



Media management

# Contents

2	Solution . . . . .	3
3	Technology and Solution Overview . . . . .	3
4	<b>Local Storage Media Source</b> . . . . .	5
5	<b>Media Browsing Requirements</b> . . . . .	6
6	<b>Media Indexing Database Requirements</b> . . . . .	8
7	Indexing Scheduling . . . . .	12
8	Thumbnailing . . . . .	18
9	DLNA (UPnP) . . . . .	19
10	Online Media Sources . . . . .	20
11	Bluetooth AVRCP . . . . .	20
12	<b>Playability check</b> . . . . .	20
13	Appendix: Media Management Technologies . . . . .	21
14	Tracker . . . . .	21
15	<b>Thumbnail Management</b> . . . . .	30
16	Grilo . . . . .	32
17	Google Data Protocol . . . . .	36
18	Librest and libsoup . . . . .	36
19	Playlists support . . . . .	37
20	Appendix: Questions & Answers . . . . .	37

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

42 used for media management without including specific requirements, additions  
43 and modifications described on [Solution](#).

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

- 50 • Local Storage.
- 51 • Removable Storage Devices.
- 52 • CD and DVD.
- 53 • DLNA (UPnP).
- 54 • Media Online Services: YouTube, Shoutcast, Dropbox, last.fm, podcasts,  
55 etc.
- 56 • Bluetooth AVRCP.

## 57 **Solution**

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

### 61 **Technology and Solution Overview**

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

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

- 68 • **Tracker** is a central repository for user information. It is made of several  
69 components: Tracker Miner, Tracker Extract and Tracker Store. Tracker  
70 Miner automatically crawls for media content files. Tracker Extract gath-  
71 ers useful metadata from these files and it stores this metadata in the  
72 Tracker Store database. Metadata can be retrieved from the Tracker Store  
73 with SPARQL queries. See [Tracker](#) for more details. Although this doc-  
74 ument will only focus on the Tracker features specific to media indexing,  
75 Tracker can be used to store other information as well, like applications,  
76 messages, calendar events, etc. or in general any information that is worth  
77 to share between applications.

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

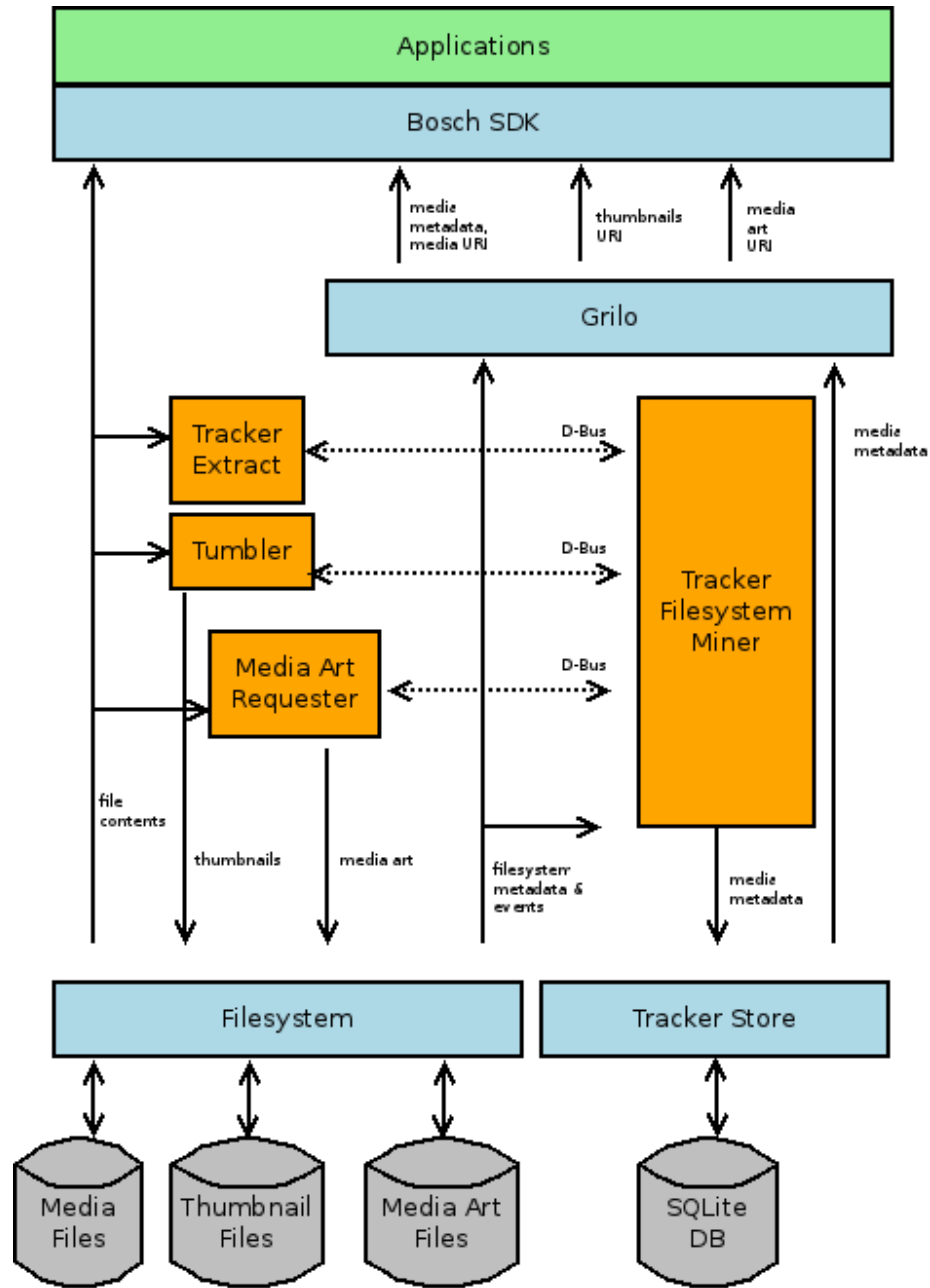
89 The proposed solution combines Grilo, Tumbler and Tracker for locating media  
90 content and retrieving its metadata from the local system and removable storage.  
91 Tracker does the heavy work: filesystem crawling, metadata extraction and  
92 metadata storage. Grilo is a simple API which lies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

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

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

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

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



115

116 **Local Storage Media Source**

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

118 **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

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

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

#### 166 **Required work.**

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

172 **Status.** Satisfied.

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

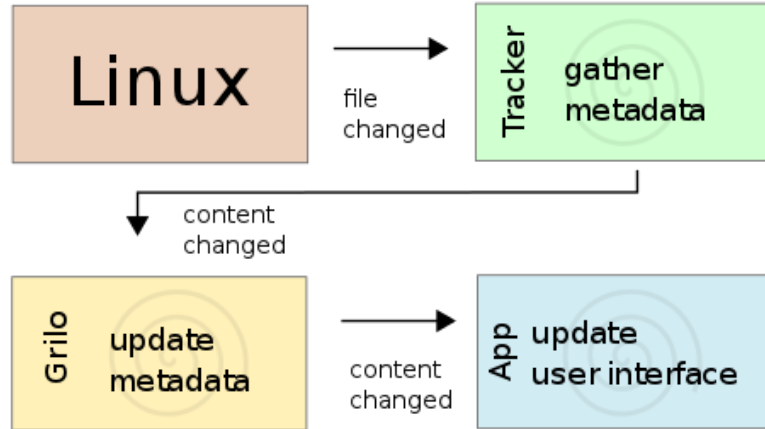
176 **Analysis.** The indexing process is asynchronous, it can happen that media  
177 content gets its metadata updated while the content is already being shown to  
178 the user.

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

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

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

194 **Status.** Satisfied.



195

196 **Paged queries Requirement R4.** Provide queries to request content infor-  
 197 mation by pages of fixed size.

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

201 **Solution:** Grilo supports paging in all requests via skip and count numbers. In-  
 202 ternally Grilo uses both mechanisms provided by Tracker SPARQL (OFFSET /  
 203 LIMIT modifiers in the SELECT SPARQL statements and TrackerSparqlCur-  
 204 sor). See **Grilo** for details on Grilo.

205 **Status.** Satisfied.

## 206 Media Indexing Database Requirements

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

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

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



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

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

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

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

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

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

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

258 **Status.** Satisfied.

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

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

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

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

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

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

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

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

318 See more details on Tracker Journal in [Tracker storage](#).

319 **Status.** Satisfied.

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

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

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

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

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

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

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

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

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

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

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

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

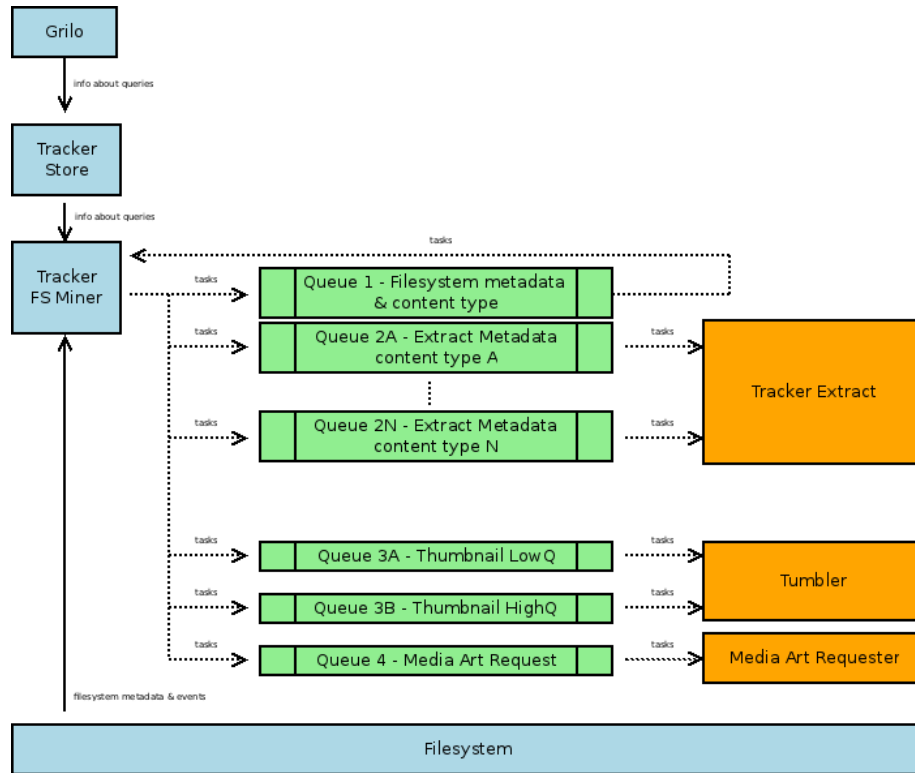
375 **Status.** Satisfied.

## 376 Indexing Scheduling

377 There are many specific requirements related with metadata extraction priori-  
378 tization. They will be analyzed in detail in the following subsections.

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

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



402

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

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

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

419 **Solution.** Collabora will add an option in Tracker's configuration to enable two  
 420 pass indexing. If enabled, tracker will first crawl the whole filesystem to store  
 421 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.

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

462 **Status.** Satisfied.

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

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

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

475 The following would be implemented as part of the solution:

- 476 • **Extract normal.** The current behavior, that is without automatic prior-  
477 itization of extraction based on queries.
- 478 • **Extract recent.** This will automatically request the metadata extraction  
479 for media content items returned in recent queries.
- 480 • **Extract next.** This will automatically request the metadata extraction  
481 for media content items that would result in next page of recent queries.  
482 This setting will imply “Extract recent” as a dependency.
- 483 • **Extract thumbnail.** This will automatically request the thumbnail com-  
484 putation for media content items returned in recent queries (or next page  
485 items if “Extract next” is also set).

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

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

491 **Status.** Satisfied.

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

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

495 **Status.** Satisfied.



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

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

503 The solution needs to be discussed in more detail with upstream Tracker main-  
504 tainers.

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

512 Thumbnail calculation is particularly expensive in CPU and storage resources.  
513 See the table in [Thumbnail management](#) for more detailed information.

514 **Status.** Satisfied.

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

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

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

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

535 **Status.** Satisfied.

536 **Thumbnailing**

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

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

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

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

550 **Status.** Satisfied.

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

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

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

557 **Status.** Satisfied.

558 **Thumbnailing algorithm configuration Requirement R16.** The algo-  
559 rithm used for calculating the thumbnails must be configurable.

560 **Analysis.** Currently Tumbler implements several plugins for thumbnail calcu-  
561 lation.

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

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

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

574       • **Bilinear**: best quality/speed balance; use this mode by default. For en-  
575       largement, it is equivalent to point-sampling the ideal bilinear-interpolated  
576       image. For reduction, it is equivalent to laying down small tiles and inte-  
577       grating over the coverage area.

578       • **Hyper**: this is the slowest and highest quality reconstruction function. It  
579       is derived from the hyperbolic filters in Wolberg’s “Digital Image Warping”  
580       .

581       **Status**. Satisfied.

## 582       DLNA (UPnP)

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

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

592       For more specific details on these topics see the UPnP AV (Universal Plug And  
593       Play Audio Video) architecture [documentation](#)<sup>1</sup>.

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

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

601       **Solution**. Collabora proposes the GUPnP framework to fulfill the requirements.  
602       The GUPnP library implements the UPnP specification: resource announce-  
603       ment and discovery, description, control, event notification, and presentation.  
604       On top of that, GUPnP\*-\*AV library is a collection of helpers for building AV  
605       (audio/video) applications using GUPnP. The GUPnP framework is licensed  
606       under LPGL v2.1 and it is written in C using GObject and libsoup. GUPnP is  
607       entirely single-threaded (though asynchronous) and integrates with the *GLib*<sup>2</sup>  
608       main loop.

---

<sup>1</sup><http://www.upnp.org/specs/av/UPnP-av-AVArchitecture-v1.pdf>

<sup>2</sup><http://gtk.org/>

609 **Status.** Satisfied.

## 610 **Online Media Sources**

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

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

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

620 **Status.** Satisfied.

## 621 **Bluetooth AVRCP**

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

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

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

628 **Status.** Moved to Connectivity design.

## 629 **Playability check**

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

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

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

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

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

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

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

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

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

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

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

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

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

668 **Status.** Satisfied.

## 669 **Appendix: Media Management Technologies**

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

### 676 **Tracker**

677 **Tracker**<sup>3</sup> is a semantic data storage for desktop and mobile devices. A semantic  
678 data storage is basically a central repository of user information, which stores  
679 relationships between pieces of data in a way that is re-usable among multiple  
680 applications.

---

<sup>3</sup><https://tracker.gnome.org/>

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

684 This makes use of several existing technologies and standards:

- 685 • **Resource Description Framework (RDF<sup>4</sup>)**. RDF is a directed, la-  
686 beled graph data format for representing information, and is a W3C stan-  
687 dard.
- 688 • **SPARQL<sup>5</sup>** is a W3C standard defining a query language for databases,  
689 able to retrieve and manipulate data stored in RDF format.
- 690 • **Ontologies<sup>6</sup>**. An ontology represents knowledge as a set of concepts  
691 within a domain, and the relationships between those concepts. It can be  
692 used to reason about the entities within that domain and may be used to  
693 describe the domain.
- 694 • **Nepomuk<sup>7</sup>** (Networked Environment for Personalized, Ontology-based  
695 Management of Unified Knowledge). Nepomuk is a research project, which  
696 defined a set of ontologies describing desktop entities like files, pictures,  
697 etc.

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

- 702 • **Tracker Storage**. The data store and daemon to interface to Tracker's  
703 databases.
- 704 • **Tracker SPARQL<sup>8</sup>**. The libtracker-sparql library is the foundation for  
705 Tracker querying and inserting data into the data store based on the Nepo-  
706 muk ontology.
- 707 • **Tracker Miner<sup>9</sup>**. The libtracker-miner library is the foundation for  
708 Tracker data miners. These miners will extract metadata and insert it  
709 in SPARQL form into the Tracker store, following the Nepomuk ontolo-  
710 gies. Developers can add new miners in order to index new data sources.
- 711 • **Tracker Extract<sup>10</sup>**. The libtracker-extract library is the foundation for  
712 Tracker metadata extraction of embedded data in files. Tracker comes  
713 with extractors written for the most common file types (like MP3, JPEG,

---

<sup>4</sup><http://www.w3.org/RDF/>

<sup>5</sup><http://www.w3.org/TR/rdf-sparql-query/>

<sup>6</sup><http://developer.gnome.org/ontology/0.12/>

<sup>7</sup><http://www.semanticdesktop.org/ontologies/>

<sup>8</sup><https://gitlab.gnome.org/GNOME/tinysparql>

<sup>9</sup><https://gitlab.gnome.org/GNOME/localsearch>

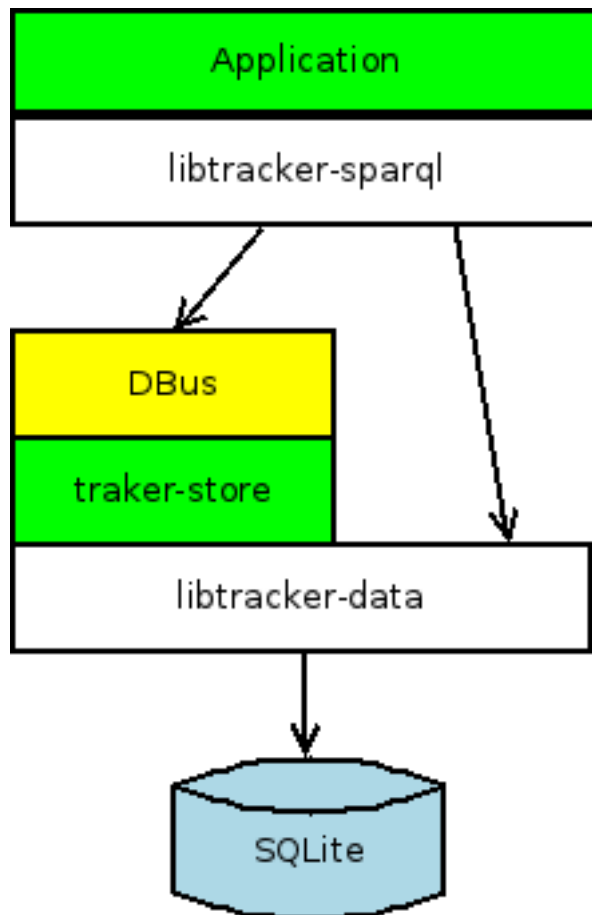
<sup>10</sup><https://gitlab.gnome.org/GNOME/localsearch>

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

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

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

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



727

- 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<sup>11</sup> 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 `XDGCACHEHOME`, as the Tracker SQLite database is stored in `$XDGCACHEHOME/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<sup>12</sup>) 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

<sup>11</sup><http://www.sqlite.org/>

<sup>12</sup><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:

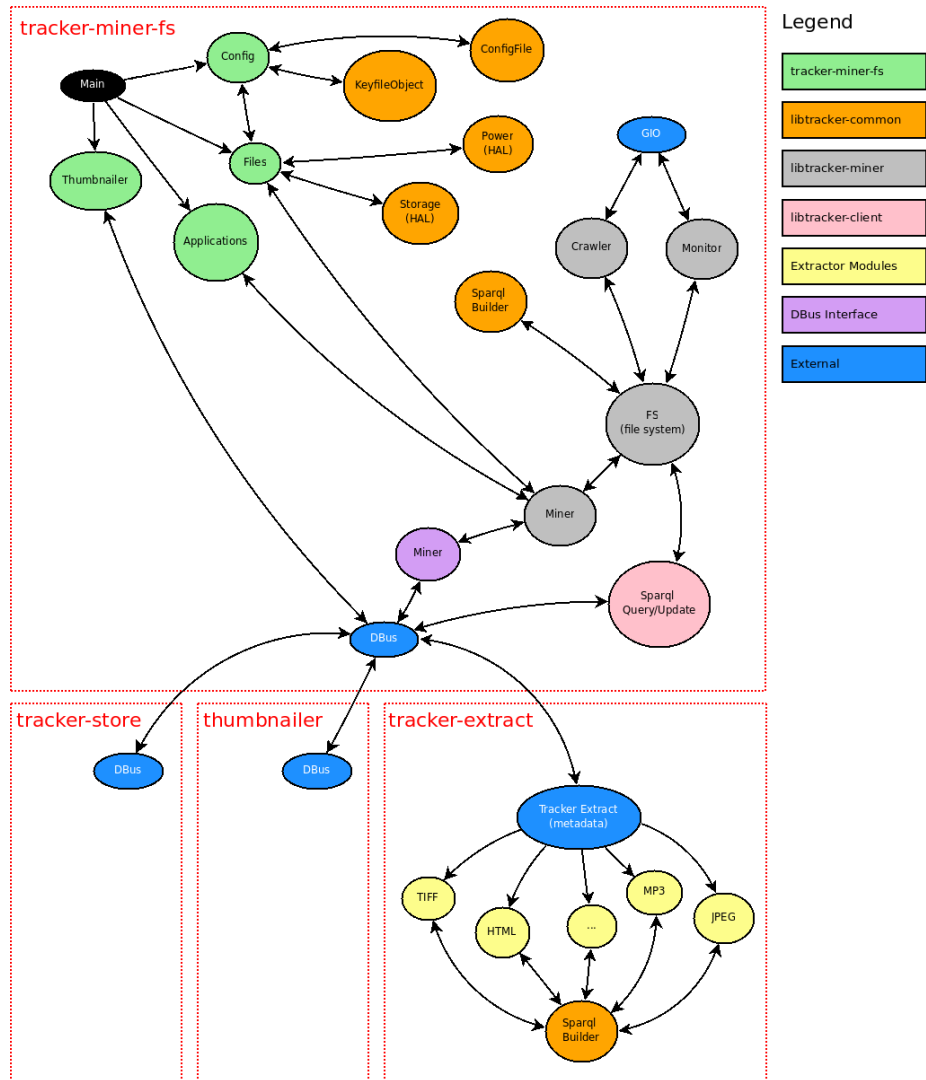
- **Startup wait time.** Primarily to avoid prevent Tracker from heavily loading the system just after boot. By default 15 seconds.

- 797 • **Scheduler priority.** Specifies the priority of indexing directories and  
798 files. There are three levels: when idle, first indexing on idle (default) and  
799 anytime.
- 800 • **Throttle.** Controls the throttle of file indexing operations. This specifies  
801 to control the overhead indexing has on the system. Of course, it is a  
802 trade-off between system load and speed, but it can be tuned to make UI  
803 applications more responsive. It is a value between 0 and 20, the higher  
804 the slower. A value of 0 denotes “as fast as possible”(default), any other  
805 number N denotes 20/N indexing operations per second. These limits can  
806 of course be adjusted internally.
- 807 • **Low disk space limit.** A configurable parameter to stop indexing in  
808 case of low free disk space. It is configurable between 0% (no limit) and  
809 100%. It is 1% by default.
- 810 • **Crawling interval.** Specifies the interval in days to check whether the  
811 filesystem is up to date with the database. A value of -1 specifies the  
812 check should only be done on unclean shutdowns and -2 specifies this  
813 check should be disabled entirely.
- 814 • **Removable days threshold.** Specifies the threshold in days after which  
815 metadata for files from removable devices will be removed if their filesys-  
816 tem is not mounted. Zero means never. Configured to 3 days by default.
- 817 • **File monitoring.** Option to track filesystem changes directly in order to  
818 know what needs to be indexed.
- 819 • **File Writeback.** Option to write information back in the files, e.g. meta-  
820 data retrieved from other sources or updated by the application, it can be  
821 stored back in the original file. It is limited to a few formats currently.
- 822 • **Index Removable Devices.** Option to enable / disable the indexing of  
823 removable devices.
- 824 • **Index Optical Discs.** Option to enable / disable the indexing of CDs,  
825 DVDs, and in general any optical media.
- 826 • **List of directories to index recursively.** It can also refer to special  
827 XDG directories like Desktop, Documents, Download, Music, Pictures,  
828 Public, Templates and Videos.
- 829 • **List of single directories to index** (non-recursively). Same notes as  
830 before.
- 831 • **List of ignored files.** Filenames can be specified with wildcards.
- 832 • **List of ignored directories.** Wildcards can be used to specify them.
- 833 • **List of ignored directories with content.** Avoid any directory con-  
834 taining a file whose name is blacklisted in this list.

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

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

846 See a general overview in the following illustration.



847

848 **Tracker Extract** Tracker extract does the actual metadata extraction. It  
849 inspects the media content and it extracts metadata information, which is stored  
850 in Tracker Store. There is a list of the currently [Tracker supported file formats](https://wiki.gnome.org/Projects/Tracker/SupportedFormats)<sup>13</sup>.  
851 It includes the main formats for all the media content types of interest (music,  
852 music playlist, video, picture, picture album and documents).

853 **Note:** in some Tracker extract plugins like the GStreamer one, the actual for-  
854 mats able to be extracted depend on the specific GStreamer plugins installed  
855 on the system.

<sup>13</sup><https://wiki.gnome.org/Projects/Tracker/SupportedFormats>

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

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

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

871 These are the configuration options for Tracker Extract:

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

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

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

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

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

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

## 907 Thumbnail Management

908 The **Thumbnail Managing Standard**<sup>14</sup> deals with the permanent storage  
909 of previews for file content. The **Thumbnail Management D-Bus spec-**  
910 **ification**<sup>15</sup> is a standardized D-Bus API to deal with thumbnailing. This D-  
911 Bus specification is currently implemented by Tumbler, which has been already  
912 used successfully in consumer products like the Nokia N9 phone. With a D-Bus  
913 specification for thumbnail management, applications don't have to implement  
914 thumbnail management themselves. If a thumbnailer is available they can dele-  
915 gate thumbnail work to a specialized service. The service then calls back when  
916 it has finished generating the thumbnail.

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

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

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

933 There are several components in the Thumbnailer service:

---

<sup>14</sup><http://specifications.freedesktop.org/thumbnail-spec/thumbnail-spec-latest.html>

<sup>15</sup><https://wiki.gnome.org/DraftSpecs/ThumbnailerSpec>

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

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

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

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

959 See [here](http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thumb-)<sup>16</sup>, to discover the mime types they currently support you  
960 need to navigate to their `provider_get_thumbnailers` implementation,  
961 for example `gst_thumbnailer_provider_get_thumbnailers` in [http://gi](http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thumb-)  
962 [t.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thum](http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thumb-)  
963 [bnailer-provider.c](http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thumb-)

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

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

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

<sup>16</sup><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**<sup>17</sup> 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**<sup>18</sup> 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

<sup>17</sup><https://wiki.gnome.org/DraftSpecs/MediaArtStorageSpec>

<sup>18</sup><https://wiki.gnome.org/Projects/Grilo>



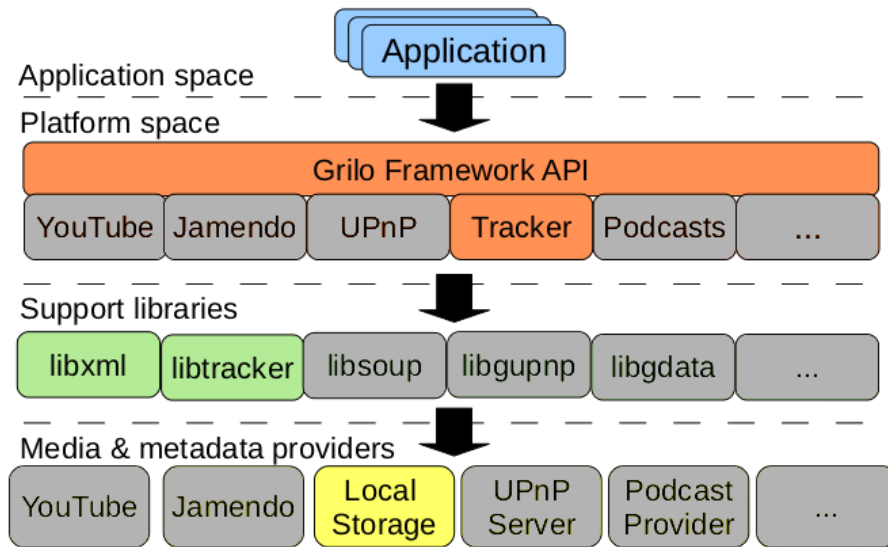
1005 various services and sources easily. Grilo comes with a collection of plugins for  
1006 accessing various media providers, like Vimeo, Flickr, YouTube etc. so they can  
1007 be presented uniformly via the Grilo API. Additionally a grilo-tracker plugin  
1008 exists, which uses the Tracker service (described in past sections), to make media  
1009 indexed by Tracker available through the Grilo API.

1010 There is an additional Grilo plugin for accessing the filesystem directly (grl-  
1011 filesystem), which checks for media content in a set of configured directories.  
1012 The defaults are the XDG user directories for pictures, music and videos.

1013 Although Grilo can be used to access many media content sources, we suggest  
1014 only using it for accessing local media content. The next sections will dig into  
1015 Grilo's details and its advantages. The main advantages of using Grilo instead  
1016 of Tracker directly for this specific use case:

- 1017 • Tracker is a semantic data storage, which can be used to store other bits  
1018 of information apart of indexing information from media content like mes-  
1019 sages, calendars, etc. In other words, it is a very general framework usable  
1020 for many purposes. Therefore, it makes sense to provide a higher level  
1021 specialized API for media browsing (Grilo) on top of Tracker to hide its  
1022 complexity from media applications.
- 1023 • Grilo has some plugins that might be useful to extract additional metadata,  
1024 e.g. album art from last.fm. Grilo is specially recommended for accessing  
1025 to metadata from the Internet, which is not meant to be indexed. In  
1026 addition, the platform could take advantage of future plug-ins which are  
1027 planned to be developed by the FOSS community like lyrics, moviedb.org,  
1028 etc.
- 1029 • Grilo would support using an indexer other than Tracker if a better one  
1030 becomes available. More importantly, applications wouldn't have to be  
1031 modified to take advantage of such a change.

1032 See the following illustration for an overview of the Grilo Architecture. Note  
1033 the boxes with grey background are not going to be used in the context of the  
1034 Apertis project.



1035

1036 **Grilo Media Source Plugins** The plugin must create at least one `GrlMediaSource`  
 1037 instance, and register it in the Grilo registry. A `GrlMediaSource`  
 1038 represents a particular source of media. These plugins provide several func-  
 1039 tions:

- 1040 • **Search** content by keywords.
- 1041 • **Browse** the media content in a hierarchical way. It is similar to exploring  
 1042 a filesystem, entering into folders (`GrlMediaBox`) and browsing files in it.
- 1043 • **Query** allows access to content using service specific language. Normally  
 1044 it provides additional filtering capabilities. This is used by applications to  
 1045 support plugin-specific functionality.
- 1046 • **Metadata** used to request additional metadata.
- 1047 • **Store** (optional), supports to push content to the source.
- 1048 • **Remove** (optional), to remove stored contents from the source.
- 1049 • **Supported keys** provides information on which metadata keys are pro-  
 1050 vided by the plugin. Typical metadata keys are: id, title, url, thumbnail,  
 1051 mime, artist, duration.
- 1052 • **Slow keys** (optional) provides info on which metadata keys are expen-  
 1053 sive to gather. So the applications could just ask for non-expensive ones  
 1054 normally, and only require the slow keys when details are required for a  
 1055 particular media content.
- 1056 • **Media from URI**. Gets `GrlMedia` from a URI. For example a file browser  
 1057 may use this to get metadata for a specific file.

1058     • **Test Media from URI** (optional). To check if the plugin can convert a  
1059         URI into a GrlMedia object.

1060     • **Notifications** on changes on media content.

1061     At least one of the content retrieval methods is expected to be implemented:  
1062     search, browse or query. Each media content result of the search/browse/query  
1063     is represented by a GrlMedia object.

1064     Plugins should be implemented in a non-blocking way to have a smooth user  
1065     experience in applications. Also threads are not recommended; splitting work  
1066     into chunks using the idle loop is encouraged.

1067     There is a standard set of metadata keys defined, but plugins can define their  
1068     own custom metadata keys.

1069     A GrlMedia can have multi-valued properties; for example a YouTube video with  
1070     different resolutions (and thus, different URIs). It is also possible to associate  
1071     different properties with each URI of a GrlMedia.

1072     **Grilo Metadata plugins** Grilo metadata source plugins do not provide ac-  
1073     cess to media content, but additional metadata information. An example would  
1074     be to provide thumbnail information for local audio content from an online  
1075     service.

1076     This plugin must create at least one GrlMetadataSource instance, and register  
1077     it in the Grilo registry. The plugin provides several functions:

1078     • **Resolve** retrieves additional information for a GrlMedia object.

1079     • **May resolve:** to check if Resolve may be performed with existing infor-  
1080         mation.

1081     • **Set metadata** (optional): set the play count or the last time a media  
1082         was played.

1083     • **Writable keys** (optional): reports which keys can be stored.

1084     • **Supported keys:** provides information on which metadata keys are pro-  
1085         vided by the plugin.

1086     • **Slow keys** (optional): provides info on which metadata keys are expensive  
1087         to gather. So the applications can ask for inexpensive keys normally, and  
1088         only request the slow keys when details are required for a particular media  
1089         content.

1090     • **Cancel operations:** cancels ongoing operations.

## 1091 Google Data Protocol

1092 **YouTube**, as well as other Google services like Picasa, use the **Google Data**  
1093 **Protocol**<sup>19</sup>. The Google Data Protocol is a REST-inspired technology for read-  
1094 ing, writing, and modifying information on the web. The protocol currently  
1095 supports two primary modes of access: AtomPub and JSON. The JSON is a  
1096 mapping of Atom items to JSON objects meant to be used for web applications  
1097 written in JavaScript.

1098 The AtomPub mode is based on the Atom Publishing protocol, with names-  
1099 paced **XML** additions. Communication between the client and server is broadly  
1100 achieved through **HTTP** requests with query parameters, and Atom feeds being  
1101 returned with result entries. Each *service* has its own namespaced additions to  
1102 the GData protocol; for example, the Google Calendar's API has specializations  
1103 for addresses and time periods.

1104 Collabora proposes **libgdata**<sup>20</sup>, which is a library to allow access to web ser-  
1105 vices using the Google Data Protocol from traditional applications. Results are  
1106 always returned in the form of result *feeds*, containing multiple *entries*. How the  
1107 entries are interpreted depends on what was queried from the service, but when  
1108 using libgdata, this is all taken care of transparently. The main dependencies  
1109 of libgdata are libsoup, libxml and liboauth.

1110 Other frameworks and applications are already using libgdata with success, e.g.  
1111 evolution-data-server, Totem's YouTube plugin and Grilo's YouTube plugin.

1112 The library libgdata already provides an implementation for the **GDataY-**  
1113 **outuTubeService**<sup>21</sup>, which provides the following functionality:

- 1114 • Query videos.
- 1115 • Query videos related to a specific video.
- 1116 • Query standard feed types: top rated, top favorites, most viewed, most  
1117 popular, most recent, most discussed, most linked, most responded, re-  
1118 cently featured and watch on mobile.
- 1119 • Upload a video.
- 1120 • Get categories.

## 1121 Librest and libsoup

1122 It is difficult to find libraries to access online media sources if they are not pro-  
1123 vided by the vendors themselves. However, most of these online media sources  
1124 are based on HTTP protocol with **REST**<sup>22</sup> interfaces. Therefore, in general, [li-

---

<sup>19</sup><http://code.google.com/apis/gdata/>

<sup>20</sup><http://developer.gnome.org/gdata/0.10/gdata-overview.html>

<sup>21</sup><http://developer.gnome.org/gdata/0.10/GDataYouTubeService.html>

<sup>22</sup>[http://en.wikipedia.org/wiki/Representational\\_state\\_transfer](http://en.wikipedia.org/wiki/Representational_state_transfer)

1125 **brestr**] and/or **libsoup**<sup>23</sup> will be useful. **Librest** is a library designed to make it  
1126 easier to access web services that are designed in a “RESTful” manner. **Libsoup**  
1127 is an HTTP client/server library for GNOME. It uses GObject and the glib  
1128 main loop, to integrate well with GNOME applications. Collabora can suggest  
1129 or provide advise for open-source ways for accessing these services on request.  
1130 This is the most effective way to access all the features.

## 1131 **Playlists support**

1132 Playlists are supported in Tracker. There is an specific Tracker Extract plugins  
1133 to handle playlists, which is using internally the **Totem Playlist Parser**<sup>24</sup> li-  
1134 brary, which is conveniently abstracted and independent of Totem. Tracker  
1135 Extract introduces the metadata retrieved in Tracker Store using the class  
1136 nmm:Playlist, which is a subclass of nfo:MediaList. The entries in the playlist  
1137 are introduced as nfo:MediaFileListEntry.

1138 The supported playlist formats in Totem Playlist Parser are: audio/x-  
1139 mpegurl, totem-plparser, audio/x-scpls, audio/x-pn-realaudio, application/ram,  
1140 application/vnd.ms-wpl, application/smil and audio/x-ms-asx.

1141 Grilo does not support playlists in the latest stable version available, so this  
1142 feature would need to be added as specified in the requirements section.

## 1143 **Appendix: Questions & Answers**

1144 These chapter contains very specific questions that have been asked during work-  
1145 shops.

1146 **Q: Will asking for a specific prioritization during metadata extrac-**  
1147 **tion increase the load by running multiple indexing jobs ?** A: No,  
1148 the Tracker scheduler will manage all metadata indexing operations in internal  
1149 queues, so prioritization will just change the sorting of the metadata indexing op-  
1150 erations, but not the overall system load. Note the scheduling system proposed  
1151 in this document is not implemented in Tracker yet. See **Indexing scheduling**  
1152 and **Tracker scheduling** for more details on prioritization and Tracker scheduling.

1153 **Q: How does the system know when to renew thumbnails ?** A: When  
1154 a thumbnail is generated, some properties are stored inside it like the original  
1155 URI and the modification time of the original file. If the original file is modified  
1156 at some point, its modification time will get changed automatically by the Linux  
1157 filesystem. So, it is possible to know when a thumbnail is outdated. Additionally,  
1158 Tracker is monitoring the filesystem for changes. In case a file is modified,  
1159 added, moved or deleted its thumbnail will be automatically updated. Note:  
1160 this feature is not fully implemented yet, but it is part of the modifications  
1161 Collabora will implement.

---

<sup>23</sup><https://gitlab.gnome.org/GNOME/libsoup>

<sup>24</sup><http://developer.gnome.org/totem-pl-parser/stable/>

1162 **Q: How the mime type of the files is determined ?** A: This is done  
1163 through glib, which finds out the mime type in a efficient way and it is used  
1164 extensively by all GNOME based software. The details of the algorithm used  
1165 can be seen in the [Shared MIME Info Specification](#)<sup>25</sup>, it has been designed to  
1166 be robust and efficient. The first thing done is to test the filename extension  
1167 to see if it is a recognized type. If this operation cannot be done or the result  
1168 is uncertain, a second check will be done using the first bytes of the file check-  
1169 ing for the signature of known files. For more details see `g_file_query_info`,  
1170 `G_FILE_ATTRIBUTE_STANDARD_CONTENT_TYPE` and `g_file_info_get_content_type`  
1171 in GNOME documentation.

1172 **Q: How the video thumbnailing works to avoid black video frames**  
1173 **or uninteresting frames in general ?** A: From [Thumbnail management](#):  
1174 “Video thumbnails can be generated using the GStreamer thumbnailing plugin.  
1175 This plugin already provides an heuristic method to extract the thumbnail from  
1176 a video stream, by selecting a frame with a wide distribution of colors (to avoid  
1177 presenting a title screen or other essentially-blank frame). Other ways could be  
1178 implemented if required, just by implementing a thumbnail plugin.

1179 **Q: How document thumbnailing works to avoid thumbnails of blank**  
1180 **pages ?** A: The existing Tumbler plugins used to extract thumbnails from  
1181 Open/LibreOffice, PDF and Microsoft Office documents gets the thumbnail  
1182 stored inside the file. It is responsibility of the office applications to write a  
1183 proper thumbnail. Typically it is just the thumbnail of the first page of the  
1184 document, which usually is the best option since the first page contains the title  
1185 in bigger font sizes, cover of the document and logos. Any other approach is  
1186 debatable, so Collabora does not recommend to make thumbnails from only text  
1187 pages since they are less likely to be useful, thumbnailing normal text would  
1188 become unreadable.

1189 **Q: How the applications can store and retrieve the last time a**  
1190 **media file was played ?** A: This functionality can be provided by the Grilo  
1191 metadata store plugin. The application must query the last values and set new  
1192 values through Grilo API. The media file is identified via the file URI. The  
1193 metadata store plugin stores these values in a Tracker database. It currently  
1194 supports the following values: last position where media item was played  
1195 (`GRL_METADATA_KEY_LAST_POSITION`), number of times a media  
1196 item has been played (`GRL_METADATA_KEY_PLAY_COUNT`) and last  
1197 date a media item was played (`GRL_METADATA_KEY_LAST_PLAYED`).  
1198 Grilo is making use of the properties already defined on the Tracker ontologies  
1199 like `nfo:lastPlayedPosition`, `nie:usageCounter` and `nie:contentAccessed`. A  
1200 benefit of using Grilo is that Tracker details are not exposed to the applications,  
1201 for example alternatively Grilo has another plugin to store these fields in a

---

<sup>25</sup><http://standards.freedesktop.org/shared-mime-info-spec/shared-mime-info-spec-latest.html>

1202 separate SQLite database in case Tracker was not used, but the API to set and  
1203 get these properties would be the same.

1204 **Q: How a thumbnail is retrieved ?** A: Thumbnails can be retrieved  
1205 through different ways depending on what specific APIs the application is  
1206 using. The best way for media applications would be through the Grilo API,  
1207 see `grl_media_get_thumbnail` and `grl_media_get_thumbnail_binary_nth`  
1208 (in case several thumbnails are available for a media item). Grilo API  
1209 is internally using glib library to retrieve this through `g_file_query_info`,  
1210 `G_FILE_ATTRIBUTE_THUMBNAIL_PATH` and `g_file_info_get_attribute_byte_string`.  
1211 Grilo API will need to be modified in case more thumbnails need to be stored  
1212 on the USB flash devices.

1213 **Q: How the system behaves on robustness on power loss ?** A: This and  
1214 other questions on system robustness will be answered on a separate document  
1215 focused on system robustness. Anyway, please see chapter [Indexing database](#)  
1216 [on removable device](#) for an advance of some issues regarding USB flash devices.

1217 **Q: How a media file from a USB Flash device is identified ?**  
1218 A: It is identified by its complete URI, e.g. `/media/D8C0-024E/Joaquin`  
1219 `Sabina/Joaquin Sabina & Fito Paez - Lluve sobre mojado.mp3`". In some  
1220 systems, USB flash devices are mounted on a directory with a hex identifier  
1221 (depending on system configuration). This identifier is the UUID (Universally  
1222 Unique Identifiers), not the label of the USB flash device. It is generated when  
1223 the filesystem is created, and it is very. Generally it is a 128 bit identifier, but  
1224 some filesystems like VFAT have smaller resolution (32 bits).

1225 **Q: Is it configurable the timeout for Tracker extract operations ?** A:  
1226 No, they are not currently, but it would be simple to make them configurable  
1227 for example through GSettings. There are two timeouts. A watchdog timeout  
1228 which checks that the tracker extract process does not hang during metadata  
1229 extraction (by default set to 20 seconds). There is an additional idle timeout,  
1230 which stops a tracker extract process if it has been idle for some time (30 seconds  
1231 by default).

1232 **Q: Does Tracker retry in case Tracker Extract fails due to the watch-**  
1233 **dog timer ?** A: By default, Tracker retries up to two times if a tracker extract  
1234 process fails. It will also retry in case the file is modified or the USB flash where  
1235 it is located is reinserted.

1236 **Q: Does Tracker store marks for the corrupted files ?** A: Currently,  
1237 there is no property to identify corrupted files in Tracker. A file whose extract  
1238 process has failed due to corruption in the file, it would just have properties  
1239 from the nfo ontology (nepomuk file object), but it would not have properties  
1240 from other subclasses like nmm (nepomuk multimedia).

1241 **Q: There are reports of performance of page queries on Tracker**  
1242 **databases is negatively affected by the number of rows in the**  
1243 **database. Collabora to double check.** A: Some tests running SPARQL  
1244 queries have been done with databases near 6000 items and the mentioned  
1245 problem was not reproducible (no performance problems found). Please  
1246 provide data set and application code reproducing this problem for further  
1247 investigation.

1248 **Q: Should Tracker be used for Radio Stations information ?** A:  
1249 Tracker has already ontologies to store radio station information. So, it would  
1250 be possible to use it to store and retrieve the user favorite radio stations.  
1251 However, the interface to access and update this information would be through  
1252 plain SPARQL, which has a steep learning curve for developers. Additionally,  
1253 the radio station information is not shared with other applications. The only  
1254 advantage of using Tracker would be that the global search would automatically  
1255 work for radio station information, so it would not be necessary to implement  
1256 an extra global search plugin to look for this info in another database. The  
1257 final decision must take into consideration how well the existing ontology for  
1258 radio stations ([nmm:RadioStation](#)<sup>26</sup>) is suited to Apertis' roadmap.

1259 **Q: What happens when a USB flash device is inserted in a USB port ?**  
1260 A: When the user inserts a USB flash device, there are three main components  
1261 participating in the action:

- 1262 • **Linux kernel** (including device drivers). The kernel will be able to com-  
1263 municate with the device as soon as it is powered up, initialized and  
1264 announced through the USB Bus.
- 1265 • **Udev**<sup>27</sup> is the device manager for the Linux kernel. Primarily, it manages  
1266 device nodes in /dev. It is the successor of devfs and hotplug, which  
1267 means that it handles the /dev directory and all user space actions when  
1268 adding/removing devices, including firmware load. The Udev daemon  
1269 listens to the netlink socket used by the kernel to communicate with user  
1270 space applications. The kernel will send a bunch of data through the  
1271 netlink socket when a device is added to, or removed from a system. The  
1272 Udev daemon catches all this data and will do the rest, i.e., device node  
1273 creation, module loading etc.
- 1274 • **UDisks**<sup>28</sup> (formerly known as DeviceKit-disks) lies on top of udev, and it  
1275 is an abstraction for enumerating disk and storage devices and performing  
1276 operations on them. It is a replacement for part of the functionality  
1277 which used to be provided by the now deprecated HAL (Hardware Abstrac-

---

<sup>26</sup><http://developer.gnome.org/ontology/0.14/nmm-ontology.html>

<sup>27</sup>[http://www.kroah.com/linux/talks/ols\\_2003\\_udev\\_paper/Reprint-Kroah-Hartman-OLS2003.pdf](http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf)

<sup>28</sup><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 indexed
2.8	2.8	(2) UDisks daemon reports a USB flash device has been inserted via D-Bus. The user is notified
3.6	0.8	(3) UDisks daemon notifies the partition in the USB Flash has been mounted automatically
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 filesystem
6	0.6	(5a) Tracker Extract has metadata for the files that have been returned in the first pass
46	40	(5b) Tracker Extract finishes gathering metadata for all files in the USB flash device

#### 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](#)<sup>29</sup> 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*”<sup>30</sup>.

<sup>29</sup><http://developer.gnome.org/gio/unstable/GFileMonitor.html>

<sup>30</sup><http://www.ibm.com/developerworks/linux/library/l-ubuntu-inotify/index.html>