Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Implementing Democracy

a large scale cross-platform desktop application

Christopher James Lahey
Participatory Culture Foundation
`clahey@clahey.net`

Abstract

Democracy is a cross-platform video podcast client. It integrates a large number of functions, including searching, downloading, and playing videos. Thus, it is a large-scale application integrating a number of software libraries, including a browser, a movie player, a bittorrent client, and an RSS reader.

The paper and talk will discuss a number of techniques used, including using PyRex to link from python to C libraries, using a web browser and a templating system to build the user interface for cross-platform desktop software (including a different web browser on each platform), and our object store used to keep track of everything in our application, store our state to disk, and bring updates to the UI.

1 Internet video

Internet video is becoming an important part of modern culture, currently through video blogs, video podcasts, and YouTube. Video podcasting gives everyone the ability to decide what they want to make available, but the spread of such systems as YouTube and Google Video suggest that large corporations will have a lot to say in the future of internet video.

The most popular use of internet video right now is YouTube. YouTube videos are popping up all over the place. Unfortunately, this gives one company a lot of power over content. It lets them take down whatever they find inconvenient or easily block certain content from reaching certain people.

Video podcasts are RSS feeds with links to videos. Podcasting allows the publisher to put whatever videos he wants on his personal webspace. Podcasting clients download these videos for display on different devices. This gets around the problem of one company controlling everything. That is, except for the fact that the most prominent podcast client is iTunes and it's used for downloading to iPods. Once again, the company has the ability to censor.

Some folks in Worcester, Massachusetts saw this as a problem and so sought funding and formed the non-profit Participatory Culture Foundation. The goal of the Participatory Culture Foundation is to make sure that everyone has a voice and that no one need be censored. We are approaching this from a number of different angles. We have a project writing tutorials for people that want to make and publish internet video. We are in planning for a server project to let people post their video podcasts. And most importantly, we write the Democracy player.

The reason the player is so important to us is that we want to make sure that publishing and viewing are independent. If they aren't, then there are two types of lock-in. Firstly, if a user wants to see a particular video, they're forced to use the particular publisher's viewing software. Secondly, once a user starts using a particular publisher's viewing software, that publisher gains control over what the viewer can see. These two could easily join together in a feedback loop that leads to a monopoly situation.

However, to separate publishing and viewing, we need a standard for communication. RSS fills this role perfectly. In fact, it's already in use for this purpose. The role we want Democracy to fill is that of a good player that encourages viewers to use RSS. Well, we also just want it to be a great video player.

2 Democracy

Democracy's main job is to download and play videos from RSS feeds. We also decided to make it able to be the heart of your video experience. Thus it plays local videos, searches for videos on the internet, and handles videos downloaded in your web browser.

To do all this we integrated a number of other tools. We included other python projects wholesale, like feed-parser and BitTorrent. We link to a number of standard libraries through either standard python interfaces or through the use of Pyrex as a glue language.

For widest adoption, we decided it was important for Democracy to be cross-platform. Windows and Mac would get us the most users, but Linux is important to us since we create free software. So far two of our new developers (myself included) have come from Linux-land and one from OSX-land.

3 Major objects in Democracy

There are two major object types that we deal with: Feeds and Items.

Feeds tend to be RSS feeds, but they can also be scrapes of HTML pages, watches on local directories, and other things. Since we don't know at feed creation time whether a URL will return an RSS feed or an HTML page, we create a Feed object which is mainly a proxy to a FeedImpl object that can be created later. python makes this proxy action almost completely transparent. We implement a `__getattr__` handler which gets called for methods and data that aren't defined in the Feed object. In this handler, we simply return the corresponding method or data for the FeedImpl object. We use this trick in a couple of other similar proxy situations in Democracy.

Items are individual entries in an RSS feed. Most of them represent a video that democracy could potentially download. You can also use democracy to download either directories of multiple videos or non-video files. They can be either available, downloading, or downloaded. We also have FileItems which are used to designate local files that don't have a corresponding URL. These can either be the children videos of a directory we download, or files found on the local disk. We have special feeds that monitor a local directory and create any found files. You can also pass Democracy a filename and it will create an item for that video.

We've spent a lot of time tweaking the behavior of all of these objects. One of the things we've discovered is that the more features that we have, the harder new features are to implement. Anyone who has any experience at all shouldn't be surprised to hear this, but it's amazing the difference between hearing about it in books and getting specific examples in your work.

4 Object Store

To keep track of everything that is happening in our application, we have a collection of objects. Every important object has a global ID. This includes all feeds and items, as well as playlists and channel guides.

However, we need fast access to different sets of objects. We need a list of all items in a particular feed, for example. To implement this, we have a system of views into the database. Each view acts like another database and can thus have subviews. We have a number of different view types. The first is *filters*, which are subsets of the database. The second is *maps*, which create a new object for each member of the original database. The third is *indexes*, which create a whole bunch of subdatabases and put each item in one of the subdatabases. Finally we have *sorts*, though these are redundant, as the other view types can be sorted as well.

The other important part of the object database is that you can monitor a view for changes. We send signals on new objects being added or removed. Each object has a signalChange function which signals that the data in that object has changed. You monitor this by watching for changes on a database view.

This in-memory object database has worked quite well for us. We have a list of all objects that we care about, while still being able to have lists of the objects that we care about right now. An example of a use of views is that each feed doesn't keep a list of its items. It just has a view into the database and as the items get created, the feed gets that item added to its item list. However, the biggest use of views is in our cross-platform UI.

5 Cross Platform UI

To implement the cross-platform user interface, we use HTML with CSS. We have two main HTML areas and platform-specific menus and buttons. The HTML is generated automatically based on the objects in our object database.

We start with XML templates that are compiled into python code that in turn generates HTML. You can pass in plain XHTML and it will work. To start with, we had a series of key-value pairs that were set in python code and could then be accessed within the templates. That proved to be a pain of having to change the python every time we needed to change a referenced key, so we switched to simply allowing the python to be embedded directly in the XML. This is safe, since we provide all the XML and don't take any from the outside world, and it's much more maintainable.

The template system can do a number of different things with the results of the python. It can embed the result directly in the HTML. It can encode as a string and then embed the result in the HTML.

Slightly more interestingly, it can hide a chunk of HTML based on the return value. It can update a section of the template whenever a view changes. The most interesting part though is that it can repeat a section of HTML for every object in a view.

`repeatForView` takes a chunk of template and repeats it for every object in a view. When an object is added to or removed from the view, it adds or removes the corresponding HTML. When an object changes, it recalculates the HTML for that object.

Some of our team members are not entirely happy with HTML as our solution. It means working within the system that the browser gives us. It also means supporting both OSX's webkit and mozilla with our code (we use `gtkmozembed` on Linux and `xul` on Windows.) Finally, it sticks us with `xul` on Windows. We originally tried using the Windows framework by hand. We decided this was just too much work from python. When that didn't work, we tried using gtk and embedding mozilla, but found that `gtkmozembed` doesn't work on Windows. Finally we switched to `xul`, but `xul` is much harder to code to than either OSX or gtk. We may switch to using gtk+ on Windows, and to support that, we would switch to using some other rendering system, perhaps our own XML language that maps to cairo on gtk+ and something else on OSX.

I personally would prefer to stick with HTML plus CSS. It gives us a wide range of developers who know our rendering model. It gives us a bunch of free code to do the rendering. The only problem is getting one of those sets of code to work on Windows.

6 LiveStorage

To save our database to disk, we originally just pickled the object database to disk. Python pickle is a library that takes a python object and encodes all the data in it to an on-disk format. The same library will then decode that data and create the corresponding objects in memory again. It handles links to other objects including reference loops and it handles all the standard string and number data types.

This worked as long as we didn't change the in-memory representation of any of our objects. We worked around a number of different issues, but in the end we decided to remove some classes, and pickle throws an exception if it has an object on disk that doesn't have a corresponding class in memory.

The next step was to add a system that copied the data into python dictionaries and then pickled that created object. To do this, we created a schema object which describes what sort of data gets stored. This makes removing a field trivial. The system that copies the data out of the python dictionaries at load time simply ignores any fields not listed in the schema.

Adding fields is a bit more complicated, but to solve this, we store a version number in the database. Every time we change the schema, we increment this version number. At load time, the system compares the version in the database to the version in the running application and runs a series of upgrades on the data. These upgrades happen on the dictionary version of the objects and thus involve no running code. They can also remove or add objects, which allows us to remove old classes that aren't necessary and add objects that are.

Our next problem with data storage was that it was super slow to save the database. The larger the database got, the slower it was, to the tune of 45 seconds on large data sets, and we want to save regularly so that the user doesn't lose any data.

To solve this, we decided to save each object individually. Unfortunately, the objects referred to one another. Pickle handles this just fine when you ask it to pickle all the objects at once, but it isn't able to do that when you want to pickle just a single object (in fact, it will basically pickle every object related to the one you requested and then at load time will not connect the objects saved in different pickle runs.) The biggest example of this was that each feed kept a list of the items in that feed.

So the first step was to make the objects not refer to each other. For the most part we wanted to do this by just replacing references to other objects with their database ID. This works, but we also decided that we didn't want to keep redundant data. For example, a simple replacement of references with IDs in the database would lead to a feed having a list of items in that feed and the items each having the ID of their feed. Luckily, our in-memory database already had filters. We just made the feeds not store their children, and instead the list of children is simply a database filter.

After this, we replaced the single pickle with a Berkeley database storing the list of objects. We still had to worry about keeping changes in sync. For example, when first creating a feed, you need to make sure that all of the items are saved as well. To solve this, we simply stored the list of changed items and did the actual save to the database in a timeout loop. We used a transaction, and since the old database save happened in a timeout loop as well, we have the exact same semantics for syncing of different objects.

This worked great for a good while. We even used the schema upgrade functions to do things other than database upgrades, such as to automatically work around bugs in old versions. In fact we had no problem with this on Linux or Windows, but on Mac OSX, we got frequent reports of database errors on load and many people lost their data. We tried reproducing the error to debug it and we asked about the issue on Berkeley DB's usenet groups (where we'd gotten useful information before,) but there was no response. From there we decided to switch to `sqlite`. We're still using it to just store a list of pickled objects, but it's working fairly well for us.

The next step we'd like to take is not to have the entire database in memory at all times. Having it in memory increases our memory usage and limits the number of feeds a user can monitor. We'd like to change to using a more standard relational database approach to storing our data. This would, however, completely change how we access our data. We've decided that the size of this change means we should wait until after 1.0 to make this change.

There are some other major obstacles to making this change other than the number of pieces of code that would have to change. The first is that we use change notification extensively. An object changes and people interested in that object are notified. Similarly, objects added to or removed are signalled. To get around this, we would need either a relational database that does change notifications of this sort, or we would need to add a change notification layer on top of the database. Currently these notifications happen based on the different filters in our database, so we would need to duplicate the SQL searches that we do as monitoring code to know who needs to be signalled. For this reason, we're hoping that we find a relational database that will handle live sql searches and send notification of changes, additions, and removals.

The second obstacle is less of a problem with the change and more a reason that it won't help. Specifically, we touch most of the database at load time anyway. It would mean that we could start the user interface having loaded less data, but we queue an update of every RSS feed at load time. To run this update, we need to load all the items in that feed so we can compare them to the items we download (so that we don't create duplicate items.) At that point we could unload all that data.

7 BitTorrent

We act as a bittorrent client as well. This can either be by loading a BitTorrent file by hand or by including it in an RSS player. As a user, I found it very pleasant to have things just download. In fact, at first, I didn't realize that I was using a BitTorrent client. I think this can help increase usage of BitTorrent since people won't be intimidated by technology if they don't know they're using it. Another good example of this is the downloader used for World of Warcraft.

Unfortunately, the primary BitTorrent source code has become non-free software. For this reason, we eventually switched to using BitTornado. Unfortunately BitTornado introduced a number of bugs, and we decided that for us, fixing those bugs would be harder than recreating the new features that BitTornado supplied. We still don't have all the features that we want, but in switching back to the old BitTorrent code, we've got a codebase that tends to work quite well.

Looking into the future, we'd love to see a free software BitTorrent client take off. Post 1.0, we're considering adding a bunch of new BitTorrent features and protocol improvements to the code base we're currently using. Our goal would certainly be to maintain the code as a separate project so we don't waste our time and so that the free software community gets a good BitTorrent client. Of course, this would depend on other developers, but we will see what happens.

8 FeedParser

For parsing RSS feeds, we've had a huge amount of success with `feedparser.py`. It's a feed parser designed to be used either separately or as a library. It parses the RSS feeds and gives them to us as useful data structures. So far the library has just worked for us in almost every situation.

The only thing we've had trouble with has been that feedparser derives a new class based on python dictionaries. It does this so that it can treat a number of different keys as the same key. For instance, `url` and `location` are treated as the same key, so that if you set either one of them, `parser_dict["url"]` will give the value. Unfortunately, this aggregation is done at read time instead of write time. This makes the dictionary a bit slower to use, but more importantly, it's meant that the == operator doesn't have the behavior that you might naively expect. We've had to write a fairly complicated replacement for it which is probably much slower than == on two dictionaries. We may change this behavior going forward to change the keys when writing to the dictionaries instead, but I will resist it until we have profiling data that shows that it slows things down.

9 Unicode

python currently has two classes to represent strings. The first is `str`, which is a list of bytes, and the second is `unicode`, which is a list of characters. These two classes automatically convert back and forth as needed, but this conversion can both get the wrong value and can cause exceptions. This usually happens because the automatic conversion isn't quite the conversion you were expecting. It actually assumes ASCII on many machines and just throws an exception when you mix a `str` object and a `unicode` object with characters greater than 127.

Unfortunately, these sorts of bugs rarely show up for an English speaker because most text is ASCII and thus converts correctly. Therefore the bugs can be hard to reproduce and since can happen all over the place.

A reasonable solution might be to use `unicode` everywhere. This was our general policy for a long time, but there were exceptions. The first was developer forgetfulness. When you write a literal string in python, unless you specify that it's `unicode`, it creates a `str` object. The second exception is that there are certain python classes that only work with `str` objects, such as `cStringIO`. For these classes we would convert into `strs` of a certain encoding, but we would sometimes forget and we would do multiple conversion steps in some cases. Thirdly, there are OS differences. Specifically, filenames are actually different classes on different OSes. When you do a `listdir` on Windows, you get `unicode` objects in the returned list, but on Linux and OSX, you get `str` objects.

Our recently introduced policy is twofold. First is that you can use different types of objects in different places. We define three object types. There's `str`, there's `unicode`, and there's `filenameType`. `filenameType` is defined in the platform-specific code, so it is different on the different platforms.

In the platform-specific code, we also provide conversion functions between the different types. There's `unicodeToFilename` and `filenameToUnicode` which do the obvious conversion. They do not do reversible conversions, but instead provide a conversion that a user would be happy to see. We also have `makeURLSafe`, and because we need to undo that change, `unmakeURLSafe`. `unmakeURLSafe (makeURLSafe(obj)) == obj` if `obj` is a `filenameType`. We will see if these conversion functions are sufficient or if we need to make more functions like this.

The second part of our policy is that we enforce the types passed to and returned from many functions. We've introduced functions that take a passed-in object and check that they're of the right type and we've introduced decorators that check the return value of the method. In both cases, an exception is thrown if an object is of the wrong type. This means that we see many bugs much sooner than if we just waited for them to be found by people using other languages.

There has been a period of transition to these new policies, since all the exposed bugs have to be fixed. We're still going through this transition phase, but it's going well so far.

10 More info

You can learn more about the democracy project at `getdemocracy.com` and more about the Participatory Culture Foundation at `participatoryculture.org`.