A MythTV Python API to Complement the JustPlay Network

by

Christian Deonier

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2008

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by .....
Stephen A. Ward
Professor, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A MythTV Python API to Complement the JustPlay Network

by

Christian Deonier

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I developed an API to control a MythTV backend. This API is called PyMythTV. It allows one to develop software that can take advantage of a PVR device, MythTV. The API was written in Python, which allows for an easy integration of the device into the JustPlay network. With this service in place, people can easily search and schedule recordings while on the network. I also developed a program to demonstrate the functionality of the API and how it could be used. I also wrote documentation describing the API. A developer could easily use this simple and flexible API to control a MythTV back-end to fulfill his needs.

Thesis Supervisor: Stephen A. Ward
Title: Professor, Department of Electrical Engineering and Computer Science

Acknowledgments

I would first like to thank all the members of the group that helped me in this project. Steve Ward, Hubert Pham, Justin Paluska are amazing– They were always willing to help me sort things out and help me figure out the tough problems. Thank you so much for your guidance and your wisdom, there is no way I could have done this without your help.

I'd like to thank my friends who supported me through this endeavor. Jin, thank you for your constant understanding and advice. Rich, you brought me out from the depths. Tri, you provided me with such inspiration, I could persevere by following your example. Shirley, you gave me hope when I thought all was lost.

Anne Hunter, I very much doubt I could have navigated this minefield without your help. Thank you for always being on my side.

Last, but certainly not least, I'd like to thank my family. Rachel, your unconditional support was and is always what keeps me going. Mom and Dad, you gave me so much advice and support (both emotionally... and financially!) that you made my dreams come true. I hope I made you guys proud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Personal Video Recorder (PVR) is a device to search for and schedule recordings for television shows. The ease-of-use provides a user with a hassle free way to manage all his television recordings. It is becoming increasingly popular [2], and both commercial and homebrew solutions are integrating into a homeowner's standard arsenal of devices.

A PVR is a perfect device to be integrated into the JustPlay network. The goal of the JustPlay network is to provide a user an easy way to control his devices. In the network, the user does not need to configure anything, he has easy access to to the device, and the device is integrated with a multitude of other devices without having to create a jungle of wires. A PVR is certainly one potential device to contribute to the JustPlay network: It's popular among many people, integration into the JustPlay network would simplify using the device, and its rich media of videos and music would allow for useful interaction with other devices, like a television or speakers.

This thesis describes the work that I did when creating a Python module to control a PVR. The API allows searching and recording by using a system of unique IDs, which gives the developer flexibility and clarity when programming with the API. The ultimate goal of the project was to create a Python module API that allows a developer to search for programs and schedule recordings in a simple fashion. This module is called "PyMythTV".

1.1 The JustPlay Project

To get a better perspective of how PyMythTV works and the scope and goals of the project, it's beneficial to get a closer look of the JustPlay network. The JustPlay network is a response to the growing problem of an increasing amount of devices in the home with no easy and suitable way to get them all configured and interacting with each other. The goal of the JustPlay network is to create an environment that's friendly to the user: easy access to devices, easy installation, virtually no configuration, and free of wires.

JustPlay achieves this by having an O2S hub (Figure 1-1) that provides the necessary configuration for each device. When one brings a device into the JustPlay network, an O2S client, the client looks for an O2S registry on the local network. The device then registers with the O2S registry and becomes configured so the user can use the device. The user could initiate some request which then translates into goals, and devices would then attempt accomplish the request if they could fulfill those goals.

There are two layers to the O2S system: A planning layer and a component layer. The planning layer deals with the user inputs and translating them to goals. The component layer deals with monitoring the devices and making them available for use. Separating the two layers is an abstraction layer (Figure 1-2) composed of many NPOPs, or Network Portable Object Packages. NPOPs are lightweight, platform-independent objects that shield the developer from having to deal with details specific to the device. One can think of them as providing a service to use the device. Whenever one wants to use a device, one can use the NPOP to access the devices features without knowing specifics of the device.

1.2 Problem Scope and Goals

My goal and thesis was to create the necessary API for an NPOP in the JustPlay network. The result provides a developer a way to use a PVR device, for tasks such

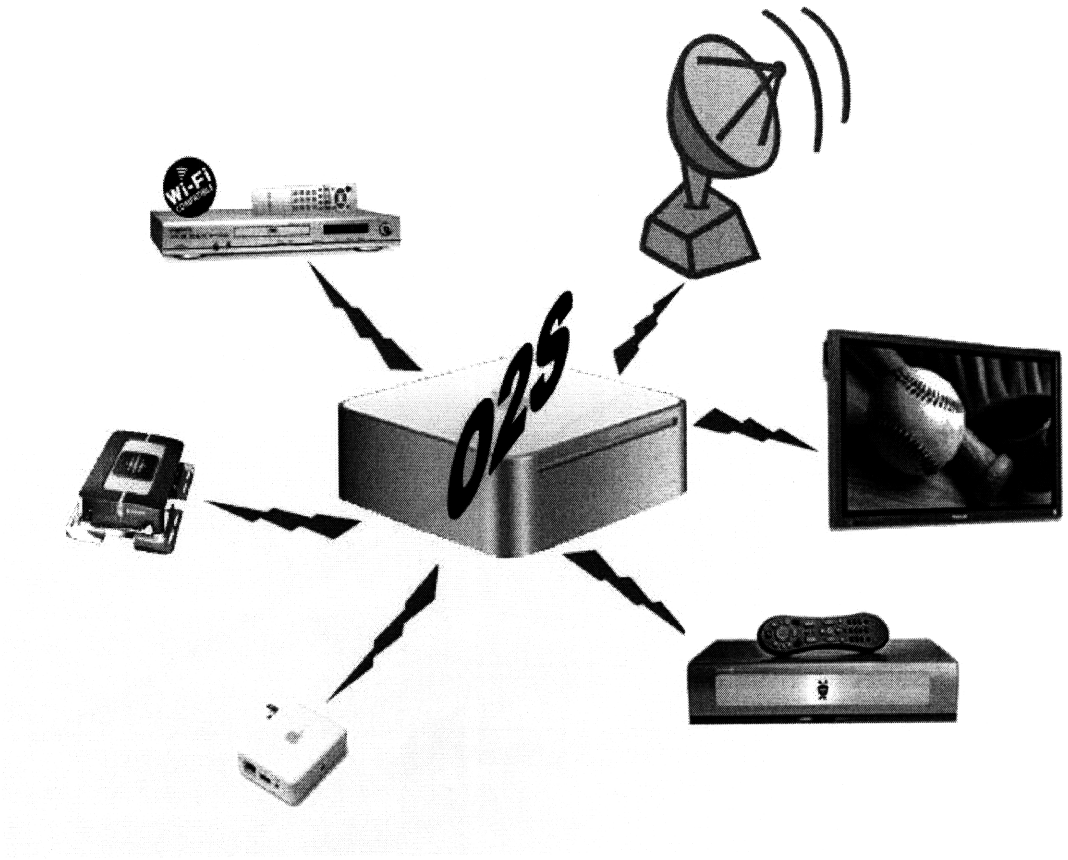

Figure 1-1: O2S acting as a hub. [9]

as searching or recording programs. The API needed to be easy to use and slim, meaning a small number of flexible methods.

There has already been a lot of work done creating other services for devices, and the API had to be similar. Because previous work related to JustPlay was coded in Python, this API had to be similarly coded in Python to maintain consistency. Also, the PVR that I incorporated into the network had to be open-source, so I could easily create an API for it. It's better for the PVR to be on a Linux platform too, for easier development and integration.

To summarize, the parameters were as follows:

1. The API should allow the searching for and recording of programs on a PVR.

2. The API should be coded in Python.

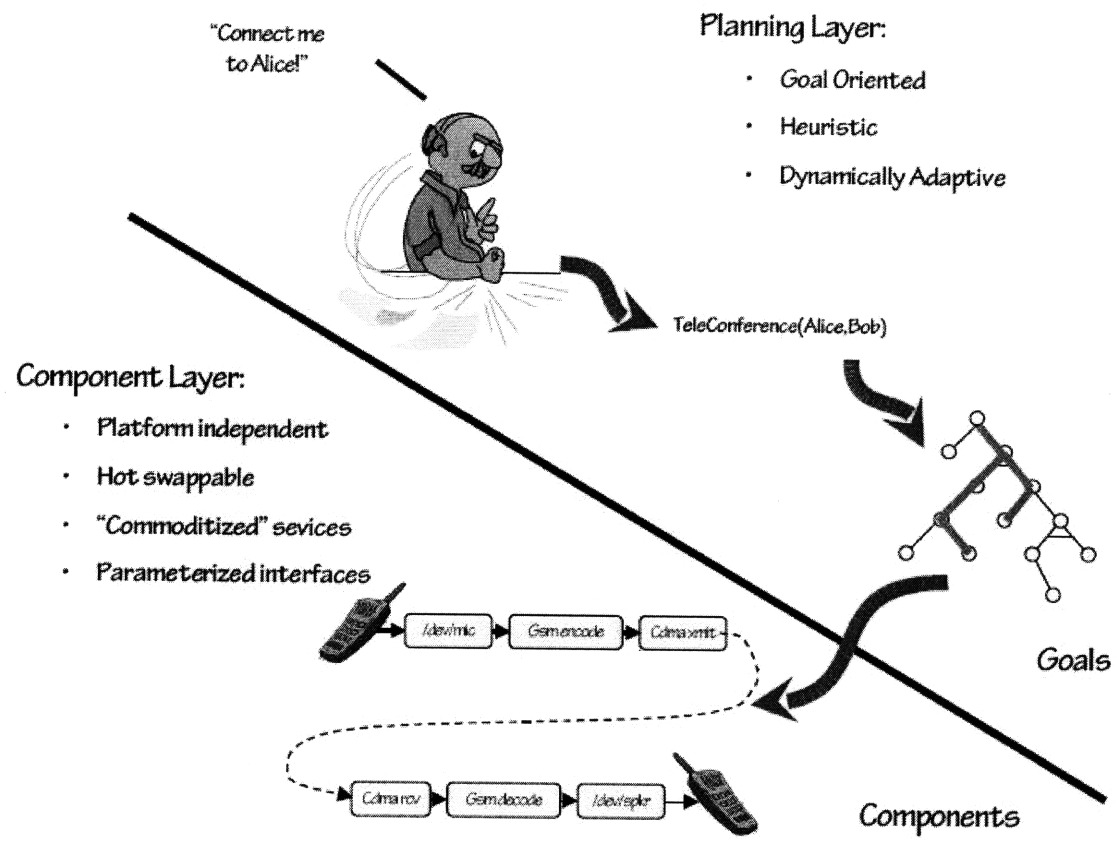

Figure 1-2: The JustPlay abstraction barrier. [9]

3. The PVR should be an open-source solution to easily build off of existing code.

4. A Linux-based PVR is desirable.

It was also useful to create documentation for the API, as well as a program that demonstrates the use of the API. Documentation was written both as an HTML format and in a Python document. The program demonstrating the possible use of the API was also written in Python.

This thesis describes how I implemented an API in Python that allows a developer to control a MythTV backend server in a simple way. The thesis also describes a sample Python application that uses the API.

1.3 Organization of Thesis

The thesis is organized as follows:

1. The first chapter provides the introduction of the thesis and provides background.

2. The second chapter describes related work to the thesis.

3. The third chapter describes preliminary research done before implementing the API.

4. The fourth chapter describes the high level API design and defines the API.

5. The fifth chapter describes the API implementation in more detail.

6. The sixth chapter describes the sample application using the API.

7. The seventh chapter concludes the thesis and discusses the results and possible future directions.

Chapter 2

Related Works

This section is an overview of previous and related work to my thesis project. Each section describes a different project, and its strengths and weaknesses. It also has a brief conclusion of what I learned from the project, as well as what I could use from it to apply it to my own thesis project. I look at each project to see whether it would be easy to develop for, whether it was based on Linux, and whether it had the features that I would need to fulfill my goals for the project.

2.1 EtiVo

EtiVo is not a PVR, but rather a service that builds off of an existing TiVo PVR. It was the first potential project that I explored. The main goal behind EtiVo is to create an efficient way to archive TV shows that one has recorded. It uses a TiVO PVR to record the shows in a ".tivo" file, and then a Windows computer copies the file and encodes it using the Windows Media Encoder to reduce the file size[6].

An obvious advantage to using EtiVo is the reduction in file size. The encoding process takes a typical hour of video that is one gigabyte is size, and reduces it down to one hundred megabytes[5]. This is a definite advantage in minimizing the network traffic when sending the stored video from the PVR to the device that is playing the video on the JustPlay network.

The downside to EtiVo is that one needs a TiVo box to record the shows, and one

needs to run Windows to run EtiVo service. TiVo didn't seem particularly easy to develop for, as it is not an open source and is created by TiVo, Inc. Also, it wasn't clear that having the files encoded with the Windows Media Encoder was the correct step, because I may have needed a different format for the video.

I decided not to continue to explore using EtiVo because of its requirements of using Windows and a TiVo recorder. In general, an open-source solution is preferable, and I concluded that further development on this project would probably be difficult. The project didn't seem to have a lot of developers working on it because it was managed by only one developer, Shahar Prish.

2.2 SageTV

SageTV is a PVR solution that supports Mac, Windows and Linux. SageTV, LLC[8] develops and maintains SageTV. SageTV requires one to have a host PC with a TV tuner card. This means that one does not have to have a TiVo box to record, and instead a normal PC would handle all the recordings.

SageTV seemed to have all the features that I wanted. It would have been flexible in case I needed the PVR to do tasks that I couldn't foresee when starting out. Also, it seemed to be able to support multiple formats, including Linux, which was a definite advantage. In all, SageTV seemed to be a complete package that would be very nice to have if I could easily develop on it and control it.

Unfortunately, I discovered rather quickly that SageTV wasnt going to be a project that I could build off of. SageTV is propriety and closed-source. While I could probably puzzle out how to develop for it, and develop a hack to get it to work, there was probably going to be a better project to build off of. Open-source software seemed to be a necessary requirement for easy development of an API.

SageTV was promising, but once again, development on it seemed problematic, and ultimately led to me not selecting it. It had the features I wanted and supported a Linux platform. I really needed a project that had an active development community that I could learn from. I also needed a project that was open-source so I could easily

code an API for it.

2.3 Freevo

Freevo [4] is an open-source PVR similar to MythTV. Freevo uses a PC as a backend, which is responsible for scheduling and managing recordings. Freevo does not need a TiVo to run, and runs on multiple platforms, including Linux. Freevo was a very competitive choice with MythTV when considering which platform to develop for.

In fact, MythTV and Freevo are very similar in what they can offer. They both support Linux, both are open-source, both have many features, and both have many developers working on them. Freevo is a very flexible system that would be easy to extend.

Freevo does not really have any downsides that many of the other previous possibilities had. The real choice was between Freevo and MythTV, and the difference of what they could offer was slight. I decided not to go with Freevo because it seemed to be not as mature a project as MythTV[10], so I might have had to contend with more bugs. Additionally, MythTV seemed to offer slightly more features than Freevo. And finally, the progress made with XBMC, described later, made a compelling case to work with MythTV over Freevo.

Freevo is flexible and open-source, and was a very promising PVR. It supported Linux, and seemed to be easy to develop for. MythTV seemed to be slightly more advantageous than Freevo, leading me to not select Freevo as the platform I would research on.

2.4 MythTV

MythTV [7] provides a PVR that is Linux based, and was ultimately the PVR I chose to further develop on.

The back-end runs only on a Linux platform, but various front-ends exist that can run on other platforms. For instance, a front-end exists that runs on OS X. MythTV

is advantageous to work on because it is open-source, unlike most PVR solutions. This allows for easy adaptation and extension of the service, which is an important trait for the scope of this project. It also had many promising features that could be adapted.

A downside to working with MythTV is the relatively limited documentation. The creators of MythTV are reluctant to spend time working on documentation, so most efforts done to document protocols are often done by other people reverse-engineering it. The lack of formal documentation means that time will often have to be spent deciphering how MythTV actually works in order to make sure applications programmed for it will work.

Despite its limitations, MythTV provides an excellent PVR foundation to research on. It's a mature program, which means one doesn't have to deal with many bugs that may exist in a new program. Protocol that is not clear can be clarified by looking at the source code. In addition, MythTV also meets the requirement that it is Linux-based, an important aspect to make sure integration into the JustPlay network is easy. These reasons led me to select MythTV as my PVR to extend.

2.5 Xbox Media Center

Xbox Media Center (XBMC) is not a PVR, but rather a project designed to use an Xbox to manage media such as pictures, videos, and audio files[11]. Unlike TiVo or a PC that the other projects use, XBMC instead uses a standard Microsoft Xbox as its hardware. XBMC is an open-source project that offers services ranging from checking local weather to displaying audio visualizations.

One such service that had been developed was a MythTV front-end. The Xbox has the CPU and video capability to display video. In effect, the Xbox could receive video from a MythTV back-end PC, and act as a front-end to browse and display recordings. The code for XBMC MythTV front-end was also open-source, which was a great benefit.

Though the service is not a complete PVR, XBMC was immeasurably helpful for

the research. For one, the XBMC front-end was coded in Python, demonstrating that MythTV could easily be accessed with a Python module. Additionally, the open-source code allowed me access to some functions to handle some of the more tedious parts of interacting with the MythTV backend. The project also provided some insight into the internals of the MythTV back-end, which proved beneficial later when I actually needed to start modifying data inside the back-end.

Overall, XBMC was very influential to my thesis work. Not only did it convince me that MythTV was a very viable choice to research, it also laid some foundation for my research to build off of. Examining the code provided me with a greater understanding how MythTV worked, a real boon given MythTV's relatively sparse documentation. The project also showed that my research could potentially be interesting to other parties because the project's work was in the same area as mine.

Chapter 3

MythTV

This section describes MythTV in greater detail. Understanding how MythTV works was important for me to be able to write an API that interacts with its back-end. The section first describes how MythTV is structured, and continues to describe how schedules are kept in MythTV, and then concludes by describing the recording process. While there are other subtleties to MythTV, these are the main areas of interest because these had to be factored in when designing my API.

3.1 MythTV Structure

MythTV is divided into two sections: a front-end and a back-end. It's pretty similar to a typical client/server architecture. In this case, the back-end is providing a service for the front-end. The back-end controls all of the data, including recordings and scheduling information. The front-end is responsible for displaying the media to the user, and providing a simple interface for the user to schedule recordings. A simple overview of this is show in Figure 3-1.

The back-end requires a Linux platform to run. In my particular case, I used Ubuntu Feisty Fawn to run MythTV, which I got from the Synaptic Package Manager. The back-end maintains a schedule of what to record and when. The back-end is also responsible for controlling all of the hardware necessary to record, e.g. the TV tuner card.

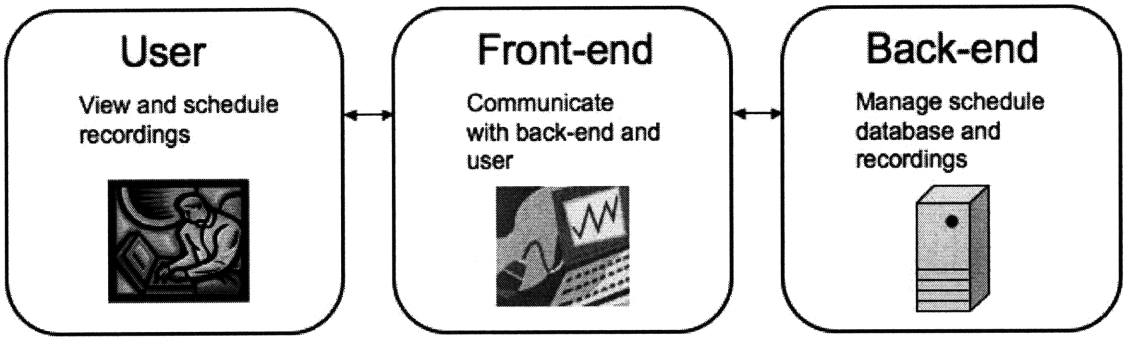

Figure 3-1: The MythTV structure.

The front-end (Figure 3-2) is not limited to a Linux platform. Front-ends have been developed for OS X, and to a much lesser extent, Windows. When the user decides to use MythTV to watch something, either a recording or live TV, the front-end connects to the backend, and streams the recording from the back-end to the front-end. This is true even for live TV. When the user watches live TV on a front-end, he is actually seeing a recording in progress. Because everything is a recording, this allows the user to easily pause, rewind, or fast-forward whatever he is watching.

I used Wireshark, a packet sniffer, to determine how information is sent between the front-end and back-end. The front-end and back-end communicate primarily through three ports (Figure 3-3). The front-end has a port that it sends commands through to the back-end. For example, when the user initiates live TV, the front-end will send a command START_RECORDING to signal the back-end to start recording with a TV tuner card. Data is managed through the two other ports. On one port, the front-end will request data of a certain size, and the back-end sends the data through the remaining third port. I also used what documentation about the MythTV protocol [3] to figure out what was going on.

3.2 Schedule Information

Most of the interesting information in the MythTV backend is managed by a MySQL database. The database holds various recordings schedules and information about programs that are going to air, and information about recordings that MythTV is

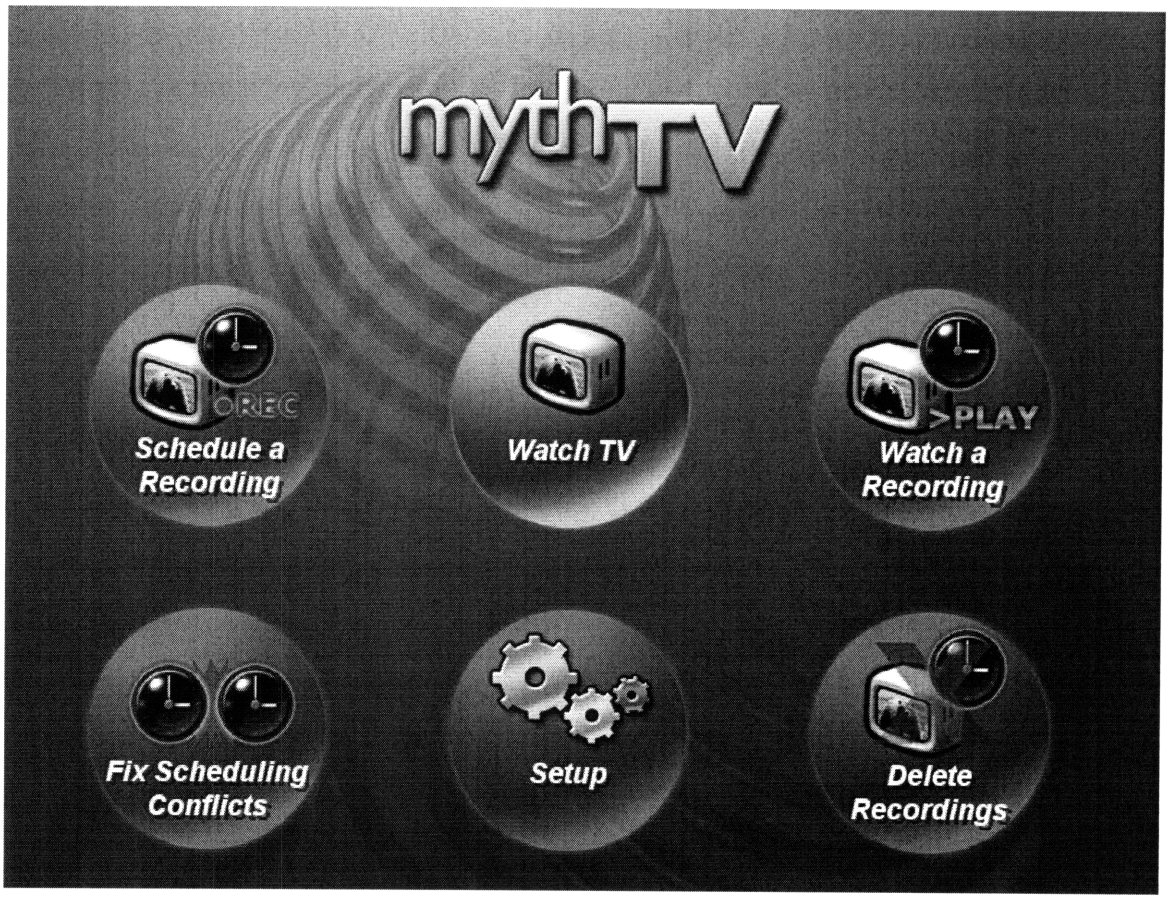

Figure 3-2: The MythTV front-end. [1]

managing. Understanding how the information is stored in MythTV allowed me to get access to the information for searching and modify the database to suit my own needs for recording.

The database in the MythTV back-end contains information about all the programs that are scheduled to be on TV within a certain amount of time. MythTV has a database entry for each program, consisting of several attributes: title, show time, description, and so on. Typically, the amount of information it has covers about two weeks, and for a typical cable line-up like MIT Cable, this amounts to about eleven thousand entries.

One of the interesting challenges of the thesis was to search all this information. Ideally, a developer wants a specific show that matches certain criteria. For instance, he may want a showing of "The Cosby Show" that airs next Tuesday. One of the goals of the API was to help the developer search through all the program listings,

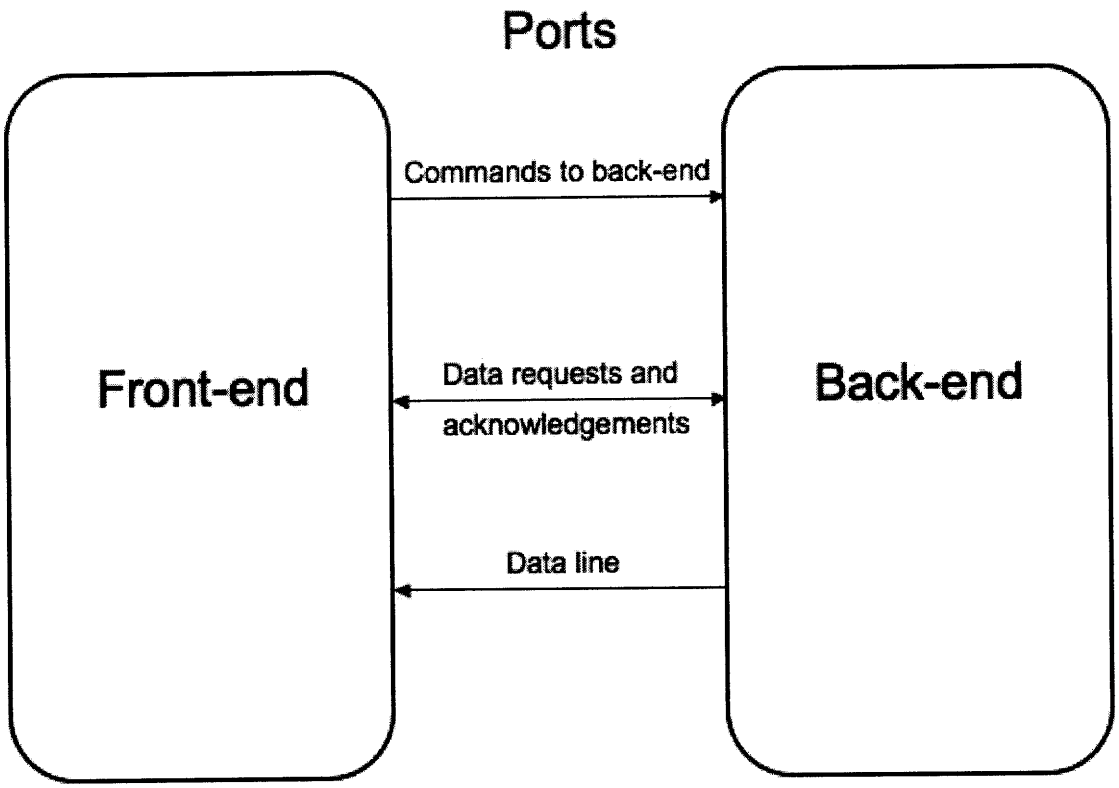

Figure 3-3: The MythTV port communication overview.

and receive results based on parameters he inputs.

The results from the search could prove interesting by themselves, but the developer probably also wants to schedule recordings based on the results from his search. It turns out that the MySQL database that MythTV manages contains recording schedules in addition to the program listings. Thus, one can schedule a recording by modifying the database and adding a new entry in the recording schedule table. This process is described in greater detail in the following section.

3.3 Recording Process

MythTV's MySQL database has several tables of information it keeps track of, for example, a table of all the channel IDs or a table for all the program listings. However, there are three tables that are important for the API: the program table, the record table, and the recorded table.

The program table contains all the program listings. An example of an entry in the program table is show below in Table 3.1. The program listings have information relevant to a specific program that is going to air, including the title and the show time. These TV listings are acquired through an outside service. Previously, a company called Zap2It maintained the listings, but Schedule Direct supplies the listings after Zap2It stopped their service. They provide MythTV with an XMLTV file, a format based off of XML. This file is then used to populate the database for all of the listings for a short time period.

Table 3.1: MySQL Program Table

Attribute	Value
'subtitle'	'Truth & Consequences'
'airdate'	0
'colorcode'	
'originalairdate'	date(2007, 11, 26)
'closecaptioned'	1
'partnumber'	0L
'title_pronounce'	
'category'	'Drama'
'title'	'Heroes'
'generic'	0
'manualid'	0L
'category_type'	'series'
'stars'	0.0
'description'	"Peter tries to destroy the virus..."
'parttotal'	0L
'subtitled'	0
'listingsource'	0L
'hdtv'	0
'previouslyshown'	0
'endtime'	datetime(2007, 11, 26, 22, 1)
'seriesid'	'EP00848361'
'stereo'	1
'showtype'	'Series'
'last'	1
'chanid'	1026L
'programid'	'EP008483610035'
'syndicatedepisodenumber'	
'starttime'	datetime(2007, 11, 26, 21, 0)
'first'	1

The record table contains all of the recording schedule information. See Table 3.2 for an example entry in the record table. MythTV knows what to record based on what schedules are in the record table. While the record table and the program table may have similar attributes, such as the title of the program, record also maintains additional information necessary for recording, such as whether it's a single or weekly recording.

The recorded table keeps track of all the recordings that MythTV has done. See Table 3.3 for an example entry in the recorded table. Each entry has similar attributes to program and record table entries, such as title, and the show time. Each entry also has differing attributes such as the file name and file size.

In effect, these three tables define a particular flow to the whole recording process. One starts out with just the program listings in the program table, which has all the information about a show you may want to record. If one decides to record a particular show, then one goes through the process of setting up a recording schedule, which is then placed in the record table. Once the program actually airs on TV and MythTV has finished recording the program, then an entry is added to the recorded table to signify an additional recording into the MythTV library. The flow is summarized in Figure 3-4.

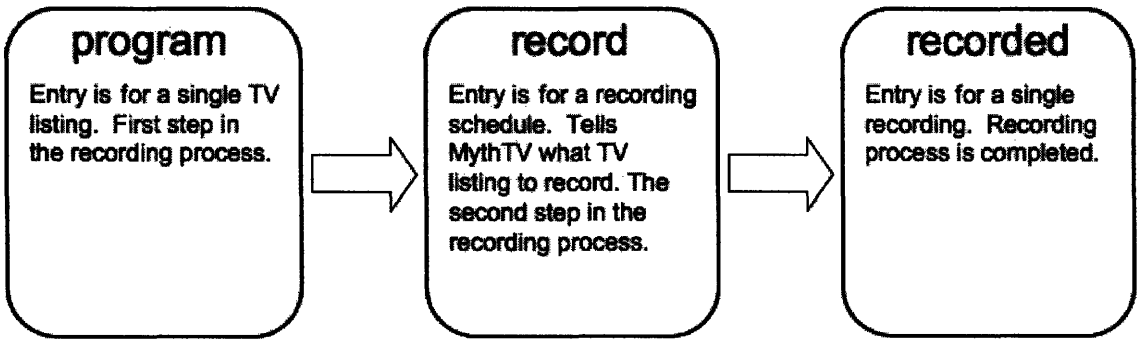

Figure 3-4: The MythTV recording process.

This process is hidden from a typical MythTV user. Their interaction usually consists of seeing some graphical display of all the possible program they can watch, and then selecting which they want to record, and then viewing the recording in

a library at a later time. However, to implement the API, I found that I had to understand how these tables worked and how to manipulate them to achieve the tasks that I wanted to do.

In general, this seemed to be the way that most projects that interact with MythTV end up doing. XBMC MythTV changed scheduling information through a Python interface while MythWeb did it through Perl. MythWeb was a web front-end to MythTV, which proved very useful for testing and debugging my own API.

The MySQL database allows control of MythTV. Because the tasks that I wanted to implement to control MythTV could be done through MySQL, I did not have to resort to trying to use low-level commands were both crytic and confusing. This was very fortunate considering there is not much documentation describing these commands, other than cursory attempts made by other developers from deciphering them. Instead, I could rely on simple MySQL commands and query and update the database to get my tasks done. With this knowledge in mind, I could then commence to design my API to start controlling MythTV.

Table 3.2: MySQL Record Table

Attribute	Value
'tsdefault'	1.0
'subtitle'	'Four Months Later'
'recordid'	10076L
'maxepisodes'	0L
'transcoder'	0L
'maxnewest'	0L
'inactive'	0
'last_record'	datetime(2007, 11, 26, 21, 0, 2)
'findtime'	timedelta(0, 75600)
'dupmethod'	6L
'category'	'Drama'
'startdate'	date(2007, 9, 24)
'dupin'	15L
'endoffset'	0L
'autotranscode	: 0
'next_record'	None
'station'	'WHDH'
'playgroup'	'Default'
'startoffset'	0L
'findday'	2
'type'	5L
'last_delete'	datetime(2007, 11, 26, 1, 55, 44)
'profile'	'Default'
'findid'	733309L
'description'	"Showdowns with Sylar..."
'prefinput'	0L
'parentid'	0L
'endtime'	timedelta(0, 79200)
'recpriority'	0L
'seriesid'	'EP00848361'
'search'	0L
'recgroup'	'Default'
'enddate'	date(2007, 9, 24)
'autoexpire'	1L
'title'	'Heroes'
'chanid'	1026L
'programid'	'EP008483610026'
'autocommflag'	1
'starttime'	timedelta(0, 75600)

Table 3.3: MySQL Recorded Table

Attribute	Value
'profile'	'Default'
'subtitle'	'Truth & Consequences'
'recordid'	10076L
'basename'	'1026_20071126210000.mpg'
'transcoder'	0L
'cutlist'	0
'originalairdate'	date(2007, 11, 26)
'watched'	0
'transcoded'	0
'playgroup'	'Default'
'category'	'Drama'
'progstart'	datetime(2007, 11, 26, 21, 0)
'endtime'	datetime(2007, 11, 26, 22, 1)
'title'	'Heroes'
'bookmark'	0
'hostname'	'christian-desktop'
'duplicate'	1
'progend'	datetime(2007, 11, 26, 22, 1)
'filesize'	2372444160L
'stars'	0.0
'preserve'	0
'findid'	0L
'description'	"Peter tries to destroy..."
'lastmodified'	datetime(2007, 11, 26, 22, 37, 36)
'timestretch'	1.0
'deletepending'	0
'previouslyshown'	0
'editing'	0L
'recpriority'	0L
'seriesid'	'EP00848361'
'commflagged'	1L
'recgroup'	'Default'
'autoexpire'	1L
'chanid'	1026L
'programid'	'EP008483610035'
'starttime'	datetime(2007, 11, 26, 21, 0)

Chapter 4

High Level Design

4.1 API Design

4.1.1 Goal

There were several goals that I tried to achieve when creating my API. Achieving a set of specific goals would yield a solid and flexible API that would be useful to developers. I wanted the API to be clear and easy to use.

My first goal was that I wanted it to be relatively simple. Simplicity helps because the developer is not deluged with a huge amount of methods he has to memorize. Instead, he would have a small number of powerful methods. When originally drafting my API, I had about fifteen methods in the API. However, I was later able to cut it down to four methods. I was able to achieve this by including more parameters in each method. The effect was each remaining function became more flexible and powerful, as opposed to doing one specific task. For instance, instead of having a method that just searched for titles, and another method that just search for categories, I could consolidate them into a general search function with title and category as some of the parameters.

One of the challenges was keeping a small and flexible API while still achieving the control I wanted over the MythTV back-end. Specifically, I broke down control in two distinct areas: one area was allowing the developer to access and search for

information about programs and recordings, and the second area was allowing the developer to act on that information and schedule recordings with the information he got from searching.

4.1.2 API Definition

The API reflects the previously stated goals by supplying four methods:

1. **search** - A method to search for recordings based on various parameters.
2. **query_id** - A method to get more information about a specific ID.
3. **record** - A method to schedule a recording.
4. **cancel_recording** - A method to cancel a previously scheduled recording.

Usage is specified in Section 5.

4.1.3 The Unique ID System

I found rather quickly that a system needed to be in place to manage all of the information being passed around. Specifically, the problem was how to refer to all of these different programs. By the very nature of the TV programs, using a simple scheme of using the program name would prove ineffective. What is there to separate one episode of The Cosby Show from another episode? How does one handle differences between TV programs and schedules for the TV program?

The system that I settled on was creating a system of unique IDs. These IDs were how one refers to programs, recording schedules, and recordings. They are important to users of the API because the user will get these as results from searches, and as parameters to record. The nomenclature is descriptive as well. A unique ID consists of the channel ID of the program, recording schedule, or recording, and concludes with the start time. To demonstrate, consider the unique ID:

```
1044_20080101090000
```

This unique ID tells you that the object the ID is referring to is located on the channel whose channel 44 and is shown on January 1, 2008 at 9am. The 1044 is the channel ID and not the channel, but typically, the channel can be surmised by looking at the last three digits, hence channel "44". The second part of the ID is actually a compressed version of a date-time, without dashes, colons, or spaces. The start times on MythTV are usually this form: "Year-Month-Day Hour:Minute:Second". So, this showing is on "2008-01-01 09:00:00," or January 1, 2008 at 9 a.m.

To fully understand the system, one needs to look a little more closely at it. Consider, for instance, the potential quandary when you record a TV program. Suddenly, one has two things one could be referring to: either the TV program listing or the recording that one just made of the program.

However, if viewed differently, it is not a problem, merely a different state. Instead of considering a program, recording schedule, and recording as separate, they are instead different states of one TV program. Namely, there is some object that exists in one of three states: One state of not being considered for recording, one state of being scheduled to be recorded, and one state of having been recorded (Figure 4-1). Said differently, a TV program is either not going to be recorded, going to be recorded, or is recorded. Note that this is very similar to the recording process described earlier.

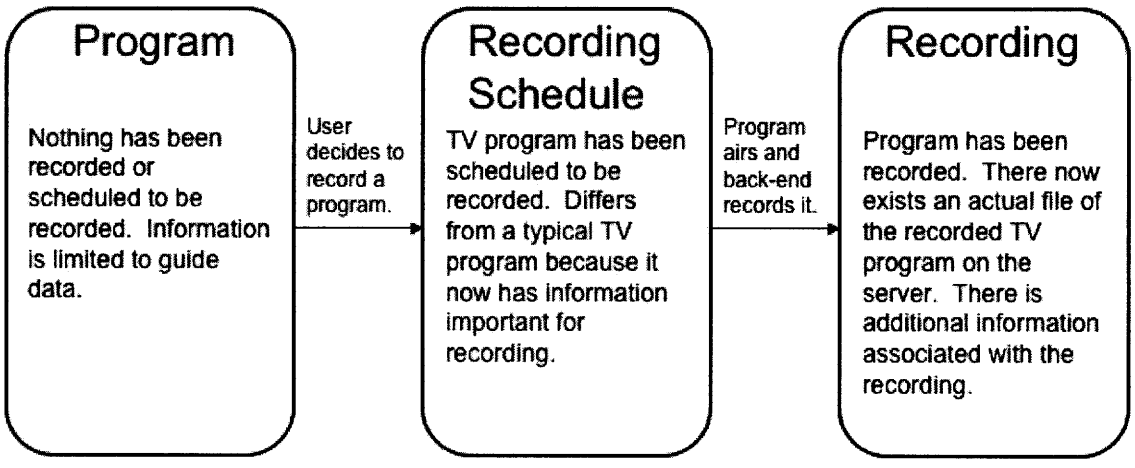

Figure 4-1: The various states of a MythTV program.

In essence, the unique ID represents one object that can transition through three

states. As it transitions through the states, the object picks up additional information. For example, if one requested information about a random TV program, one would receive information about a TV program. If one were to schedule that same TV program, and then request information relating to that unique ID, one would also get information about the recording schedule. It continues in the same fashion for recordings.

In summary, the unique ID is a way to keep track of TV programs in their various states. Searching will return unique IDs, which in turn refer to TV programs that fit the search criteria. When scheduling a recording, specifying a unique ID allows scheduling of a particular program.

4.2 API Functionality

4.2.1 Search

The API is designed to allow the developer to search for TV programs. This is particularly relevant for the JustPlay network. One of the concepts behind the JustPlay network is the user can supply a verbal command, and the network can meet the user's demand. For instance, a verbal command of "Show me 'The Simpsons' on TV" requires that the JustPlay network be intelligent enough to search through the TV listings to see if it can meet the user's demand. Allowing searches for specific programs based on a variety of parameters is one of the core abilities of the API.

I also included a way for the developer to get more information associated with a particular unique ID. Querying a unique ID yields information available about the program, recording schedule, or recording.

4.2.2 Record

The API is also designed to allow the developer to easily record TV programs. Generally, one would do a search first to get a unique ID, and then one could pass that unique ID along to the recording function in the API to schedule the recording. One

of the challenges of implementation was to make the recording process fairly automatic to the developer. This prevents him from having to learn too much about the inner workings of MythTV, and leads to a painless experience. However, I did try to allow some flexibility for the developer in case he needed to do more specific kinds of recordings that the default one would not allow.

I also included a way for the developer to cancel recordings. Like recording, he can simply pass along a unique ID, and the recording will be canceled.

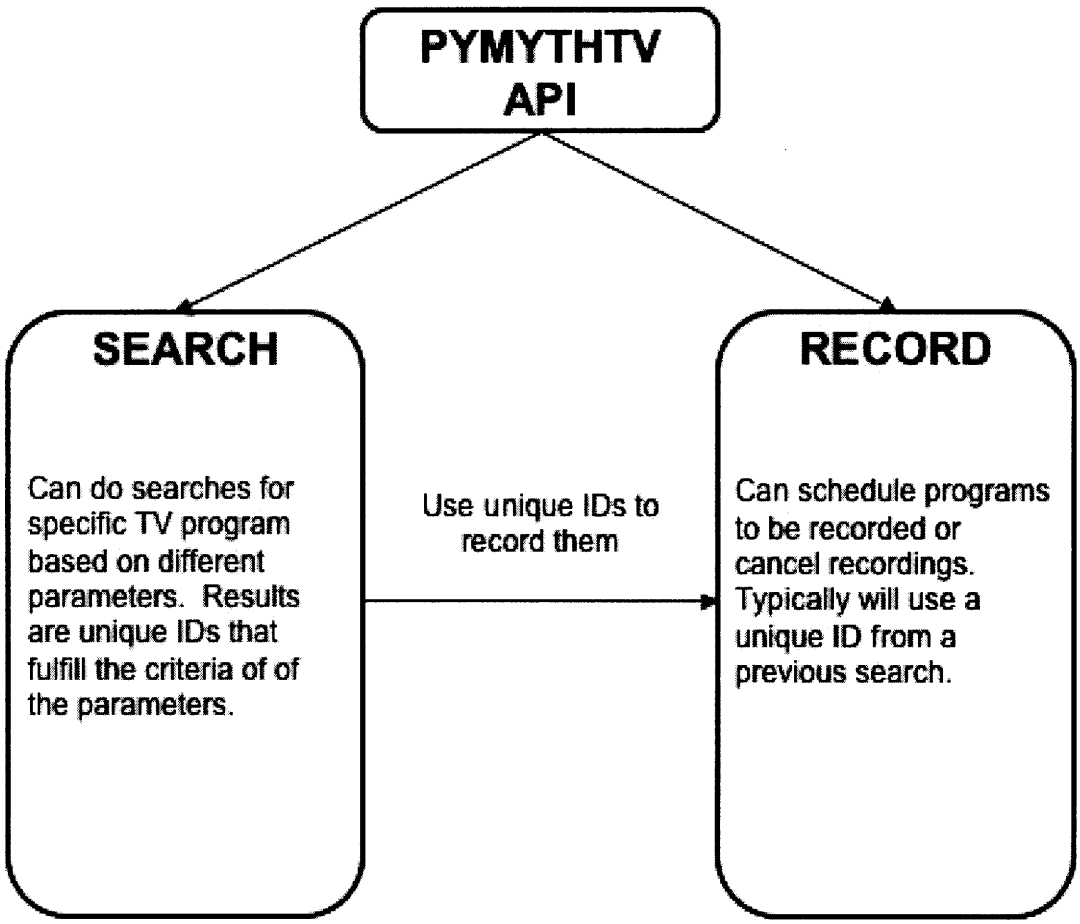

Figure 4-2: The different areas of the PyMythTV API.

A summary of the areas of the PyMythTV API are shown in Figure 4-2

Chapter 5

PyMythTV

The following section gives a more detailed description of the PyMythTV Python module that I created. I will first start off by providing descriptions and examples of how to use the API. Afterward, I describe some lower-level details that were important to creating the API. This API provides control of a MythTV back-end, allowing searching and recording of schedules.

In the next section, I will describe the four methods of the API:

1. **search**
2. **query_id**
3. **record**
4. **cancel_recording**

5.1 PyMythTV Methods

The API consists of four methods. Two methods, *search* and *query_id* deal with searching and retrieving information, and *record* and *cancel_recording* deal with scheduling recordings. The following will describe these methods in greater detail, how to use the methods, and interesting implementation details.

5.1.1 search

Description

The *search* method is designed to find programs, recording schedules, or recordings based on several different parameters. The *search* method searches through the MySQL database, finds matches, and returns the unique IDs of what matches the search criteria. The method returns a list of unique IDs that match the parameters.

The *search* method has eight parameters:

1. **search_all** (type: bool): Signifies whether one is searching for programs, recording schedules, and recordings at the same time.

   True (default) indicates one is searching in all three states at once. False indicates one is searching in only one particular state.

2. **recorded** (type: bool): Indicates which state one is searching in, if search_all is False.

   None (default) indicates one is searching for a program. False indicates one is searching for a recording schedule. True indicates one is searching for a recording.

3. **title** (type: string): Indicates one wants to search for an item based on its title. The *search* method uses substring searching by default, but can do exact matching if specified.

   None (default) indicates one does not want to search based on a title. A string indicates one wants to search for items that have the title containing the given string. A string with a "+" in front indicates one wants to search for items with an exact title matching the given string.

4. **category** (type: string): Indicates one wants to search for an item based on its category, such as comedy or drama. The *search* method uses substring searching by default, but can do exact matching if specified.

None (default) indicates one does not want to search based on a title. A string indicates one wants to search for items that have the category containing the given string. A string with a "+" in front indicates one wants to search for items with an exact category matching the given string.

5. **starttime** (type: string): Indicates one wants to search for something with a particular start time. Start times should be in for format "Year-Month-Day Hours:Minutes:Seconds", e.g., "2008-01-01 09:00:00".

   None (default) indicates one does not want to search based on a particular start time. A string indicates one wants to search for an item with a start time matching the given string.

6. **endtime** (type: string): Indicates one wants to search for something with a particular end time. End times should be in for format "Year-Month-Day Hours:Minutes:Seconds", e.g., "2008-01-01 09:00:00".

   None (default) indicates one does not want to search based on a particular end time. A string indicates one wants to search for an item with a end time matching the given string.

7. **description** (type: string): Indicates one wants to search for an item based on its description. The *search* method uses substring searching by default, but can do exact matching if specified.

   None (default) indicates one does not want to search based on the description. A string indicates one wants to search for items that have the description containing the given string. A string with a "+" in front indicates one wants to search for items with an exact description matching the given string.

8. **hdtv** (type: bool): Indicates one wants to search for an item based on whether or not it is HDTV.

   None (default) indicates one does not want to search based on HDTV. False indicates one wants items that are not HDTV. True indicates one wants items that are HDTV.

Usage

After creating a Myth object with PyMythTV, one can begin using the *search* method to start returning results. The result is a list of unique IDs. If the list is empty, then there are no matches based on the parameters. One can specify more than one parameter. The following are examples demonstrating the use of the method, assuming one has a Myth object, "m":

```
m.search(title="Family Guy")
```

returns a list of unique IDs representing programs that have the title containing "Family Guy".

```
m.search(title="Family Guy", hdtv=True)
```

returns a list of unique IDs representing programs that have the title containing "Family Guy" that are in HDTV.

```
m.search(recorded=True, title="Family Guy", hdtv=True)
```

returns a list of unique IDs representing recordings that have the title containing "Family Guy" that are in HDTV.

```
m.search(search_all=True, starttime="2008-01-01 09:00:00")
```

returns a list of unique IDs representing programs, recording schedules, and recordings that start on January 1st, 2008 at 9 a.m.

5.1.2 query_id

Description

The *query_id* method is designed to retrieve information associated with a particular unique ID. For example, one may have done a search to see which programs are airing at a specific time. One can then use *query_id* on the returned unique_ids to retrieve information about each unique ID. One can then retrieve the titles of all

shows that are showing for one's given start time. Similarly, one can also retrieve information about recording schedules and recordings.

The *query_id* method returns a dictionary. The keywords in the dictionary are the attributes available describing the program.

There are two parameters for *query_id*:

1. **unique_id** (type: string): Indicates which unique ID you would like to get more information about. The returned dictionary contains all information available, meaning it might contain program information, recording schedule information, or recording information. All the information is gathered from the MySQL database.

   The string you specify is the unique ID that you would like to get more information about.

2. **query** (type: string): Designed to get additional information about the unique ID that a normal query does not provide. This information is not readily available from the MySQL database. This parameter is designed to further extend possible information retrieval. There is presently only one additional query available:

   The string "progress" generates a float value representing how much of the recording has been completed, assuming a recording is in progress.

Usage

One needs a unique ID that one wants information about before one can use this method effectively. Typically, one can get such a unique ID from the *search* method. The following are examples of how to use this method, assuming one has a Myth object, "m":

```
m.query_id("1044_20080101090000")
```

returns a dictionary of information about the unique ID "1044_20080101090000".

```
m.query_id("1044_20080101090000", query="progress")
```

returns a percentage of the completion of the recording of the unique ID "1044_20080101090000".

5.1.3 record

Description

The *record* method is designed to schedule a recording. The process is fairly automatic, and one merely needs to supply a unique ID. The method creates a schedule, submits it to the back-end, and updates the back-end without further interaction. I did allow for more control over the type of recording schedule one generates, in the eventuality that one wants to create a more custom schedule that the automatically generated one cannot satisfy. A description of the Recording_Schedule object follows the section describing how to use the *record* method.

The *record* method requires either a unique ID or a custom Recording_Schedule. One can also specify the type of recording schedule, if desired.

The *record* method returns a Recording_Schedule that it automatically generated.

There are three parameters for the *record* method:

1. **unique_id** (type: string): If one supplies a unique ID instead of a Recording_Schedule, then *record* automatically schedules the recording on the back-end and creates a Recording_Schedule.

   The string one specifies indicates the unique ID of the program one would like to record.

2. **recording_schedule** (type: Recording_Schedule): If one supplies a Recording_Schedule, then the *record* method uses the supplied Recording_Schedule to

schedule the recording, instead of automatically scheduling a recording and creating its own Recording_Schedule.

The Recording_Schedule one specifies indicates one would like to use a custom schedule instead of the default.

3. **record_type** (type: int): One can control the type of recording schedule using record_type. The record_type of a schedule is how often a schedule records something. The type can range from recording a program every week, to every day, to anytime it is on. The default record_type is a single one-time recording. To make things easier, I included several global variables so one does not have to input integers.

   The integer specified indicates the recording type one would like to do. What follows are the global variable names along with their values.

```
RECORD_ONCE = 1
RECORD_DAILY = 2
RECORD_WEEKLY = 5
FIND_AND_RECORD_ONCE = 6
FIND_AND_RECORD_DAILY = 9
FIND_AND_RECORD_WEEKLY = 10
RECORD_ANY_TIME_ON_CHANNEL = 3
RECORD_ANY_TIME = 4
```

Usage

One needs either a unique ID or Recording_Schedule object. Supplying a unique ID automatically records the program associated with the unique ID. Supplying a Recording_Schedule records whatever the Recording_Schedule specifies. The following are examples of how to use the *record* method, assuming one has a Myth object, "m":

```
m.record(unique_id="1044_20080101090000")
```

automatically schedules the back-end to record the program associated with the unique ID "1044_20080101090000". A Recording_Schedule is generated by *record* and is returned.

```
m.record(unique_id="1044_20080101090000", record_type=RECORD_WEEKLY)
```

automatically schedules the back-end to record the program associated with the unique ID "1044_20080101090000" on a weekly basis. A Recording_Schedule is generated by *record* and is returned.

If one wanted to schedule a recording based on a more complex schedules one could take the following steps:

```
sch = m.record(unique_id="1044_20080101090000")
```

returns a Recording_Schedule object, which one can save and modify. One could modify the record_type to record on a weekly basis, for instance:

```
sch.schedule[record_type] = RECORD_WEEKLY
```

```
m.record(recording_schedule=sch)
```

is called after canceling the previous recording with the *cancel_recording* method, because no unique_id can refer to two different recording schedules. The end result is the same as having called:

```
m.record(unique_id="1044_20080101090000", record_type=RECORD_WEEKLY)
```

Of course, one is not limited to only modifying the record type. One can view possible variables from the record table specified later, though this requires some knowledge of how MythTV works and how it uses those values.

Recording_Schedule

Recording_Schedule is an object for recording schedules. The actual recording schedule is no more than a dictionary, and Recording_Schedule is a wrapper for the dictionary. The Recording_Schedule object has a **schedule** variable, which is the actual recording schedule dictionary.

5.1.4 cancel_recording

Description

The *cancel_recording* method is designed to easily allow one to cancel existing recording schedules. Its a simple and automatic method. One merely supplies a unique ID, and the *cancel_recording* method cancels the recording schedule associated with that unique ID.

The *cancel_recording* method requires a unique ID.

The *cancel_recording* method does not return anything.

There is only one parameter:

1. **unique_id** (type: string): Indicates which unique IDs recording schedule one wishes to cancel.

   The string one supplies is the unique ID, and the recording schedule of that unique ID will be canceled.

Usage

Using the *cancel_recording* method is simple and automatic. One merely supplies the unique ID, and the corresponding recording schedule is canceled. The following is an example, assuming one has a Myth object, "m":

```
m.cancel_recording("1044_20080101090000")
```

cancels the recording schedule associated with the unique ID "1044_20080101090000".

5.2 API Background and Reference

The following subsections are not necessary to use the API, but are provided as reference to see how the API was created an other background details in case the user is interested. I'll first describe the low-level connection and commands to control the hardware, and then describe the MySQL connection.

5.2.1 Establishing a Connection

As mentioned earlier, one type of communication that PyMythTV does with the MythTV back-end is a low-level connection. These are commands that assume a more direct control over the MythTV back-end, such as starting and stopping actual hardware, or updating the status of the MythTV back-end.

The actual low-level connection is used sparingly in comparison with the high level MySQL connection. Anytime needs to do a particular task that requires actual modification of the back-end, one need to establish a temporary low-level connection. This low level connection is established in the API using the Python module "socket".

The API first establishes a socket between the API object and the back-end, which I found to be port 6543. After creating the connection between the two, messages can be sent back and forth. MythTV requires a rather specific sequence when first establishing a connection that allows control over the back-end. One first has to announce the connection to the backend in a specific mode, either "monitoring" or "playback" mode. The API only needed to use "monitoring" mode.

After establishing the mode, one could then send messages to the back-end to update the back-end as needed. This is necessary for scheduling recordings, because one has to notify the back-end that its database has changed.

The following commands were sufficient for the API. They are strings that are sent over a socket to control various components of the back-end on a very low level, mainly hardware. Note that these are not actual methods provided by the API, merely the commands that allow the API methods to work:

```
#      ANN Monitor [back-end] 0
#      RESCHEDULE_RECORDINGS [record-id]
```

where "#" indicates the length of the message in bytes. The length varies based on the values [back-end] and [record-id], which correspond to the MythTV back-end address and recording ID respectively. For example, I had to use the command:

```
25      ANN Monitor 18.95.2.126 0
```

each time I established a low-level connection with the MythTV back-end, and I used:

```
27      RESCHEDULE_RECORDINGS 18254
```

if I wanted to update the status of the recording schedule with the record ID 18254.

Again, a user of the API need not worry about these details, and instead use the four methods described above. I used XBMC for the core foundation of this low-level connection. They had several functions managed the connection quite well, including connecting and updating the MythTV backend.

5.2.2 MySQL

MySQL proved to be the optimal way to communicate with MythTV. The database had everything necessary to accomplish the goals of the API. Not only could one parse the database to retrieve specific TV programs from searching, but one could also modify the database to input custom schedules, and simply notify the back-end to update using low-level communication.

MySQL has many tables, including a table that associates channel IDs with the actual channel number one sees on TV; however, the most interesting are the program, record, and recorded tables. The three states of a TV program mentioned earlier correspond with these tables. TV programs that have had no action taken upon them can be accessed in the program table, recording schedules can be accessed in the record table, and information about recordings can be accessed in the recorded table.

The API tries to hide the specifics of the tables as much as possible. Instead, it relies on the user specifying the unique ID of a TV program. The searching one would do to find specific programs based on parameters would typically be done in

the program table, but one could search any of the tables by changing a parameter. Recording only alters the record table. The MythTV back-end modifies the recorded table as needed, and the API does not modify it.

Unlike the low-level connection, MySQL does not directly use the "sockets" module. Instead, it uses the MySQLdb module, which handles the connections to a MySQL database, as well as facilitating the transfer of messages to and from the database.

Again, much of the communication done was through MySQL. Access to the database meant one could search all the program listings and schedules recordings.

Chapter 6

A PyMythTV Application

The following describes my work in creating an application to use my PyMythTV API. I also wrote this application in Python. In this section, I step through the motivation of this application, describe my approach, point out implementation details, and conclude with my analysis.

6.1 Motivation

The motivation behind the application is two-fold. One is to actually test the API and see how useful it is, and the other is to demonstrate that the API can indeed work. The API by itself doesn't do anything. It needs an application to actually invoke its methods. When I designed the API, I tried to predict what sort of methods would be useful for the developer. I implemented the necessary and cut the superfluous. However, I thought it would be very useful to actually try and write an application that used my API. By writing the application, I could see if there were any methods that I need to include, and I could see if any existing methods needed improvement.

Additionally, creating an application makes a deliverable and clear demonstration. Because the API does not do anything by itself, its not as clear to see what the APIs purpose is. Seeing an application that uses the API is the best way to demonstrate how effective the API is.

6.2 Approach

My approach was to create a relatively simple application that directly invoked different aspects of my API. The most logical sort of application that used all of the different methods was a MythTV front-end. I didn't want an overly complex front-end, mainly because I wanted the focus to be on how to use the API and what it's directly capable of, instead of pursuing further extension of the API.

I decided to create a text-based front-end. Not only is it easy to implement, it saves time, and clearly demonstrates the abilities of the API. The application needed to demonstrate its ability to search and its ability to record. The following describes the implementation.

6.3 Implementation

The resulting application was very simple and typical. The interface was text-based, as show in Figure 6-1.

I ended up having a main menu with options that directly corresponded to the four methods in the PyMythTV API. The user could search, get information about unique IDs, schedule recordings, and cancel recordings.

One additional feature that I added was allowing the user to save a unique ID. This allows the user to actually see and better understand the flow from the search area of the API to the record area. It also demonstrated how the unique IDs were used to reference things, and how one could use the unique IDs.

The overall goal in the application is allowing the user to record something. The first step he is expected to do is to search for a particular program. He inputs which parameters he wants to use in his search, and a unique ID is automatically saved. He can then use another option of the program to get information about the unique ID. He can then select the recording option to record the program associated with the saved unique ID. If he so desires, he can use the final option of the program to cancel it.

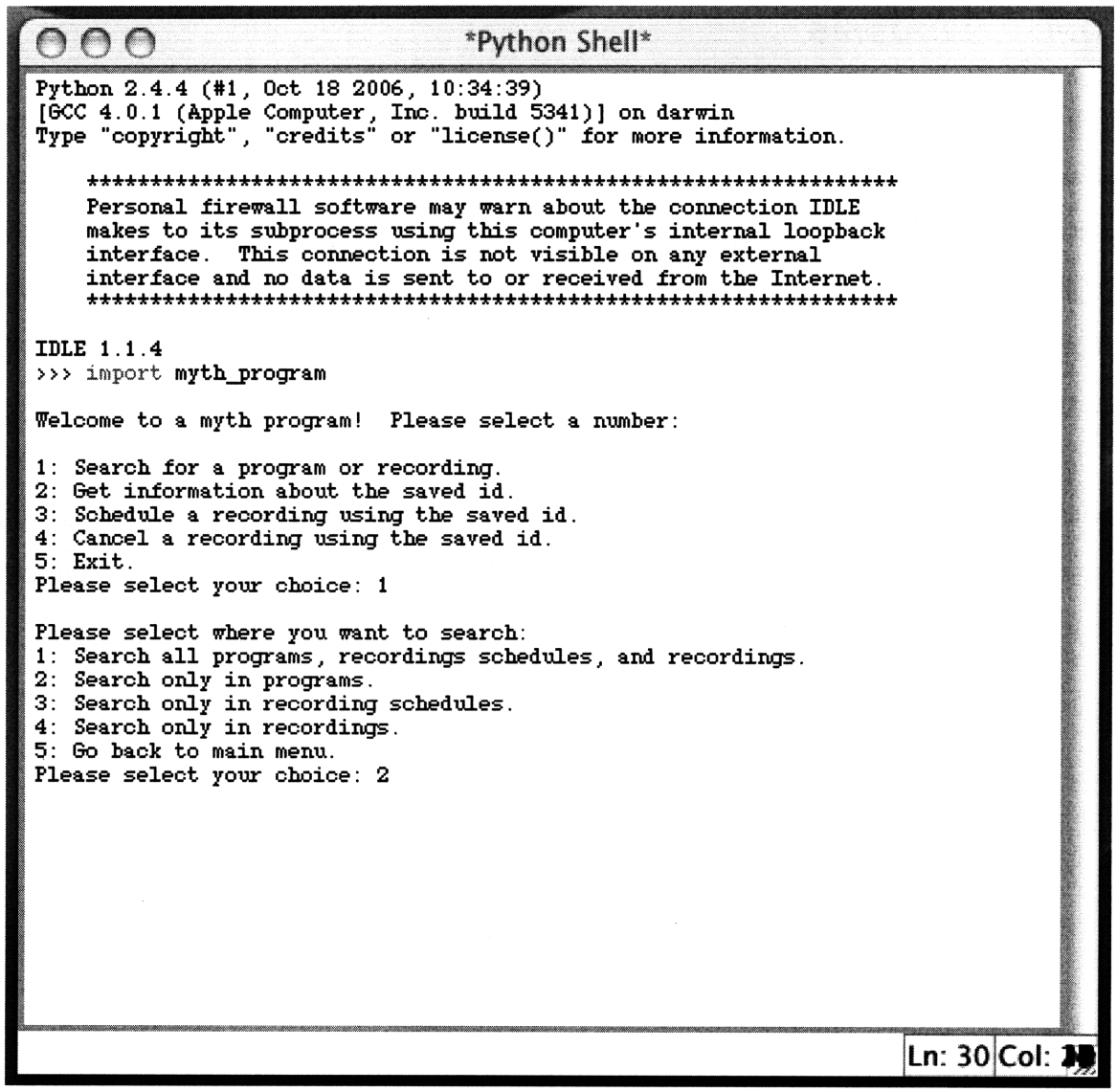

Figure 6-1: PyMythTV Application.

The implementation of the program itself was fairly simple. The program was basically a state machine, and the user could move to various states. The program loops first displayed information associated with the state, such as displaying the options available to the user or returning information the user requested. The loop then asked the user to input a command. The third and final part of the loop then did an action, which was usually an invocation of an API method, such as *record* or *search*.

6.4 Analysis

The end result of the application was successful. After having created the program, I am confident that the API is flexible enough for a developer while providing a slim set of methods facilitating development. I did not see any problems in achieving the tasks that the API is meant to address, specifically, searching and recording. I did notice that it might have been helpful to create a clear way to display information returned from querying a unique ID.

The actual implementation of the application itself was not too challenging. However, the application was designed to be simple, so a more complex application might use the API in ways I had not anticipated. I think that the API is flexible and simple enough to successfully adapt to more demanding applications.

Chapter 7

Discussion

7.1 Conclusion

PyMythTV is a very flexible and capable API. Testing showed that the API did meet its initial goals. For instance, one can search for specific TV programs, recordings or schedules very easily. The number of parameters that one can specify gives it great latitude in one's searches.

Additionally, it is very easy to use the information from searching in a useful way. One usually performs the search to find TV programs one is interested in recording. The API easily handles the transition of taking that information and allowing one to record TV programs one found from searching. The unique ID system is an easy way to keep track and refer to all of the TV programs, schedules, and recordings.

The PyMythTV application showed that the API could be used for a typical application. While the application was not particularly complex, it did demonstrate the features of the API successfully. Because of the simplicity of the API, it should be fairly simple to use the API in a more complex application.

All in all, the PyMythTV API should help out the JustPlay network a lot. It allows control of MythTV to a reasonable degree, meaning it should be very easy to incorporate a PVR into the network.

7.2 Lessons Learned

Probably the most challenging aspect of the project was not the API implementation, but rather the API design. It is difficult to fully anticipate what a developer would need from his API. It's also very easy to overwhelm the developer by creating an API that is complex to use by having a multitude of methods or unclear specifications.

Additionally, I learned how to cope with creating a part of a larger project. The JustPlay network is too large for anyone one person to do individually, and my contribution is just one part of a larger whole. This was challenging because I couldn't treat my part as stand-alone. Eventually, this API would be incorporated and used in the JustPlay network, meaning I had to be cognizant of what other people would need. Things I thought would be useful might actually be not, and things I didn't think of had to be considered. If anything, I learned how to design an API for others, and not just myself.

With regards to the implementation, I learned a lot about researching a particular project, and learning how to extend it. There wasn't a how-to guide to explain all of the details of MythTV, and learning how to find out the details I needed was an interesting and important part of the project. For instance, I learned to use Wireshark to do packet-sniffing, or learned to look at other's people code to figure out how to adapt my code to do certain tasks.

7.3 Possible Extension

The API is simple and flexible, but could probably be extended. One thing that I think might be useful is helping to present information is a better way. For instance, when the developer requests information about a particular unique ID, I present data in a rather raw format, and it's up to the developer to deal with that information. If one could find a way to anticipate better what the developer needed, the API might even further hide implementation details of MythTV.

Of the two areas, searching and recording, searching is probably the more interest-

ing to extend. Recording is pretty straightforward; one simply updates the back-end and sends various commands. Searching, however, has many directions it can go. I provided a search function that is flexible, but not very smart. That is, it does not really automate things for the developer in ways that might be helpful. When one uses the search function, one has to enter the parameters in a very specific manner.

It would be interesting if the search function could search more intelligently. For instance, perhaps instead of searching only based on the attributes, one could better interpret the needs of the person invoking it to provide an intelligent solution. For instance, perhaps there could be some way that the search function analyzes how it is being used, and then could use the knowledge to obtain better or faster results.

This fits in with the concept of the JustPlay network. There is already a goal-system part of the project, where the user's command is broken down into very specific goals. If the API could be adapted to better mesh with the goal system, it could provide a better service.

Another extension is generalizing the module. Presently, the module only accommodates MythTV. However, one isn't guaranteed to always use MythTV as a PVR. It would be nice if the API could be generalized to extend to a variety of PVRs. For instance, when one invokes the *search* method, perhaps the API can see what PVR it is interacting with, and invoke a specialized search for that PVR.

This generalized API could better leverage the existing hardware. For instance, suppose the user has a TiVo and a MythTV box. The API could search for results on the TiVo, and find a program to record only to discover there is no room to record the show on the TiVo box. The API could then adapt and instead schedule the recording on the MythTV box based on the results from the TiVo box.

In conclusion, the PyMythTV leads to many interesting avenues, both in extending the API and how one can use the API. The module provides a solid foundation for present use and future development.

Appendix A

PyMythTV Code

Note: Code slightly modified to fit within margins. Use for reference only.

```
import MySQLdb
import string
import socket
import random
import datetime

SEP = "[]:[]"

__author__ = "Christian Deonier <cdeonier@MIT.EDU>"

class Myth:
    """A class allowing searching and scheduling a MythTV backend."""

    db = None
    cursor = None

    def __init__(self, host, user, password, database):
        self.host = host
        self.user = user
        self.password = password
        self.database = database
        self._connect()

    ############################################################
    # External, Public API
    ############################################################

    def search( self, search_all = False,  recorded = None,
```

```
        title = None, category = None, starttime = None,
        endtime = None, description = None, hdtv = None ):

    """ The way to search the mythtv database. One supplies various
        parameters to do more specific types of searching.  Can
        specify whether one wants to search substrings or
        exact searches."""

    if search_all == True:
        li = []
        self._execute(self._sql_query( Table = None,
                                        Title = title,
                                        Category = category,
                                        Starttime = starttime,
                                        Endtime = endtime,
                                        Description = description,
                                        Hdtv = hdtv))
        schList = self._fetch()
        for x in schList:
            new_id = self._make_id(x[0], x[1])
            li.append(new_id)
        self._execute(self._sql_query( Table = False,
                                        Title = title,
                                        Category = category,
                                        Starttime = starttime,
                                        Endtime = endtime,
                                        Description = description,
                                        Hdtv = hdtv))
        schList = self._fetch()
        for x in schList:
            new_id = self._make_id(x[2], str(x[4]) +\
                                            " " + str(x[3]))
            if li.count(new_id) > 0:
                continue
            else:
                li.append(new_id)
        self._execute(self._sql_query( Table = True,
                                        Title = title,
                                        Category = category,
                                        Starttime = starttime,
                                        Endtime = endtime,
                                        Description = description,
                                        Hdtv = hdtv))
        schList = self._fetch()
        for x in schList:
```

```
            new_id = self._make_id(x[0], x[1])
            if li.count(new_id) > 0:
                continue
            else:
                li.append(new_id)
        return li
    else:
        li = []
        self._execute(self._sql_query(Table=recorded, Title=title,
                                       Category=category,
                                       Starttime=starttime,
                                       Endtime=endtime,
                                       Description = description,
                                       Hdtv=hdtv ))
        schList = self._fetch()
        if recorded == None:
            for x in schList:
                li.append(self._make_id(x[0], x[1]))
        elif recorded == False:
            for x in schList:
                li.append(self._make_id(x[2], str(x[4])+\
                 " " + str(x[3])))
        elif recorded == True:
            for x in schList:
                li.append(self._make_id(x[0], x[1]))
        else:
            print "bad table entry"
        return li

def query_id(self, unique_id, query = None):

    """ A method to retrieve various pieces of information about
                unique_ids such as progress of a recording, or the ids
                attributes.  Supply a unique id, and get back some
                information.  """

    if query == None:
        #Default just returns title of recording e.g., 'House'
        info = self._get_info(unique_id)
        return info

    elif query == 'progress':
        # Check to see percentage complete of a recording
        info = self._get_progress(unique_id)
        return info
```

```
        else:
            raise AttributeError("invalid query type")

    def record(self, unique_id = None, recording_schedule = None,
               record_type = 1):
        """ Given a recording schedule, this function will modify
                        the backend database to include recording_schedule,
                        and then notify the backend to update, so that the
                        recording will be recorded. """
        # Check to make sure we don't get duplicate ids
        present_recording_schedules = self.search(recorded=False)
        if present_recording_schedules.count(unique_id) > 0:
            raise AttributeError(\
                "Recording schedule with that id already exists!")

        if unique_id == None and recording_schedule == None:
            raise AttributeError("Need unique_id or recording_schedule.")
        else:
            if recording_schedule == None:
                dictionary = self.query_id(unique_id)
                dictionary['station'] = self._get_station(\
                                        str(dictionary['chanid']))
                dictionary['type'] = record_type
                recording_schedule = Recording_Schedule(dictionary, unique_id)

        sql = """REPLACE INTO record
                (type,
                chanid, starttime,
                startdate, endtime,
                enddate, title,
                subtitle, description,
                category, profile,
                recpriority, autoexpire,
                maxepisodes, maxnewest,
                startoffset, endoffset,
                recgroup, dupmethod,
                dupin, station,
                seriesid, programid,
                search, autotranscode,
                autocommflag, autouserjob1,
                autouserjob2, autouserjob3,
                autouserjob4, findday,
                findtime, findid,
                inactive, parentid,
```

```
            transcoder, tsdefault,
            playgroup, prefinput,
            next_record,
            last_record,
            last_delete)
            VALUES """ +\
          self._make_schedule_string(recording_schedule.schedule)
    self._execute(sql)
    if unique_id != None:
        updated_values = self.query_id(unique_id)
        recording_schedule.schedule['recordid'] = \
                                    updated_values['recordid']
        recording_schedule.schedule['next_record'] = \
                                    updated_values['next_record']
        recording_schedule.schedule['last_record'] = \
                                    updated_values['last_record']
        recording_schedule.schedule['last_delete'] = \
                                    updated_values['last_delete']
    else:
        unique_id = str(recording_schedule.schedule['chanid']) +\
                    "_" +\
                    recording_schedule.schedule['startdate'] +\
                    recording_schedule.schedule['starttime']
        updated_values = self.query_id(unique_id)
        recording_schedule.schedule['recordid'] = \
                                    updated_values['recordid']
        recording_schedule.schedule['next_record'] = \
                                    updated_values['next_record']
        recording_schedule.schedule['last_record'] = \
                                    updated_values['last_record']
        recording_schedule.schedule['last_delete'] = \
                                    updated_values['last_delete']
    self._reschedule_notify(recording_schedule.schedule)
    return recording_schedule


def cancel_recording(self, unique_id):
    """ Cancel a particular recording schedule. Supply the unique id,
                and this does the rest. """
    chanid, date = self._break_id(unique_id)
    temp = str(date).split(" ")
    startdate = temp[0]
    starttime = temp[1]
    sql = "DELETE FROM record WHERE " +\
        "chanid = (\'"+str(chanid)+"\') AND starttime = (\'" +\
```

```
        str(starttime)+"\') AND startdate = (\'" + str(startdate) +\
        "\')"
    self._execute(sql)
    self._reschedule_notify()

#Use these for your 'type' value in your schedule for more complex
#recordings.
RECORD_ONCE = 1
RECORD_DAILY = 2
RECORD_WEEKLY = 5
FIND_AND_RECORD_ONCE = 6
FIND_AND_RECORD_DAILY = 9
FIND_AND_RECORD_WEEKLY = 10
RECORD_ANY_TIME_ON_CHANNEL = 3
RECORD_ANY_TIME = 4

############################################################
# Interal, Private Methods
############################################################
# very low-level backend code thanks to xbmc team code
############################################################

def _connect( self ):
    #Establish connection with MySQL db, sets up cursor
    db = MySQLdb.connect(self.host,
                         self.user,
                         self.password,
                         self.database)
    self.db = db
    self.cursor = db.cursor()

def _execute( self, sql ):
    self.cursor.execute(sql)

def _fetch( self ):
    temp = self.cursor.fetchall()
    return temp

def _make_id(chanid, starttime):
    uniqueID = None
    if isinstance( starttime, datetime.datetime ):
        year = str(starttime.year)
        if starttime.month < 10:
            month = "0"+str(starttime.month)
        else:
```

```
            month = str(starttime.month)
        if starttime.day < 10:
            day = "0"+str(starttime.day)
        else:
            day = str(starttime.day)
        if starttime.hour < 10:
            hour = "0"+str(starttime.hour)
        else:
            hour = str(starttime.hour)
        if starttime.minute < 10:
            minute = "0"+str(starttime.minute)
        else:
            minute = str(starttime.minute)
        if starttime.second < 10:
            second = "0"+str(starttime.second)
        else:
            second = str(starttime.second)
        uniqueID = str(chanid)+"_"+year+month+day+hour+minute+second
    elif isinstance( starttime, str ):
        temp = starttime.split(" ")
        date =  temp[0]
        time =  temp[1]
        if(len(time) == 7):
            time = "0" + time
        start = date.replace("-","") + time.replace(":","")
        uniqueID = str(chanid)+"_"+start
    else:
        print "Given start time not valid."
    return uniqueID

_make_id = staticmethod(_make_id)

def _break_id(uniqueID):
    chanid = int(uniqueID.rsplit("_")[0])
    starttime = uniqueID.rsplit("_")[1]
    year = int(starttime[0:4])
    month = int(starttime[4:6])
    day = int(starttime[6:8])
    hour = int(starttime[8:10])
    minute = int(starttime[10:12])
    second = int(starttime[12:14])
    date = datetime.datetime(year, month, day, hour, minute, second)
    return chanid, date

_break_id = staticmethod(_break_id)
```

```
def _sql_query(Table=None, Title=None, Category=None,
               Starttime=None, Endtime=None,
               Description = None, Hdtv=None):
    sql = "SELECT * FROM"

    conditionSpecified = False
    #Select which table to access in db: program, record, or recorded
    if Table == None:
        sql = sql+" program"
    elif Table == False:
        sql = sql+" record"
    elif Table == True:
        sql = sql+" recorded"
    else:
        print "bad ID"
        return
    #Specify conditions
    if Title != None:
        conditionSpecified = True
        if Title[0] != "+":
            Title = Title[1:len(Title)]
            sql = sql + " WHERE title LIKE (\'%" + str(Title)+"%\')"
        else:
            sql = sql + " WHERE title = (\'" + str(Title) + "\')"
    if Category != None:
        if Category[0] != "+":
            Category = Category[1:len(Category)]
            if conditionSpecified:
                sql = sql + " AND category LIKE (\'%" +\
                      str(Category) + "%\')"
            else:
                conditionSpecified = True
                sql = sql + " WHERE category LIKE (\'%" +\
                      str(Category) + "%\')"
        else:
            if conditionSpecified:
                sql = sql + " AND category = (\'" +\
                      str(Category) + "\')"
            else:
                conditionSpecified = True
                sql = sql + " WHERE category = (\'" +\
                      str(Category) + "\')"
    if Description != None:
        if Description[0] != "+":
```

```
            Description = Description[1:len(Description)]
            if conditionSpecified:
                sql = sql + " AND description LIKE (\'%" +\
                      str(Description) + "%\')"
            else:
                conditionSpecified = True
                sql = sql + " WHERE description LIKE (\'%" +\
                      str(Description) + "%\')"
        else:
            if conditionSpecified:
                sql = sql + " AND description = (\'" +\
                      str(Description) + "\')"
            else:
                conditionSpecified = True
                sql = sql + " WHERE description = (\'" +\
                      str(Description) + "\')"
    if Starttime != None:
        if conditionSpecified:
            sql = sql + " AND starttime = (\'" + str(Starttime)+"\')"
        else:
            conditionSpecified = True
            sql = sql + " WHERE starttime = (\'"+str(Starttime)+"\')"
    if Endtime != None:
        if conditionSpecified:
            sql = sql + " AND endtime = (\'" + str(Endtime) + "\')"
        else:
            conditionSpecified = True
    if Hdtv != None:
        if conditionSpecified:
            sql = sql + " AND hdtv = (\'" + str(int(Hdtv)) + "\')"
        else:
            conditionSpecified = True
            sql = sql + " WHERE hdtv = (\'" + str(int(Hdtv)) + "\')"
    return sql

_sql_query = staticmethod(_sql_query)

def _get_progress( self, uniqueID ):
    now = datetime.datetime.now()
    info = self.query_id(uniqueID)
    starttime = info['starttime']
    endtime = info['endtime']
    if now > endtime:
        return 1
    elif now < starttime:
```

```
            return 0
        elif starttime.day < endtime.day:
            delta1 = 86400 - self._get_seconds( starttime ) + \
                     self._get_seconds( endtime )
            if now.day == endtime.day:
                delta2 = 86400 - self._get_seconds( starttime ) + \
                         self._get_seconds( now )
            else:
                delta2 = self._get_seconds( now ) - \
                         self._get_seconds( starttime )
            percentageComplete = float(delta2) / delta1
            percentageFormatted = '%.2f' % percentageComplete
            return percentageFormatted
        else:
            delta1 = self._get_seconds( endtime ) - \
                     self._get_seconds( starttime )
            delta2 = self._get_seconds( now ) -\
             self._get_seconds(starttime)
            percentageComplete = float(delta2) / delta1
            percentageFormatted = '%.2f' % percentageComplete
            return percentageFormatted

    def _get_seconds(_time):
        return _time.hour * 3600 + _time.minute * 60 + _time.second

    # this makes _get_seconds a static method:
    _get_seconds = staticmethod(_get_seconds)

    def _get_info( self, uniqueID ):
        gotProgram=False
        gotRecord=False
        gotRecorded=False
        #Get schedules from mysql database
        chanid, date = self._break_id(uniqueID)

        sql_program = "SELECT * FROM program WHERE " +\
                      "chanid=(\'"+str(chanid)+"\') AND starttime=(\'"+\
                      str(date)+"\')"

        sql_record = "SELECT * FROM record WHERE chanid=(\'" +\
                     str(chanid)+"\') AND starttime=(\'"+str(date)+"\')"

        sql_recorded = "SELECT * FROM recorded WHERE chanid=(\'" +\
                       str(chanid)+"\') AND starttime=(\'"+str(date)+"\')"
```

```
self._execute(sql_program)
result = self._fetch()
if len(result) > 0:
    program = result[0]
    gotProgram = True
self._execute(sql_record)
result = self._fetch()
if len(result) > 0:
    record = result[0]
    gotRecord = True
self._execute(sql_recorded)
result = self._fetch()
if len(result) > 0:
    recorded = result[0]
    gotRecorded = True
program_dictionary = {}
record_dictionary = {}
recorded_dictionary = {}
dictionary = {}
#Make dictionary and fill it with values
if gotProgram:
    program_dictionary['chanid'] = program[0]
    program_dictionary['starttime'] = program[1]
    program_dictionary['endtime'] = program[2]
    program_dictionary['title'] = program[3]
    program_dictionary['subtitle'] = program[4]
    program_dictionary['description'] = program[5]
    program_dictionary['category'] = program[6]
    program_dictionary['category_type'] = program[7]
    program_dictionary['airdate'] = program[8]
    program_dictionary['stars'] = program[9]
    program_dictionary['previouslyshown'] = program[10]
    program_dictionary['title_pronounce'] = program[11]
    program_dictionary['stereo'] = program[12]
    program_dictionary['subtitled'] = program[13]
    program_dictionary['hdtv'] = program[14]
    program_dictionary['closecaptioned'] = program[15]
    program_dictionary['partnumber'] = program[16]
    program_dictionary['parttotal'] = program[17]
    program_dictionary['seriesid'] = program[18]
    program_dictionary['originalairdate'] = program[19]
    program_dictionary['showtype'] = program[20]
    program_dictionary['colorcode'] = program[21]
    program_dictionary['syndicatedepisodenumber'] = program[22]
    program_dictionary['programid'] = program[23]
```

```
        program_dictionary['manualid'] = program[24]
        program_dictionary['generic'] = program[25]
        program_dictionary['listingsource'] = program[26]
        program_dictionary['first'] = program[27]
        program_dictionary['last'] = program[28]
    if gotRecord:
        record_dictionary['recordid'] = record[0]
        record_dictionary['type'] = record[1]
        record_dictionary['chanid'] = record[2]
        record_dictionary['starttime'] = record[3]
        record_dictionary['startdate'] = record[4]
        record_dictionary['endtime'] = record[5]
        record_dictionary['enddate'] = record[6]
        record_dictionary['title'] = record[7]
        record_dictionary['subtitle'] = record[8]
        record_dictionary['description'] = record[9]
        record_dictionary['category'] = record[10]
        record_dictionary['profile'] = record[11]
        record_dictionary['recpriority'] = record[12]
        record_dictionary['autoexpire'] = record[13]
        record_dictionary['maxepisodes'] = record[14]
        record_dictionary['maxnewest'] = record[15]
        record_dictionary['startoffset'] = record[16]
        record_dictionary['endoffset'] = record[17]
        record_dictionary['recgroup'] = record[18]
        record_dictionary['dupmethod'] = record[19]
        record_dictionary['dupin'] = record[20]
        record_dictionary['station'] = record[21]
        record_dictionary['seriesid'] = record[22]
        record_dictionary['programid'] = record[23]
        record_dictionary['search'] = record[24]
        record_dictionary['autotranscode'] = record[25]
        record_dictionary['autocommflag'] = record[26]
        record_dictionary['autouserjob1'] = record[27]
        record_dictionary['autouserjob2'] = record[28]
        record_dictionary['autouserjob3'] = record[29]
        record_dictionary['autouserjob4'] = record[30]
        record_dictionary['findday'] = record[31]
        record_dictionary['findtime'] = record[32]
        record_dictionary['findid'] = record[33]
        record_dictionary['inactive'] = record[34]
        record_dictionary['parentid'] = record[35]
        record_dictionary['transcoder'] = record[36]
        record_dictionary['tsdefault'] = record[37]
        record_dictionary['playgroup'] = record[38]
```

```
        record_dictionary['prefinput'] = record[39]
        record_dictionary['next_record'] = record[40]
        record_dictionary['last_record'] = record[41]
        record_dictionary['last_delete'] = record[42]
    if gotRecorded:
        recorded_dictionary['chanid'] = recorded[0]
        recorded_dictionary['starttime'] = recorded[1]
        recorded_dictionary['endtime'] = recorded[2]
        recorded_dictionary['title'] = recorded[3]
        recorded_dictionary['subtitle'] = recorded[4]
        recorded_dictionary['description'] = recorded[5]
        recorded_dictionary['category'] = recorded[6]
        recorded_dictionary['hostname'] = recorded[7]
        recorded_dictionary['bookmark'] = recorded[8]
        recorded_dictionary['editing'] = recorded[9]
        recorded_dictionary['cutlist'] = recorded[10]
        recorded_dictionary['autoexpire'] = recorded[11]
        recorded_dictionary['commflagged'] = recorded[12]
        recorded_dictionary['recgroup'] = recorded[13]
        recorded_dictionary['recordid'] = recorded[14]
        recorded_dictionary['seriesid'] = recorded[15]
        recorded_dictionary['programid'] = recorded[16]
        recorded_dictionary['lastmodified'] = recorded[17]
        recorded_dictionary['filesize'] = recorded[18]
        recorded_dictionary['stars'] = recorded[19]
        recorded_dictionary['previouslyshown'] = recorded[20]
        recorded_dictionary['originalairdate'] = recorded[21]
        recorded_dictionary['preserve'] = recorded[22]
        recorded_dictionary['findid'] = recorded[23]
        recorded_dictionary['deletepending'] = recorded[24]
        recorded_dictionary['transcoder'] = recorded[25]
        recorded_dictionary['timestretch'] = recorded[26]
        recorded_dictionary['recpriority'] = recorded[27]
        recorded_dictionary['basename'] = recorded[28]
        recorded_dictionary['progstart'] = recorded[29]
        recorded_dictionary['progend'] = recorded[30]
        recorded_dictionary['playgroup'] = recorded[31]
        recorded_dictionary['profile'] = recorded[32]
        recorded_dictionary['duplicate'] = recorded[33]
        recorded_dictionary['transcoded'] = recorded[34]
        recorded_dictionary['watched'] = recorded[35]
    for x in program_dictionary.keys():
        if dictionary.has_key(x):
            continue
        else:
```

```
            dictionary[x] = program_dictionary[x]
    for x in record_dictionary.keys():
        if dictionary.has_key(x):
            continue
        else:
            dictionary[x] = record_dictionary[x]
    for x in recorded_dictionary.keys():
        if dictionary.has_key(x):
            continue
        else:
            dictionary[x] = recorded_dictionary[x]
    return dictionary

def _backend_connect(self, playback = False, monitor = True ):
    s = None
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((self.host, 6543))
    if monitor:
            s = self._ann_monitor(s)
    elif playback:
            s = self._ann_playback(s)
    return s

def _ann_monitor(self, s):
    reply = self._send_request( s, ["ANN Monitor %s 0" % self.host] )
    if not self._is_ok(reply):
        raise ServerException, "backend monitor refused: " +\
        str(reply)
    return s

def _ann_playback(self, s):
    reply = self._send_request(s, ["ANN Playback %s 0" % self.host])
    if not self._is_ok(reply):
        raise ServerException, "backend playback refused: " +\
         str(reply)
    return s

def _is_ok(self, msg):
    if msg == None or len(msg) == 0:
        return False
    else:
        return string.upper(msg[0]) == "OK"

def _send_msg(self, s, req):
    msg = self._build_msg(req)
```

```
    s.send(msg)

def _send_request(self, s, msg):
    if s == None:
        s = self._backend_connect(self.host, False, False)
    self._send_msg(s, msg)
    reply = self._read_msg(s)
    return reply

def _build_msg(self, msg):
    msg = string.join(msg, SEP)
    return "%-8d%s" % (len(msg), msg)

def _read_msg(self, s):
    retMsg = ""

    retMsg = s.recv(8)
    reply = ""
    if string.upper(retMsg) == "OK":
        return "OK"

    n = int(retMsg)
    i = 0
    while i < n:
        reply += s.recv(n - i)
        i = len(reply)
    return reply.split(SEP)

def _reschedule_notify(self, schedule = None):
    s = self._backend_connect(False, True)
    recordId = None
    if schedule:
        recordId = schedule['recordid']
    if recordId == None:
        recordId = -1
    reply =
            self._send_request(s, ["RESCHEDULE_RECORDINGS %d" % recordId])

def _get_time(self, _date):
    date_string = str(_date)
    temp = date_string.split(" ")
    return temp[1].replace(":","")

def _get_date(self, _date):
    date_string = str(_date)
```

```
        temp = date_string.split(" ")
        return temp[0].replace("-","")

    def _get_station(self, chanid):
        sql = "SELECT * FROM channel WHERE chanid = " + str(chanid)
        self._execute(sql)
        channel = self._fetch()
        station = channel[0][4]
        return station

    def _make_schedule_string(self, sch):
        sch_string = "(" +\
          str(sch['type']) + ", " +\
          str(sch['chanid']) + ", " +\
          str(sch['starttime']) + ", " +\
          str(sch['startdate']) + ", " +\
          str(sch['endtime']) + ", " +\
          str(sch['enddate']) + ", " +\
          "\'" + str(sch['title']).replace("\'", "\\\'") +\
          "\', " +\
          "\'" + str(sch['subtitle']).replace("\'", "\\\'") +\
          "\', " +\
          "\'" + str(sch['description']).replace("\'", "\\\'") +\
          "\', " +\
          "\'" + str(sch['category']).replace("\'", "\\\'") +\
          "\', " +\
          "\'" + str(sch['profile']).replace("\'", "\\\'") +\
          "\', " +\
          str(sch['recpriority']) + ", " +\
          str(sch['autoexpire']) + ", " +\
          str(sch['maxepisodes']) + ", " +\
          str(sch['maxnewest']) + ", " +\
          str(sch['startoffset']) + ", " +\
          str(sch['endoffset']) + ", " +\
          "\'" + str(sch['recgroup']).replace("\'", "\\\'") +\
          "\', " +\
          str(sch['dupmethod']) + ", " +\
          str(sch['dupin']) + ", " +\
          "\'" + str(sch['station']).replace("\'", "\\\'") +\
          "\', " +\
          "\'" + str(sch['seriesid']).replace("\'", "\\\'") +\
          "\', " +\
          "\'" + str(sch['programid']).replace("\'", "\\\'") +\
          "\', " +\
          str(sch['search']) + ", " +\
```

```
            str(sch['autotranscode']) + ", " +\
            str(sch['autocommflag']) + ", " +\
            str(sch['autouserjob1']) + ", " +\
            str(sch['autouserjob2']) + ", " +\
            str(sch['autouserjob3']) + ", " +\
            str(sch['autouserjob4']) + ", " +\
            str(sch['findday']) + ", " +\
            str(sch['findtime']) + ", " +\
            str(sch['findid']) + ", " +\
            str(sch['inactive']) + ", " +\
            str(sch['parentid']) + ", " +\
            str(sch['transcoder']) + ", " +\
            str(sch['tsdefault']) + ", " +\
            "\'" + str(sch['playgroup']).replace("\'", "\\\'") +\
            "\', " +\
            str(sch['prefinput']) + ", " +\
            "\'" + str(sch['next_record']) + "\', " +\
            "\'" + str(sch['last_record']) + "\', " +\
            "\'" + str(sch['last_delete']) + "\'" +\
            ")"
        return sch_string

class Recording_Schedule:
    """ Simple class to manage a recording schedule. """

    schedule = dict()
    unique_id = None

    def __init__(self, dictionary, unique_id):
        self.unique_id = unique_id

        #schedule['recordid'] = -1
        self.schedule['type'] = dictionary['type']
        self.schedule['chanid'] = dictionary['chanid']
        self.schedule['starttime'] =
                self._get_time(dictionary['starttime'])
        self.schedule['startdate'] =
                self._get_date(dictionary['starttime'])
        self.schedule['endtime'] =
                self._get_time(dictionary['endtime'])
        self.schedule['enddate'] =
                self._get_date(dictionary['endtime'])
        self.schedule['title'] = dictionary['title']
        self.schedule['subtitle'] = dictionary['subtitle']
        self.schedule['description'] = dictionary['description']
```

```
        self.schedule['category'] = dictionary['category']
        self.schedule['profile'] = "Default"
        self.schedule['recpriority'] = 0
        self.schedule['autoexpire'] = 1
        self.schedule['maxepisodes'] = 0
        self.schedule['maxnewest'] = 0
        self.schedule['startoffset'] = 0
        self.schedule['endoffset'] = 0
        self.schedule['recgroup'] = "Default"
        self.schedule['dupmethod'] = 6
        self.schedule['dupin'] = 15
        self.schedule['station'] = dictionary['station']
        self.schedule['seriesid'] = dictionary['seriesid']
        self.schedule['programid'] = dictionary['programid']
        self.schedule['search'] = 0
        self.schedule['autotranscode'] = 0
        self.schedule['autocommflag'] = 1
        self.schedule['autouserjob1'] = 0
        self.schedule['autouserjob2'] = 0
        self.schedule['autouserjob3'] = 0
        self.schedule['autouserjob4'] = 0
        self.schedule['findday'] = 0
        self.schedule['findtime'] = 0
        self.schedule['findid'] = 0
        self.schedule['inactive'] = 0
        self.schedule['parentid'] = 0
        self.schedule['transcoder'] = 0
        self.schedule['tsdefault'] = 1
        self.schedule['playgroup'] = "Default"
        self.schedule['prefinput'] = 0
        self.schedule['next_record'] = dictionary['starttime']
        self.schedule['last_record'] = datetime.datetime(2000,1,1)
        self.schedule['last_delete'] = datetime.datetime(2000,1,1)

    def _get_time(self, _date):
        date_string = str(_date)
        temp = date_string.split(" ")
        return temp[1].replace(":","")

    def _get_date(self, _date):
        date_string = str(_date)
        temp = date_string.split(" ")
        return temp[0].replace("-","")

    def _get_station(self, chanid):
```

```
        sql = "SELECT * FROM channel WHERE chanid = " + str(chanid)
        self._execute(sql)
        channel = self._fetch()
        station = channel[0][4]
        return station


#############################################################
# Deprecated Code
#############################################################

def _get_schedule_deprecated( self, uniqueID, recorded=None ):
    """ Deprecated """

    chanid, date = self._break_id(uniqueID)
    sql = "SELECT * FROM"

    #Select which table to access in db
    if recorded == None:
        sql = sql+" program"
    elif recorded == False:
        sql = sql+" record"
    elif recorded == True:
        sql = sql+" recorded"
    else:
        print "bad ID"
        return

    sql = sql+" WHERE chanid=(\'"+str(chanid)+\
                              "\') AND starttime=(\'" +\
          str(date)+"\')"
    self._execute(sql)
    sch = self._fetch()[0]
    return sch

def _generate_recordid(self):
    sql = "SELECT * FROM record"
    self._execute(sql)
    record = self._fetch()
    used_record_ids = []
    for x in record:
        used_record_ids.append(x[0])
    rand_num = random.randint(10000, 19999)
    num = used_record_ids.count(rand_num)
    while num != 0:
```

```
    rand_num = randint(10000, 19999)
    num = used_record_ids.count(rand_num)
return rand_num
```

Appendix B

pyMythTV Application Code

```
import myth
import pprint

saved_id = None
result = None
search_parameters = None

def main():
    exit = False
    state = "INITIAL"
    user_input = None
    m = myth.Myth('18.95.2.126', 'mythtv', 'ravdjqqf', 'mythconverg')
    while exit != True:
        display(state)
        user_input = request_input(user_input, state)
        state = do_action(user_input, state, m)

def display(state):
    if state == 'INITIAL':
        print "\nWelcome to a myth program!  Please select a number:\n"
        if saved_id != None:
            print "Your saved id: " + saved_id
        print "1: Search for a program or recording."
        print "2: Get information about the saved id."
        print "3: Schedule a recording using the saved id."
        print "4: Cancel a recording using the saved id."
        print "5: Exit."
    elif state == 'SEARCH':
        print "\nPlease select where you want to search:"
        print "1: Search all programs, recordings schedules, and recordings."
```

```
        print "2: Search only in programs."
        print "3: Search only in recording schedules."
        print "4: Search only in recordings."
        print "5: Go back to main menu."
    elif state == 'SEARCH_PARAMETERS':
        print "\nPlease specify which parameters you would like (seperated by" +\
              "spaces)."
        print "Example: \'1 6\' Would select title and hdtv."
        print "1: Title."
        print "2: Category."
        print "3: Start Time."
        print "4: End Time."
        print "5: Description."
        print "6: HDTV."
        print "7: Go back to main menu."
    elif state == 'SEARCH_RESULTS':
        print "Your new saved_id is: ", globals()['saved_id']
    elif state == 'GET_INFO':
        if globals()['saved_id'] == None:
            print "Need a saved id to get information about it!"
        else:
            print "Information associated with saved id", globals()['saved_id']
        state = 'GET_INFO2'
    elif state == 'GET_INFO2':
        p = pprint.pprint(globals()['result'], depth=1)
    elif state == 'SCHEDULE':
        if globals()['saved_id'] == None:
            print "Need a saved id to get information about it!"
        else:
            print "\nSaving a recording schedule for", globals()['saved_id']
    elif state == 'SCHEDULE2':
        print "Schedule saved."
    else:
        print "Error."

def request_input(user_input, state):
    if state == 'SEARCH_RESULTS' or\
       state == 'GET_INFO' or\
       state == 'GET_INFO2' or\
       state == 'SCHEDULE' or\
       state == 'CANCEL':
        return None
    user_input = raw_input("Please select your choice: ")
    user_input = user_input.rsplit(" ")
    if len(user_input) == 1:
```

```
        user_input = user_input[0]
    return user_input

def do_action(user_input, state, m):
    if state == 'INITIAL':
        if user_input == '1':
            state = 'SEARCH'
        elif user_input == '2':
            state = 'GET_INFO'
        elif user_input == '3':
            state = 'SCHEDULE'
        elif user_input == '4':
            state = 'CANCEL'
        elif user_input == '5':
            import os
            print "Exiting."
            os._exit(99)
        else:
            print "Sorry, please input again."
    elif state == 'SEARCH':
        globals()['search_parameters'] = [False, None, None, None,
                                          None, None, None, None]
        if user_input == '1':
            state = 'SEARCH_PARAMETERS'
            globals()['search_parameters'][0] = True
        elif user_input == '2':
            state = 'SEARCH_PARAMETERS'
            globals()['search_parameters'][1] = None
        elif user_input == '3':
            state = 'SEARCH_PARAMETERS'
            globals()['search_parameters'][1] = False
        elif user_input == '4':
            state = 'SEARCH_PARAMETERS'
            globals()['search_parameters'][1] = True
        elif user_input == '5':
            state = 'INITIAL'
        else:
            print "Sorry, please input again."
    elif state == 'SEARCH_PARAMETERS':
        if user_input.count('1') > 0:
            globals()['search_parameters'][2] =\
                raw_input("Please input title: ")
        if user_input.count('2') > 0:
            globals()['search_parameters'][3] =\
                raw_input("Please input category: ")
```

```
    if user_input.count('3') > 0:
        globals()['search_parameters'][4] =\
            raw_input("Please input starttime: ")
    if user_input.count('4') > 0:
        globals()['search_parameters'][5] =\
            raw_input("Please input endtime: ")
    if user_input.count('5') > 0:
        globals()['search_parameters'][6] =\
            raw_input("Please input description: ")
    if user_input.count('6') > 0:
        globals()['search_parameters'][7] =\
            raw_input("Please input hdtv: ")
    result = m.search(search_all=globals()['search_parameters'][0],
                      recorded=globals()['search_parameters'][1],
                      title=globals()['search_parameters'][2],
                      category=globals()['search_parameters'][3],
                      starttime=globals()['search_parameters'][4],
                      endtime=globals()['search_parameters'][5],
                      description=globals()['search_parameters'][6],
                      hdtv=globals()['search_parameters'][7])
    if len(result) > 0:
        globals()['saved_id'] = result[0]
        print "\nSelecting first matching unique id..."
    else:
        print "No matches found for given parameters"
    state = 'SEARCH_RESULTS'
elif state == 'SEARCH_RESULTS':
    state = 'INITIAL'
elif state == 'GET_INFO':
    if saved_id == None:
        state = 'GET_INFO2'
        print "No saved id!"
    else:
        state = 'GET_INFO2'
        globals()['result'] = m.query_id(saved_id)
elif state == 'GET_INFO2':
    state = 'INITIAL'
elif state == 'SCHEDULE':
    if saved_id == None:
        state = 'INITIAL'
        print 'No saved id!'
    else:
        state = 'INITIAL'
        m.record(saved_id)
elif state == 'CANCEL':
```

```
        if saved_id == None:
            state = 'INITIAL'
            print "No saved id!"
        else:
            state = 'INITIAL'
            m.cancel_recording(saved_id)
    return state


main()
```

Bibliography

[1] Baldrick. Mythtv front-end picture., 2008.

[2] ECT News Business Desk. A description of tivo growth., 2004.

[3] Bill Jackson. Mythtv protocol, 2006.

[4] John Molohan. Freevo home theatre platform, 2008.

[5] Shahar Prish. Etivo description, 2005.

[6] Shahar Prish. Etivo service, 2005.

[7] Isaac Richards. Mythtv description, 2008.

[8] LLC. SageTV. Sagetv description, 2008.

[9] Justin Mazzola Paluska Steve Ward. Justplay project overview., 2006.

[10] Joe Stump. The state of home-brew pvrs on linux, 2003.

[11] XBMC Team. Xbox media center description., 2008.