
# Media management

# Contents



This document covers the management of media content in the Apertis platform. There are several types of media content to handle in the platform: images, audio, video and documents. We can identify the following operations with media:

- **Media Indexing**: extracting metadata from media content and store it in a format that allows fast retrieval.
- **Media Browsing**: locate the media content and access its metadata.

Solution provides a general overview of the technologies used, like an executive summary of Appendix: Media management technologies, as well as a high level view of the solution proposed. Additionally, it exposes in detail the media management requirements in the Apertis platform, providing an analysis as well as a solution to each requirement, which might involve modifying existing technologies or even create new ones.

Although this document is mostly focused on the media content, the technologies introduced are related with other features in the platform like global search, which allows to search not only in media content but also in applications, messages, calendar events, etc. For details on global search please check its specific design.

Appendix: Media management technologies is mostly used as reference material from other sections of the document, so it is not necessary to read from start-to-finish. It has a detailed description of the current state of the technologies

used for media management without including specific requirements, additions and modifications described on Solution.

This document assumes the adoption of a media-centric approach for applications (every media source provider will have its own application for browsing and playback). This provides a customized fully-featured experience for each of the media provider services. See below the list of media content providers that have been identified as requirements, these services will be analyzed in more detail in chapter 2 Solution.

- Local Storage.
- Removable Storage Devices.
- CD and DVD.
- DLNA (UPnP).
- Media Online Services: YouTube, Shoutcast, Dropbox, last.fm, podcasts, etc.
- Bluetooth AVRCP.

## Solution

The following sections will provide a high level view of the technologies and solutions followed by a detailed analysis of the requirements for media content sources supported.

### Technology and Solution Overview

This document looks at what changes could be made to the open source components to better support the Apertis use cases, it is important to note that those changes may not be possible for the scope of this project and may not be accepted upstream.

See below an enumeration and a brief overview of the main technologies used in the design:

- **Tracker** is a central repository for user information. It is made of several components: Tracker Miner, Tracker Extract and Tracker Store. Tracker Miner automatically crawls for media content files. Tracker Extract gathers useful metadata from these files and it stores this metadata in the Tracker Store database. Metadata can be retrieved from the Tracker Store with SPARQL queries. See Tracker for more details. Although this document will only focus on the Tracker features specific to media indexing, Tracker can be used to store other information as well, like applications, messages, calendar events, etc. or in general any information that is worth to share between applications.

- **Grilo** is a simple API for browsing media content and provide media content metadata. Grilo layer helps to hide the complexities of Tracker and its query language, by focusing on media content (since Tracker is much more generic). See Grilo for more details.
- **Tumbler**. It is a service for accessing and caching thumbnails. See Thumbnail management for more details.
- **libsoup** and **librest** are libraries simplifying the creation of HTTP client/servers and the access to REST-based services respectively. See Librest and Libsoup.
- **libgdata** is a library implementing the Google Data Protocol. It provides access to Google Services like YouTube and Picasa, among others.

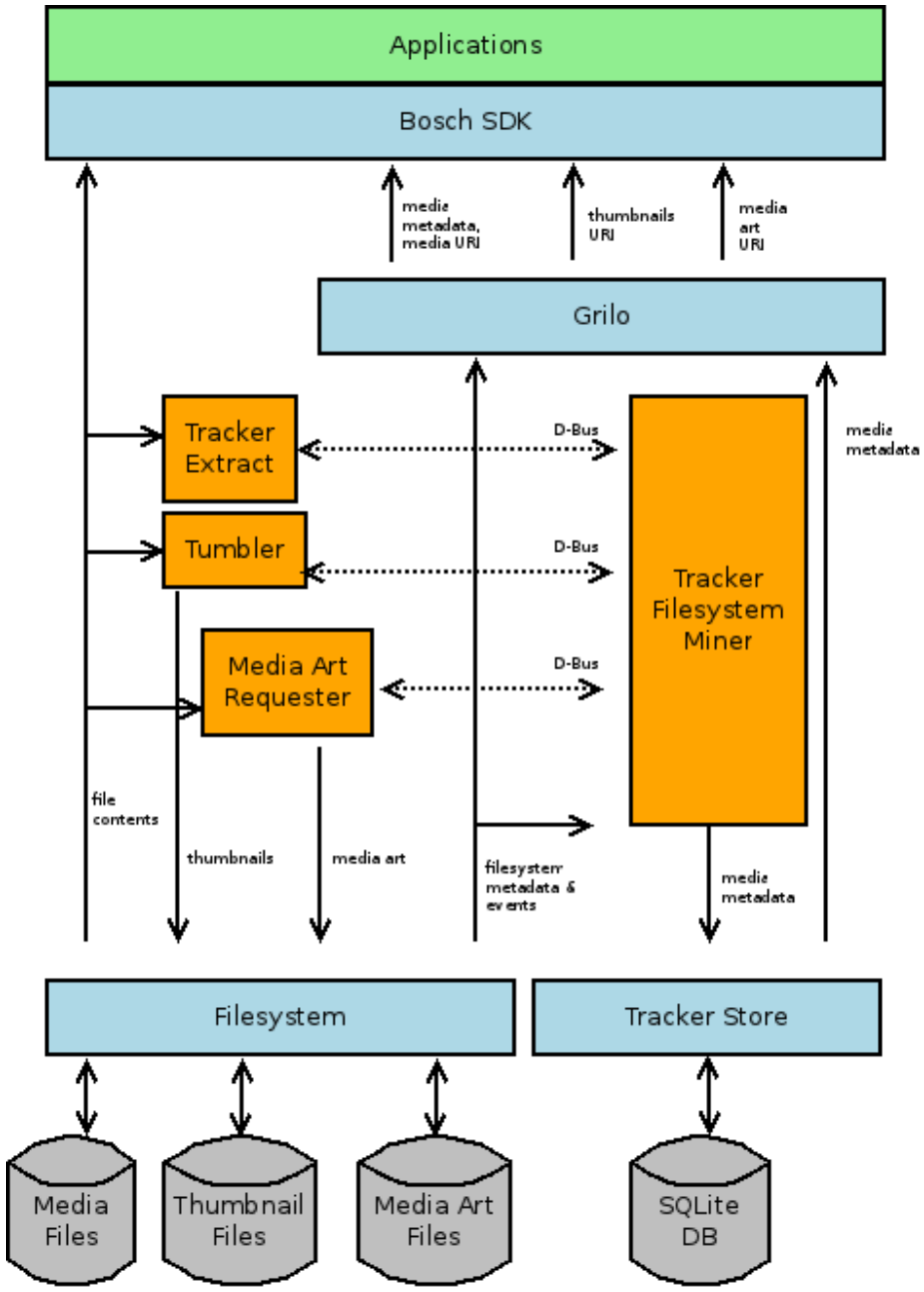
The proposed solution combines Grilo, Tumbler and Tracker for locating media content and retrieving its metadata from the local system and removable storage. Tracker does the heavy work: filesystem crawling, metadata extraction and metadata storage. Grilo is a simple API which lies on top of Tracker, used by applications to discover media content and its metadata. Tumbler is responsible of thumbnail generation.

Tracker's scheduling algorithms needs to be modified to support the requirements. The goal is to prioritize the different tasks of information retrieval, so what applications need first must be retrieved first. There are different cases depending on the specific requirements:

- Prioritization done automatically by Tracker in a hard-coded way (not configurable), like gathering all metadata from filesystem (filename, size, modification time, etc.) before extracting metadata from the file contents.
- Prioritization done automatically but configurable, like prioritizing the indexing of music files over video files.
- Prioritization influenced or requested by upper layers. In some cases, upper layers need to provide some clues about what needs to be done first or what is more important, like a picture viewer application boosting priority to metadata extraction of image files (instead of the default which could be music files).

The details on Grilo API stability can be checked in the API stability design. In summary, it is still a young API and its API will be broken on version 0.2. Under this situation, it might be convenient to layer an Apertis SDK API on top of the Grilo API to improve API stability for the application layer.

See this illustration for an overview of the general architecture. Some of the components listed will be introduced with more detail in the following chapters.




**Local Storage Media Source**

**Requirement R1**. Support local storage as a media source.

**Analysis**. The system has storage memory to store media locally. Locating

media content in the system local storage and retrieving its metadata is required.

**Solution**. Collabora proposes a combination of Tracker and Grilo, as a powerful solution for this endeavor (see section 2.1). Tracker can be reviewed in detail in Technology and solution overview, and Grilo in chapter 3.3. Upper layers will just interact with the Grilo layer, which is a simple API specialized in media browsing hiding the complexity of Tracker.

Grilo allows to browse, search and locate the media content in the system. The application can access the media content through the filesystem API via the URI (Uniform Resource Identifier), e.g. file://home/username/Music/song1.ogg.

See requirement R5 for comments on public and private content.

**Status**. Satisfied.

### Media Browsing Requirements

**File-system based browsing Requirement R2.** Support filesystem based browsing for early access.

**Analysis.** This is required in order to quickly render a user interface to the user, for example when plugging in a USB flash device. Removable devices are potentially slow and it takes time to actually index and capture all metadata, so information like author and album could not be available on time. Therefore, a filesystem view should be available through the media browsing framework itself at least, in order to provide quick access to the media content by browsing the filesystem structure; as opposed to other ways to browse content using the metadata (by author, album, etc.).

**Solution.**

There is a Grilo Filesystem plugin. This is the fastest way to access the filesystem entries in the device. Content would be available soon after the filesystem is mounted on the system. Additionally, this plugin already monitors and reports for changes on the directories or files. One disadvantage of the Grilo Filesystem plugin is that it could be hard to access the metadata or get notified about changes in an efficient way.

Another solution would be to use Grilo Tracker plugin. Grilo plugins provide access to the media content in a hierarchical way. Grilo Tracker plugin has two modes of hierarchical navigation, one based on categories and another one based on the filesystem. The latter one provides the info in the same structure as it is stored in the filesystem. It allows to browse from a root folder or from specific folders. However, the information has to be previously available in Tracker Store for this to work. To minimize this delay, Tracker scheduler will be changed to get filesystem information before other media metadata. Obtaining the filesystem information is very fast compared to the extraction of the metadata (which involves reading the file contents). Some timings have been gathered to show this

fact, check the table in [Appendix: Questions Appendix: Questions (#appendix-questions–answers) Answers Answers] for the details. This solution plays nicely with requirement R3 (to get notifications of ready metadata as soon as it is available) and with R8 and R13 (regarding the scheduling of operations like crawling, metadata extraction, etc.).

The last solution provided looks more promising than the first one, since it integrates better with the overall architecture and it does not have a negative impact in other requirements.

**Required work**.

Grilo Tracker plugin will need to be modified to operate as specified in the solution, and it actually depends on requirement R8 and R13 related to Tracker scheduling. Additionally, an API would need to be provided to change easily from one hierarchical model to the other on run-time. See Grilo Media Source Plugins for more information about Grilo.

**Status**. Satisfied.

**Notification on metadata changes** **Requirement R3.** Metadata info can change during run-time, so the media browsing API has to notify whoever is interested through some mechanism when these changes happen.

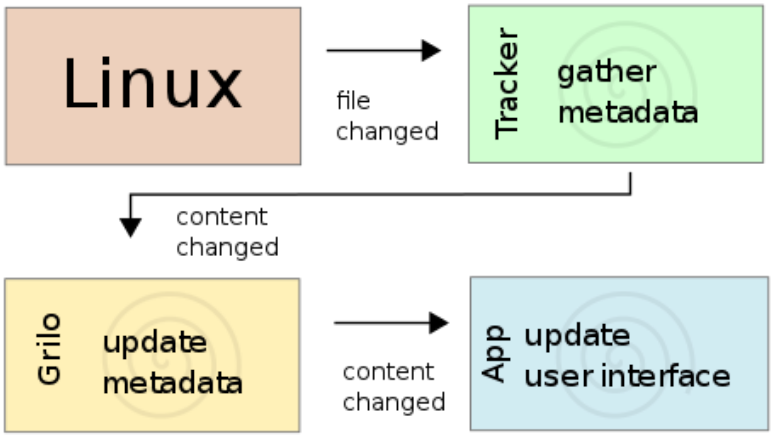
**Analysis**. The indexing process is asynchronous, it can happen that media content gets its metadata updated while the content is already being shown to the user.

Tracker internally uses the file system monitor service provided by the Linux kernel, which is a very efficient way to get notified about changes on the filesystem and it is not doing active polling.

Once Tracker Miner gets notified about a change in the filesystem, it will check what needs to be done depending on the specific type of change. For example, if a new file is added it will determine if the new file is interesting for Tracker or not, much in the same way it does when crawling through the filesystem looking for files to index. In the case of a notification of a deleted file, it would remove its associated information in Tracker Store. In the case of modified files, it would extract the information again.

**Solution**: Grilo tracks changes in Tracker Store by subscribing to the **GraphUpdated** D-Bus signal from the Tracker Store service (see Tracker storage for more details). Grilo processes this information and provides notifications of changes on media content. See the following illustration for an overview of the interaction between the components involved.

**Status**. Satisfied.




**Paged queries** **Requirement R4**. Provide queries to request content information by pages of fixed size.

**Analysis**. There are potentially lots of results in a query for browsing media content. Therefore, a mechanism to get the results incrementally as needed is required.

**Solution**: Grilo supports paging in all requests via skip and count numbers. Internally Grilo uses both mechanisms provided by Tracker SPARQL (OFFSET / LIMIT modifiers in the SELECT SPARQL statements and TrackerSparqlCursor). See Grilo for details on Grilo.

**Status**. Satisfied.

### Media Indexing Database Requirements

**Media indexing of shared and private files** **Requirement R5**: The system must be capable of indexing shared and private files. Shared files can be accessed by all users in the system. Private files are only accessible for the user who created them initially.

**Analysis**. The reason of this requirement is to guarantee a minimum level of data confidentiality among the users in the system (for example regarding personal photos and documents). This would be even more important if we consider Tracker could be used to store other information as well.

We assume there are folders which are public (shared and accessible to all users in the system) and folders which are private (only accessible to the owner of the folder). Due to the existence of private content, each user must have its own Tracker database for storing metadata.

In the future, the device may have different configurations for privacy. First case would be that all user files are public, and they should be available for indexing by all other users. Second case, where each user's files are private. A third case would be that the user would be prompted which files to make public. Those public files should be available for indexing by all.

**Solution**. Due to Tracker's architecture, it is not neither easy nor efficient to add the capability to have more than one database managed by a Tracker instance. Due to the nature of SPARQL queries, it would require very complex database joins and performance would suffer. SQLite is known to be very slow in such setup. Additionally, Tracker developers are not keen on accepting this change, since Tracker had a similar behavior in the past, and it was abandoned due to multiple problems. Therefore, this would probably produce a fork of the Tracker version in the middleware and it would be a huge increase on maintenance cost. In summary, Tracker managing multiple databases does not seem feasible for now.

The proposed solution is to have a just a Tracker instance for each user, which holds both the metadata for private files belonging to the user and the metadata for public files.

A drawback of this solution is the additional space needed, since the metadata for the public files is stored in each Tracker instance. Due to the local system storage in the automotive industry being very expensive, we could think there will not be really many public files to index. Additionally, the database space used to index those public files is really minimal (0.03% as shown in Table 1) and the number of potential users in a system is very reduced. In the case of removable storage files, that will be treated as public files. The solution for indexing and thumbnailing will be covered in Indexing database on removable device.

Another drawback is the extra processing required to index the public contents for each user. There are also some risks about overloading too much the system in this case, but those could be managed in the Tracker scheduler.

In the case of the thumbnails, it is possible to share the thumbnails objects, since they are stored in files. Also note a Tracker instance would need to run for every user logged in into the system; only Tracker Store and Tracker Miner though, not Tracker Extract which automatically shuts down when idle.

To handle future privacy configurations, file permissions should be set accordingly, and Tracker configured to index files of all users. Thumbnails should be generated and stored in a central location where they could be retrieved by all Grilo instances. Also, AppArmor profiles should be probably tweaked to allow Tracker instances to read other users'files.

**Status**. Satisfied.

**Database version management** **Requirement R6**. The system should be able to cope with database version updates.

**Analysis**. Database version updates is very tricky regarding Tracker, since the updates could happen in different levels:

- **SQLite database level**. Every effort is made to keep SQLite fully backwards compatible from one release to the next. Rarely, however, some enhancements or bug fixes may require a change to the underlying file format. There are two types of updates, and you can differentiate by comparing the version numbers of the old and new libraries.
- First digit update on the version number. A reload of the database will be required. Therefore, the contents of the database has to be dumped into a portable ASCII representation using the old version of the library and then reload the data using the new version of the library. So we would need either a backup done with the old version or have the old version distributed to do a dump of the database. Last first digit change was on June 2004.
- Second digit update on the version number. It is backwards compatible, so newer versions will be able to read and write older database files. But there is no guarantee of forward compatibility. Last second digit change was on July 2010. Provided we want to upgrade to the new version, the update of the database could be done with just the new version.
- **Tracker RDF mapping level and Ontology level**. First is related with the mapping from RDF database model to a relational database model (SQLite in this case). Second is related with changes on the models defining the domains, objects, its properties and links. Both of these changes are tracked by the Tracker database version. If the version is different, then Tracker must perform a full re-index, as there is no backwards compatibility. However, by using the Tracker journal, it would just be like a reload of information, since the journal is like a log of all transactions done in the database. This does not guarantee all the information will be retained, since due to changes in the ontology, some data might be invalid on the new model. There is also another way to cope with ontology changes, via ALTER TABLE directly in SQLite, but this requires some custom coding to be done and it is very complex to handle all the cases in ontology changes. The last time the Tracker database version was changed was in version 0.9.38 (February 2011). See Tracker storage.

It is clear that changes in the Tracker database version is a larger risk than changes in SQLite. Let us analyze various scenarios:

- If Tracker Store just holds indexing information, this could be regenerated by re-indexing, so there would be no real data loss on an database version update.
- If Tracker Store keeps information entered by the user, like user tags, then

it would be lost during a full re-index. To prevent this, an ad-hoc tool could be implemented to convert this information to the new database version.

- Often the manufacturers or distribution maintainers decide to not deploy new changes on the ontologies to avoid these database update problems. Anyhow, some changes could be supported via some custom code, like adding / removing properties; but others affecting the domains or class hierarchy are much harder to handle. Each case of ontology change needs to be analyzed particularly.

**Solution**. It is a bit of a case by case trade-off between storage space for the Tracker journal vs CPU time for re-indexing. Assuming we cannot use unlimited storage space on the device, then using the Tracker journal is not an option. The way to handle database version updates is to analyze them on a case by case basis. There are several points to evaluate like what is the impact of the update in the existing database, what type of data it is (generated data vs user data), and what solutions are possible to keep the data (either implementing ad-hoc tools to migrate data or make use of already available tools).

See more details on Tracker Journal in Tracker storage.

**Status**. Satisfied.

**Indexing database on removable device** **Requirement R7.** Storage of the indexing information for removable storage in the removable storage itself.

**Analysis**. The main motivation for this requirement is to avoid using the scarce expensive storage in the system. Here are some general problems and risks with this approach:

- **Data corruption**. The user can disconnect the removable device at any time without properly syncing. For a holistic view on robustness see Robustness document. See points below to consider:
  - Risk of corruption for user files and filesystem metadata. The device could have been ejected in the middle of a write operation. The device would not be usable unless its filesystem is recovered, and the user could lose some or all the files.
  - Journalled filesystems work more reliably, guaranteeing at least the filesystem will not be left in an inconsistent state. In any case, the user is the one who chooses the filesystem for its own USB flash devices, and not the system, so there is not much to do here since the FAT filesystem is typically the de facto standard used in USB flash devices, which is not a journalled filesystem. Another point is that USB flash devices are typically optimized for FAT filesystems.
  - Write cache disabling for the USB flash device decreases the data corruption risk, but the risk does not disappear. The user could

still eject on the middle of a write operation. As a result of the disabled cache write operations will be slower. Additionally, USB flash manufacturers tend to lie regarding sync requests.

- Note: the size of thumbnails has not been considered in this section, since the thumbnail storage is independent from the metadata storage. However, as we can see in the modeling spreadsheet, the size of the thumbnails is really significant, even more than the metadata size, so most probably it would make sense to store thumbnails and album art in the USB flash device. Therefore the risk of data corruption cannot be avoided in the end, just minimized.

**Solution.** The alternative to use a dedicated metadata database in removable storage devices was discarded due to data corruption and maintainability problems. However, thumbnails and album art will be stored in the removable storage. That is a large portion of the metadata, and will help save local storage space.

A single Tracker instance per user in local storage holding the metadata for media content in the USB flash devices.

The thumbnails and album art will be stored in the USB flash device. As we saw before, any write to a USB flash device could end up into corruption if the user does not behave correctly. A check should be added when generating thumbnails to use local storage when the removable device is full.

*Note: In the current implementation, If the device does not have enough free space, thumbnails will be generated. Album art will be generated in the local storage cache.*

The disk space usage can be controlled by removing metadata of unmounted external devices when the disk space is low and/or when the DB size exceeds a given limit.

Currently Tracker removes metadata only after 3 days, and when the disk space is low, the indexing engine simply stops. A trigger shall be added to remove metadata if the disk space is low, starting with data from removable storage devices.

Also, the default for the database size limit is unlimited. A limit will be set, to prevent waste of local disk space, and the database will purge old data when the limit is hit.

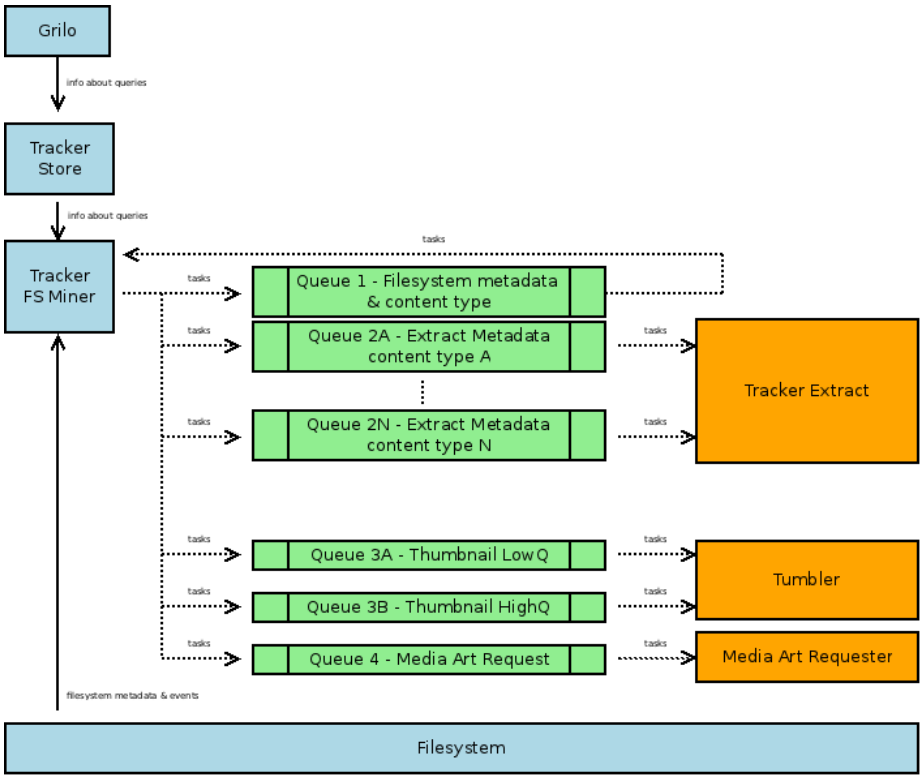
**Status**. Satisfied.

### Indexing Scheduling

There are many specific requirements related with metadata extraction prioritization. They will be analyzed in detail in the following subsections.

The Tracker Scheduler will need modifications to be able to specify priorities as well as separate the operations on different stages. Additionally some extra hooks might be needed in order to provide hints from the browsing applications. There are several ways to implement this prioritization. One way would be by an API that allows the application to explicitly give priority to certain operations or use cases. Another way would be a heuristic way based on recent queries done to the media framework. This automatic approach although initially interesting looks a bit risky, as there could be unpredictable interactions between applications. See Tracker scheduling for more details on how Tracker Scheduling works in the upstream version.

The following illustration shows an overview of how the scheduling and priorities of indexing operations works. There is a main component, the Tracker Filesystem Miner, feeding the task queues. Generating new tasks is based on previous queries, filesystem events (e.g. new file created) and as a result of crawling the filesystem. Tasks are consumed from the queues by different components in order, the lower the priority the first it gets executed. The priority of a task is determined by the type of task, which defines the queue where the task belongs. Additionally, tasks resulting from recent queries are normally placed in the front of the queue since they will most likely be a result of user interaction. Also note this design allows to do some configuration regarding the type of tasks and their priority, as well as test for other ideas during the development. Requirement R12 has more details about the abstraction of different types of tasks in the queues.




**Media Content Counters** **Requirement R8**. Provide the number of items per content type as soon as possible.

**Analysis.** To determine the number of items per content type, all files must be crawled first, and its mime type must be determined. It is not needed to do a full extract of metadata to determine the mime type, but in some cases it might be needed to read the first few bytes of a file (see Q&A for more details about determining the mime type).

Tracker crawls the filesystem for new files to be indexed, and adds these files to a internal queue. Each time a file from the queue is processed, there are two steps. The first step, which is done by the Tracker filesystem Miner, gathers metadata from the filesystem attributes without actually inspecting the file contents. In a second step, more information is extracted by Tracker Extract by inspecting the file contents, which is a more expensive operation. These steps are done for every file processed. However to meet the requirements above, we would perform the first pass for all the items found before starting the second pass for every item.

**Solution.** Collabora will add an option in Tracker's configuration to enable two pass indexing. If enabled, tracker will first crawl the whole filesystem to store files'attributes but won't try to get embedded information (e.g. MP3 metadata,

etc). A boolean property will be added in Tracker's database for files that need a 2nd pass, so Tracker knows which files needs a 2nd pass when it is done crawling the filesystem. That property needs to be written into the database (and not only in-memory) so Tracker is able to correctly resume its indexing after a system reboot. Additionally, directories containing partially indexed files will be flagged (in memory), to avoid re-crawling the whole filesystem when doing the 2nd pass (a list of all partially indexed files would be too big and consume too much memory).

This solution has been discussed with upstream developers and has great chances to be accepted.

**Status**. Satisfied.

**Prioritized extraction per content type Requirement R9**. prioritize metadata extraction per content type: first music play-list, music, video, pictures and documents. Default prioritization can be adjusted on run-time depending on user activity, e.g. if user starts browsing pictures.

**Analysis**. Current Tracker scheduling does the metadata extraction in no specific order.

**Solution**. A D-Bus interface will be added on Tracker. That interface will be used by applications to tell Tracker about their current priorities. For example, a music application will ask Tracker to index "audio/*"mime-type first.

If an application requests priority for a certain mime-type, Tracker will skip any other file while crawling the filesystem. Additionally, directories containing skipped files will be flagged (in memory), to avoid re-crawling the whole filesystem when Tracker is done indexing all files that have the priority (a list of all skipped files would be too big and consume too much memory).

When Tracker is done crawling the whole filesystem, it will do the 2nd pass indexing (see 2.5.1) on the files that have the priority (e.g. if the music application is running, the 2nd pass is done only on audio files at this point). When done, it will do the 2nd pass on all files, ignoring the filters.

If an external storage device is plugged while Tracker is doing the 2nd pass, it stops and crawls the new media first (doing first pass on prioritized files). When done, Tracker will resume doing the 2nd pass.

If priorities changes while Tracker is doing the 2nd pass, it stops and crawl directories where files have been skipped earlier. When done, Tracker will resume doing the 2nd pass.

In summary, Tracker will do the 1st pass indexing (file attributes only, no embedded metadata) on prioritized files, then 2nd pass on prioritized files, then 1st pass on not prioritized files, and finally the 2nd pass on not prioritized files.

This solution has been discussed with upstream developers and has great chances to be accepted.

**Status**. Satisfied.

**Selective prioritized extraction Requirement R10**. Prioritize metadata extraction for certain files, e.g. music files currently shown to the user.

**Analysis**. The goal is to influence the scheduling of extract operations in Tracker based on the user behavior. for example, If a user is browsing a specific folder in the filesystem, the metadata extraction of the files currently displayed to the user, must have priority over others. Additionally, the system could anticipate the needs of the user, by trying to extract metadata for next media content items in the page. This can be done by influencing the priority of extract operations in Tracker by checking the results of recent queries.

**Solution**. The D-Bus interface proposed in 2.5.2's solution will be extended to let applications give the priority on some specific files, in addition to the general mime-type priority.

The following would be implemented as part of the solution:

- **Extract normal**. The current behavior, that is without automatic prioritization of extraction based on queries.
- **Extract recent**. This will automatically request the metadata extraction for media content items returned in recent queries.
- **Extract next**. This will automatically request the metadata extraction for media content items that would result in next page of recent queries. This setting will imply "Extract recent"as a dependency.
- **Extract thumbnail**. This will automatically request the thumbnail computation for media content items returned in recent queries (or next page items if "Extract next"is also set).

The application or SDK layer would be the responsible for enabling the settings more appropriate for every specific case. Alternatively, Grilo could have extract recent, new and thumbnails enabled by default. This is a trivial change that could be decided later on during the development phase.

Solution needs to be discussed in more detail with upstream Tracker maintainers.

**Status**. Satisfied.

**Selective prioritized thumbnailing Requirement R11**. Prioritize thumbnails depending on user activity.

**Solution**. This is already covered by requirement R10.

**Status**. Satisfied.

**Multi pass metadata extraction Requirement R12**. Iterative process for metadata extraction in multiple passes: blank entry just file names, textual information, graphical information like thumbnails, information from internet, etc.

**Solution**. The proposed solution in 2.5.1 already describe 2 pass indexing. A third pass can be added the same way to create thumbnails, get information from internet, etc.

The solution needs to be discussed in more detail with upstream Tracker maintainers.

Collabora proposes Tumbler to generate and manage the thumbnailings (but not scheduling the thumbnailing). In current version, Tumbler provides a D-Bus service with schedulers to manage the thumbnails. Tumbler does not do any crawling to look for contents to be thumbnailed; Tracker will request thumbnailing operations to Tumbler. Although Tumbler has several schedulers to keep track of the thumbnailing requests with different priorities, it will be Tracker who takes care of the scheduling.

Thumbnail calculation is particularly expensive in CPU and storage resources. See the table in Thumbnail management for more detailed information.

**Status**. Satisfied.

**Concurrency configurable Requirement R13.** The scope (e.g. quantity of extracted data) within one step, grabbing the data concurrent for multiple files.

**Solution**. Tracker has a scheduler priority parameter which allows to issue new operations when the CPU is idle. Additionally there is an internal setting to set the task pool limit, which controls the number of concurrent tasks that can run at the same time. Currently this value is hard-coded to one, but it could be exposed via configuration or make it dependent on the number of cores in the system depending on Apertis' needs. Additionally there is support to adjust the amount of work to do concurrently, in order to avoid overloading the system. This is set by the throttle parameter, which basically allows to specify how many extract operations can be carried per second (see Tracker miner for more details on throttle and scheduler priority).

The operations handled by the scheduler have small granularity (a single file), so it is expected the whole system can react in time to get in / out from the idle state. The management of the idle status is done directly by the kernel, by setting the appropriate input / output priorities and CPU priorities to idle. Additionally, a specific cgroup could be set up to have more control over the resources used for media indexing.

Solution needs to be discussed in more detail with upstream Tracker maintainers.

**Status**. Satisfied.

### Thumbnailing

**Two-step thumbnailing** **Requirement R14.** Provide an additional iteration to generate metadata which is not already embedded within the content, such as thumbnails for pictures. First, use a very fast algorithm (time beats quality). At a later time, use a better more time-consuming algorithm.

**Solution**. This is dependent on requirement R12. The Thumbnailer service already supports several flavors for a thumbnail. It currently provides a normal and large size which could fulfill this requirement by using different algorithms for each size.

Requirement R12 solution includes an abstract mechanism to add additional passes. The first and second pass for thumbnail extraction could be considered as additional passes to be configured in this abstract mechanism. This mechanism will provide enough flexibility to connect to different algorithms.

Solution needs to be discussed in more detail with upstream Tracker maintainers.

**Status**. Satisfied.

**Thumbnail resolution configuration** **Requirement R15.** Resolutions for thumbnail flavors normal and high must be configurable.

**Analysis**. Currently the resolution sizes are hard-coded in Tumbler source code.

**Solution**. The list of flavors for thumbnails, as well as its resolution will be exposed through configuration files or via an API.

**Status**. Satisfied.

**Thumbnailing algorithm configuration** **Requirement R16.** The algorithm used for calculating the thumbnails must be configurable.

**Analysis**. Currently Tumbler implements several plugins for thumbnail calculation.

**Solution**. It is possible to add new plugins with specific algorithms or modify existing plugins to use other algorithms. The algorithm used for thumbnailing should be configurable. As an example, see the list of algorithms available currently through gdk_pixbuf_scale() functions:

- **Nearest**: nearest neighbor sampling. This is the fastest and lowest quality mode. Quality is normally unacceptable when scaling down, but may be OK when scaling up.
- **Tiles**: this is an accurate simulation of the PostScript image operator without any interpolation enabled. Each pixel is rendered as a tiny parallelogram of solid color, the edges of which are implemented with an-

tialiasing. It resembles nearest neighbor for enlargement, and bilinear for reduction.

- **Bilinear**: best quality/speed balance; use this mode by default. For enlargement, it is equivalent to point-sampling the ideal bilinear-interpolated image. For reduction, it is equivalent to laying down small tiles and integrating over the coverage area.
- **Hyper**: this is the slowest and highest quality reconstruction function. It is derived from the hyperbolic filters in Wolberg's "Digital Image Warping" .

**Status**. Satisfied.

### DLNA (UPnP)

**Requirement R17**. Browsing DLNA (Digital Living Network Alliance) media sources.

**Analysis.** There will be a player application in the Apertis platform to access and control DLNA media sources. This application plays the role of Controller in DLNA spec, it would be able to browse the media collection of remote Media Servers. This information is provided by the Content Directory service on the Media Server. The information provided about media content includes metadata like name, artist, date created, size, album art, etc., as well as the protocols and data formats supported by the server for that particular content item.

For more specific details on these topics see the UPnP AV (Universal Plug And Play Audio Video) architecture documentation[1].

Metadata indexing of media content in remote Media Servers is not required. Indexing is not desirable normally, since enough metadata is normally provided by the Content Directory service for browsing purposes, and local storage is scarce. Apart the amount of storage needed could be in practice very high due to the usage of remote sources.

Providing the Media Server and Media Renderer roles are out of scope for this document of the Apertis platform.

**Solution**. Collabora proposes the GUPnP framework to fulfill the requirements. The GUPnP library implements the UPnP specification: resource announcement and discovery, description, control, event notification, and presentation. On top of that, GUPnP*-*AV library is a collection of helpers for building AV (audio/video) applications using GUPnP. The GUPnP framework is licensed under LPGL v2.1 and it is written in C using GObject and libsoup. GUPnP is entirely single-threaded (though asynchronous) and integrates with the *GLib*[2] main loop.

[1] http://www.upnp.org/specs/av/UPnP-av-AVArchitecture-v1.pdf
[2] http://gtk.org/

**Status**. Satisfied.

### Online Media Sources

**Requirement R18**. Access to online media sources.

**Analysis**. Depending on the actual media source, the specific functionality and the API style provided will be different. For example, Google services like YouTube and Picasa are accessed through the Google Data Protocol. In general, most of these media sources are based on a REST based interface.

**Solution.** With few exceptions, like **libgdata** for Google Data Protocol, there are not many good options in FOSS to access specific media source online providers. However, in the worst case scenario we could use librest and libsoup, which are described in Librest and Libsoup.

**Status.** Satisfied.

### Bluetooth AVRCP

**Requirement R19**. Browsing of media content from Bluetooth devices.

**Analysis**. Bluetooth AVRCP 1.4 allows to browse media contents in the Bluetooth device. Indexing of this contents is not required.

**Solution**. This can be implemented by using the BlueZ API. Exact status about AVRCP 1.4 implementation will be covered in more detail in Connectivity design document.

**Status**. Moved to Connectivity design.

### Playability check

**Requirement R20.** Playability check. Determine if a file is playable or not.

**Analysis**. We want to avoid showing the user a file which cannot be played. It is not enough to do it through simple mime type checking, since this might lead to false positives. Minimal check for corruption and codecs is required.

**Solution**. The playability has two steps:

1) At indexing time. During the Tracker indexing process, Tracker Extract is able to extract information information about the mime type and audio / video codec for a media content file. Additionally Tracker Extract process should be able to mark the file in Tracker Store if any corruption is found on the file during the process of metadata extraction.

As an example, during the process of thumbnail extraction for a video file something similar happens, corruption or inability to decode a frame could be found when trying to decode a specific frame to use it as a thumbnail. This file would be marked as corrupted in Tracker Store.

Although the last example was about a video file, this applies to other types as well, like audio files, and in general to any file where metadata extraction makes sense. The metadata extraction process will be responsible to mark those files as corrupted in the case it was not possible to extract metadata from them.

Tracker has the flexibility to change or add new extract plugins. Therefore, it will be possible to customize or replace the plugins with more robust ones in case it is needed.

2) At browsing time. There are some checks to do for media content files before showing to the user. Check the file is not marked as corrupted. Check the file is from a known mime type. Check a compatible decoder exists in the system for the codec of the audio / video file. The list of codecs available can be obtained through the GStreamer registry.

There is an special case at browsing time, in the case where the required metadata is not available yet (probably due to the reason the file has not been processed yet). In this case, the default would be to show the file until the metadata is retrieved.

The solution comprehends changes in the two layers. Tracker (mostly Tracker Extract) for the metadata retrieved at indexing time. And also at a higher level for using the information and determine if the file is ultimately playable or not.

Note that the system is not 100% safe, since to guarantee that we would have to decode all the frames.

Additionally, applications will be able to mark specific files as non-playable for those cases playability cannot be determined until playback time.

Solution needs to be discussed in more detail with upstream Tracker maintainers.

**Status**. Satisfied.

## Appendix: Media Management Technologies

This chapter is focused on describing the **current status** of the various technologies, without really including the specific additions or modifications discussed on the requirements, which are covered in Solution. Therefore, some of the technologies do not fully obey the requirements yet in its current status, the modifications or additions needed to make them work as desired are described on Solution.

### Tracker

**Tracker**[3] is a semantic data storage for desktop and mobile devices. A semantic data storage is basically a central repository of user information, which stores relationships between pieces of data in a way that is re-usable among multiple applications.

[3] http://projects.gnome.org/tracker/

The concept is quite wide and applicable to different types of information like pictures, messages, etc. But this document is just focused on media content, the indexing of which is one of Tracker's primary functions.

This makes use of several existing technologies and standards:

- **Resource Description Framework (RDF[4])**. RDF is a directed, labeled graph data format for representing information, and is a W3C standard.
- **SPARQL**[5] is a W3C standard defining a query language for databases, able to retrieve and manipulate data stored in RDF format.
- **Ontologies**[6]. An ontology represents knowledge as a set of concepts within a domain, and the relationships between those concepts. It can be used to reason about the entities within that domain and may be used to describe the domain.
- **Nepomuk**[7] (Networked Environment for Personalized, Ontology-based Management of Unified Knowledge). Nepomuk is a research project, which defined a set of ontologies describing desktop entities like files, pictures, etc.

Tracker is a data store, an indexer and a search engine that allows the user to find and link data easily. Tracker is typically used for searching the local storage. By default Tracker comes with several indexing services called "miners" . Tracker is made up of several components:

- **Tracker Storage**. The data store and daemon to interface to Tracker's databases.
- **Tracker SPARQL**[8]. The libtracker-sparql library is the foundation for Tracker querying and inserting data into the data store based on the Nepomuk ontology.
- **Tracker Miner**[9]. The libtracker-miner library is the foundation for Tracker data miners. These miners will extract metadata and insert it in SPARQL form into the Tracker store, following the Nepomuk ontologies. Developers can add new miners in order to index new data sources.
- **Tracker Extract**[10]. The libtracker-extract library is the foundation for Tracker metadata extraction of embedded data in files. Tracker comes with extractors written for the most common file types (like MP3, JPEG,

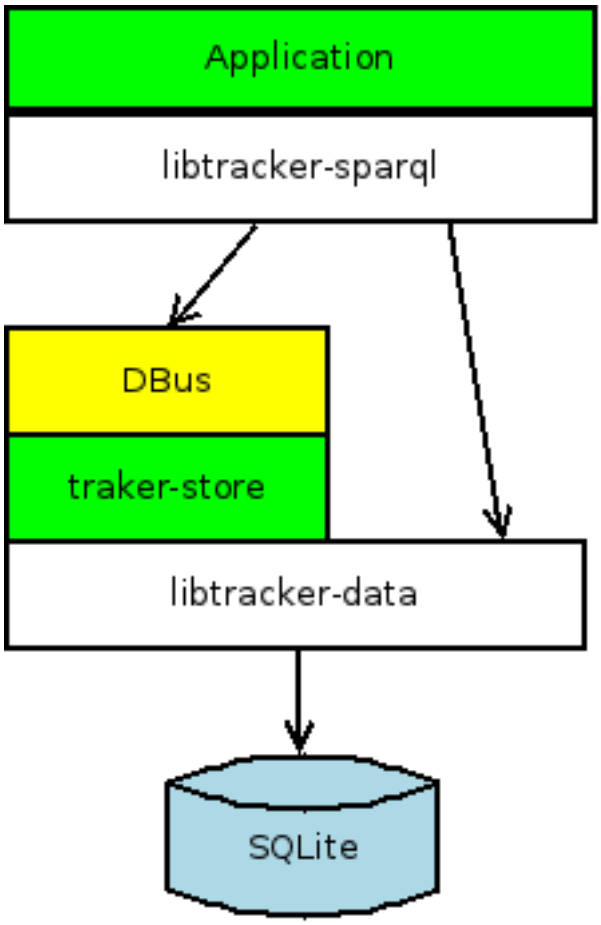
[4] http://www.w3.org/RDF/
[5] http://www.w3.org/TR/rdf-sparql-query/
[6] http://developer.gnome.org/ontology/0.12/
[7] http://www.semanticdesktop.org/ontologies/
[8] http://developer.gnome.org/libtracker-sparql/0.12/
[9] http://developer.gnome.org/libtracker-miner/0.12/
[10] http://developer.gnome.org/libtracker-extract/0.12/

PNG, etc.). However, for rarer formats, it is possible to write plugins to extract the metadata.

Ubuntu 12.04 currently has Tracker version 0.12.10, while the Apertis platform was using 0.10.6. During these versions many fixes have been done as well as some enhancements and improvements, but nothing really substantial. The performance of several components, specially the Tracker filesystem miner has improved in the 0.12 release. The limitations of Tracker are exposed in the context of the requirements in Solution.

The preferences for each Tracker component can be managed through GSettings, although there is also a UI application which is not interesting in the scope of this project (tracker-preferences).

**Tracker Storage** The Tracker storage is divided in several parts as shown in the following illustration.



- The public **libtracker-sparql** is the API layer used by the applications to access the Tracker storage using SPARQL. Internally, it uses the D-Bus interface when writing access to the database is required. However, it allows a more direct access to the database for read-only access (through libtracker-data), which reduces the D-Bus traffic.
- The **Tracker store daemon (tracker-store)** provides a D-Bus interface to access the RDF storage, and it also provides also a mechanism to notify when changes happen in the RDF storage.
- **libtracker-data** is the library interfacing directly with SQLite database, used by both tracker store and libtracker-sparql.

Below, there are listed the ontologies related with media content which are supported by Tracker:

- Nepomuk File Ontology (nfo).
- Nepomuk ID3 (nid3).
- Nepomuk MultiMedia (nmm).

See below more details about the storage needs required by Tracker:

- **SQLite**[11] **database**. The common configuration is to have separate Tracker storage for each user. However, this can be set up depending on the requirements of the platform, by changing environment variable XDG_CACHE_HOME, as the Tracker SQLite database is stored in $XDG_CACHE_HOME/tracker. Here are some rough numbers on SQLite database space usage:
  - Empty SQLite database. The database with initialized data, but without indexing files requires about 1.2 Mbytes.
  - Indexing Photos. As an approximate figure, our measurements show about 800 Kbytes of database size is used for every 500 photos (aprox. 3 Gbytes of media). Note, the size in Gbytes is just an approximate figure, since the amount of metadata scales with number of media items and not with their size.
  - Indexing Music. As an approximate figure, our measurements show about 800 Kbytes of database size is used for every 300 mp3 songs (3 Gbytes of media).
- **Write Ahead Log (WAL**[12]**) files**. The Tracker database is stored in SQLite using WAL. The WAL option allows better performance, concurrency and reliability; at a cost of consuming extra disk space. This file is part of SQLite. which is limited to 10,000 pages maximum, i.e. max of 10 Mbytes. Furthermore, this space used is temporal since it will get deleted

[11] http://www.sqlite.org/
[12] http://www.sqlite.org/draft/wal.html

as soon as the the database is checkpointed, which happens automatically or when the limit is reached. There is an additional relatively small file for shared memory, but that is transient and it does not even use disk space, just memory.

- **Ontologies**. The file ontologies.gvdb is stored in the same directory as the SQLite files. It is about 350 Kbytes, created on initialization. The size does not depend on the data indexed, but on the ontology models.
- **Tracker Journals**. It stores all inserts, updates and deletes. Basically it is a file that grows without bound, a reason why it has received some criticism. It is meant for data redundancy and backup. The journal is also used to cope with ontology changes. It can be disabled at compile time. In fact, it was disabled on the Nokia N9, mainly due to the ever-growing problem and privacy. Tracker journal can be a reasonable choice for a desktop system, but in case of embedded devices it is better disabled. It is stored in the $XDG_DATA_HOME/tracker/data directory.

| Tracker Use Case | Media in GiB | Index in MiB | Index in % |
|---|---|---|---|
| Empty database | 0 GiB | 11.5 | NA |
| 500 photos or 300 songs | 3 GiB | 12.3 | 0.4 % |
| 5K photos or 3K songs | 30 GiB | 19.5 | 0.06% |
| 5K photos and 3K songs | 60 GiB | 27.5 | 0.04% |
| 83K photos and 50K songs | 1000 GiB | 277 | 0.03% |

Tracker use cases for storage utilization

Note: at the time of this writing, Ubuntu 12.04 was currently using SQLite 3.7.9 (November 2011), while the latest stable version available is 3.7.10 (January 2012).

Here are some configuration parameters for the Tracker Storage:

- **Tracker DB Journal size**. Size of the journal at rotation. By default 50 Mbytes.
- **Tracker DB Journal rotate destination**. Where to store the journal chunk when it hits the max size.

**Tracker Miner** Tracker miners are responsible of finding content to index. Although in the context of this document we are normally just interested in files, it could be any resource able to be stored in Tracker. Tracker already comes with a filesystem miner. Additionally other miners can be implemented for specific data sources (either local or remote sources). Here are some configuration parameters for the filesystem miner:

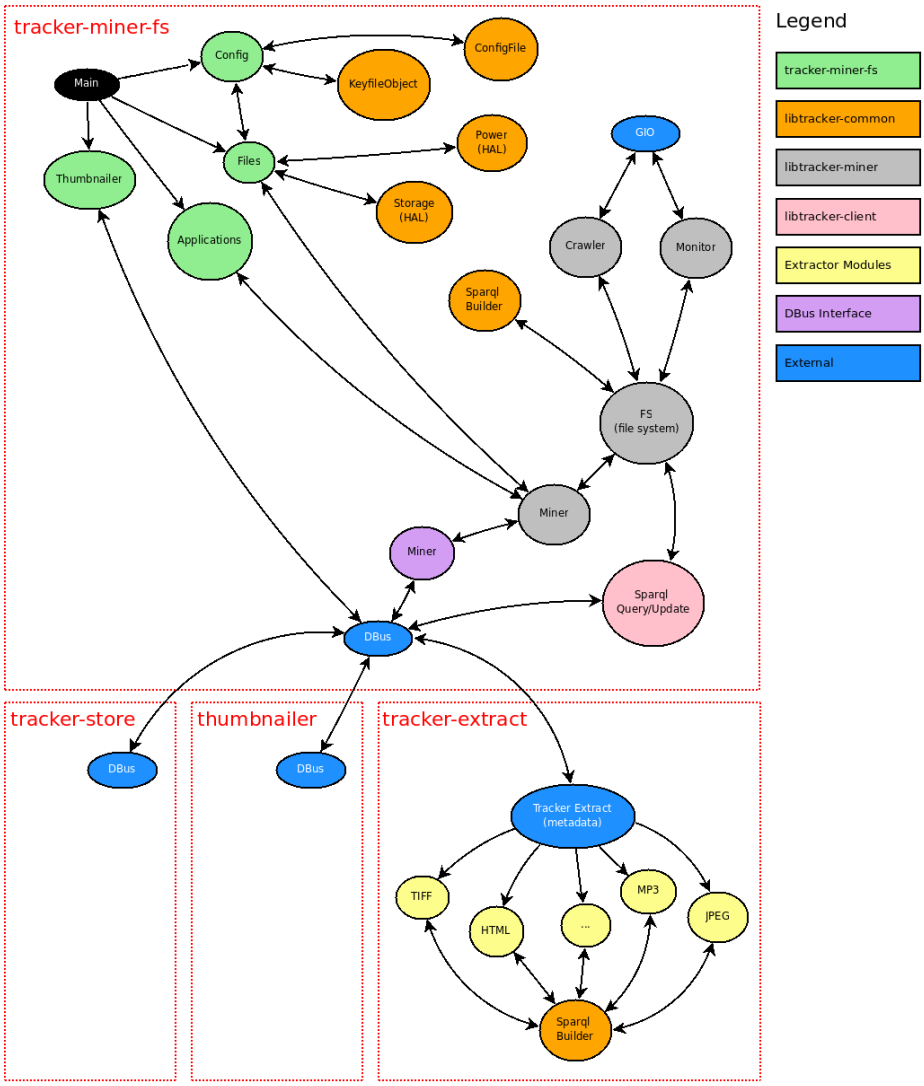
- **Startup wait time**. Primarily to avoid prevent Tracker from heavily loading the system just after boot. By default 15 seconds.

- **Scheduler priority**. Specifies the priority of indexing directories and files. There are three levels: when idle, first indexing on idle (default) and anytime.
- **Throttle**. Controls the throttle of file indexing operations. This specifies to control the overhead indexing has on the system. Of course, it is a trade-off between system load and speed, but it can be tuned to make UI applications more responsive. It is a value between 0 and 20, the higher the slower. A value of 0 denotes "as fast as possible"(default), any other number N denotes 20/N indexing operations per second. These limits can of course be adjusted internally.
- **Low disk space limit**. A configurable parameter to stop indexing in case of low free disk space. It is configurable between 0% (no limit) and 100%. It is 1% by default.
- **Crawling interval**. Specifies the interval in days to check whether the filesystem is up to date with the database. A value of -1 specifies the check should only be done on unclean shutdowns and -2 specifies this check should be disabled entirely.
- **Removable days threshold**. Specifies the threshold in days after which metadata for files from removable devices will be removed if their filesystem is not mounted. Zero means never. Configured to 3 days by default.
- **File monitoring**. Option to track filesystem changes directly in order to know what needs to be indexed.
- **File Writeback**. Option to write information back in the files, e.g. metadata retrieved from other sources or updated by the application, it can be stored back in the original file. It is limited to a few formats currently.
- **Index Removable Devices**. Option to enable / disable the indexing of removable devices.
- **Index Optical Discs**. Option to enable / disable the indexing of CDs, DVDs, and in general any optical media.
- **List of directories to index recursively**. It can also refer to special XDG directories like Desktop, Documents, Download, Music, Pictures, Public, Templates and Videos.
- **List of single directories to index** (non-recursively). Same notes as before.
- **List of ignored files**. Filenames can be specified with wildcards.
- **List of ignored directories**. Wildcards can be used to specify them.
- **List of ignored directories with content**. Avoid any directory containing a file whose name is blacklisted in this list.

The Tracker Miner Manager keeps track of available miners, their current progress/status, and also allows basic external control of them, such as pausing or resuming data processing. It controls the scheduling of the different operations through the configuration parameters already specified before. The miner only does the crawling operation for files and sequencing the metadata extraction scheduling. The actual metadata extraction is accomplished by Tracker Extract, described in the next section.

The most widely used miner is the filesystem miner, responsible for indexing local files. Other miners exist like UPnP miner, which indexes UPnP servers. The way to create new filesystem miners will not be shown in this document, since there is no requirement for it in this project.

See a general overview in the following illustration.




**Tracker Extract** Tracker extract does the actual metadata extraction. It inspects the media content and it extracts metadata information, which is stored in Tracker Store. There is a list of the currently Tracker supported file formats[13]. It includes the main formats for all the media content types of interest (music, music playlist, video, picture, picture album and documents).

**Note**: in some Tracker extract plugins like the GStreamer one, the actual formats able to be extracted depend on the specific GStreamer plugins installed on the system.

[13] https://wiki.gnome.org/Projects/Tracker/SupportedFormats

The extract plugins are built as dynamic libraries which are load at run-time. There is a text file to configure what mime types an extract plugin understands and which library file is. There are two types of extract plugins, specific and generic. Specific extractors are preferred if they exist, otherwise generic ones are used (e.g. like audio/*).

In case more formats need to be supported, they can be easily added to Tracker by implementing extra plug-ins. They are relatively simple to implement; the function **tracker_extract_get_metadata()** simply has to be provided. For more details, check the example in the Tracker Extract [documentation][Tracker Extract-doc].

Tracker Extract is a D-Bus daemon with a very simple interface, to get metadata and to cancel existing tasks. Tracker Extract daemon can be configured to automatically shutdown when idle after a certain period of time, allowing to free resources. Also, it detects extract operations that take too much time and aborts them.

These are the configuration options for Tracker Extract:

- **Scheduler priority**. Specify the priority of extracting metadata. There are three levels: when idle, first indexing on idle (default) and anytime.
- **Max bytes**. Maximum number of bytes to extract for text files. This is used just for text extraction (when full text search is enabled), since it can make grow the index database significantly. The default is 1 MByte, and the maximum 10 MBytes.

**Tracker Scheduling** Tracker employs several background processes: Tracker Store, Tracker Miner and Tracker Extract. Tracker Miner and Extract do the heavier work in a autonomous way and they can potentially consume a lot of resources. **Tracker Miner Manager** controls and monitors Tracker Miners, scheduling all their operations, including crawling the filesystem and invoking metadata extract operations.

Tracker Miner and Extract can have their CPU scheduling priority configured (as described before). Tracker Store daemon does not need its CPU priority configured since it works on demand; it must always be running and process any request by user apps or other processes. Additionally, all Tracker daemons have IO priority set to minimum, to interfere the least possible with other applications.

The Tracker Filesystem Miner sets up a filesystem notifier with the directories to index. The filesystem notifier is responsible for finding the directories and files to index, and to monitor and notify of any changes. Tracker Filesystem Miner has several priority queues; one per type of operation. Tracker Miner processes items from these queues when it becomes idle. The priority of the types of operations from highest to lowest is: writeback operations, deleted items, created items, updated items, moved items.

After the operation is removed from the queue, it gets added to the task pool while it is running. The length of the task pools is checked before adding new operations to it to avoid overloading the system. The items in the task pools are processed in several steps. Initially, the information is captured without inspecting the content files, properties like mime type, size, modification and creation time, etc. In a second step, a request is done to Tracker Extract to extract more information from the file.

Thumbnails are not requested by the Tracker Miner Manager. But if a file with an existing thumbnail gets moved or deleted, the thumbnail will be updated too (so the thumbnail filename will get renamed or deleted too).

### Thumbnail Management

The **Thumbnail Managing Standard**[14] deals with the permanent storage of previews for file content. The **Thumbnail Management D-Bus specification**[15] is a standardized D-Bus API to deal with thumbnailing. This D-Bus specification is currently implemented by Tumbler, which has been already used successfully in consumer products like the Nokia N9 phone. With a D-Bus specification for thumbnail management, applications don't have to implement thumbnail management themselves. If a thumbnailer is available they can delegate thumbnail work to a specialized service. The service then calls back when it has finished generating the thumbnail.

Thumbnailing is an expensive operation. Therefore, it is meant to be requested by applications on-demand, i.e. If the application needs a thumbnail for a file it should request explicitly for it to the Thumbnailer service.

Some features provided by the Thumbnailing service that can be interesting in our context:

- Provide the ability to handle different thumbnail flavors (sizes). By default two flavors exist:
    1. Normal configured by default as 128x128.
    2. Large configured by default as 256x256.
- Possibility to implement thumbnailers for closed formats or with customized features.
- Complexity of a LIFO queue and setting I/O and scheduling priorities for background thumbnailing is no longer the responsibility of the application developer.
- Extensibility with plug-ins. This is useful to support for additional file types or when different interpolation algorithms are required.

There are several components in the Thumbnailer service:

[14] http://specifications.freedesktop.org/thumbnail-spec/thumbnail-spec-latest.html
[15] https://wiki.gnome.org/DraftSpecs/ThumbnailerSpec

- **Thumbnailer**. Calculates the thumbnail for a specific file format.
- **Thumbnailer Manager**. A register of available Thumbnailers is available at runtime.
- **Thumbnail Cache**. This avoids regeneration of thumbnails when files are copied or moved and cleans up the cache sporadically and when a file is deleted. This is managed automatically by Tracker Filesystem Miner.

The thumbnails are stored in *$XDG_CACHE_HOME/thumbnails/[SIZE]/(md5sum of original URI).png*. Thumbnails for files on removable devices may instead be stored in a *shared thumbnail repository* on the removable device, as *.sh_thumbnails/[SIZE]/(md5sum of original filename not including path).png*, relative to the file. See §10 of the Thumbnail Managing Standard.

One of the advantages of Tumbler is that the scheduler is abstracted, there are two options implemented: a background scheduler using a first-in-first-out (FIFO) queue and a foreground one using a last-in-first-out (LIFO) queue. Tumbler has been used successfully in several environments including XFCE, Maemo and MeeGo. GNOME uses GnomeThumbnail API to generate thumbnails. EFL is using ethumb. Although there are not many differences between the different Thumbnailing services, Tumbler is one of the most advanced since it is a real service and not a library, and it provides scheduling features. Additionally, Tumbler comes packaged for popular distributions like Ubuntu and Fedora, and it has the extra advantage of being already integrated with Tracker, as we saw in previous section.

Tumbler can be extended to support new thumbnails types as needed with plugins. There are already existing plugins for GStreamer, JPEG, font, a large collection of image formats (GDK pixbuf), PDFs (libpoppler), etc.

> See here[16], to discover the mime types they currently support you need to navigate to their `provider_get_thumbnailers` implementation, for example `gst_thumbnailer_provider_get_thumbnailers` in http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thumbnailer-provider.c

Keep in mind that if a given format is not supported by Tumbler, support can be added through its plugin API.

Video thumbnails can be generated using the GStreamer thumbnailing plugin. This plugin already provides an heuristic method to extract the thumbnail from a video stream, by selecting a frame with a wide distribution of colors (to avoid presenting a title screen or other essentially-blank frame).

It is interesting to keep a look on the disk space utilization for thumbnails. After doing some measures, we found out that thumbnails occupy 13 kilobytes for 128x128 pixel size, and about 29 kilobytes for 256x256 size.

[16] http://git.xfce.org/xfce/tumbler/plain/plugins/

| Thumbnail Use Case | Media in Gb | Thumbnail size in Mb normal + large = total | Usage in % |
|---|---|---|---|
| 500 photos | 3 Gb | 6.3 + 13.7 = 20 | 0.65 % |
| 5K photos | 30 Gb | 63.5 + 141.6 = 205.1 | 0.67 % |
| 166K photos | 1000 Gb | 2107.4 + 4701.2 = 6808.4 | 0.66 % |

Thumbnail storage utilization

**Media Art Storage** **Media Art Storage**[17] provides a mechanism for applications to store and retrieve artwork associated with media content, like music from an album, the logo for a radio station, or a graphic representing a podcast. The storage medium for artwork is the filesystem inside a user's home directory or in $XDG_CACHE_HOME/media-art/. Tracker manages and requests media art for the albums and artists.

In some situations it is desirable to have a *local media art repository* (for example, for read-only media or for USB removable devices). The location for local media art will be a subdirectory named .mediaartlocal/ within the same directory as the album's files.

Tracker already checks for media art present in the indexed folders. Additionally it is able to request the downloading of album art to the album art provider installed in the system. There is already a FOSS album art provider example using Google Images, but it can be replaced by other implementations extracting album art from other sources just by implementing a D-Bus service with the interface com.nokia.albumart.Requester.

Thumbnails of media art follow the Thumbnail Specification. The URI used to determine the thumbnail path is the full URI pointing to the original media art. For the path to the thumbnail refer to the Thumbnail Specification itself. A media art fetcher is allowed to store the normal and large thumbnails immediately after download of the media art is completed. A media art fetcher is, however, not required to do this by itself (the thumbnail infrastructure will or should take care of this if the media art is not thumbnailed yet).

### Grilo

**Grilo**[18] is a simple API for browsing and searching media content from various sources using a single API. Applications will be able to browse and discover media content by using the Grilo API. This API will provide media content and its metadata, and GStreamer framework will be able to play video or audio content (either local or remote).

A single, high-level API that abstracts the differences among various media content providers, allowing application developers to integrate content from

[17] https://wiki.gnome.org/DraftSpecs/MediaArtStorageSpec
[18] https://wiki.gnome.org/Projects/Grilo

various services and sources easily. Grilo comes with a collection of plugins for accessing various media providers, like Vimeo, Flickr, YouTube etc. so they can be presented uniformly via the Grilo API. Additionally a grilo-tracker plugin exists, which uses the Tracker service (described in past sections), to make media indexed by Tracker available through the Grilo API.

There is an additional Grilo plugin for accessing the filesystem directly (grl-filesystem), which checks for media content in a set of configured directories. The defaults are the XDG user directories for pictures, music and videos.

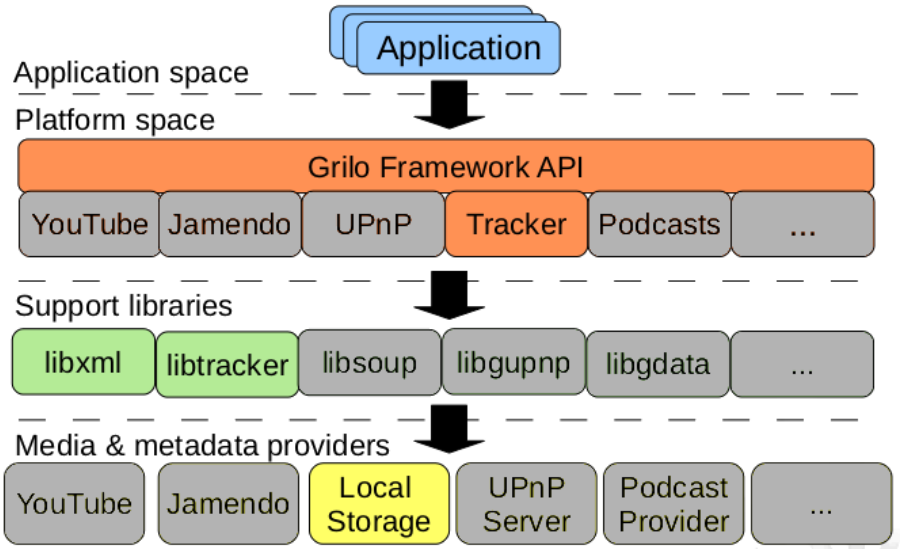
Although Grilo can be used to access many media content sources, we suggest only using it for accessing local media content. The next sections will dig into Grilo's details and its advantages. The main advantages of using Grilo instead of Tracker directly for this specific use case:

- Tracker is a semantic data storage, which can be used to store other bits of information apart of indexing information from media content like messages, calendars, etc. In other words, it is a very general framework usable for many purposes. Therefore, it makes sense to provide a higher level specialized API for media browsing (Grilo) on top of Tracker to hide its complexity from media applications.

- Grilo has some plugins that might be useful to extract additional metadata, e.g. album art from last.fm. Grilo is specially recommended for accessing to metadata from the Internet, which is not meant to be indexed. In addition, the platform could take advantage of future plug-ins which are planned to be developed by the FOSS community like lyrics, moviedb.org, etc.

- Grilo would support using an indexer other than Tracker if a better one becomes available. More importantly, applications wouldn't have to be modified to take advantage of such a change.

See the following illustration for an overview of the Grilo Architecture. Note the boxes with grey background are not going to be used in the context of the Apertis project.




**Grilo Media Source Plugins** The plugin must create at least one GrlMediaSource instance, and register it in the Grilo registry. A GrlMediaSource represents a particular source of media. These plugins provide several functions:

- **Search** content by keywords.
- **Browse** the media content in a hierarchical way. It is similar to exploring a filesystem, entering into folders (GrlMediaBox) and browsing files in it.
- **Query** allows access to content using service specific language. Normally it provides additional filtering capabilities. This is used by applications to support plugin-specific functionality.
- **Metadata** used to request additional metadata.
- **Store** (optional), supports to push content to the source.
- **Remove** (optional), to remove stored contents from the source.
- **Supported keys** provides information on which metadata keys are provided by the plugin. Typical metadata keys are: id, title, url, thumbnail, mime, artist, duration.
- **Slow keys** (optional) provides info on which metadata keys are expensive to gather. So the applications could just ask for non-expensive ones normally, and only require the slow keys when details are required for a particular media content.
- **Media from URI**. Gets GrlMedia from a URI. For example a file browser may use this to get metadata for a specific file.

- **Test Media from URI** (optional). To check if the plugin can convert a URI into a GrlMedia object.
- **Notifications** on changes on media content.

At least one of the content retrieval methods is expected to be implemented: search, browse or query. Each media content result of the search/browse/query is represented by a GrlMedia object.

Plugins should be implemented in a non-blocking way to have a smooth user experience in applications. Also threads are not recommended; splitting work into chunks using the idle loop is encouraged.

There is a standard set of metadata keys defined, but plugins can define their own custom metadata keys.

A GrlMedia can have multi-valued properties; for example a YouTube video with different resolutions (and thus, different URIs). It is also possible to associate different properties with each URI of a GrlMedia.

**Grilo Metadata plugins** Grilo metadata source plugins do not provide access to media content, but additional metadata information. An example would be to provide thumbnail information for local audio content from an online service.

This plugin must create at least one GrlMetadataSource instance, and register it in the Grilo registry. The plugin provides several functions:

- **Resolve** retrieves additional information for a GrlMedia object.
- **May resolve:** to check if Resolve may be performed with existing information.
- **Set metadata** (optional): set the play count or the last time a media was played.
- **Writable keys** (optional): reports which keys can be stored.
- **Supported keys:** provides information on which metadata keys are provided by the plugin.
- **Slow keys** (optional): provides info on which metadata keys are expensive to gather. So the applications can ask for inexpensive keys normally, and only request the slow keys when details are required for a particular media content.
- **Cancel operations:** cancels ongoing operations.

### Google Data Protocol

**YouTube,** as well as other Google services like Picasa, use the **Google Data Protocol**[19]. The Google Data Protocol is a REST-inspired technology for reading, writing, and modifying information on the web. The protocol currently supports two primary modes of access: AtomPub and JSON. The JSON is a mapping of Atom items to JSON objects meant to be used for web applications written in JavaScript.

The AtomPub mode is based on the Atom Publishing protocol, with namespaced **XML** additions. Communication between the client and server is broadly achieved through **HTTP** requests with query parameters, and Atom feeds being returned with result entries. Each *service* has its own namespaced additions to the GData protocol; for example, the Google Calendar's API has specializations for addresses and time periods.

Collabora proposes **libgdata**[20], which is a library to allow access to web services using the Google Data Protocol from traditional applications. Results are always returned in the form of result *feeds*, containing multiple *entries*. How the entries are interpreted depends on what was queried from the service, but when using libgdata, this is all taken care of transparently. The main dependencies of libgdata are libsoup, libxml and liboauth.

Other frameworks and applications are already using libgdata with success, e.g. evolution-data-server, Totem's YouTube plugin and Grilo's YouTube plugin.

The library libgdata already provides an implementation for the **GDataYouTubeService**[21], which provides the following functionality:

- Query videos.
- Query videos related to a specific video.
- Query standard feed types: top rated, top favorites, most viewed, most popular, most recent, most discussed, most linked, most responded, recently featured and watch on mobile.
- Upload a video.
- Get categories.

### Librest and libsoup

It is difficult to find libraries to access online media sources if they are not provided by the vendors themselves. However, most of these online media sources are based on HTTP protocol with REST[22] interfaces. Therefore, in general, [**li-**

[19] http://code.google.com/apis/gdata/
[20] http://developer.gnome.org/gdata/0.10/gdata-overview.html
[21] http://developer.gnome.org/gdata/0.10/GDataYouTubeService.html
[22] http://en.wikipedia.org/wiki/Representational_state_transfer

**brest**] and/or **libsoup**[23] will be useful. **Librest** is a library designed to make it easier to access web services that are designed in a "RESTful"manner. **Libsoup** is an HTTP client/server library for GNOME. It uses GObjects and the glib main loop, to integrate well with GNOME applications. Collabora can suggest or provide advise for open-source ways for accessing these services on request. This is the most effective way to access all the features.

### Playlists support

Playlists are supported in Tracker. There is an specific Tracker Extract plugins to handle playlists, which is using internally the **Totem Playlist Parser**[24] library, which is conveniently abstracted and independent of Totem. Tracker Extract introduces the metadata retrieved in Tracker Store using the class nmm:Playlist, which is a subclass of nfo:MediaList. The entries in the playlist are introduced as nfo:MediaFileListEntry.

The supported playlist formats in Totem Playlist Parser are: audio/x-mpegurl, totem-plparser, audio/x-scpls, audio/x-pn-realaudio, application/ram, application/vnd.ms-wpl, application/smil and audio/x-ms-asx.

Grilo does not support playlists in the latest stable version available, so this feature would need to be added as specified in the requirements section.

## Appendix: Questions & Answers

These chapter contains very specific questions that have been asked during workshops.

**Q: Will asking for a specific prioritization during metadata extraction increase the load by running multiple indexing jobs ?** A: No, the Tracker scheduler will manage all metadata indexing operations in internal queues, so prioritization will just change the sorting of the metadata indexing operations, but not the overall system load. Note the scheduling system proposed in this document is not implemented in Tracker yet. See Indexing scheduling and Tracker scheduling for more details on prioritization and Tracker scheduling.

**Q: How does the system know when to renew thumbnails ?** A: When a thumbnail is generated, some properties are stored inside it like the original URI and the modification time of the original file. If the original file is modified at some point, its modification time will get changed automatically by the Linux filesystem. So, it is possible to know when a thumbnail is outdated. Additionally, Tracker is monitoring the filesystem for changes. In case a file is modified, added, moved or deleted its thumbnail will be automatically updated. Note: this feature is not fully implemented yet, but it is part of the modifications Collabora will implement.

[23] http://developer.gnome.org/libsoup/
[24] http://developer.gnome.org/totem-pl-parser/stable/

**Q: How the mime type of the files is determined ?** A: This is done through glib, which finds out the mime type in a efficient way and it is used extensively by all GNOME based software. The details of the algorithm used can be seen in the Shared MIME Info Specification[25], it has been designed to be robust and efficient. The first thing done is to test the filename extension to see if it is a recognized type. If this operation cannot be done or the result is uncertain, a second check will be done using the first bytes of the file checking for the signature of known files. For more details see g_file_query_info, G_FILE_ATTRIBUTE_STANDARD_CONTENT_TYPE and g_file_info_get_content_type in GNOME documentation.

**Q: How the video thumbnailing works to avoid black video frames or uninteresting frames in general ?** A: From Thumbnail management: "Video thumbnails can be generated using the GStreamer thumbnailing plugin. This plugin already provides an heuristic method to extract the thumbnail from a video stream, by selecting a frame with a wide distribution of colors (to avoid presenting a title screen or other essentially-blank frame). Other ways could be implemented if required, just by implementing a thumbnail plugin.

**Q: How document thumbnailing works to avoid thumbnails of blank pages ?** A: The existing Tumbler plugins used to extract thumbnails from Open/LibreOffice, PDF and Microsoft Office documents gets the thumbnail stored inside the file. It is responsibility of the office applications to write a proper thumbnail. Typically it is just the thumbnail of the first page of the document, which usually is the best option since the first page contains the title in bigger font sizes, cover of the document and logos. Any other approach is debatable, so Collabora does not recommend to make thumbnails from only text pages since they are less likely to be useful, thumbnailing normal text would become unreadable.

**Q: How the applications can store and retrieve the last time a media file was played ?** A: This functionality can be provided by the Grilo metadata store plugin. The application must query the last values and set new values through Grilo API. The media file is identified via the file URI. The metadata store plugin stores these values in a Tracker database. It currently supports the following values: last position where media item was played (GRL_METADATA_KEY_LAST_POSITION), number of times a media item has been played (GRL_METADATA_KEY_PLAY_COUNT) and last date a media item was played (GRL_METADATA_KEY_LAST_PLAYED). Grilo is making use of the properties already defined on the Tracker ontologies like nfo:lastPlayedPosition, nie:usageCounter and nie:contentAccessed. A benefit of using Grilo is that Tracker details are not exposed to the applications, for example alternatively Grilo has another plugin to store these fields in a

[25] http://standards.freedesktop.org/shared-mime-info-spec/shared-mime-info-spec-latest.html

separate SQLite database in case Tracker was not used, but the API to set and get these properties would be the same.

**Q: How a thumbnail is retrieved ?** A: Thumbnails can be retrieved through different ways depending on what specific APIs the application is using. The best way for media applications would be through the Grilo API, see grl_media_get_thumbnail and grl_media_get_thumbnail_binary_nth (in case several thumbnails are available for a media item). Grilo API is internally using glib library to retrieve this through g_file_query_info, G_FILE_ATTRIBUTE_THUMBNAIL_PATH and g_file_info_get_attribute_byte_string. Grilo API will need to be modified in case more thumbnails need to be stored on the USB flash devices.

**Q: How the system behaves on robustness on power loss ?** A: This and other questions on system robustness will be answered on a separate document focused on system robustness. Anyway, please see chapter Indexing database on removable device for an advance of some issues regarding USB flash devices.

**Q: How a media file from a USB Flash device is identified ?** A: It is identified by its complete URI, e.g. /media/D8C0-024E/Joaquin Sabina/Joaquin Sabina & Fito Paez - Llueve sobre mojado.mp3". In some systems, USB flash devices are mounted on a directory with a hex identifier (depending on system configuration). This identifier is the UUID (Universally Unique Identifiers), not the label of the USB flash device. It is generated when the filesystem is created, and it is very. Generally it is a 128 bit identifier, but some filesystems like VFAT have smaller resolution (32 bits).

**Q: Is it configurable the timeout for Tracker extract operations ?** A: No, they are not currently, but it would be simple to make them configurable for example through GSettings. There are two timeouts. A watchdog timeout which is checks that the tracker extract process does not hang during metadata extraction (by default set to 20 seconds). There is an additional idle timeout, which stops a tracker extract process if it has been idle for some time (30 seconds by default).

**Q: Does Tracker retry in case Tracker Extract fails due to the watchdog timer ?** A: By default, Tracker retries up to two times if a tracker extract process fails. It will also retry in case the file is modified or the USB flash where it is located is reinserted.

**Q: Does Tracker store marks for the corrupted files ?** A: Currently, there is no property to identify corrupted files in Tracker. A file whose extract process has failed due to corruption in the file, it would just have properties from the nfo ontology (nepomuk file object), but it would not have properties from other subclasses like nmm (nepomuk multimedia).

**Q: There are reports of performance of page queries on Tracker databases is negatively affected by the number of rows in the database. Collabora to double check.** A: Some tests running SPARQL queries have been done with databases near 6000 items and the mentioned problem was not reproducible (no performance problems found). Please provide data set and application code reproducing this problem for further investigation.

**Q: Should Tracker be used for Radio Stations information ?** A: Tracker has already ontologies to store radio station information. So, it would be possible to use it to store and retrieve the user favorite radio stations. However, the interface to access and update this information would be through plain SPARQL, which has a step learning curve for developers. Additionally, the radio station information is not shared with other applications. The only advantage of using Tracker would be that the global search would automatically work for radio station information, so it would not be necessary to implement an extra global search plugin to look for this info in another database. The final decision must take into consideration how well the existing ontology for radio stations (nmm:RadioStation[26]) is suited to Apertis'roadmap.

**Q: What happens when a USB flash device is inserted in a USB port ?** A: When the user inserts an USB flash device, there are three main components participating in the action:

- **Linux kernel** (including device drivers). The kernel will be able to communicate with the device as soon as it is powered up, initialized and announced through the USB Bus.
- **Udev**[27] is the device manager for the Linux kernel. Primarily, it manages device nodes in /dev. It is the successor of devfs and hotplug, which means that it handles the /dev directory and all user space actions when adding/removing devices, including firmware load. The Udev daemon listens to the netlink socket used by the kernel to communicate with user space applications. The kernel will send a bunch of data through the netlink socket when a device is added to, or removed from a system. The Udev daemon catches all this data and will do the rest, i.e., device node creation, module loading etc.
- **UDisks**[28] (formerly known as DeviceKit-disks) lies on top of udev, and it is an abstraction for enumerating disk and storage devices and performing operations on them. It is is a replacement for part of the functionality which used be provided by the now deprecated HAL (Hardware Abstrac-

[26] http://developer.gnome.org/ontology/0.14/nmm-ontology.html
[27] http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf
[28] http://www.freedesktop.org/wiki/Software/udisks

tion Layer). UDisks is a user daemon with D-Bus interface which gets notifications from udev.

See the following table for an idea of what happens when a USB flash device is inserted. The table provides a general idea about the timings for different operations in the system. Note, although the timings are based on real measures, are not guaranteed since the all the software components have not been completely built yet and timings depend on the actual hardware used.

| Timeline (s) | Delay (s) | Event |
|---|---|---|
| 0 | - | (1) User inserts a USB flash device in the system, one which has never been indexe |
| 2.8 | 2.8 | (2) UDisks daemon reports a USB flash device has been inserted via D-Bus. The u |
| 3.6 | 0.8 | (3) UDisks daemon notifies the partition in the USB Flash has been mounted auto |
| 4.9 | 1.3 | (4a) Media files in the root directory of the USB flash device are shown to the user |
| 5.4 | 0.5 | (4b) Tracker has finished crawling the filesystem to find out all entries in the filesys |
| 6 | 0.6 | (5a) Tracker Extract has metadata for the files that have been returned in the first |
| 46 | 40 | (5b) Tracker Extract finishes gathering metadata for all files in the USB flash devi |

**Q: How does the monitoring of filesystem changes work in Tracker ?**
A: The monitoring of changes in files and directories of the filesystem is handled internally by Tracker Miner via the GFileMonitor[29] API. Note GFileMonitor is just an abstraction in glib, which abstracts the file monitoring functionality, since there are several backends available implementing such functionality depending on the specific operating system. Note, this mechanism is a very efficient way to get notified about changes on the filesystem, since it is directly provided by the kernel, instead of doing active polling. Linux uses the **inotify** backend. For a more detailed view of the inotify API see the tutorial "*Monitor filesystem activity with inotify*[30]".

[29] http://developer.gnome.org/gio/unstable/GFileMonitor.html
[30] http://www.ibm.com/developerworks/linux/library/l-ubuntu-inotify/index.html