Entertainer - Developer's documentation

Lauri Taimila

05/27/2007

Contents

1 INTRODUCTION

This document is written for those, who are interested in technical side of the Entertainer Media Center application. Entertainer aims to be a full solution for accessing your multimedia easily via remote control right from your livingroom couch. In this document multimedia refers to music, photographs and videos. Entertainer also includes a private video recorder, which means that Entertainer can be used to watch Live TV and it records your favourite shows for later viewing.

Entertainer is designed to be easily extensible, robust, easy to use and install. The most important keywords of the prject are *simplicity* and *usability*. Media Center application shouldn't require any config file modifications from user. That's why Entertainer provides easy to use GUI-applications for this purpose. Entertainer uses GTK-library for graphical user interface, which means that it's most suitable for GNOME and XFce environments, but it also works on other desktop environments like KDE. Why I chose GTK over QT... well Kentertainer just sounds stupid. ;)

Entertainer uses many existing projects like `Gstreamer`, `Clutter`, `SQLite` and `Universal feed parser` just to name few. Most of the external projects are implemented with C, but Entertainer itself is implemented with a high level language called *Python*. The whole application is implemented strictly in object-oriented manner. This means that everything is modeled as an object.

Entertainer is an open source project released under GPL-licence version 2. This means that you don't have to pay for it and you are allowed to modify the source code and redistribute it under certain limitations. Please, see GPL-licence for more information.

2 HIGH LEVEL ARCHITECTURE

Entertainer uses client-server achitecture and it has two main components called *frontend* and *backend*. The backend is a daemon (kind of) server process that runs in the background all the time. It is respossible of keeping the *media library cache*, RSS-feeds and TV-Guide up to date and recording scheduled TV-shows. In other words, backend is respossible of everything else, but displaying media on the screen and interacting with the user. Frontend is a GUI part of the application and it displays photographs, plays music and videos. Frontend can be used via remote control and keyboard. Figure 1 illustrates the high level architecture of the Entertainer media center application. There are also other processes included to the Entertainer. Those extra processes offer stuff like system tray icon and GTK-interfaces for preferences.

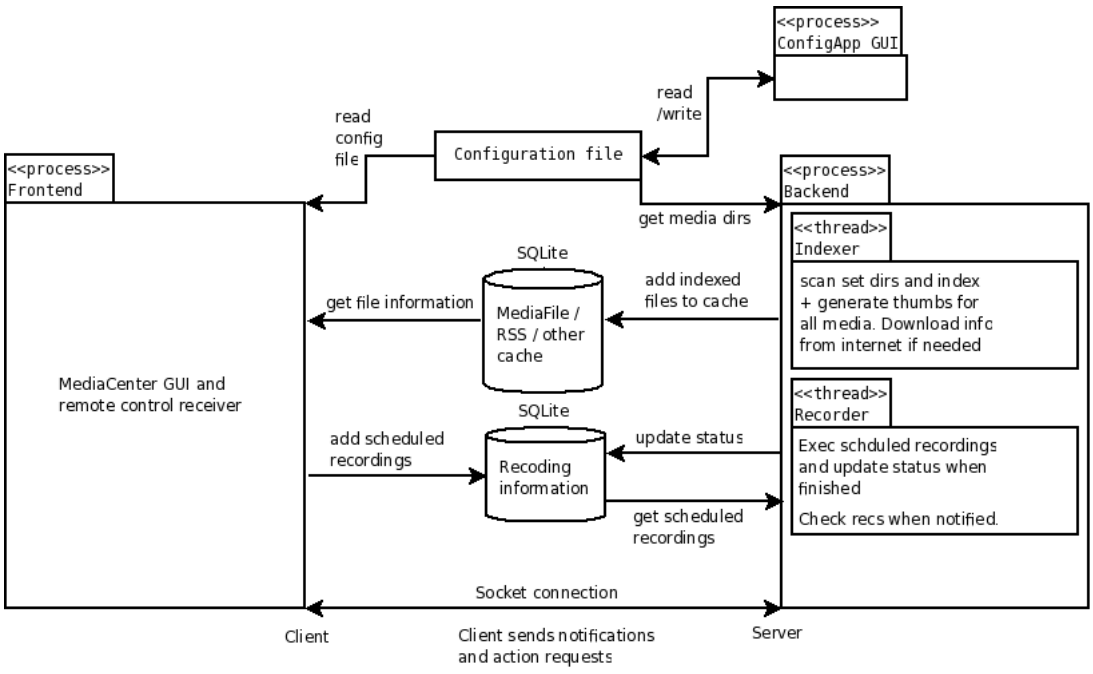

Figure 1: High-level architecure

As figure 1 shows, the high-level components of the Entertainer are *Frontend*, *Backend*, *Media library Cache* and *Configuration tools*. Backend can communicate with other processes via sockets. Frontend connects to backend when it's executed. It sends simple messages to backend as needed. For example, when user selects a TV-

show for recording the frontend sends a message to the backend, which handels the scheduling and recording of the show. Also configration tools notify backend that config files has been changed. This way backend knows to read them again and update it's internal state.

Content can be imagined in between of backend and frontend processes. Backend process indexes files, downloads metadata and generates thumbnails. This activity keeps cache files, which are SQLite databases, up to date. Frontend reads the cache and doesn't care about backend. Backend does also scheduled operations including updating feed cache and guide, which both are SQLite databases as well.

3 CACHE

Entertainer has a cache almost for every data type that it handles including images, videos, music, feeds, tv-guide and recordings. All caches are implemented as SQLite databases. Every cache is stored in it's own file that has `db` extension. Caches are stored in the following directory: `~/.entertainer/cache`.

3.1 Descriptions

3.1.1 Image cache

Image cache contains all information of the images. It has two tables: `Image` and `Album`. Album has a title and description and it contains images. Image table contains information on images. One row matches to one file. Entertainer considers folder as an album and images in that folder belog to that album. Subfolder are own albums and there is no deep hierarcy in image cache. In figure **??** is ER-diagram of the image cache database. This database is stored in `~/.entertainer/cache/image.db`.

3.1.2 Music cache

Music cache contains everything related to the music library. Entertainer's music library contains Albums, Tracks, Artists and Playlists. All this data is stored in music cache database, which is stored in `~/.entertainer/cache/music.db` -file. In figure **??** you can see the structure of the cache. As you can see, there is only `Track` and `Playlist` tables. MusicLibrary handels the abstraction of albums and artists. This cache is created in `backend/components/mediacache/music_cache.py`.

3.1.3 Video cache

Video cache contains information on video files. These do NOT include recorded TV-shows, but only the files that are in video library. Video cache contains two tabels `videofile` and `metadata` (actually these should be merged). `Videofile` table contains basic information on video and `metadata` some extrnal information. Metadata table contains type field, which determinse the type of the videofile. Allowed values are `MOVIE`, `CLIP` and `TV-SERIES`. According to this field the video is represnted as a `Movie`, `TVEpisode` or `VideoClip` object in VideoLibrary (client side). This cache database is stored in ∼`/.entertainer/cache/video.db`. In figure **??**

3.1.4 Feed cache

Feed cache includes all feeds and entries of those feeds. Entry is a one "post" of the feed. Backend fetches feeds time to time and updates the cache. Cache keeps up to 50 entries per feed. In figure **??** you can see the structure of the Feed cache. Cache is stored in ∼`/.entertainer/cache/feed.db`

3.2 ER-diagrams

4 BACKEND

4.1 Core

Backend uses *message-bus architecure*. Core of the backend consists of *message bus*, *messages*, *message scheduler*, *connection server* and *message bus proxy*. All components of the backend are registered to message bus. Backend components communicate with each other by sending messages to the message bus. Also client processes can register them selves to the message bus via message-bus proxy. This way clients can trasparently communicate with backend components. Behind the curtains this communication is implemented using sockets, but clients doesn't need to know that. All they know is that they can send and receive messages. Figure 2 illustrates the architecture with a high level diagram. In the following sections we dicuss each component in more detail.

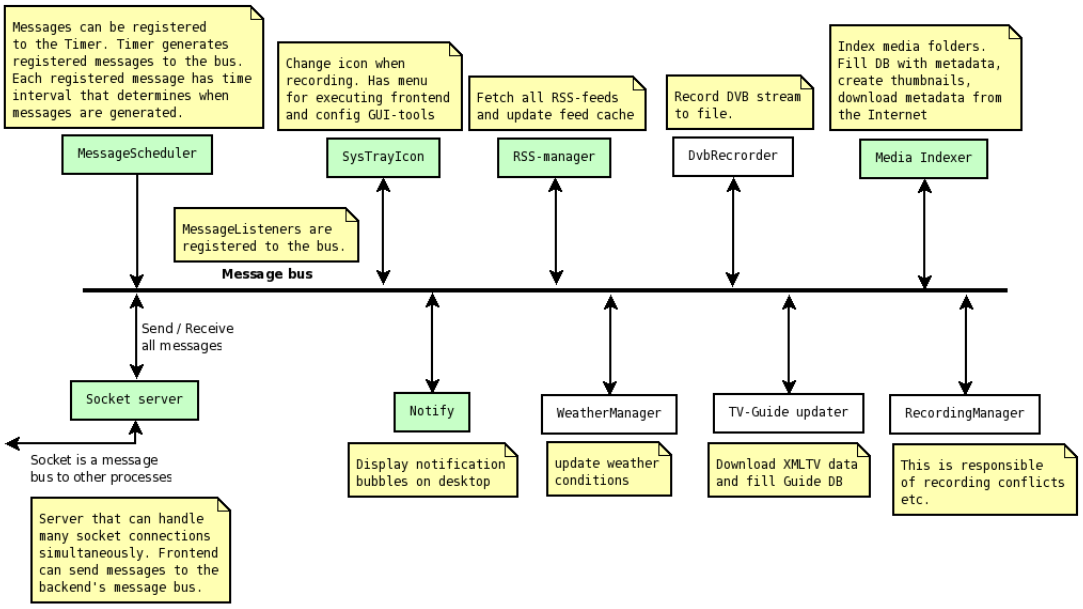

Figure 2: Backend architecure

4.1.1 Message bus

Message bus is an object (type of `MessageBus`) that is resposible of transmitting messages between registered components. Every component has to implement `MessageHandler` interface so that it can receive messages properly. `MessageBus` has more also more advanced features than just trasmitting messages to all components. Every component has to determine message types that it is interested in. Components also determine the priority level for each message type. This way `MessageBus` doesn't have to transmit all messages to all components. Instead, it notifies only those components that are really interested in some particular message type. For more, `MessageBus` notifies those components in the order of priorites. For example, it is more critical to start recording as soon as possible than display notification bubble that recording has been started. That's why DVB-recorder tells the `MessageBus` that it has `VERY_HIGH` priority for `START_RECORDING` message type and similary notification system says that it has `VERY_LOW` priority. Now, when `START_RECORDING` message occurs, the `MessageBus` knows that it should first notify DVB-recorder and after that notification system. MessageBus is locked in such a way that only one message can occur at the time on the bus.

4.1.2 Message

Message is an object of `Message` type. These are the objects that are transmitted via `MessageBus`. Each Message has a message type and possibly some user data. Data part can contain arbitary data or it can be empty. Message types are defined in `MessageType` class. I suggest to take a look of that class, because it provides more information about the meaning of different message types. When component wants to send a `Message` to the `MessageBus` it needs to create a Message object and after that call `MessageBus`-object's `notifyMessage(message_obj)` method.

4.1.3 Message scheduler

`MessageScheduler` object generates messages to the message bus. When backend starts, it registers message types and intervals to the `MessageScheduler`. After that

messages are generated to the message bus in given time intervals. When message types are registered to the `MessageScheduler`, it creates a random time interval for each message type. This prevents all messages to be generated at the same time to the message bus. Random time interval is waited only the first time. After this messages are generated with given time interval. This scheduler is used for updateing guide and feed cache every now and then. It is also easy to add new features that require frequent updates.

4.1.4 Connection server

`ConnectionServer` listens incoming client connections and binds connected clients to the MessageBus. Every client is handled in it's own thread. This class is strictly binded to the MessageBusProxy class which we will discuss in next section.

4.1.5 Message bus proxy

`MessageBusProxy` object hides the complexity of sockets. `MessageBusProxy` is used in client process (Frontend for example) and it is abstraction of backend's `MessageBus`. In other words, client processes can act like backend components simply by using object of this class. This makes communication between components very easy no matter in which process the components are actually running.

4.2 Components

4.2.1 System tray icon

(Partly depricated information) System tray icon is a small icon that is usually displayed in notification area. `SystemTrayIcon` class implements this icon and it's pop-up menu, which allows user to manage Entertainer media center. Pop-up menu

contains Log viewer, which can be used to see Entertainer's log. It also allowes easy way to execute configuration GUIs and frontend process. SystemTrayIcon class implements MessageHandler and it is interested of messages `RECORDING_STARTED` and `RECORDING_STOPPED`. When Entertainer is recording a TV-show the system tray icon changes to the recording icon. This way user can easily see that Entertainer is currently recording.

4.2.2 Notification system

Notification system displays notification bubbels on the desktop. Notification system is implemented with `Notify` class that uses *libnotify* library. Notification bubbles are displayed when Entertainer backend begins or ends recording and also if there are conflicts between scheduled recordings. It's easy to add new notifications by modifying the `Notify` class. `Notify` implements `MessageHandler` and it's registered to the message bus when backend starts.

4.2.3 Feed manager

Feed manager is responsible of updateing feed cache. At the moment, only RSS-feeds are supported. `FeedManager` and `FeedFetcher` classes implements this feature. `FeedManager` implements `MessageHandler` interface and it is registered to the message bus. When `FeedManager` receives `UPDATE_FEEDS` message, it executes a new `FeedFetcher` thread that does the actually cache updating. When update is done, `FEED_CACHE_UPDATED` message is emitted to the message bus.

5 FRONTEND

5.1 Architecture

Not written yet...

5.2 Graphical User Interface

Graphical user interface is one of the most important parts of the Media Center application. Entertainer user interface is implemented with Clutter and GTK. Frontend consists of GTK-window that contains toolbar, menu bar, status bar and the most important part ,*Clutter stage*, which is displayed via Clutter-GTK widget. Frontend also includes preferences dialog, content management dialog and log viewer. All these three sub components can be also executed as separated processes without executing the whole frontend. These dialogs are discussed more depth in chapter 6.

In this chapter we concentrate to the Clutter part of the GUI. The main class of the GUI is *UserInterface* that is responsible of managing screen changes and creating all screens. *UserInterface* object also receives all user actions. When key press is received, UserInterface object method `handle_key_press_event()` is called. This method decides how to react to key press action. Usually it just forwards event to the current screen. Current screen has internal logic for events.

Entertainer UI layers

Entertainer UI can be thinked as a set of layers that are piled on to each other. This screen stack is illustrated in the figure 3.

Figure above shows how video is always displayd under menu overlay and screen widgets. If video is not running the whole layer doesn't exists. In this case the

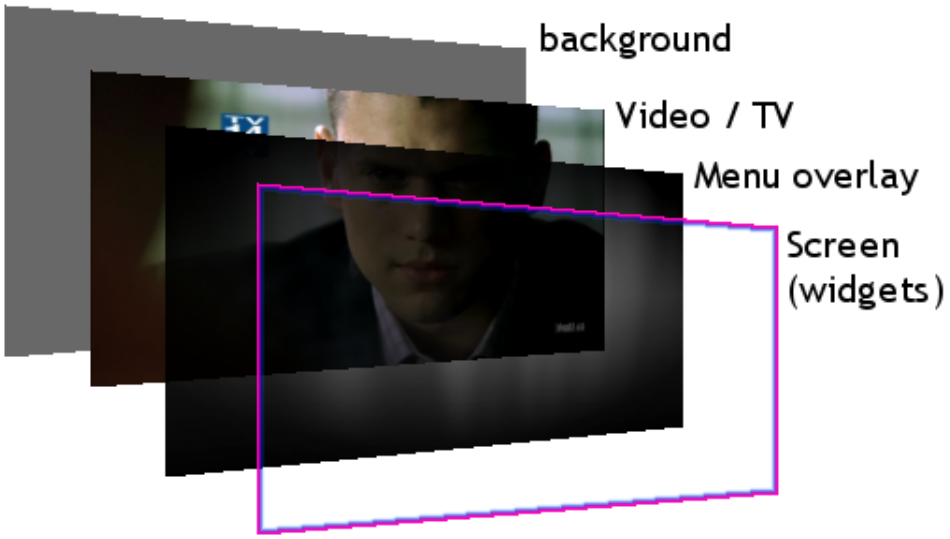

Figure 3: Entertainer GUI layers.

background layer is displayed through menu overlay image. If current screen is OSD type, then menu overlay layer is hidden. This way OSD screens allow user to see video playback on screen as it should be seen. Menu overlay layer is drawn only when user is navigating in menus. This figure omits the possibility of two screens to be displayed at the same time. This happends when another screen is type of DIALOG. Dialogs are shown on top of the current screen.

Screen life cycle

Entertainer has two important concepts: *Screen* and *Transition*. These both are also interfaces that many classes implement. Screen is a set of Clutter actors, which together create one view. View is what users sees on screen. Transition is a mechanism to animate screen switches. Say, when user selects photo album from the photo album list, we need to switch from the photo screen to the album screen. When this happens. We call UserInterface objects's `changeScreen()`. This call creates a new screen (in this case album screen) and then gives current screen and new screen to the Transition object, which animates screen switch.

Screen is created when it is needed, in other words, when user enters to that screen. When screen object is created it is added to the Clutter stage object. It's also pushed into the *Screen history*, which is a stack of Screen objects. `ScreenHistory` object keeps record of recent screens. This allows user to navigate "back" just like in web-browsers. History size can be changed from the preferences dialog. Screen is added to the stage when it is created, but when we remove it? There are two possibilites: When history size is exceeded the oldest screen is removed. Also when user navigates to "back", the current screen is NOT added into the history and it is removed from stage immediately. If we would add screens into the history when navigating back, then we would create an endless loop of two screens.

Creating a new Screen or Transition effect

If you want to create a new view to the Entertainer, say e-mail reader, you need to write a new `EmailScreen` class that implements `Screen` interface. If you want to create a new transition effect you just implement `Transition` interface which contains only two methods. It's recommended to look into the `FadeTransition` class to get the idea how transitions effects work. **Notice** that if animation direction is "backwards" then you need to remove old screen from the stage after animation has been displayed. This can be done by using 'completed' signal with Clutter `Timeline` object.

6 EXTRA TOOLS FOR ENTERTAINER

6.1 Preferences GUI

Preferences GUI allows user to configure Entertainer the easy way. Actually it's just a simple GUI for editing `/.entertainer/preferences.cfg` -configuration file and all the same things can be done with text editor directly. Regardless of this, it is still recommended to use GUI application, because it notifies the backend process that changes has been made. This way backend knows to reload configurations. If changes are made by hand with the text editor, then backend needs to be restarted to changes take effect. Preferences GUI is a standalone application and it can be used even if Entertainer is not running.

6.2 Content management GUI

Content Management GUI is exactly like Preferences GUI, but it is used to manage content of the Entertainer. This GUI application allows user to add and remove content of the Entertainer easily. Content Management GUI edits the `/.entertainer/content.cfg` -file and also notifies the backend when changes occur. Just like the Preferences GUI.

6.3 Message bus notifier

This is a small command-line application that allows user (or other processes) to send messages to the backend's message bus. This was written mainly for debug purposes, but it can be used for other things too. The usage of this application is straight forward. Command line tool takes only one paramter, which is the type of the message that will be emitted to the message bus. Here is an example how to force feed cache to be updated immediately:

```
entertainer-messagebus-notifier.py --message=UPDATE_FEEDS
```