse engineering work may be required.

The main disadvantage of DVB-IPI is its missing support for media types other than MPEG streams, making it unsuitable for the distribution of locally stored content (photos, music) in the home network. Mechanisms such as Multiprotocol Encapsulation (MPE, which allows transmission of arbitrary data embedded in an MPEG stream) could theoretically be used to extend the capabilities, but they are not part of the DVB-IPI specifications. Its focus lies more on providing the same features (with the addition of Video-on-demand) that the existing DVB transports provide.

[1] Although no reliable source for a naming transition was found, many of the cited sources use the two abbreviations in parallel

[2] http://www.dvb.org/products_registration/dvb_compliant_products/by_specification/product_list/index.xml?csID=51 lists mainly broadcasting/carrier hardware

Nevertheless, a universal device can implement a DVB-IP HNED in addition to other frameworks to gain compatibility with existing DVB-IP services.

2.2 DVB-HN

The DVB Project seems to have actually understood the limitations of DVB-IPI2.1, and they are now setting out to combine the properties of DVB-IPI and DLNA into a new framework called DVB-HN[22] (DVB Home Network). In the essence, DVB-HN takes the DLNA guidelines and makes all these optional parts mandatory that are necessary for broadcast content. They also introduce new device classes:

- HNED (Home Network End Devices): These reuse the existing DLNA device classes (DMS, DMP, DMC), extended to DVB-DMS, DVB-DMP and DVB-DMC, respectively.
- Gateway Devices (GD) link the home network to the broadcast source. A DVB-UGD (Unidirectional Gateway Device) contains a tuner and a DMS, and makes received DVB channels available on the home network. A DVB-BGD (Bidirectional Gateway Device) connects the home network to a service provider via a broadband connection. The latter may also include a DVB-IPI-compliant Delivery Network Gateway to support existing DVB-IPI HNEDs (this can be as simple as bridging the network through directly to the service provider).
- A Remote Device (DVB-RMC) can control devices inside the home network, while not necessarily being inside the home network (connected via VPN). For example be used to schedule a TV recording while being not at home.

DVB-HN also introduces new device capabilities (sets of features that are not bound to a certain device class). The most notable ones are:

- **dvb-sdns** specifies support for the SD&S protocol specified by DVB-IPI, making the device compatible with DVB-IPI DNGs or HNEDs.
- **dvb-cpcm** implements DVB Content Protection and Copy Management compatible DRM.

Of all of the frameworks presented herein, DVB-HN is the one that comes closest to the goals of this project. But again, the framework is quite new and still in draft state, and searching the world wide web for devices implementing it yields no results. If no conflicts arise between the DLNA and DVB-HN specifications, it would be the best course of action for a truly universal device to support both standards.

2.3 Proprietary solutions

As all of the standards mentioned are quite new, many vendors have in the meantime developed their own, proprietary home network systems. While systems like AppleTV[3], Microsoft Win-

[3] http://www.apple.com/de/appletv/

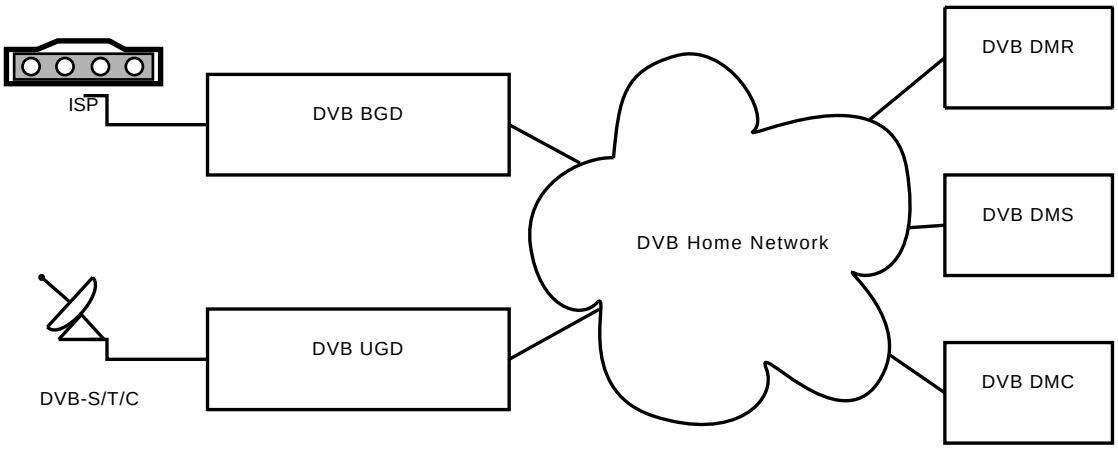


Figure 2.1: DVB-HN architecture diagram

dows Media Center[4] etc. may work very well within their ecosystem, they inevitably lock the user into a world controlled by one single vendor. It is quite likely that the AppleTV appliance will never display DVB-IPTV, just as likely as that the Microsoft Mediaroom-based[5] T-Home Entertain receiver will never play a single song from iTunes. The only way out of this is the consequent promotion and use of open standards.

2.4 VDR

One notable framework, although being a proprietary solution, is the Video Disk Recorder[18] (VDR). It is a Free Software project licensed under the GNU Public License, that in the beginning was designed to turn a otherwise deprecated PC system into a personal video recorder. It required a so-called full featured DVB receiver card (which includes a hardware MPEG decoder and the capability to display an on screen display). Other than that, it has low requirements, a 300MHz CPU, 256MB RAM and of course a hard disk big enough to record TV shows. The main cause for it's widespread usage[6] is probably it's good extensibility via plugins.

Soon, the DVB tuner card vendors introduced cheaper cards ("budget cards") lacking the hardware decoder. Shortly after that, several plugins emerged that simulated the decoding frontend in software. Another widespread plugin is the `streamdev` plugin, which allows a VDR instance to act as a server (offering the DVB hardware on the network) or a client (accessing a vdr-streamdev server on the network). `Streamdev` offers both a proprietary protocol called "VTP" and a standard HTTP streaming server. With these plugins, streaming live broadcast to the home network is possible, but setting it up is definitely too difficult for the average end user.

[4] http://www.microsoft.com/germany/windows/products/winfamily/mediacenter/default.mspx
[5] http://www.microsoft.com/germany/lifestyle/tv_filme/mediaroom.mspx
[6] The German community `http://www.vdr-portal.de` alone has more than 20000 registered users, the big German computer magazine c't offers a live CD distribution `http://www.heise.de/ct/projekte/machmit/ctvdr/`.

Nevertheless, a running VDR instance with the `streamdev` server is one of the possible sources for the streaming media server implemented for this thesis.

2.5 UPnP

The *Universal Plug and Play* architecture[7][17] intends to make networked devices[7] from different vendors interoperable by specifying IP address acquisition, device discovery and description, remote action invocation and event notification. Thus, a UPnP compatible device can find other UPnP devices on the network, call methods provided by them and be notified of events, everything without the user having to configure the device.

UPnP AV[6] defines a collection of UPnP device types to enable networked devices to share media content across the home network. A *Media Server* stores and provides content items, while a *Media Renderer* is capable of receiving and displaying them. A *Control Point* discovers these devices on the network and initiates media transfer (playback). UPnP AV allows many transports, including HTTP, RTSP/RTP, point-to-point links (e.g. IEEE1394) and proprietary connections.

As UPnP it is the framework of choice for this project, let us now look closer at its design and the protocols used:

2.5.1 Devices and services

In UPnP terms, a device can be an UPnP *device* that provides services, or a *control point* that invokes the services on other *devices*. Thus, *device* is written in italics throughout this document whenever the logical entity “device” as specified by the UPnP device architecture is meant.

Each UPnP *device* offers one or more *services*, and may even contain other *devices*. The topmost *device* in this hierarchy is called *root device*, while the others are *embedded devices*. This way, one appliance may provide different, maybe even unrelated features at once. Every *root device* has a Universally Unique Identifier (UUID) that is unique for a single device and does not change across device reboots.

Each *service* provides several *service actions* that can be invoked remotely.

2.5.2 IP address acquisition

Although this component is not handled by `rtpserver` but by the underlying operating system, it shall still be described here.

First, the UPnP network is currently restricted to IPv4; neither IPv6 nor other non-IP protocols are part of the specification. UPnP specifies that upon connecting to the network, each device must first try to acquire an IP address through DHCP. Only if that fails, it must assign itself a link-local IPv4 address[1] (also known as “Zeroconf”).

The entire UPnP network communication uses IP addresses, neither the use of DNS domain names nor of configuration-free resolution schemes such as multicast DNS is required or allowed.

[7] A *device* in UPnP terms is not necessarily a hardware device, but it can also be a piece of software running on a desktop computer

Note that if the device has multiple network interfaces (e.g. 802.3 and 802.11), each interface is handled independently.

2.5.3 Device discovery

UPnP devices find each other by the means of the Simple Service Discovery Protocol (SSDP)[12]. SSDP specifies announcements other devices can listen to as well as an active search for specific devices.

```
NOTIFY * HTTP/1.1
HOST: 239.255.255.250:1900
CACHE-CONTROL: max-age=1800
LOCATION: http://10.20.0.76:10080/volatile/description.xml
NT: urn:schemas-upnp-org:device:MediaServer:1
NTS: ssdp:alive
SERVER: Linux/2.6.27-7-generic, UPnP/1.0, Portable SDK \
        for UPnP devices/1.6.6
X-User-Agent: redsonic
USN: uuid:a6a69884-7a6d-413b-a955-75cba6a8b07d:\
        :urn:schemas-upnp-org:device:MediaServer:1
```

Figure 2.2: SSDP alive message of an UPnP device

Announcement Whenever a *device* or *service* enters or leaves the network, it announces this by a multicast datagram to 239.255.255.250:1900. Figure 2.2 contains an example. A device entering the network sends an Notification Sub Type (`NTS`) header of `ssdp:alive`, a device leaving sends `ssdp:byebye`. The messages contain:

- An expiration time (in the `CACHE-CONTROL` header): To prevent stale entries left in the internal state of devices by devices leaving the network without sending a *byebye* message (e.g. because they are suddenly powered down or disconnected from the network), each SSDP message has a timeout, after which the entity that was announced expires, unless it is renewed by another *alive* message. The timeout is usually set to 1800 seconds (30 minutes).
- A device description URL (in the `LOCATION` header): Here, interested network entities can find an XML document with further information about the *device* or *service*. See also 2.5.4
- A notification type (`NT`) and a unique service name (`USN`) specifying the type of device or service that sent the notification. Their contents depend on the type of notification.

An UPnP root device announcement consists of 3 notifications for the *root device*, 2 for each embedded *device*, and one for each service. The exact makeup of these messages can be found in [7].

Search UPnP *control points* can also actively search the network for specific *device* or service types. They send a search request such as the one in figure 2.3.

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 60
ST: upnp:rootdevice
```

Figure 2.3: SSDP search for a *root device*

The search contains a maximum wait time in seconds (`MX` header) that specifies the upper bound for a random response delay. This way, multiple devices responding to the search are less likely to flood the searching party at the same time. Additionally, the search contains a search target (`ST` header), specifying what to search for (in the simplest case, it is `ssdp:all`, searching for any entity). Usually, the search requests are sent multiple times to ensure that they are received even on lossy networks.

Each *device* that considers itself or one of its services matching the search request now returns a unicast search response to the IP address the search request came from.

2.5.4 Description

Both the description for *devices* and *services* are published via HTTP as XML documents. These contain all information a *control point* needs to interact with the *device* or *service*.

A device description contains among others the device type, a friendly name that can be displayed to the user, a UUID that uniquely identifies a single instance of this device, and several other free form fields such as the manufacturer, an URL to the manufacturer website, a model name, model number, serial number and UPC. It also contains a list of the services the *device* provides, and a list of `embedded devices`. See Figure 2.4 for an example.

A service description contains the list of available actions (methods that can be called remotely) and service state variables. See Figure 2.5 for an example.

2.5.5 Control

After it has discovered a *device* and fetched its description, a *control point* can now invoke actions provided by the *device*. Action invocation is done using SOAP remote procedure calls. The caller sends the SOAP request as an HTTP POST request to the `controlURL` of the service. If the request is valid, the *device* executes the action and returns the response. If the execution would take longer than 30 seconds, the service is required to return instantly, and to send an event when the execution is complete.

The UPnP standard once included a mechanism to directly query for state variables, but this is now deprecated[7].

The actions a *service* provides are usually referred to in the form *<service>:<action>* or *<service abbreviation>:<action>* , e.g. *CMS:GetProtocolInfo* for the ConnectionManager's `GetProtocolInfo` action.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<root xmlns="urn:schemas-upnp-org:device-1-0"
      xmlns:dlna="urn:schemas-dlna-org:device-1-0">
<specVersion>
  <minor>0</minor>
  <major>1</major>
</specVersion>
<URLBase>http://192.168.80.2:49152</URLBase>
<device>
  <deviceType>
    urn:schemas-upnp-org:device:MediaServer:1
  </deviceType>
  <friendlyName>DVB streaming media server</friendlyName>
  <manufacturer>Martin Emrich</manufacturer>
  <manufacturerURL>
    http://www.emmes-world.de
  </manufacturerURL>
  <modelDescription>Martins RTP-Server</modelDescription>
  <modelName>rtpserver</modelName>
  <modelNumber>rtpserver 0.0.2svn</modelNumber>
  <modelURL>http://www.emmes-world.de</modelURL>
  <serialNumber>00123456789</serialNumber>
  <presentationURL>
    http://192.168.80.2:10080
  </presentationURL>
  <UDN>uuid:a40777e1-385a-4b47-8f60-295cf71aa02b</UDN>
  <dlna:X_DLNADOC>DMS-1.00</dlna:X_DLNADOC>
  <serviceList>
    <service>
      <serviceType>
        urn:schemas-upnp-org:service:ConnectionManager:1
      </serviceType>
      <serviceId>
        urn:upnp-org:serviceId:ConnectionManager
      </serviceId>
      <SCPDURL>
        http://192.168.80.2:10080/volatile/cmsservice.xml
      </SCPDURL>
      <controlURL>/volatile/cmscontrol</controlURL>
      <eventSubURL>/volatile/cmsevent</eventSubURL>
    </service>
    <service>
      <serviceType>
        urn:schemas-upnp-org:service:ContentDirectory:1
      </serviceType>
      <serviceId>
        urn:upnp-org:serviceId:ContentDirectory
      </serviceId>
      <SCPDURL>
        http://192.168.80.2:10080/volatile/cdsservice.xml
      </SCPDURL>
      <controlURL>/volatile/cdscontrol</controlURL>
      <eventSubURL>/volatile/cdsevent</eventSubURL>
    </service>
  </serviceList>
</device>
```

Figure 2.4: Example device description for a UPnP AV media server

2.5.6 Eventing

Instead of constantly polling for the state of a *service*, the *control point* may subscribe to events emitted by the *service*. UPnP uses the General Event Notification Architecture[2] (GENA) for notifications. This protocol is based on HTTP, but introduces new methods for the different message types.

A client interested in eventing sends a subscription request to the *service*'s `eventSubURL`. The subscription includes a callback URL and the duration of the subscription. Note that there is no way to limit the subscription to a certain subset *service state variables*. The *service* responds with a unique subscription ID, and sends a first, *initial* eventing including all evented state variables. After that (until either the subscription expires or the client unsubscribes), the *service* sends an event notification to each subscriber whenever a variable changes.

2.6 UPnP AV

The UPnP AV[6] infrastructure specifies a set of UPnP device profiles and their interaction to facilitate the transfer of media content (audio, video, images, ...) between devices. These device profiles are:

- A **Media Server** that is typically the source of a media stream. An example is a networked storage device that publishes the contained media files via UPnP AV. Intel has provided a helpful set of design guidelines[15].
- A **Media Renderer** receives media for playback. This can be a networked TV set, audio player or set-top-box.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major><minor>0</minor>
  </specVersion>
  <actionList>
    <action>
      <name>GetSortCapabilities</name>
      <argumentList>
        <argument>
          <name>SortCaps</name><direction>out</direction><relatedStateVariable>
            SortCapabilities
          </relatedStateVariable>
        </argument>
      </argumentList>
    </action>
    <action>
      <name>GetSystemUpdateID</name>
      <argumentList>
        <argument>
          <name>Id</name><direction>out</direction><relatedStateVariable>
            SystemUpdateID
          </relatedStateVariable>
        </argument>
      </argumentList>
    </action>
  </actionList>
  <serviceStateTable>
    <stateVariable sendEvents="no">
      <name>SearchCapabilities</name><dataType>string</dataType>
    </stateVariable>
    <stateVariable sendEvents="no">
      <name>SortCapabilities</name><dataType>string</dataType>
    </stateVariable>
    <stateVariable sendEvents="yes">
      <name>SystemUpdateID</name><dataType>ui4</dataType><defaultValue>
        0
      </defaultValue>
    </stateVariable>
  </serviceStateTable>
</scpd>
```

Figure 2.5: Excerpt from a service description (UPnP AV media server's ContentDirectory service). Some actions/service variables removed.

- A **Control Point** refers to a device capable of discovering other UPnP AV devices, matching the media available from Media Servers with the capabilities of available Media Renderers and controlling the media transfers. It can be thought of as a networked remote control.

The device profiles can be combined, e.g. a Media Renderer and a Control Point could be together in a networked TV set, where the user can choose available content on screen and play it back.

2.6.1 Media Server

The media server provides two to three *services*: A ContentDirectory[5] to access the list of available content items, a ConnectionManager[4] to initiate or stop media transports, and an AVTransport[3] service to control individual media transports. As most of the popular transport methods (HTTP or RTSP/RTP) provide transport management themselves, some ConnectionManager actions and the complete AVTransport service are optional.

2.6.2 Media Renderer

The media renderer also provides two to three *services*: A RenderingControl to influence the rendering itself (e.g. volume control, brightness, contrast), a ConnectionManager and again an optional AVTransport service.

2.6.3 Control Point

The control point brings the former two together. It locates available media servers and renderers and retrieves the content directory for display to the user. It matches the source's and sink's supported protocols and media formats to display only combinations that can be successfully played back. Finally, it initiates and controls the media transfer.

2.6.4 Content Directory

The content directory[5] contains references to all media items available on a media server, arranged in a tree structure. A control point can navigate through the content directory using the *CDS:Browse* action, or search it using the *CDS:Search* action. Optionally, the content directory can even be modified by the control point, e.g. to upload new media to the device.

Each object is required to have at least a unique ID, the ID of its parent (The root object has to have an ID of 0, and a parentID of -1), a title and a *restricted* flag (if set to true, the object is read-only for control points). Derived classes may add additional fields and attributes.

The object types are all inherited from the same base type `object`. Here are the most basic types and some derived examples:

- `object.container` corresponds to a file system folder. It can have child objects. Mandatory fields are the ID, a name
- `object.item` is the most basic media item type. All other media types inherit from this.
- `object.item.audioItem` A generic audio item. Adds attributes like *genre* or *publisher*.
- `object.item.audioItem.musicTrack` refers to one single music item, e.g. one song on an album. Added attributes are e.g. *artist* or *album*

A *control point* accessing the content directory via the *CDS:Browse* method can influence the result with these parameters:

- *ObjectID* where to start
- *BrowseFlag*: One of “BrowseMetadata” or “BrowseDirectChildren”. The former returns only information about the object requested, while the latter returns objects directly below the specified ObjectID.
- *Filter*: If set to “*”, all required properties are returns. Alternatively, a list of properties to be returned can be specified.
- *StartingIndex*: for “BrowseDirectChildren”, this is a result offset, for example to fetch object information page by page.
- *RequestCount*: Limit the number of returned objects. Together with *StartingIndex*, a device with a small screen can fetch only the subset of objects that currently fit on the screen.
- *SortCriteria*: Specifies how the results shall be sorted. Example: “`+upnp:album,+upnp:originalTrackNumber`”.

The media server then returns the requested content objects as a DIDL-Lite XML document. DIDL-Lite[5] itself is a subset of the *Digital Item Declaration Language* specified in MPEG-21 (ISO/IEC 21000). This translates the object structure described above into XML.

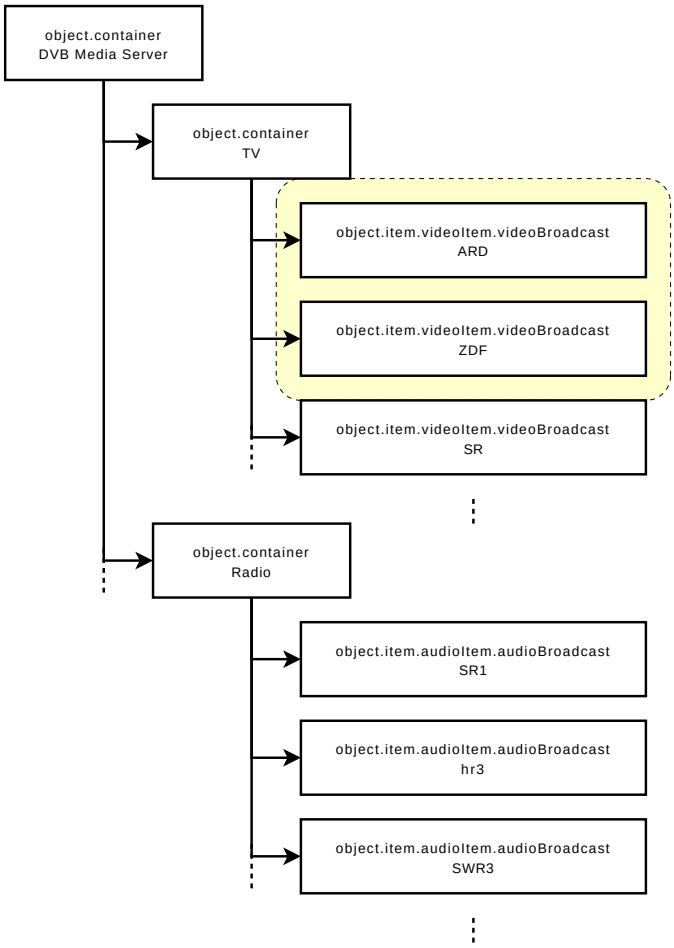


Figure 2.6: Content directory structure. An XML representation of the marked area can be seen in figure 2.7

2.6.5 Content resources

Each content item object contains one or more *resources* pointing to the actual content. Multiple resources for the same content item may point to different available transports, or different representations of the content (e.g. an additional smaller version of a photograph, or a a video encoded with different codecs). Each resource contains at least the URL where the resource can be obtained, and a *protocolInfo*[4] string which is made up of four fields:

1. **Protocol**: The transport protocol used: The two protocols used by this project are `http-get` for HTTP and `rtsp-rtp-udp` for RTP. Also possible are `internal` for a connection internal to the device, `iec61883` for an IEEE1394 link or the vendor's ICANN-registered domain name for a vendor-specific transport.

2. **Network**: Not needed for HTTP or RTP, thus set to "`*`".

```
<DIDL-Lite
      xmlns="urn:schemas-upnp-org:metadata-1-0/DIDL-Lite/"
      xmlns:dc="http://purl.org/dc/elements/1.1/"
      xmlns:dlna="urn:schemas-dlna-org:metadata-1-0"
      xmlns:upnp="urn:schemas-upnp-org:metadata-1-0/upnp/">
  <item id="3" parentID="2" restricted="1">
    <dc:title>Das Erste</dc:title>
    <upnp:channelNr>1</upnp:channelNr>
    <upnp:channelName>Das Erste</upnp:channelName>
    <upnp:class>object.item.videoItem.videoBroadcast</upnp:class>
    <res bitrate="1875000"
protocolInfo="http-get:*:video/mpeg:\
DLNA.ORG_PN=MPEG_PS_PAL;\
DLNA.ORG_FLAGS=8D10000000000000000000000000000000">

      http://134.96.63.118:10080/192.168.80.1/tv/\
S19.2E-1-1101-28106.mpg?apid=102&amp;audio=MP2&amp;\
container=MPEGPS&amp;ppid=101&amp;video=MPEG2&amp;vpid=101
    </res>
    <res protocolInfo="rtsp-rtp-udp:*:video/mpeg:\
DLNA.ORG_PN=MPEG_TS_SD_EU_ISO;\
DLNA.ORG_FLAGS=8D10000000000000000000000000000000">

      rtsp://134.96.63.118:10554/192.168.80.1/tv/\
S19.2E-1-1101-28106.mpg?apid=102&amp;audio=MP2&amp;\
container=MPEGTS&amp;ppid=101&amp;video=MPEG2&amp;vpid=101
    </res>
  </item>
  <item id="4" parentID="2" restricted="1">
    <dc:title>ZDF</dc:title>
    <upnp:channelNr>2</upnp:channelNr>
    <upnp:channelName>ZDF</upnp:channelName>
    <upnp:class>object.item.videoItem.videoBroadcast</upnp:class>
    <res bitrate="1875000" protocolInfo="http-get:*:video/mpeg:\
DLNA.ORG_PN=MPEG_PS_PAL;\
DLNA.ORG_FLAGS=8D10000000000000000000000000000000">

      http://134.96.63.118:10080/192.168.80.1/tv/\
S19.2E-1-1079-28006.mpg?apid=120&amp;audio=MP2\
&amp;container=MPEGPS&amp;ppid=110&amp;video=MPEG2&amp;vpid=110
    </res>
    <res protocolInfo="rtsp-rtp-udp:*:video/mpeg:\
DLNA.ORG_PN=MPEG_TS_SD_EU_ISO;\
DLNA.ORG_FLAGS=8D10000000000000000000000000000000">

      rtsp://134.96.63.118:10554/192.168.80.1/tv/\
S19.2E-1-1079-28006.mpg?apid=120&amp;audio=MP2&amp;\
container=MPEGTS&amp;ppid=110&amp;video=MPEG2&amp;vpid=110
    </res>
  </item>
</DIDL-Lite>
```

Figure 2.7: Example DIDL-Lite XML document

3. **Content Format**: Usually the MIME type of the content, e.g. **video/mpeg** or **image/jpeg**.

4. **Additional Info**: May contain further information about the resource. If not needed, it is set to "*".

2.7 DLNA

The Digital Living Network Alliance[8] (DLNA) is a consortium of leading consumer entertainment vendors. Its goal is to provide the end user with networked home entertainment devices that work together seamlessly even if they are from different vendors, as long as they bear the DLNA CERTIFIED™logo.

The DLNA is practically the heir of the UPnP AV group, and the DLNA architecture is basically UPnP AV with many additional specifications and clarifications, all designed to avoid compatibility problems between different devices. These are some of the more notable additions:

- More device profiles: DLNA distinguishes not only between Media Server, Media Renderer and Control Point, but between different device categories such as home network devices (HND) or mobile handheld devices (MHD), and different media classes (e.g. Digital Media Player (DMP) or Mobile Digital Media Server (M-DMS)). There is also a printing architecture (built on top of the UPnP Printer Definition). This way, the single device capabilities can be specified much more thoroughly.

- DLNA Media Controllers (DMC) not only match content and Renderer based on transport protocol and format, but on standardized DLNA **Media Profiles**, that specify containers, codecs and even image size. This avoids many compatibility problems, e.g. attempts to play back 1080p video content on a mobile phone.

- Mandatory transports, containers and codecs: For a device to be certified, it must at least support HTTP transport and a minimum set of media formats (including MPEG 2 video, MPEG Layer 3 Audio, JPEG and LPCM). DLNA even specifies a Media Interoperability Unit (MIU), being essentially a transcoder device to make content compatible to other devices.
- Detailed media transport specifications: The DLNA device guidelines also specify how to access and transport content that is not randomly accessible, such as live broadcasts or growing recordings on a hard disk recorder.

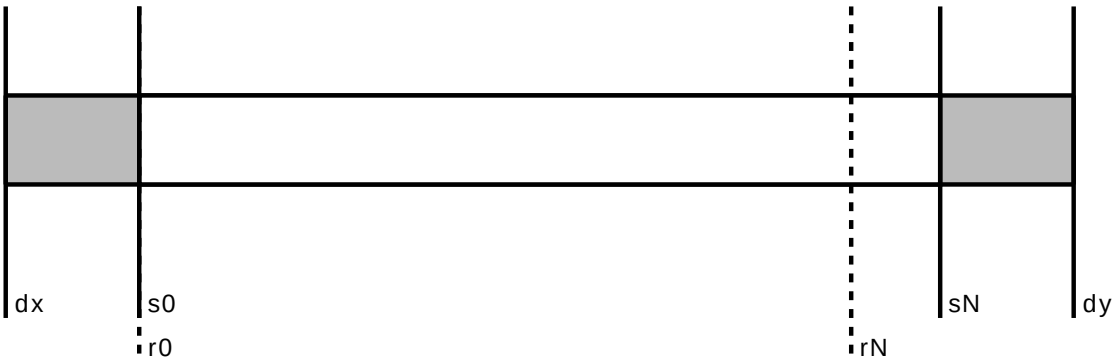
For a device to be DLNA CERTIFIED™, it must conform to hundreds of requirements, which is tested using a conformance test tool (CTT) provided by the DLNA to its members. The DLNA homepage[8] lists already over 1000 certified devices, including TV sets, notebooks (usually by the means of a certified DLNA software) and of course the Sony PlayStation 3 video game console.

Although DLNA specifies everything needed for bringing live broadcast into the home network (RTSP/RTP transport or tuner management), as of the end of 2008, no devices implementing these portions of the guidelines have actually hit the market. This may also be due to the fact that these features are optional for the certification process and thus not tested by the CTT (it tests mostly only the mandatory requirements). Another cause could be the increased development effort needed for the RTP transport.

Besides constraining or clarifying the existing UPnP AV infrastructure, the DLNA guidelines[9] primarily add new headers and fields to the existing protocols and formats. The motto is always “parse and interpret, or parse and ignore”, meaning that any information encountered in communication that is not understood (e.g. a vendor-specific XML element in a DIDL-Lite document) or seems malformed, the peer shall always be as forgiving as possible and act as if the element was never there. This way, communication can be augmented with additional information without disturbing legacy clients.

2.7.1 Data Availability Models

DLNA knows about the different availability properties of stored and live content, and thus has defined the *Uniform Client Data Availability Model* (UCDAM) to describe stream boundaries and available ranges:



Uniform Client Data Availability Model

- $[d_x, d_y]$ is the whole content object, with parts possibly unreachable (e.g. for live broadcast content).
- $[s_0, s_N]$ is the data range the content source is actually able to transmit. For content stored on disk, $[d_x, d_y] = [s_0, s_N]$ is usually the case. For live broadcast content (without buffering or recording to disk running), both are equal and constantly moving forward. Depending on that, DLNA draws a distinction between *fixed-s_0*, *s_0-increasing* and *fixed-s_N*, *s_N-increasing*, respectively.
- $[r_0, r_N]$ is the range available for random access. r_0 equals to s_0, and for seekable content r_N usually equals to s_N. For non-random-access content (certain containers do only allow playback from the beginning, such as AVI files with a damaged index, or only to certain intervals, e.g. MPEG groups of pictures), r_N equals s_0.
- $[d_0, d_N]$ is the range available to the content receiver. If it locally buffers the received data, this range may exceed $[s_0, s_N]$.

Based on this description model, the DLNA guidelines specify content under three different data availability models. In all cases, it is assumed that $[s_0, s_N] = [r_0, r_N]$. Note that DLNA specifies seeking both by byte offset and by native play time (npt).

Full Random Access Data Availability Model

This model is used for *fixed-s_0*, *fixed-s_N* content where the complete content binary is always seekable, such as content stored on disk (including growing content that is still being recorded). Here, the client can request arbitrary content ranges from the server, as long as the container format and the codec(s) used permit it.

Limited Random Access Data Availability Model (Mode 0)

This is the model for *s_0-increasing* content, such as live broadcasts. Here, seeking is limited to a subrange of the content binary, maybe even to a single point in time.

Limited Random Access Data Availability Model (Mode 1)

This is the model for *fixed-s_0* content. The example the DLNA guidelines give is a content item currently being encoded by the media server. Here, the content from the beginning to the point where the encoder is currently working is timely available, while the part that is still to be encoded is not.

2.7.2 *protocolInfo* fields

DLNA specifies the fourth field of the content directory resources' *protocolInfo* (See section 2.6.5) as a comma-separated list of key-value pairs to provide further information about the resource. The available keys are

- The `DLNA.ORG_PN` field specifies the DLNA media profile of the content.
- The conversion indicator `DLNA.ORG_CI` specifies whether the content has been recoded. If the content provides several resources, the client can use it to find the content in its original format which usually has the highest quality, as long as the client is compatible with the format.
- `DLNA.ORG_PS`: lists the available extra play speeds (beyond the normal 1x playback speed) to indicate support for e.g. fast forward or slow-motion playback.
- `DLNA.ORG_OP`: specifies the seek capabilities, i.e. if the resource supports seeking via byte- or npt-offsets.
- `DLNA.ORG_MAXSP`: Indicates the support for the RTSP *Speed* header and its maximum value.
- `DLNA.ORG_FLAGS` is a 32-character hexadecimal string containing dozens of boolean flags. The most important are
  - *sp-flag*: if true, the content is sender-paced (server acts as the clock source), so the receiver cannot assume the ability to control the transmission speed.
  - s_0-*increasing* and s_N-*increasing*: if true, the content's s_0 (or s_N respectively) boundary is increasing with time.
  - *tm-s*, *tm-b* and *tm-i*: Selects the DLNA transfer mode, one of "Streaming" (real-time foreground transfer of audio/video content), "Background" (low-priority bulk transfer) and "Interactive" (best-effort foreground transfer for images).
  - *http-stalling* indicates if the server allows the client to pause the transmission by simply stalling the HTTP connection.

Using this information, the receiver can determine what capabilities each content item has: Is it possible to fetch the content in blocks or is a continuous transfer necessary? Can the content be randomly accessed using byte ranges or even time codes? If these features are available, the receiver indicates its requests using standard HTTP or RTSP headers, or if necessary, using special DLNA headers.

2.7.3 HTTP headers

Several new HTTP headers are introduced by the DLNA to specify transfer parameters:

`contentFeatures.dlna.org` : Is sent by the server whenever a client sends a GET or HEAD request with a header line of "`getContentFeatures.dlna.org: 1`". It contains the content resource's fourth field (the DLNA protocolInfo fields).

`availableSeekRange.dlna.org` : A client can request the currently available seek range for a given content item by issuing a GET or HEAD request with "`getAvailableSeekRange.dlna.org: 1`".

The server responds with a `availableSeekRange.dlna.org` header indicating the data availability mode (see above) and available seek ranges in bytes and/or npt. Example values are "`0 bytes=0-0`" (Limited Random Access Data Availability Model, Mode 0, no seeking whatsoever possible), or "`1 npt=00:00:00-00:08:31 bytes=0-204356823`" (Limited Random Access Data Availability Model, Mode 1, seeking available both in NPT and bytes).

`transferMode.dlna.org` : Using the `transferMode.dlna.org` header, the client specifies whether the transfer is a Streaming, Interactive or Background transfer. Then the server responds with the given transfer mode using the same header, and applies the necessary QoS mechanisms for the transfer mode. For backwards compatibility (with previous DLNA versions and non-DLNA devices), omitting the header means Streaming transfer.

`Range` and `timeSeekRange.dlna.org` : In addition to the normal HTTP `Range` header for byte seeking, DLNA defines the `timeSeekRange.dlna.org` header, which implements seeking using NPT.

`PlaySpeed.dlna.org` : This header sent by the client specifies the intended playback speed. If the server confirms the request by sending the same header back, it will then send the content in time-scaled form (e.g. by doubling frames and slowing down audio for slow-motion playback).

`realTimeInfo.dlna.org` : For content with realtime character (such as live broadcasts), client and server can communicate additional information in key-value pairs. The only currently used key is `DLNA.ORG_TLAG`. If it is a finite (positive real) number, it specifies the time in seconds between the current time (e.g. when a piece of content is received from the tuner hardware) and the time it must have been sent to the client. If this time is exceeded (because the client did not accept it in time), the server is allowed to drop the data to keep the realtime character of the transmission. If it is unlimited, the tag is set to "*".

2.7.4 RTSP headers

`Supported` (`dlna.announce, ...`) can be issued by client or server. It contains a comma-separated list of capabilities of the sending endpoint, e.g. `dlna.announce` if the server can send RTSP ANNOUNCE messages.

`Buffer-Info.dlna.org` : When the server reports support for RTCP Buffer Fullness Reports, this is sent by the client during the RTSP SETUP request to indicate the client buffer characteristics. It includes dejitter buffer size in bytes, coded data buffer size in bytes, receiver buffer transfer mechanism (whether the dejitter buffer will push down to the coded data buffer immediately or based on the packet's time stamp) and the receiver's buffer size in milliseconds.

`WCT.dlna.org` : If set by the client to `1`, the server must add wallclock time samples to the data stream.

`Max-Prate.dlna.org` is sent by the client to indicate its maximum incoming packet rate.

`Event-Type.dlna.org` in an RTSP ANNOUNCE message from the server indicates the event. E.g. if the end of the stream is reached, the server sets it to 2000.

`Available-Range.dlna.org` is used in the RTSP DESCRIBE response by the server to indicate the available seek range (UCDAM r_0 and r_N), similar to the HTTP `availableSeekRange.dlna.org` header.

`Predec-Buffer-Size.dlna.org`, `Initial-Buffering.dlna.org` specify the buffer size required by the receiver for the RTP streams.

2.8 Transport protocols

2.8.1 HTTP

The HTTP protocol, probably accounting for most of the internet traffic (except maybe peer-to-peer filesharing networks), surely does not need much introduction. It's simple and extensible header structure, independence of the transmitted content types and its reliability have made it the favorite transfer protocol for the home media network. Both request and response use a plain text header before any transmitted content ("body"). The first line of a request specifies the *method*, the resource and the protocol identifier (e.g. `HTTP/1.1`).

Example request:

```
GET /hello.txt HTTP/1.1\r\n
Host: www.example.org\r\n
\r\n
```

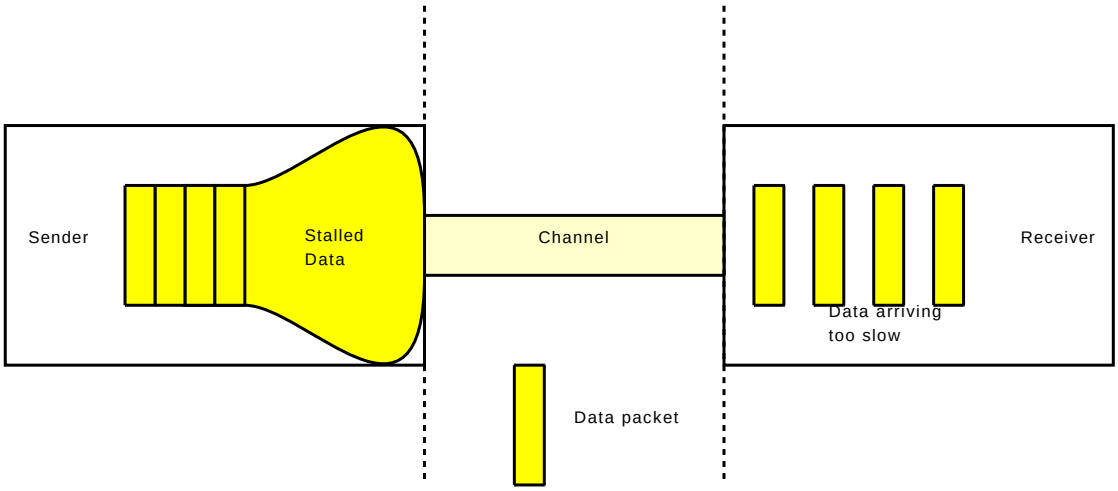
And the corresponding response:

```
HTTP/1.1 200 OK\r\n
Content-Length: 12\r\n
Connection: close\r\n
Content-Type: text/plain\r\n
\r\n
Hello World!
```

Based on the TCP protocol, which is designed for best-effort, reliable in-order data transfers, HTTP it does not make any real-time guarantees. For the classic world wide web, this is of course not necessary, but for media streaming applications, a certain amount of work is necessary to make HTTP play along. As long as the receiver maintains a sufficiently large buffer, it is

acceptably suited for the transmission of stored content, as long as the net bandwidth provided by the channel between sender and receiver is perfectly above the data rate of the content being played.

As soon as the available bandwidth is exceeded, TCP's stream flow control turns to a disadvantage.



A HTTP transmission over an insufficient channel stalls the sender and starves the receiver.

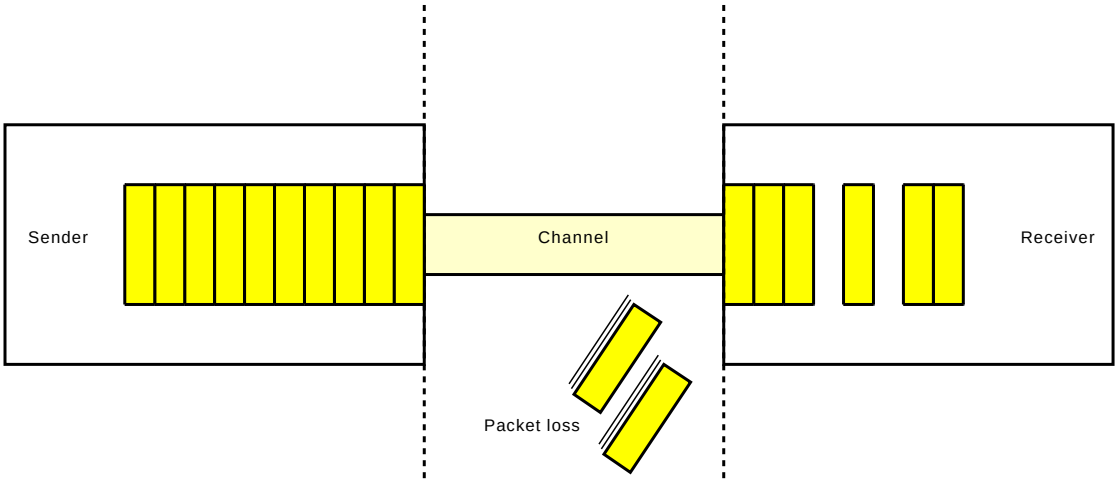
In this case, the client will receive the data too slow (although still complete and in-order), and can no longer play without interruptions. To minimize this problem, most players buffer a fair amount of data before starting playback, to compensate for short data rate drops.

The biggest advantage of HTTP is that the transmitted content does not need to be aware of the channel characteristics, because HTTP guarantees that nothing is lost. So HTTP remains the transport protocol of choice for background transfers and transfers of non-realtime-sensitive data.

2.8.2 RTSP/RTP

In contrast, the Realtime Transport Protocol (RTP[19]) and it's companion protocols RTSP[20] and RTCP are from the ground up designed for realtime streaming applications.

RTP is the protocol carrying the actual data. It is based on UDP, which by itself neither takes care of data ordering nor timing. RTP thus adds a sequence number for each packet to keep them in order and detect losses. Additionally, each packet contains a timestamp for its first payload byte for synchronization purposes, and a payload type to identify the type of data (e.g. MPEG video or PCM audio). The application must of course take the lossy nature of the protocol into account, either by error correction schemes or a robust decoder. As it is UDP-based, it can also be used in a multicast environment, when multiple clients want to receive the same content simultaneously.



A RTP transmission over an lossy channel will only result in packet loss, but the server is still able to send.

RTSP (Real Time Streaming Protocol) is used for session management, i.e. to determine content properties and control the transmission (setup, playback, pause, stopping,...). It is designed syntactically quite similar to HTTP, but unlike that it is stateful, so successive operations on the same session do not need to happen on the same TCP connection. Therefore, it includes a session id and a sequence number.

A typical RTSP session usually begins with an `OPTIONS` request by the client. The server responds with a list of methods it supports. Next, the client will issue a `DESCRIBE` request for a particular content item. The server responds with a description to let the client know about the type of the streams to expect. Then, the client sends a `SETUP` request. Only now, the server will allocate resources for the session, and issue a *Session ID* to the client, who now must include the session ID in each subsequent request regarding this session. Now, the client may issue `PLAY` or `PAUSE` requests to control the transmission. Finally, when the client wants to end the session, it sends a `TEARDOWN` request.

While a streaming session is playing (or recording), the sender and the receiver may *optionally* exchange connection information over RTCP (Realtime Transmission Control Protocol). It is mostly used for sender or receiver reports, e.g. notifying the peer about missed packets, or giving the sender a buffer fill report.

One problem with RTP remains, and that is dead peer detection (DPD). As there is no persistent connection during a running session, a server cannot passively detect when a client just leaves the network, and will just continue to send the RTP stream. One possibility is to use regular receiver reports via RTCP, and to tear down the session if no report is received within a certain amount of time. Another “workaround” used by many current players[8] is to keep an RTSP connection open, and to issue a no-op request (e.g. `OPTIONS`) including the session ID every few seconds.

[8] e.g. VLC or mplayer

Session Description Protocol (SDP)

The Session Description Protocol[14] is a format to present the properties of a certain session. It is a plain text format, consisting of an *ordered* list of key-value pairs.

```
v=0
o=- 1224610665 1224610665 IN IP4 192.168.80.2
s=Das Erste
c=IN IP4 224.0.1.2
t=0 0
m=video 0 RTP/AVP 33
a=control:rtsp://192.168.80.2:10554/video.mpg
```

This example describes a multicast video transmission. The session ID is 1224610665, originating from the IP address 192.168.80.2 over an IPv4 network. The stream's session name is "Das Erste", and it is transmitted on the multicast channel 224.0.1.2 with no predefined UDP port (0). It consists of one MPEG Transport Stream (RTP payload type 33), and is controlled via RTSP at `rtsp://192.168.80.2:10554/video.mpg`.

The DLNA specifies rules for the usage of existing SDP headers, as well as some new SDP headers, including `a=contentFeatures.dlna.org` (again containing the content item resource's fourth field) or `a=scmsFlag.dlna.org` (specifying copyright and copy status).

Error correction

Due to the lossy nature of RTP, the content should be made more robust against transmission errors, especially when unreliable (Wireless LAN) or shared (home network) networks are used. There are two approaches here, either adding redundancy up-front (forward error correction, FEC), or by providing the receiver with a way to rerequest lost packets (automatic rerequest, ARQ).

FEC works by adding a certain amount of error correction data to each packet. When a packet is lost, the receiver is still able to reconstruct it by using the correction data of the surrounding packets. The obvious disadvantage is that the increased bandwidth is always required, even if the there are no errors to be corrected. But if data corruption occurs, the client is able to quickly reconstruct the data. This is ideal for lossy channels where excess bandwidth is available.

ARQ gives the client the possibility to request a copy of a lost packet from the server within a short timeframe. The main drawback is that the receiver has to buffer the stream at least for twice the transmission time plus some delay in the server for sending the retransmission. It also does not scale well if used for multicast over a lossy shared medium (such as wireless LAN), where many clients request retransmissions for different blocks.

Hybrid error correction schemes combine these two methods, and ideally adapt the parameters of the FEC to the current channel characteristics. The goal is to avoid as many retransmissions as possible with the FEC, while still having the possibility to rerequest uncorrectable packets. The *adaptive hybrid error correction* (AHEC) used in this project falls in this category.

In any case, the streaming media format should be suitable for lossy channels, i.e. the stream should still be decodable if some data is corrupt (of course with visible or audible losses). The MPEG 2 video codec is a good example, if data corruption occurs, small blocks may appear garbled on the screen, but the playback remains running.

All of these schemes work only if the channel bandwidth is greater or equal to the bandwidth required by the content. If this is not the case, the only option is to reduce the content's requirements, e.g. by requantizing the MPEG video (dropping detail information) or completely recoding the stream (possibly to a lower video resolution or lower audio sampling rate).

A fairly new approach are scalable (video) coding schemes, which require additional support by the physical layer. Here, a lower quality version of the content is transmitted in a more robust way (for example with a lower bitrate in a wireless LAN), while the extra information necessary to recreate a higher-quality version together with the base information is transmitted with higher bitrate. Whenever the client fails to receive the high-quality component, it still can display the low-quality version.

3 Architecture

One of this project's principles is to make use of existing standards to maximize the interoperability with existing and upcoming devices on the market. The DLNA Media Server architecture offers all that is needed, including device discovery, content enumeration, and media transmission. The DLNA/UPnP AV architecture is built using existing internet standards such as HTTP, SOAP and XML.

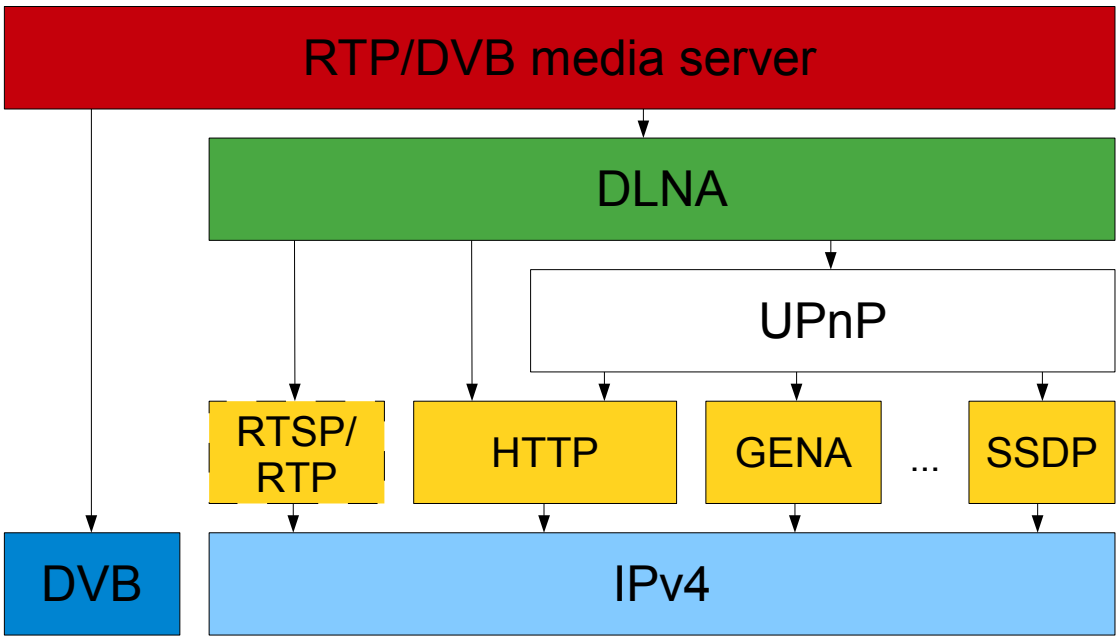


Figure 3.1: Overview of the protocols and standards `rtpserver` uses.

The resulting media server prototype, currently just called `rtpserver` , is UPnP AV and DLNA compatible and runs on the GNU/Linux operating system. It offers broadcast channels received via a tuner backend as DLNA content items to UPnP AV or DLNA clients. The streams can be transmitted via HTTP or RTSP/RTP. If necessary, the stream will be transformed into the required format (remultiplexing or transcoding).

3.1 Stream processing

As `rtpserver` might support more stream types than just DVB MPEG transport streams in the future, its stream processing features a modular design, borrowing from the well-known Unix

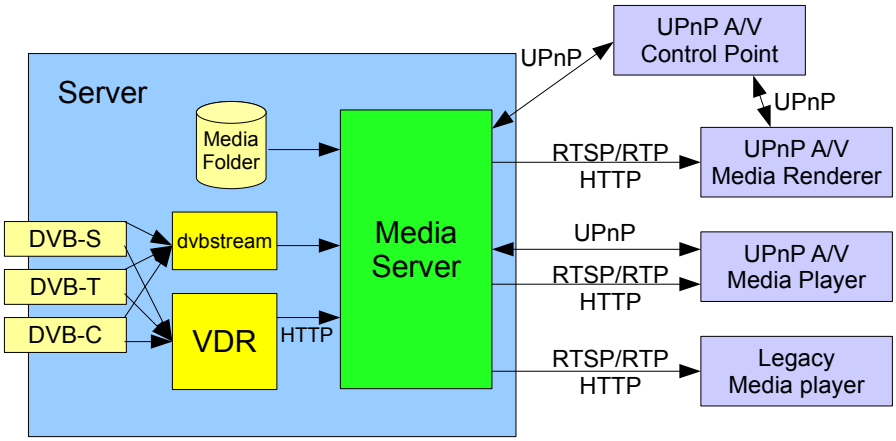


Figure 3.2: `rtpserver` architecture overview

philosophy of “do one thing and do it well”. Streams are processed in a *pipeline* consisting of a source, multiple translators (being both source and sink) and an exit sink. Pipeline elements can be dynamically added or removed from sources (allowing e.g. multiple clients sharing a single DVB-S tuner if they watch channels on the same transponder).

Each source sends data down in discrete chunks, tagged with a payload type and an extra identification depending on the payload type. Sinks are expected not to delay their source in any way, thus most translators work asynchronously by processing the data in an extra thread.

Each Source produces a specified type of output, sent down in discrete chunks to accommodate whatever subdivision the payload requires. It is identified by *payload type*, *container type* and an optional multipurpose ID. While the *container type* describes the container format used by the data (e.g. AVI, MKV, MPEG-PS), the *payload type* describes what the source sends down in each chunk (e.g. one MPEG transport stream packet, an MPEG PS pack, or raw, unaligned data).

3.1.1 Sources

Each source type consists of a content item class, which represents a content item this source can provide, a *SourceProvider* class that creates source objects for a given content item, and the source class itself, that provides the data.

VDRSource

As mentioned above, `rtpserver` can access the DVB hardware via a running VDR with the `streamdev` server. It is even possible to run VDR and `rtpserver` on two different, networked machines, although it is not recommended.

This method has several advantages:

- The available DVB hardware can be shared between the VDR itself and possibly multiple `rtpserver` instances.

- The tedious task of tuning the DVB hardware directly and managing multiple cards is left to VDR.
- VDR provides a list of channels, so doing a channel scan from `rtpserver` is not necessary.

A minor disadvantage is the slightly increased delay between initiating a streaming session and its beginning.

Upon startup, the VDRSourceProvider will fetch the channel list from the VDR. VDR provides a simple, text-based network protocol called SVDRP. After connecting to the SVDRP port and sending the `LSTC` (LiST Channels) command, VDR sends the list of available channels:

```
> LSTC
< 220 sauron SVDRP VideoDiskRecorder 1.6.0-1; Fri Apr 24 19:30:39 2009
< 250-1 Das Erste;ARD:11837:hC34:S19.2E:27500:101:102=deu,103=2ch;106=dd:104:0:28106:1:1101:0
< 250-2 ZDF;ZDFvision:11954:hC34:S19.2E:27500:110:120=deu,121=2ch;125=dd:130:0:28006:1:1079:0
< 250-3 ProSieben;ProSiebenSat.1:12544:hC56:S19.2E:22000:511:512=deu;515=deu:33:0:17501:1:1107:0
< 221 sauron closing connection
```

This list contains channel number and name, tuning parameters (frequency, symbol rate, FEC ratio,...), and the stream PIDs. Each channel line is translated into a content item for the content directory.

When a channel is to be streamed, the VDRSource connects to the configured VDR instance via HTTP and starts pushing down the data received from VDR to all connected sinks. If multiple clients request the same content, usually an existing VDRSource object for this content is reused, thus for multiple streaming sessions of the same content only one connection to the VDR is used. A VDRSource can provide MPEG transport stream, packetized elementary stream or, for audio source, elementary stream payload.

For TV streams, MPEG transport stream is chosen. VDR will typically provide a full SI stream, consisting of PAT/PMI, video, audio, teletext and PCR PIDs.

DVBStreamSource

This source reads from the DVB device using the command line tool `dvbstream`. Again, for multiple transports requesting the same channel the DVBStreamSource object is reused for multiple concurrent clients, but not for different channels on the same frequency.

The channel list is parsed from a file with the same format as the one VDR uses, it looks similar to the example above.

FileSource

This reads files from the local file system and sends it down the pipeline unaltered. It exists primarily to stream the DLNA test media for the compliance test tool (See section 4.5.1), but can also be used to stream music files[1].

[1] As the content directory is hold in memory and regenerated at every server start, it is not yet usable to serve a bigger music collection.

3.1.2 Translators

As not every sink or client accepts the content in the format the source provides, `rtpserver` provides many translators that modify the data stream in a certain way. Each one usually does only one specific job, as they can be put in sequence for more complex tasks.

MPEGTSPacketizer

It locks itself on MPEG transport stream packet headers[16], and sends down single MPEG TS packet. It does not make any assumptions about the alignment or completeness of the incoming data. If the synchronization is lost, all data is discarded until synchronization is regained.

MPEGTStoPESExtractor

Reassembles the single packetized elementary streams from incoming transport stream packets.

MPEGPEStoESExtractor

Extracts one specific elementary stream from a packetized elementary stream.

MPEGAudioDecoder

Decodes MPEG1 Layer 2/3 Audio elementary stream to a stereo LPCM audio stream. This is mostly used for streaming clients that do not support MPEG Layer 2 audio directly[2].

MPEGTStoPSReplex

Remultiplexes an MPEG transport stream into an MPEG program stream. It is basically the `replex`[3] tool with a C++ class wrapper around it. It extracts a given set of PIDs from the transport stream, recreates

As replex was not designed to work on a live stream, it requires a significant amount of buffering (ca. 6MB) to work. It also does not work with all channels[4].

Unfortunately, this translator is the only way to make the Sony PlayStation 3 play live broadcast TV, as it does not support MPEG transport streams. Note that even with this remultiplexer, the PS3 still shows synchronization problems, resulting in occasional frame skips and audio/video desynchronization. Interestingly, the same stream that exhibits this effect when being streamed live works flawlessly when first saved to disk and then served using the FileSource. This leads to the conclusion that the problem is caused by the PS3's buffers running empty. So far no solution was found.

[2] This translator was written for the Terratec Noxon2Audio streaming radio client, which only supports MPEG Layer 3 audio, but not Layer 2

[3] http://www.metzlerbros.org/dvb/

[4] One example is the German channel Pro7, which does not work

LiveBuffer

Many HTTP streaming clients fill their local buffer until an upper bound is reached before they start rendering. Then they stall the HTTP connection until a lower bound of their buffer is reached, and fetch the next chunk of data. Due to this non-uniform behavior, a *LiveBuffer* element is inserted into the pipeline for all HTTP sessions involving live content. It decouples the incoming stream from the outgoing sink to absorb short moments of HTTP connection stalling. It can also delay the transmission start until a configured buffer fill level is reached, to keep a certain buffer reserve if the client reads the data slightly faster as the incoming source provides it.

3.1.3 Sinks

For each transmission protocol, a separate sink is available. See Section 3.2 for details on the session management.

RTPSink

Sends out the incoming data via RTP over UDP to a client or a multicast channel. For an MPEG transport stream, it adapts the RTP timestamps to the stream's PCR, and packs 7 TS packet together into one RTP packet, as recommended by DVB-IPI[11]. Figure 3.3 shows an illustration.

Support for the DLNA-defined *vnd.dlna.mpeg-tts* payload type (which adds a 32bit timestamp in front of each 188 byte TS packet is not yet included[5].

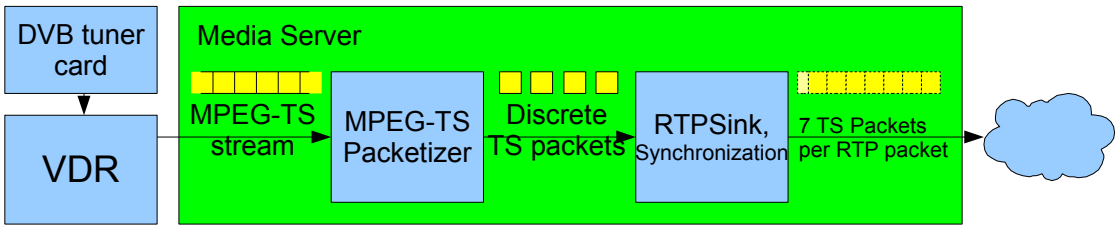


Figure 3.3: Flow of an MPEG transport stream from source to sink

HTTPStreamingSession

Part of the HTTP session object is a sink component. When the client has sent a valid HTTP request for a content item, it attaches itself at the end of the pipeline and sends the data to the client.

[5] No rendering endpoint was available that supports this format, and the relevant documents were only available on short notice.

3.2 Session Management

For each streaming process initiated by a client, a session object keeps all the state related to the streaming. While this is quite easy for HTTP, as signaling is in-band with the actual data, it gets more complicated with RTSP and/or multicast streaming.

3.2.1 HTTP Session

For HTTP, the session starts with the incoming HTTP GET request, and ends when the connection is terminated by either side. After parsing the incoming GET request, the HTTPServer instance asks each registered HTTPHandler to process the request. If it is a valid URL pointing to a streaming media object, the session registry will create a new HTTPStreamingSession and hand the connection over to it. the HTTPStreamingSession parses the header, creates the Source object and the translator pipeline, and starts streaming. The HTTPStreamingSession thus is not only the session object, but also the pipeline sink.

3.2.2 RTSP/RTP Session

For an RTSP session, the RTSPStreamingSession object is created upon a SETUP request by the client. Depending on the negotiated transport, either an RTPUnicastSession or an RTPMulticastSession object is created, and a session ID is generated. This ID is sent back to the client and from now on refers to this session. The client could now close the RTSP connection, and send subsequent requests regarding this session (PLAY, PAUSE, TEARDOWN,...) over new connections. Only when the client sends a TEARDOWN request for this session, the session resources are freed and the session ID is no longer valid.

Here, the session object (RTSPStreamingSession) is separated from the sink object (RTPUnicastSession/RTPMulticastSession). This is necessary for multicast sessions, where multiple clients may request the very same channel simultaneously. The StreamingSessionRegistry keeps track of the number of subscribers for each multicast session, and will destroy it only when the last client ends the connection.

4 Implementation

The core of this project was the desire to implement a working prototype of a media server that could stream live broadcast to an unaltered DLNA client, as well as to a prototype media renderer with RTP support. In the beginning of the `rtpserver` development, a UPnP library was required. There were several Free Software UPnP frameworks available:

- *libupnp*[1]: C library based on the original Intel Linux SDK.
- *libgupnp*[2]: Object-oriented C library, based on GObject, glib.
- *coherence*[3]: UPnP framework written in Python.
- *libdlna*[4]: C library, provides support for DLNA media profiles. Depends on *libupnp*.

At first, *libupnp* was used, but it soon became clear that a native C++ library would have many advantages. *libupnp*'s own use of POSIX threads regularly conflicted with with `rtpserver`, and so more and more aspects of the UPnP framework were rewritten in C++. This inevitably led to a new, native C++ UPnP library called `libupnp++`.

4.1 `libupnp++`

After the UPnP components rewritten from scratch for `rtpserver` grew bigger, they were split off into a standalone library called `libupnp++`. A short time later, `libupnp++` included all features necessary for a Media Server, and the dependency on *libupnp* was removed.

`libupnp++` contains an HTTP web server, that serves all the generated description XML documents. It also handles incoming SOAP calls. The developer can implement the *HTTPHandler* interface to serve own URLs (e.g. for streaming media or own documents), or use a *UPNPServerVirtualDirs* object to serve static items from memory.

The built-in GENA server (together with the *UPNPService* base class) handles event subscriptions and notifications. The SSDP server automatically announces new and disappearing *devices* and *services*.

All servers are started in background threads, so the main application does neither have to run a special main loop nor do any regular calls.

It also includes all *services* necessary for a Media Server (ContentDirectory and ConnectionManager), as well as base classes for content containers and items. Support for *control points* and *embedded devices* is planned, but not yet available.

[1] http://pupnp.sourceforge.net
[2] http://www.gupnp.org
[3] http://www.coherence.beebits.net
[4] http://libdlna.geexbox.org

4.1.1 Requirements

`libupnp++` depends on:

- *GNU Common C++* for multithreading.[5]
- *Apache Xerces-C* 2.8 for parsing/generating XML[6]
- *uuid* to generate UUIDs
- *libdlna* (optional) to classify media files.
- *libmhash* to generate short URLs from arbitrary paths using a MD5 hash table.

4.2 `rtpserver`

`rtpserver` is the UPnP Media Server. Although the first goal was to stream live broadcast TV via RTP, it is built in a modular fashion to include more features in the future.

4.2.1 Requirements

Hardware

A working `rtpserver` installation needs ca. 10 MB hard disk, 256 MB RAM and a 500 MHz processor to stream DVB data untransformed. If a client requires realtime remultiplexing to MPEG program stream, a 1GHz CPU is recommended.

Operating system

At present, only Linux is supported. If a VDR instance (see below) is available on the network, porting `rtpserver` to *BSD, MacOS X or Windows may be possible, but there are currently no plans to do so (Most embedded systems in current set-top-boxes are running Linux anyways). The preferred Linux distribution is Ubuntu 8.10, but others should work if the software requirements below are satisfied.

Software

`rtpserver` needs the following libraries:

- *libupnp++* (See section 4.1)
- *libccrtp-ahec* (provides Advanced Hybrid Error Correction[21]) or *libccrtp* for RTP support

[5] According to http://www.gnu.org/software/commoncpp/, it is to be merged with *uCommon*. As both projects appear to be inactive for some time now, I plan to remove the dependency on GNU Common C++ by rewriting the multithreading classes.

[6] Transition to the more lightweight *libxml2* is being considered

- GNU Common C++ (*libcommoncpp2*)
- *libconfig++* for the configuration file
- *libmad* (optional) for decoding MPEG Layer 2 audio to LPCM
- *libdlna* (optional) to identify DLNA media profiles of stored media files.
- *uuid*

4.3 The demo setup

The demonstration setup as it was successfully showcased at the CeBIT 2009 fair consists of the following components:

- Media Server: Intel Atom-based HTPC, running Ubuntu 8.10 and `rtpserver` . It can have a variety of tuners, including a DVB-S PCI card or a DVB-T USB dongle. As the tuning backend, the VDR is used.
- Media Renderer: Intel Atom-based HTPC, running Ubuntu 8.10 and the media renderer by Jochen Grün[13]. Decoding is done in hardware on a full-featured DVB-S card.
- Media Player: Sony PlayStation III game console. Although it supports only the HTTP transport, it both demonstrated the interoperability between `rtpserver` and a DLNA hardware device, and served as a general eye-catcher.
- Control Point: Apple iPhone 3G running the PlugPlayer[7], a UPnP AV control point application.
- Network: In the role of the home network node, an ordinary wireless router was used, including a small ethernet switch and a DHCP server. All devices except the iPhone were connected via 802.3 ethernet.

The setup demonstrated the great advantages of using the RTP protocol for live broadcast instead of HTTP: While the PS3 needed almost 10 seconds to start playing a station due to the buffering requirements for HTTP, the RTP-based media renderer was able to switch between channels almost instantaneously, just as one is used to from a conventional DVB set-top box.

4.4 Use case: Streaming live TV via RTSP/RTP

This is what happens when a user initiates playback of a TV station from the control point on the media renderer.

[7] `http://plugplayer.com/`

4.4.1 Device startup and discovery

Upon startup, all devices will be assigned an IP address from the DHCP server. The UPnP *devices* will then be discoverable and announce their presence via SSDP. When started, `rtpserver` will scan the configured media folder for media files, and map them into the content directory. The DVB source is queried for available channels, and they are inserted into the content directory, too. `rtpserver` distinguishes between TV and radio stations, and sorts them into two separate content containers.

4.4.2 Content discovery

A control point retrieves the content directory from the media server by issuing SOAP requests for the media server's *ContentDirectory::Browse()* action. This usually happens on-demand, i.e. whenever the user scrolls in the content item list or enters a subdirectory, the newly-visible content objects are fetched from the media server's content directory. Each content item on `rtpserver` that represents a broadcast channel has two *resources*, one for HTTP transport, one for RTSP/RTP. Figure 2.7 shows an example entry for a TV station.

4.4.3 Session initiation

The user usually selects not only the content to be played, but also the renderer where it should be displayed. The control point queries the selected media renderer for a list of possible protocols and formats using the *ConnectionManager:GetProtocolInfo()* action, to see if the renderer would actually be capable of displaying the selected content. The media renderer used here supports these protocols:

```
rtsp-rtp-udp:*:video/mpeg2:,rtsp-rtp-udp:*:video/mpeg:
```

In this case, the intersection between this set of supported protocols and the resources provided for the given TV station leaves only one resource over. Now, the control point will initiate the playback by sending the resource to the media renderer via its *AVTransport::SetAVTransportURI()* action. Then the *AVTransport::Play()* method is invoked to start the playback.

For both HTTP and RTSP/RTP, the transport setup is done outside of UPnP's SOAP framework. For the RTSP/RTP transport, the media renderer[8] first gets the available methods using the RTSP `OPTIONS` method:

```
> OPTIONS * RTSP/1.0
> CSeq: 0
>
< RTSP/1.0 200 OK
< CSeq: 0
< public: DESCRIBE, OPTIONS, PLAY, SETUP, TEARDOWN
< server: rtpserver 0.0.2svn
<
```

[8] The example communication shown her was in part captured using a non-UPnP client (VLC, available at http://www.videolan.org)

rtpserver currently only supports the most basic methods that are necessary for playback.

As the renderer got an RTSP URL from the control point, it will now fetch additional information about it from the server using the DESCRIBE method:

```
> DESCRIBE /rtpserver/radio/S19.2E-1-1093-28422.mpa?\
  apid=431&audio=MP2&container=MPEGES&ppid=0 RTSP/1.0
> Host: 192.168.80.5:10554
> CSeq: 1
>
< RTSP/1.0 200 OK
< CSeq: 1
< server: rtpserver 0.0.2svn
< Content-Type: application/sdp
< Content-Length: 238
<
< v=0
< o=- 1240165012 1240165012 IN IP4 192.168.80.5
< s=hr4
< c=IN IP4 239.35.129.11
< t=0 0
< m=audio 0 RTP/AVP 33
< a=control:rtsp://192.168.80.5:10554/rtpserver/radio/\
  S19.2E-1-1093-28422.mpa?apid=431&audio=MP2&container=MPEGES&ppid=0
<
```

The server responds with an SDP document describing the content. Note that until now, no preparation for a streaming session has happened in rtpserver . This will only happen if the renderer sends the SETUP request:

```
> SETUP rtsp://192.168.80.5:10554/rtpserver/tv/\
  S19.2E-1-1079-28006.mpg?apid=120&audio=MP2&\
  container=MPEGTS&ppid=110&video=MPEG2&vpid=110 RTSP/1.0
> CSeq: 3
> Transport: RTP/AVP;multicast;ttl=1
> user-agent: VLC media player (LIVE555 Streaming Media v2008.07.24)
>
< RTSP/1.0 200 OK
< CSeq: 3
< Session: 1240167190
< server: rtpserver 0.0.2svn
< transport: RTP/AVP;multicast;mode="PLAY";destination=239.35.129.11;ttl=1;port=10004-10008
<
```

Here, the client requested the session setup for a TV channel. It indicated support for multicast using the Transport header. rtpserver will now look up the content item for the requested URL.

4.4.4 Setting up the pipeline

rtpserver will now assemble the pipeline associated with the chosen content item. To make real use of a multicast transport, rtpserver keeps track of all running multicast transmissions,

and would return an existing instance if the given broadcast channel was already being streamed. But it is assumed here that this transmission being set up is the only one.

First, a source is created by the source provider associated with the content item, in this case the *VDRSourceProvider*. To save bandwidth and work efficiently with the number of DVB tuners available, the *VDRSourceProvider* also keeps track of all VDR sources, and will reuse sources of the same channel. Here, a new *VDRSource* will be created, which will connect to the VDR for the media stream.

Next, the sink is created, in this case an `RTPSink`. A multicast channel and a set of RTP ports is assigned to it.

Finally, the chain of translators between the source and the sink is assembled, currently using a hard-coded set of rules. As the *VDRSource* already produces an MPEG transport stream, only the *MPEGTSPacketizer* is needed, so the final pipeline will be *VDRSource→MPEGTSPacketizer→RTPSink*.

The client is now assigned a unique session ID, and the server is starting to stream the TV channel on the assigned multicast channel. As it is a live stream over an unidirectional UDP transport, it is no problem to start streaming right away without waiting for the client). Nevertheless, an ordinary client will now still issue the *PLAY* request to start the stream:

```
> PLAY rtsp://192.168.80.5:10554/rtpserver/tv/S19.2E-1-1079-28006.mpg\
  ?apid=120&audio=MP2&container=MPEGTS&ppid=110&video=MPEG2&vpid=110 RTSP/1.0
> CSeq: 4
> Session: 1240167190
> range: npt=0,000-
> user-agent: VLC media player (LIVE555 Streaming Media v2008.07.24)
>
< RTSP/1.0 200 OK
< CSeq: 4
< Session: 1240167190
< server: rtpserver 0.0.2svn
<
```

The client will now listen on the negotiated UDP multicast channel for the media stream and render it.

4.4.5 Ending the session

When the user ends the playback, the renderer will terminate the RTSP session with a `TEARDOWN` request:

```
> TEARDOWN rtsp://192.168.80.5:10554/rtpserver/tv/S19.2E-1-1079-28006.mpg\
  ?apid=120&audio=MP2&container=MPEGTS&ppid=110&video=MPEG2&vpid=110 RTSP/1.0
> CSeq: 5
> Session: 1240167190
> user-agent: VLC media player (LIVE555 Streaming Media v2008.07.24)
>
< RTSP/1.0 200 OK
< CSeq: 5
< Session: 1240167190
< server: rtpserver 0.0.2svn
```

Now, the server will deallocate all resources of this session, unless they are also in use by another session (e.g. shared DVB channels or multicast sessions).

4.5 DLNA compliance tests

4.5.1 The DLNA Compliance Test Tool

The DLNA provides their members with a test suite application to verify that the "device under test" (DUT) complies with the DLNA guidelines. The Compliance Test Tool (CTT) is a Windows application that runs a test against the DUT for most of the guidelines. The guidelines are categorized into "must" rules that are required for compliance, "should" rules that are considered good behavior and "optional" rules that the DUT may or may not comply with. While almost all "must" and most "should" guidelines have a test case, most "optional" guidelines do not.

After starting the CTT, the user must create a device profile by providing the class of the DUT (e.g. DMS for a digital media server), and check boxes for additional features (e.g. wireless network support, multiple network support or media upload). The CTT then compiles a set of tests for the chosen features.

If the guideline to be tested is applicable for the device (i.e. the guideline applies for the device's class, category and capabilities), the test will either pass or, depending on the category, will either fail or get a warning state. If the test detects that the device does not claim to support an optional feature the guideline describes, it will return "N/A". Some tests also require manual intervention, such as rebooting or power-cycling the DUT.

4.5.2 `rtpserver` 's test status

The CTT has several disadvantages that made testing `rtpserver` a little bit more difficult. The biggest problem was its limited support for endless media. Most of the tests that involve fetching broadcast media from `rtpserver` wait for the stream to end, what would never happen under normal circumstances. Thus `rtpserver` has a special option to limit broadcast streams to 30 seconds. Also, some tests and their corresponding guidelines were not documented clearly enough, so it was not possible to satisfy a few tests. Some tests were included into the chosen test set even if they were for features `rtpserver` does neither provide nor advertise. Finally, as the whole RTSP/RTP support is optional, no tests for this protocol were included in the CTT.

Nevertheless, `rtpserver` satisfies the CTT tests with only a handful exceptions, where the cause for the failure could not be determined. The results are:

- **568** Guidelines/Guideline blocks[9] were relevant for our device profile (DMS).
- For **124** of them, the CTT did not provide a test. If the guideline was applicable to `rtpserver` , it was attempted to satisfy the guideline anyways.
- **237** guidelines were related to optional features `rtpserver` does not (yet) provide, such as media upload. The CTT correctly identified them as Not Applicable.
- **188** tests passed without problems.

[9] For the cases where an entire feature comprised of multiple guidelines were irrelevant for `rtpserver` , these were grouped together as only one

- **8** tests returned a “warning” result. Among them are missing icons for content items (Album covers, station logos) or a too small SSDP `CACHE-CONTROL`[10] value that was set intentionally to speed up the tests.
- **11** tests actually failed. For 5 of them, the CTT finds faults in data that is actually never sent on the network. 4 tests claim that `rtpserver` supports media upload (any attempts to upload fail of course), although neither `rtpserver` nor the CTT profile claim media upload support. The remaining two were not satisfiable as the referenced documents were either ambiguous or not available.

[10] The maximum interval in seconds between SSDP re-advertisements is recommended to be 1800, but during testing it was reduced to 60

5 Conclusion

It has been shown that a broadcast TV experience similar to that of traditional transports can be provided in the home network while using existing standards. Even playback of live TV on a commercially available DLNA client is possible via HTTP, although with some remaining AV synchronization problems and limited convenience. What is still missing is the complete TV-like experience with a quickly responding remote control that can switch channels up an down just like with a normal TV set.

5.1 Future Work

Even though the essential features are working, some topics still need to be addressed:

5.1.1 Direct DVB tuner access

The current DVB backends (*VDRSource* and *DVBStreamSource* both rely on third-party software to tune the DVB hardware and access the received stream. A future `rtpserver` version will certainly include direct access to the DVB tuner hardware using the Linux DVB API.

5.1.2 DVB-SI mapping and EPG

DVB-HN specifies how the service information embedded in the DVB streams (DVB-SI), for example the EPG (Electronic Program Guide) data, is to be mapped into the content directory. When this is done, each individual TV or radio program is mapped to a separate content item. These could then be requested individually by a recording client.

5.1.3 Dedicated control point

Neither the Sony PlayStation 3 nor the PlugPlayer on the iPhone offer the possibility to “zap” up and down between channels, as these devices are designed with stored content in mind. So a future project could include a DLNA control point with the TV experience in mind. The content directory provided by `rtpserver` already includes the `upnp:channelNr` property to arrange the available broadcast channels in a sequential list just like on a legacy TV set.

5.1.4 Embedded media server and player

Currently, both media server and media renderer run on a complete Ubuntu installation installed on a hard disk, that requires several minutes to boot up as well as manual operation to start the media server or renderer. A more user-friendly design could consist of a small, dedicated Linux

distribution that boots quickly off a flash drive directly into the application. Such a design could provide faster booting even than some of the existing commercial IPTV platforms[1]. Based on the connection status (Whether a display and/or a tuner device is connected), either media server, media renderer or both could be started automatically.

5.1.5 Content directory search

The UPnP *ContentDirectory* specifies an optional, powerful *Search* action, that can return, based on user-specified search parameters, a list content items that even do not need to be reachable using normal *Browse* requests. This opens the possibility to include web-based media platforms such as YouTube, Google Video or Last.fm, if an appropriate source class is implemented.

5.1.6 Intelligent transcoding engine

The translator framework (Section 3.1.2 currently has only a few translators that became necessary during the development of `rtpserver` . The ruleset which defines the pipeline for certain source/sink combinations is also hardcoded. More translators could reencode the content or change the container using external libraries such as *ffmpeg*[2]. The ruleset could be replaced by graph algorithms that automatically determine the fastest or most quality-preserving route.

5.1.7 Rewrite RTP library with error correction

The currently used RTP-with-AHEC[21], *libccrtp-ahec*, is an older fork of the GNU libccrtp. This library was originally designed with voice over IP telephony in mind, and is not easy to integrate on newer Linux distributions[3]. A new library designed from the ground up with AHEC in mind could increase stability and would be prepared for new technologies such as scalable video codecs.

[1] The Microsoft Mediaroom-based T-Home receivers need over 5 minutes to boot when powered up
[2] A powerful Free Software media codec library, available at http://www.ffmpeg.org/
[3] It was not possible to build the latest *libccrtp-ahec* on Ubuntu 9.04

Bibliography

[1] S. Cheshire, B. Aboba, and E. Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927 (Proposed Standard), May 2005. http://www.ietf.org/rfc/rfc3927.txt.

[2] J. Cohen and S. Aggarwal. General Event Notification Architecture Base. IETF Draft (expired), July 1998. http://quimby.gnus.org/internet-drafts/draft-cohen-gena-p-base-01.txt.

[3] Contributing Members of the UPnP Forum. AVTransport:1 Service Template Version 1.01, June 2002.

[4] Contributing Members of the UPnP Forum. ConnectionManager:1 Service Template Version 1.01, June 2002.

[5] Contributing Members of the UPnP Forum. ContentDirectory:1 Service Template Version 1.01, June 2002.

[6] Contributing Members of the UPnP Forum. UPnP AV Architecture:0.83 v1.0, 2002.

[7] Contributing Members of the UPnP Forum. UPnP Device Architecture v1.0, July 2006.

[8] Digital Living Network Alliance. Digital Living Network Alliance: official web site. http://www.dlna.org.

[9] Digital Living Network Alliance. DLNA Networked Device Interoperability Guidelines, October 2006.

[10] Digital Living Network Alliance. DLNA Overview and Vision Whitepaper 2007, 2007.

[11] European Telecommunications Standards Institute. ETSI TS 102 034 V1.3.1: Digital Video Broadcasting (DVB); Transport of MPEG-2 TS Based DVB Services over IP Based Networks, 2007.

[12] Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu, and Shivaun Albright. Simple Service Discovery Protocol/1.0 Operating without an Arbiter. IETF Draft (expired), October 1999. ftp://ftp.pwg.org/pub/pwg/ipp/new_SSDP/draft-cai-ssdp-v1-03.txt.

[13] Jochen Grün. Study Thesis: Implementierung eines DVB Digital Media Renderers, November 2008.

[14] M. Handley and V. Jacobson. SDP: Session Description Protocol, April 1998. http://www.ietf.org/rfc/rfc2327.txt.

[15] Intel Corporation. Designing a UPnP AV MediaServer, 2003.

[16] ISO/IEC. ISO/IEC 13818-1: Information technology — Generic coding of moving pictures and associated audio information: Systems, December 2000.

[17] Michael Jeronimo and Jack Weast. *UPnP Design by Example*. Intel Press, April 2003.

[18] Klaus Schmidinger. Video Disk Recorder. `http://www.cadsoft.de/vdr`.

[19] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, July 2003. `http://www.ietf.org/rfc/rfc3550.txt`.

[20] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol, April 1998. `http://www.ietf.org/rfc/rfc2326.txt`.

[21] Guoping Tan and Thorsten Herfet. Optimization of an RTP Level Hybrid Error Correction Scheme for DVB Systems Over Wireless Home Networks Under Restrict Delay Constraint. In *IEEE Transactions on Broadcasting, Vol 53, Issue 1, Part 2*, pages 297–307. IEEE, March 2007.

[22] The DVB Project. DVB-HN (Home Network) Reference Model Phase 1. `http://www.dvb.org/technology/standards/a109.tm3690r2.DVB-HN_ref_model.pdf`, February 2007.