



Media management

1 Contents

| | | |
|----|---|----|
| 2 | Solution | 3 |
| 3 | Technology and Solution Overview | 3 |
| 4 | Local Storage Media Source | 5 |
| 5 | Media Browsing Requirements | 6 |
| 6 | Media Indexing Database Requirements | 8 |
| 7 | Indexing Scheduling | 13 |
| 8 | Thumbnailing | 18 |
| 9 | DLNA (UPnP) | 19 |
| 10 | Online Media Sources | 20 |
| 11 | Bluetooth AVRCP | 20 |
| 12 | Playability check | 20 |
| 13 | Appendix: Media Management Technologies | 22 |
| 14 | Tracker | 22 |
| 15 | Thumbnail Management | 30 |
| 16 | Grilo | 32 |
| 17 | Google Data Protocol | 36 |
| 18 | Librest and libsoup | 36 |
| 19 | Playlists support | 37 |
| 20 | Appendix: Questions & Answers | 37 |

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

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

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

34 Although this document is mostly focused on the media content, the technolo-
35 gies introduced are related with other features in the platform like global search,
36 which allows to search not only in media content but also in applications, mes-
37 sages, calendar events, etc. For details on global search please check its specific
38 design.

39 **Appendix: Media management technologies** is mostly used as reference material
40 from other sections of the document, so it is not necessary to read from start-
41 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:

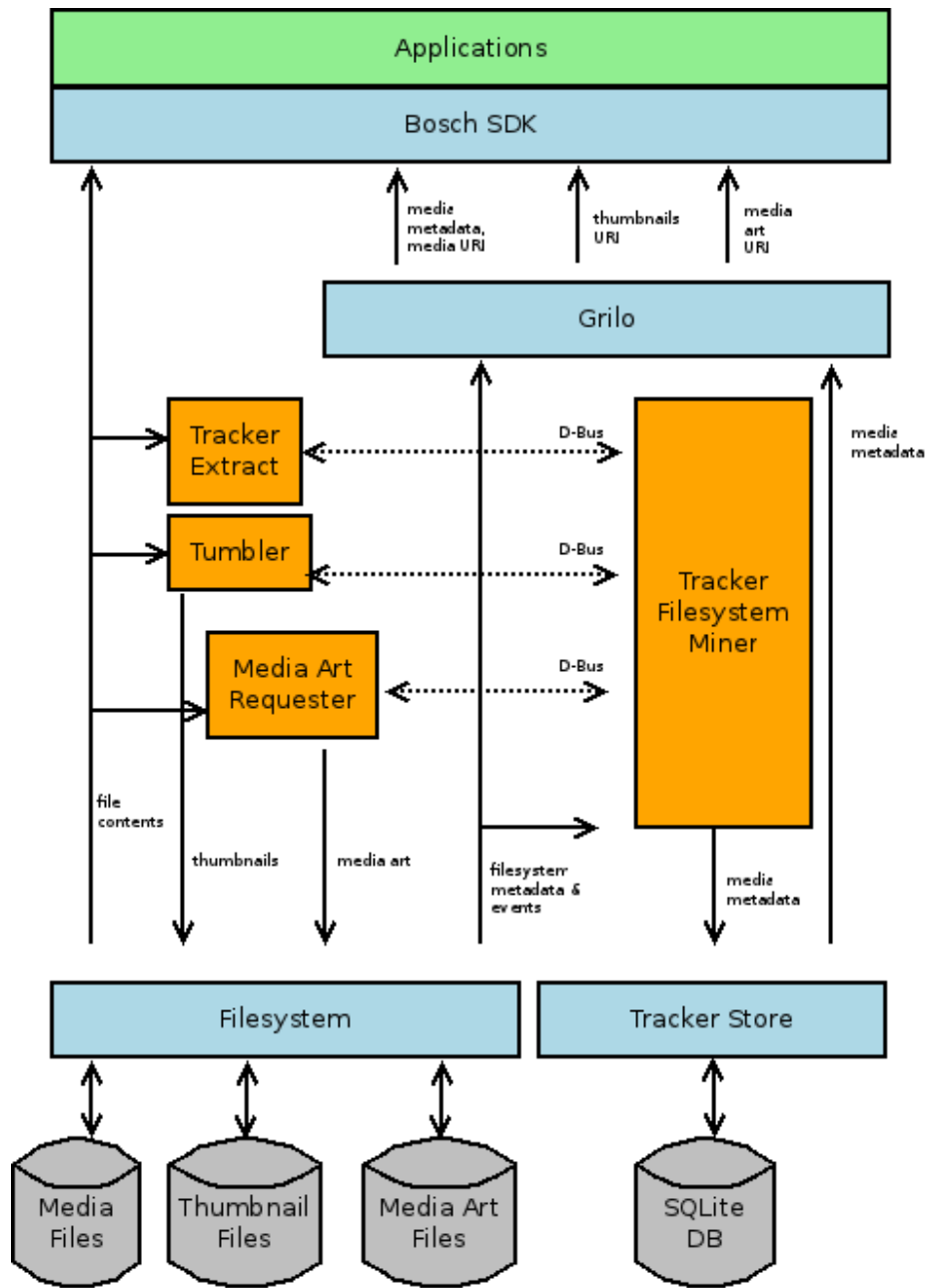
99 • Prioritization done automatically by Tracker in a hard-coded way (not
100 configurable), like gathering all metadata from filesystem (filename, size,
101 modification time, etc.) before extracting metadata from the file contents.

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,
105 upper layers need to provide some clues about what needs to be done
106 first or what is more important, like a picture viewer application boosting
107 priority to metadata extraction of image files (instead of the default which
108 could 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

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

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

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

128 See requirement R5 for comments on public and private content.

129 **Status.** Satisfied.

130 **Media Browsing Requirements**

131 **File-system based browsing**

132 **Requirement R2.** Support filesystem based browsing for early access.

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

141 **Solution.**

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

148 Another solution would be to use Grilo Tracker plugin. Grilo plugins provide
149 access to the media content in a hierarchical way. Grilo Tracker plugin has two
150 modes of hierarchical navigation, one based on categories and another one based
151 on the filesystem. The latter one provides the info in the same structure as it is
152 stored in the filesystem. It allows to browse from a root folder or from specific
153 folders. However, the information has to be previously available in Tracker Store
154 for this to work. To minimize this delay, Tracker scheduler will be changed to get
155 filesystem information before other media metadata. Obtaining the filesystem
156 information is very fast compared to the extraction of the metadata (which
157 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**

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

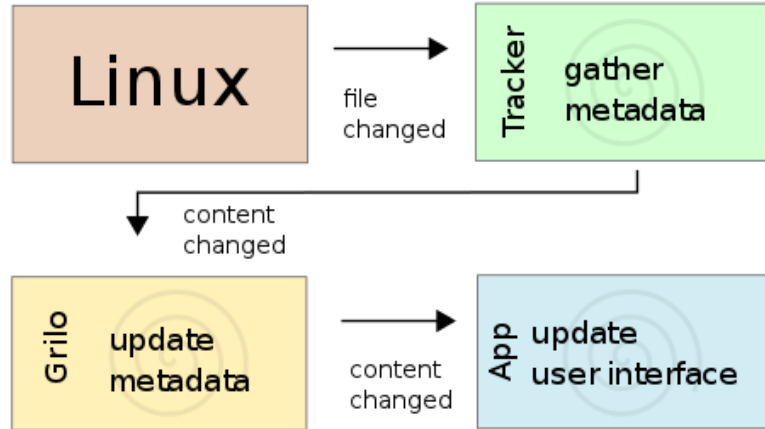
177 **Analysis.** The indexing process is asynchronous, it can happen that media
178 content gets its metadata updated while the content is already being shown to
179 the user.

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

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

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

195 **Status.** Satisfied.



196

197 **Paged queries**

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

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

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

207 **Status.** Satisfied.

208 **Media Indexing Database Requirements**

209 **Media indexing of shared and private files**

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

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

217 We assume there are folders which are public (shared and accessible to all users
 218 in the system) and folders which are private (only accessible to the owner of the

219 folder). Due to the existence of private content, each user must have its own
220 Tracker database for storing metadata.

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

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

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

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

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

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

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

260 **Status.** Satisfied.

261 **Database version management**

262 **Requirement R6.** The system should be able to cope with database version
263 updates.

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

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

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

- 300 • If Tracker Store just holds indexing information, this could be regenerated
301 by re-indexing, so there would be no real data loss on an database version
302 update.

- 303 • If Tracker Store keeps information entered by the user, like user tags, then
304 it would be lost during a full re-index. To prevent this, an ad-hoc tool
305 could be implemented to convert this information to the new database
306 version.
- 307 • Often the manufacturers or distribution maintainers decide to not deploy
308 new changes on the ontologies to avoid these database update problems.
309 Anyhow, some changes could be supported via some custom code, like
310 adding / removing properties; but others affecting the domains or class
311 hierarchy are much harder to handle. Each case of ontology change needs
312 to be analyzed particularly.

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

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

322 **Status.** Satisfied.

323 **Indexing database on removable device**

324 **Requirement R7.** Storage of the indexing information for removable storage
325 in the removable storage itself.

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

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

343 – Write cache disabling for the USB flash device decreases the data
344 corruption risk, but the risk does not disappear. The user could
345 still eject on the middle of a write operation. As a result of the
346 disabled cache write operations will be slower. Additionally, USB
347 flash manufacturers tend to lie regarding sync requests.

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

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

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

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

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

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

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

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

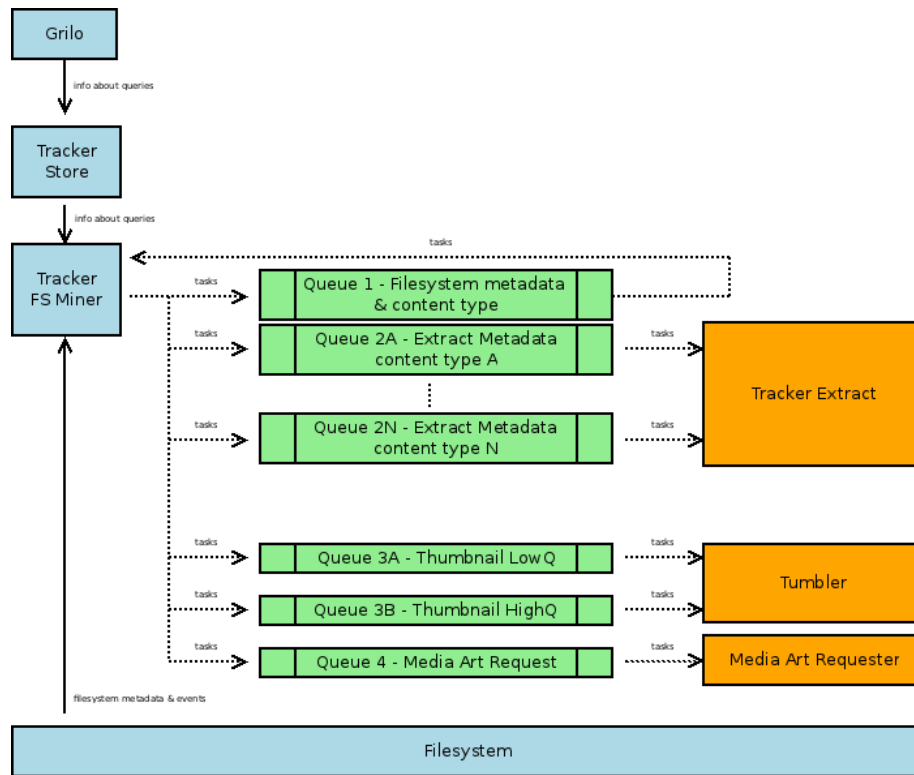
379 **Status.** Satisfied.

380 **Indexing Scheduling**

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

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

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



406

407 **Media Content Counters**

408 **Requirement R8.** Provide the number of items per content type as soon as
 409 possible.

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

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

424 **Solution.** Collabora will add an option in Tracker's configuration to enable two

425 pass indexing. If enabled, tracker will first crawl the whole filesystem to store
426 files' attributes but won't try to get embedded information (e.g. MP3 metadata,
427 etc). A boolean property will be added in Tracker's database for files that need
428 a 2nd pass, so Tracker knows which files needs a 2nd pass when it is done
429 crawling the filesystem. That property needs to be written into the database
430 (and not only in-memory) so Tracker is able to correctly resume its indexing after
431 a system reboot. Additionally, directories containing partially indexed files will
432 be flagged (in memory), to avoid re-crawling the whole filesystem when doing
433 the 2nd pass (a list of all partially indexed files would be too big and consume
434 too much memory).

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

437 **Status.** Satisfied.

438 **Prioritized extraction per content type**

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

443 **Analysis.** Current Tracker scheduling does the metadata extraction in no spec-
444 ific order.

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

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

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

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

460 If priorities changes while Tracker is doing the 2nd pass, it stops and crawl di-
461 rectories where files have been skipped earlier. When done, Tracker will resume
462 doing the 2nd pass.

463 In summary, Tracker will do the 1st pass indexing (file attributes only, no em-
464 bedded metadata) on prioritized files, then 2nd pass on prioritized files, then
465 1st pass on not prioritized files, and finally the 2nd pass on not prioritized files.

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

468 **Status.** Satisfied.

469 **Selective prioritized extraction**

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

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

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

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

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

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

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

498 **Status.** Satisfied.

499 **Selective prioritized thumbnailing**

500 **Requirement R11.** Prioritize thumbnails depending on user activity.

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

502 **Status.** Satisfied.

503 **Multi pass metadata extraction**

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

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

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

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

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

521 **Status.** Satisfied.

522 **Concurrency configurable**

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

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

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

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

542 **Status.** Satisfied.

543 **Thumbnailing**

544 **Two-step thumbnailing**

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

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

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

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

558 **Status.** Satisfied.

559 **Thumbnail resolution configuration**

560 **Requirement R15.** Resolutions for thumbnail flavors normal and high must
561 be configurable.

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

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

566 **Status.** Satisfied.

567 **Thumbnailing algorithm configuration**

568 **Requirement R16.** The algorithm used for calculating the thumbnails must
569 be configurable.

570 **Analysis.** Currently Tumbler implements several plugins for thumbnail calcu-
571 lation.

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

- 576 • **Nearest:** nearest neighbor sampling. This is the fastest and lowest quality
577 mode. Quality is normally unacceptable when scaling down, but may be
578 OK when scaling up.
- 579 • **Tiles:** this is an accurate simulation of the PostScript image operator
580 without any interpolation enabled. Each pixel is rendered as a tiny par-
581 allelogram of solid color, the edges of which are implemented with an-
582 tialiasing. It resembles nearest neighbor for enlargement, and bilinear for
583 reduction.
- 584 • **Bilinear:** best quality/speed balance; use this mode by default. For en-
585 largement, it is equivalent to point-sampling the ideal bilinear-interpolated
586 image. For reduction, it is equivalent to laying down small tiles and inte-
587 grating over the coverage area.
- 588 • **Hyper:** this is the slowest and highest quality reconstruction function. It
589 is derived from the hyperbolic filters in Wolberg’s “Digital Image Warp-
590 ing”.

591 **Status.** Satisfied.

592 DLNA (UPnP)

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

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

602 For more specific details on these topics see the UPnP AV (Universal Plug And
603 Play Audio Video) architecture [documentation](http://www.upnp.org/specs/av/UPnP-av-AVArchitecture-v1.pdf)¹.

604 Metadata indexing of media content in remote Media Servers is not required.
605 Indexing is not desirable normally, since enough metadata is normally provided
606 by the Content Directory service for browsing purposes, and local storage is

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

607 scarce. Apart the amount of storage needed could be in practice very high due
608 to the usage of remote sources.

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

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

619 **Status.** Satisfied.

620 Online Media Sources

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

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

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

630 **Status.** Satisfied.

631 Bluetooth AVRCP

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

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

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

638 **Status.** Moved to Connectivity design.

639 Playability check

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

²<http://gtk.org/>

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

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

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

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

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

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

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

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

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

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

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

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

678 **Status.** Satisfied.

679 Appendix: Media Management Technologies

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

686 Tracker

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

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

694 This makes use of several existing technologies and standards:

- 695 • **Resource Description Framework (RDF)**⁴. RDF is a directed, la-
696 beled graph data format for representing information, and is a W3C stan-
697 dard.
- 698 • **SPARQL**⁵ is a W3C standard defining a query language for databases,
699 able to retrieve and manipulate data stored in RDF format.
- 700 • **Ontologies**⁶. An ontology represents knowledge as a set of concepts
701 within a domain, and the relationships between those concepts. It can be
702 used to reason about the entities within that domain and may be used to
703 describe the domain.
- 704 • **Nepomuk**⁷ (Networked Environment for Personalized, Ontology-based
705 Management of Unified Knowledge). Nepomuk is a research project, which
706 defined a set of ontologies describing desktop entities like files, pictures,
707 etc.

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

- 712 • **Tracker Storage**. The data store and daemon to interface to Tracker’s
713 databases.

³<http://projects.gnome.org/tracker/>

⁴<http://www.w3.org/RDF/>

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

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

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

- 714 • **Tracker SPARQL**⁸. The libtracker-sparql library is the foundation for
715 Tracker querying and inserting data into the data store based on the Nepo-
716 muk ontology.
- 717 • **Tracker Miner**⁹. The libtracker-miner library is the foundation for
718 Tracker data miners. These miners will extract metadata and insert it
719 in SPARQL form into the Tracker store, following the Nepomuk ontolo-
720 gies. Developers can add new miners in order to index new data sources.
- 721 • **Tracker Extract**¹⁰. The libtracker-extract library is the foundation for
722 Tracker metadata extraction of embedded data in files. Tracker comes
723 with extractors written for the most common file types (like MP3, JPEG,
724 PNG, etc.). However, for rarer formats, it is possible to write plugins to
725 extract the metadata.

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

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

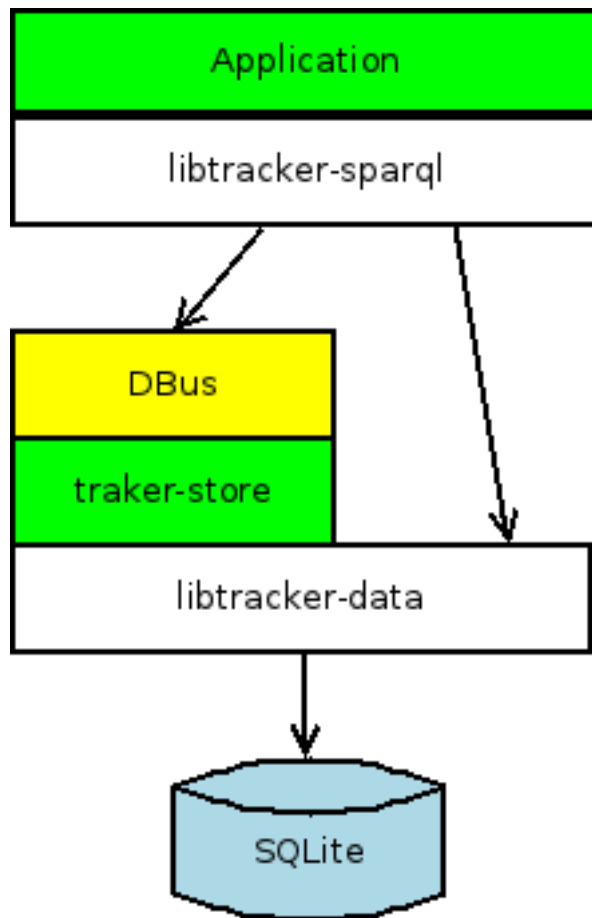
735 **Tracker Storage**

736 The Tracker storage is divided in several parts as shown in the following illus-
737 tration.

⁸<http://developer.gnome.org/libtracker-sparql/0.12/>

⁹<http://developer.gnome.org/libtracker-miner/0.12/>

¹⁰<http://developer.gnome.org/libtracker-extract/0.12/>



738

739 • The public **libtracker-sparql** is the API layer used by the applications
 740 to access the Tracker storage using SPARQL. Internally, it uses the D-Bus
 741 interface when writing access to the database is required. However, it
 742 allows a more direct access to the database for read-only access (through
 743 **libtracker-data**), which reduces the D-Bus traffic.

744 • The **Tracker store daemon (tracker-store)** provides a D-Bus interface
 745 to access the RDF storage, and it also provides also a mechanism to notify
 746 when changes happen in the RDF storage.

747 • **libtracker-data** is the library interfacing directly with SQLite database,
 748 used by both **tracker store** and **libtracker-sparql**.

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

- 751 • Nepomuk File Ontology (nfo).
- 752 • Nepomuk ID3 (nid3).

753 • Nepomuk MultiMedia (nmm).

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

755 • **SQLite¹¹ database.** The common configuration is to have separate
756 Tracker storage for each user. However, this can be set up depending
757 on the requirements of the platform, by changing environment variable
758 XDG_CACHE_HOME, as the Tracker SQLite database is stored in
759 \$XDG_CACHE_HOME/tracker. Here are some rough numbers on
760 SQLite database space usage:

761 – Empty SQLite database. The database with initialized data, but
762 without indexing files requires about 1.2 Mbytes.

763 – Indexing Photos. As an approximate figure, our measurements show
764 about 800 Kbytes of database size is used for every 500 photos (approx.
765 3 Gbytes of media). Note, the size in Gbytes is just an approximate
766 figure, since the amount of metadata scales with number of media
767 items and not with their size.

768 – Indexing Music. As an approximate figure, our measurements show
769 about 800 Kbytes of database size is used for every 300 mp3 songs
770 (3 Gbytes of media).

771 • **Write Ahead Log (WAL¹²) files.** The Tracker database is stored in
772 SQLite using WAL. The WAL option allows better performance, concu-
773 rency and reliability; at a cost of consuming extra disk space. This file is
774 part of SQLite. which is limited to 10,000 pages maximum, i.e. max of 10
775 Mbytes. Furthermore, this space used is temporal since it will get deleted
776 as soon as the the database is checkpointed, which happens automatically
777 or when the limit is reached. There is an additional relatively small file
778 for shared memory, but that is transient and it does not even use disk
779 space, just memory.

780 • **Ontologies.** The file ontologies.gvdb is stored in the same directory as
781 the SQLite files. It is about 350 Kbytes, created on initialization. The
782 size does not depend on the data indexed, but on the ontology models.

783 • **Tracker Journals.** It stores all inserts, updates and deletes. Basically
784 it is a file that grows without bound, a reason why it has received some
785 criticism. It is meant for data redundancy and backup. The journal is also
786 used to cope with ontology changes. It can be disabled at compile time.
787 In fact, it was disabled on the Nokia N9, mainly due to the ever-growing
788 problem and privacy. Tracker journal can be a reasonable choice for a
789 desktop system, but in case of embedded devices it is better disabled. It
790 is stored in the \$XDG_DATA_HOME/tracker/data directory.

¹¹<http://www.sqlite.org/>

¹²<http://www.sqlite.org/draft/wal.html>

| 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% |

791 Tracker use cases for storage utilization

792 Note: at the time of this writing, Ubuntu 12.04 was currently using SQLite 3.7.9
793 (November 2011), while the latest stable version available is 3.7.10 (January
794 2012).

795 Here are some configuration parameters for the Tracker Storage:

- 796 • **Tracker DB Journal size.** Size of the journal at rotation. By default
797 50 Mbytes.
- 798 • **Tracker DB Journal rotate destination.** Where to store the journal
799 chunk when it hits the max size.

800 Tracker Miner

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

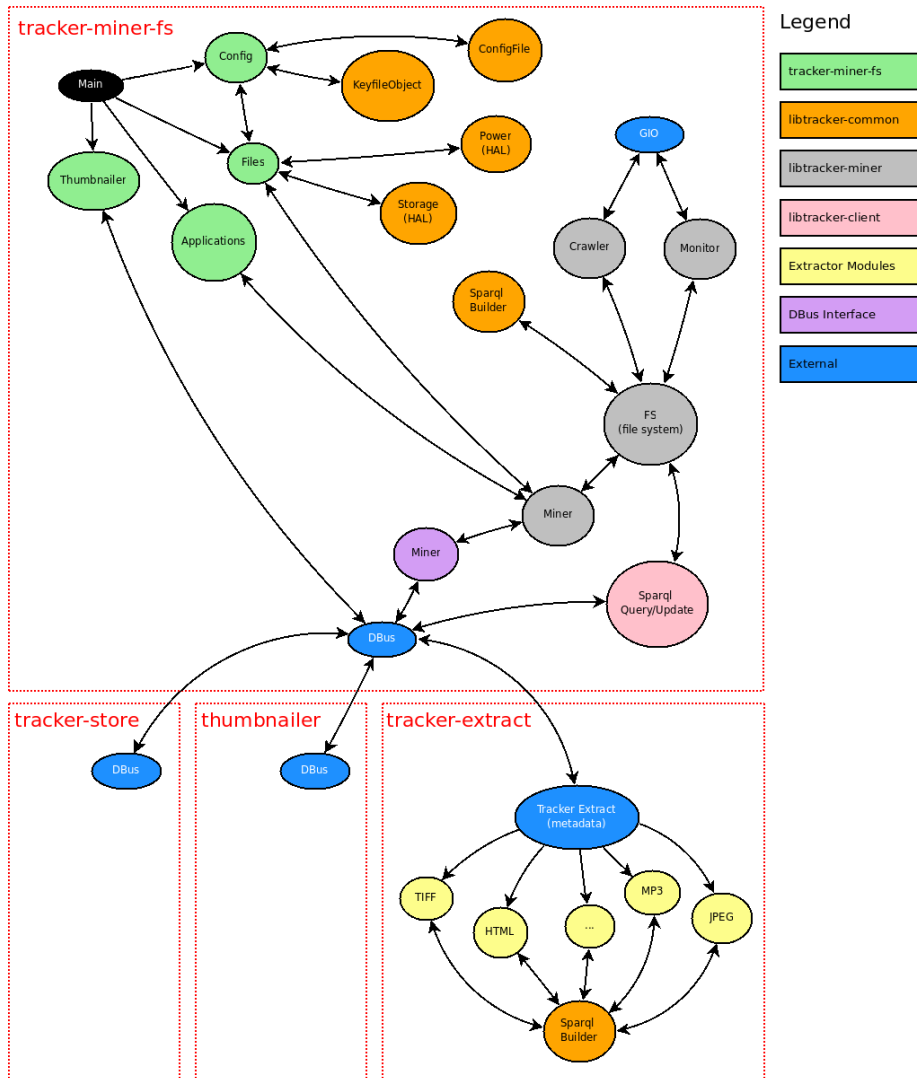
- 807 • **Startup wait time.** Primarily to avoid prevent Tracker from heavily
808 loading the system just after boot. By default 15 seconds.
- 809 • **Scheduler priority.** Specifies the priority of indexing directories and
810 files. There are three levels: when idle, first indexing on idle (default) and
811 anytime.
- 812 • **Throttle.** Controls the throttle of file indexing operations. This specifies
813 to control the overhead indexing has on the system. Of course, it is a
814 trade-off between system load and speed, but it can be tuned to make UI
815 applications more responsive. It is a value between 0 and 20, the higher
816 the slower. A value of 0 denotes “as fast as possible” (default), any other
817 number N denotes 20/N indexing operations per second. These limits can
818 of course be adjusted internally.
- 819 • **Low disk space limit.** A configurable parameter to stop indexing in
820 case of low free disk space. It is configurable between 0% (no limit) and
821 100%. It is 1% by default.

- 822 • **Crawling interval.** Specifies the interval in days to check whether the
823 filesystem is up to date with the database. A value of -1 specifies the
824 check should only be done on unclean shutdowns and -2 specifies this
825 check should be disabled entirely.
- 826 • **Removable days threshold.** Specifies the threshold in days after which
827 metadata for files from removable devices will be removed if their filesys-
828 tem is not mounted. Zero means never. Configured to 3 days by default.
- 829 • **File monitoring.** Option to track filesystem changes directly in order to
830 know what needs to be indexed.
- 831 • **File Writeback.** Option to write information back in the files, e.g. meta-
832 data retrieved from other sources or updated by the application, it can be
833 stored back in the original file. It is limited to a few formats currently.
- 834 • **Index Removable Devices.** Option to enable / disable the indexing of
835 removable devices.
- 836 • **Index Optical Discs.** Option to enable / disable the indexing of CDs,
837 DVDs, and in general any optical media.
- 838 • **List of directories to index recursively.** It can also refer to special
839 XDG directories like Desktop, Documents, Download, Music, Pictures,
840 Public, Templates and Videos.
- 841 • **List of single directories to index** (non-recursively). Same notes as
842 before.
- 843 • **List of ignored files.** Filenames can be specified with wildcards.
- 844 • **List of ignored directories.** Wildcards can be used to specify them.
- 845 • **List of ignored directories with content.** Avoid any directory con-
846 taining a file whose name is blacklisted in this list.

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

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

858 See a general overview in the following illustration.



859

860 **Tracker Extract**

861 Tracker extract does the actual metadata extraction. It inspects the media
 862 content and it extracts metadata information, which is stored in Tracker Store.
 863 There is a list of the currently [Tracker supported file formats](https://wiki.gnome.org/Projects/Tracker/SupportedFormats)¹³. It includes the
 864 main formats for all the media content types of interest (music, music playlist,
 865 video, picture, picture album and documents).

866 **Note:** in some Tracker extract plugins like the GStreamer one, the actual for-
 867 mats able to be extracted depend on the specific GStreamer plugins installed

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

868 on the system.

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

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

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

884 These are the configuration options for Tracker Extract:

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

891 **Tracker Scheduling**

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

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

904 The Tracker Filesystem Miner sets up a filesystem notifier with the directories
905 to index. The filesystem notifier is responsible for finding the directories and
906 files to index, and to monitor and notify of any changes. Tracker Filesystem
907 Miner has several priority queues; one per type of operation. Tracker Miner

908 processes items from these queues when it becomes idle. The priority of the
909 types of operations from highest to lowest is: writeback operations, deleted
910 items, created items, updated items, moved items.

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

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

921 **Thumbnail Management**

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

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

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

- 936 • Provide the ability to handle different thumbnail flavors (sizes). By default
937 two flavors exist:
 - 938 1. Normal configured by default as 128x128.
 - 939 2. Large configured by default as 256x256.
- 940 • Possibility to implement thumbnailers for closed formats or with cus-
941 tomized features.
- 942 • Complexity of a LIFO queue and setting I/O and scheduling priorities for
943 background thumbnailing is no longer the responsibility of the application
944 developer.

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

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

- 945 • Extensibility with plug-ins. This is useful to support for additional file
946 types or when different interpolation algorithms are required.

947 There are several components in the Thumbnailer service:

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

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

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

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

973 See [here](#)¹⁶, to discover the mime types they currently support
974 you need to navigate to their `provider_get_thumbnailers` implemen-
975 tation, for example `gst_thumbnailer_provider_get_thumbnailers` in
976 [http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/
977 gst-thumbnailer-provider.c](http://git.xfce.org/xfce/tumbler/plain/plugins/gst-thumbnailer/gst-thumbnailer-provider.c)

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

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

¹⁶<http://git.xfce.org/xfce/tumbler/plain/plugins/>

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

| 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 % |

987 Thumbnail storage utilization

988 Media Art Storage

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

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

999 Tracker already checks for media art present in the indexed folders. Additionally
 1000 it is able to request the downloading of album art to the album art provider
 1001 installed in the system. There is already a FOSS album art provider example
 1002 using Google Images, but it can be replaced by other implementations extracting
 1003 album art from other sources just by implementing a D-Bus service with the
 1004 interface `com.nokia.albumart.Requester`.

1005 Thumbnails of media art follow the Thumbnail Specification. The URI used
 1006 to determine the thumbnail path is the full URI pointing to the original media
 1007 art. For the path to the thumbnail refer to the Thumbnail Specification itself.
 1008 A media art fetcher is allowed to store the normal and large thumbnails imme-
 1009 diately after download of the media art is completed. A media art fetcher is,
 1010 however, not required to do this by itself (the thumbnail infrastructure will or
 1011 should take care of this if the media art is not thumbnailed yet).

1012 Grilo

1013 **Grilo**¹⁸ is a simple API for browsing and searching media content from various
 1014 sources using a single API. Applications will be able to browse and discover

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

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

1015 media content by using the Grilo API. This API will provide media content
1016 and its metadata, and GStreamer framework will be able to play video or audio
1017 content (either local or remote).

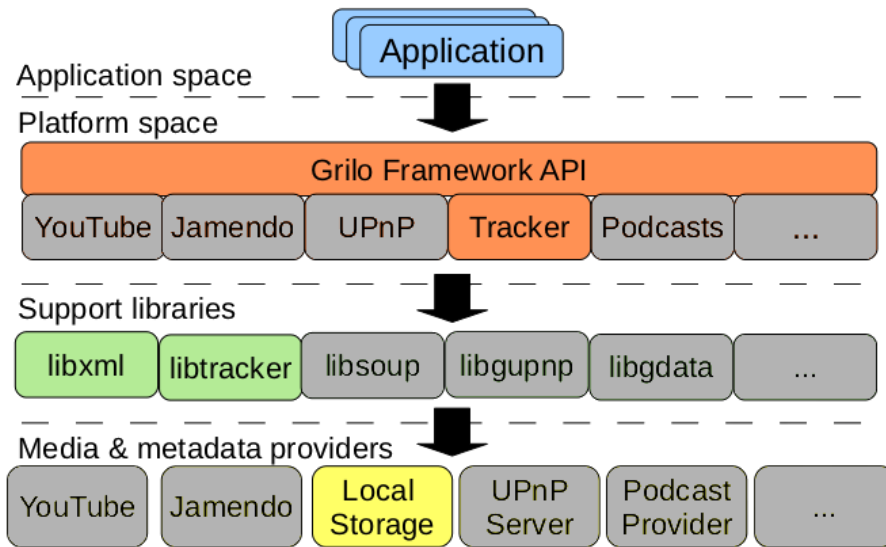
1018 A single, high-level API that abstracts the differences among various media
1019 content providers, allowing application developers to integrate content from
1020 various services and sources easily. Grilo comes with a collection of plugins for
1021 accessing various media providers, like Vimeo, Flickr, YouTube etc. so they can
1022 be presented uniformly via the Grilo API. Additionally a grilo-tracker plugin
1023 exists, which uses the Tracker service (described in past sections), to make media
1024 indexed by Tracker available through the Grilo API.

1025 There is an additional Grilo plugin for accessing the filesystem directly (grl-
1026 filesystem), which checks for media content in a set of configured directories.
1027 The defaults are the XDG user directories for pictures, music and videos.

1028 Although Grilo can be used to access many media content sources, we suggest
1029 only using it for accessing local media content. The next sections will dig into
1030 Grilo's details and its advantages. The main advantages of using Grilo instead
1031 of Tracker directly for this specific use case:

- 1032 • Tracker is a semantic data storage, which can be used to store other bits
1033 of information apart of indexing information from media content like mes-
1034 sages, calendars, etc. In other words, it is a very general framework usable
1035 for many purposes. Therefore, it makes sense to provide a higher level
1036 specialized API for media browsing (Grilo) on top of Tracker to hide its
1037 complexity from media applications.
- 1038 • Grilo has some plugins that might be useful to extract additional meta-
1039 data, e.g. album art from last.fm. Grilo is specially recommended for
1040 accessing to metadata from the Internet, which is not meant to be in-
1041 dexed. In addition, the platform could take advantage of future plug-ins
1042 which are planned to be developed by the FOSS community like lyrics,
1043 moviedb.org, etc.
- 1044 • Grilo would support using an indexer other than Tracker if a better one
1045 becomes available. More importantly, applications wouldn't have to be
1046 modified to take advantage of such a change.

1047 See the following illustration for an overview of the Grilo Architecture. Note
1048 the boxes with grey background are not going to be used in the context of the
1049 Apertis project.



1050

1051 **Grilo Media Source Plugins**

1052 The plugin must create at least one `GrlMediaSource` instance, and register it in
 1053 the Grilo registry. A `GrlMediaSource` represents a particular source of media.
 1054 These plugins provide several functions:

- 1055 • **Search** content by keywords.
- 1056 • **Browse** the media content in a hierarchical way. It is similar to exploring
 1057 a filesystem, entering into folders (`GrlMediaBox`) and browsing files in it.
- 1058 • **Query** allows access to content using service specific language. Normally
 1059 it provides additional filtering capabilities. This is used by applications to
 1060 support plugin-specific functionality.
- 1061 • **Metadata** used to request additional metadata.
- 1062 • **Store** (optional), supports to push content to the source.
- 1063 • **Remove** (optional), to remove stored contents from the source.
- 1064 • **Supported keys** provides information on which metadata keys are pro-
 1065 vided by the plugin. Typical metadata keys are: id, title, url, thumbnail,
 1066 mime, artist, duration.
- 1067 • **Slow keys** (optional) provides info on which metadata keys are expen-
 1068 sive to gather. So the applications could just ask for non-expensive ones
 1069 normally, and only require the slow keys when details are required for a
 1070 particular media content.
- 1071 • **Media from URI**. Gets `GrlMedia` from a URI. For example a file browser
 1072 may use this to get metadata for a specific file.

1073 • **Test Media from URI** (optional). To check if the plugin can convert a
1074 URI into a GrlMedia object.

1075 • **Notifications** on changes on media content.

1076 At least one of the content retrieval methods is expected to be implemented:
1077 search, browse or query. Each media content result of the search/browse/query
1078 is represented by a GrlMedia object.

1079 Plugins should be implemented in a non-blocking way to have a smooth user
1080 experience in applications. Also threads are not recommended; splitting work
1081 into chunks using the idle loop is encouraged.

1082 There is a standard set of metadata keys defined, but plugins can define their
1083 own custom metadata keys.

1084 A GrlMedia can have multi-valued properties; for example a YouTube video with
1085 different resolutions (and thus, different URIs). It is also possible to associate
1086 different properties with each URI of a GrlMedia.

1087 **Grilo Metadata plugins**

1088 Grilo metadata source plugins do not provide access to media content, but
1089 additional metadata information. An example would be to provide thumbnail
1090 information for local audio content from an online service.

1091 This plugin must create at least one GrlMetadataSource instance, and register
1092 it in the Grilo registry. The plugin provides several functions:

1093 • **Resolve** retrieves additional information for a GrlMedia object.

1094 • **May resolve:** to check if Resolve may be performed with existing infor-
1095 mation.

1096 • **Set metadata** (optional): set the play count or the last time a media
1097 was played.

1098 • **Writable keys** (optional): reports which keys can be stored.

1099 • **Supported keys:** provides information on which metadata keys are pro-
1100 vided by the plugin.

1101 • **Slow keys** (optional): provides info on which metadata keys are expensive
1102 to gather. So the applications can ask for inexpensive keys normally, and
1103 only request the slow keys when details are required for a particular media
1104 content.

1105 • **Cancel operations:** cancels ongoing operations.

1106 Google Data Protocol

1107 **YouTube**, as well as other Google services like Picasa, use the **Google Data**
1108 **Protocol**¹⁹. The Google Data Protocol is a REST-inspired technology for
1109 reading, writing, and modifying information on the web. The protocol currently
1110 supports two primary modes of access: AtomPub and JSON. The JSON is a
1111 mapping of Atom items to JSON objects meant to be used for web applications
1112 written in JavaScript.

1113 The AtomPub mode is based on the Atom Publishing protocol, with names-
1114 paced **XML** additions. Communication between the client and server is broadly
1115 achieved through **HTTP** requests with query parameters, and Atom feeds being
1116 returned with result entries. Each *service* has its own namespaced additions to
1117 the GData protocol; for example, the Google Calendar's API has specializations
1118 for addresses and time periods.

1119 Collabora proposes **libgdata**²⁰, which is a library to allow access to web ser-
1120 vices using the Google Data Protocol from traditional applications. Results are
1121 always returned in the form of result *feeds*, containing multiple *entries*. How the
1122 entries are interpreted depends on what was queried from the service, but when
1123 using libgdata, this is all taken care of transparently. The main dependencies
1124 of libgdata are libsoup, libxml and liboauth.

1125 Other frameworks and applications are already using libgdata with success, e.g.
1126 evolution-data-server, Totem's YouTube plugin and Grilo's YouTube plugin.

1127 The library libgdata already provides an implementation for the **GDataY-**
1128 **ouTubeService**²¹, which provides the following functionality:

- 1129 • Query videos.
- 1130 • Query videos related to a specific video.
- 1131 • Query standard feed types: top rated, top favorites, most viewed, most
1132 popular, most recent, most discussed, most linked, most responded, re-
1133 cently featured and watch on mobile.
- 1134 • Upload a video.
- 1135 • Get categories.

1136 Librest and libsoup

1137 It is difficult to find libraries to access online media sources if they are not pro-
1138 vided by the vendors themselves. However, most of these online media sources
1139 are based on HTTP protocol with **REST**²² interfaces. Therefore, in general,

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

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

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

²²http://en.wikipedia.org/wiki/Representational_state_transfer

1140 [librest] and/or [libsoup](#)²³ will be useful. **Librest** is a library designed to
1141 make it easier to access web services that are designed in a “RESTful” manner.
1142 **Libsoup** is an HTTP client/server library for GNOME. It uses GObject and
1143 the glib main loop, to integrate well with GNOME applications. Collabora can
1144 suggest or provide advise for open-source ways for accessing these services on
1145 request. This is the most effective way to access all the features.

1146 **Playlists support**

1147 Playlists are supported in Tracker. There is an specific Tracker Extract plugins
1148 to handle playlists, which is using internally the [Totem Playlist Parser](#)²⁴ li-
1149 brary, which is conveniently abstracted and independent of Totem. Tracker
1150 Extract introduces the metadata retrieved in Tracker Store using the class
1151 nmm:Playlist, which is a subclass of nfo:MediaList. The entries in the playlist
1152 are introduced as nfo:MediaFileListEntry.

1153 The supported playlist formats in Totem Playlist Parser are: audio/x-
1154 mpegurl, totem-plparser, audio/x-scpls, audio/x-pn-realaudio, application/ram,
1155 application/vnd.ms-wpl, application/smil and audio/x-ms-asx.

1156 Grilo does not support playlists in the latest stable version available, so this
1157 feature would need to be added as specified in the requirements section.

1158 **Appendix: Questions & Answers**

1159 These chapter contains very specific questions that have been asked during work-
1160 shops.

1161 **Q: Will asking for a specific prioritization during metadata extraction
1162 increase the load by running multiple indexing jobs ?**

1163 A: No, the Tracker scheduler will manage all metadata indexing operations in
1164 internal queues, so prioritization will just change the sorting of the metadata
1165 indexing operations, but not the overall system load. Note the scheduling system
1166 proposed in this document is not implemented in Tracker yet. See [Indexing
1167 scheduling](#) and [Tracker scheduling](#) for more details on prioritization and Tracker
1168 scheduling.

1169 **Q: How does the system know when to renew thumbnails ?**

1170 A: When a thumbnail is generated, some properties are stored inside it like the
1171 original URI and the modification time of the original file. If the original file is
1172 modified at some point, its modification time will get changed automatically by
1173 the Linux filesystem. So, it is possible to know when a thumbnail is outdated.
1174 Additionally, Tracker is monitoring the filesystem for changes. In case a file is

²³<http://developer.gnome.org/libsoup/>

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

1175 modified, added, moved or deleted its thumbnail will be automatically updated.
1176 Note: this feature is not fully implemented yet, but it is part of the modifications
1177 Collabora will implement.

1178 **Q: How the mime type of the files is determined ?**

1179 A: This is done through glib, which finds out the mime type in a efficient way and
1180 it is used extensively by all GNOME based software. The details of the algorithm
1181 used can be seen in the [Shared MIME Info Specification](#)²⁵, it has been designed
1182 to be robust and efficient. The first thing done is to test the filename extension
1183 to see if it is a recognized type. If this operation cannot be done or the result
1184 is uncertain, a second check will be done using the first bytes of the file check-
1185 ing for the signature of known files. For more details see `g_file_query_info`,
1186 `G_FILE_ATTRIBUTE_STANDARD_CONTENT_TYPE` and `g_file_info_get_content_type`
1187 in GNOME documentation.

1188 **Q: How the video thumbnailing works to avoid black video frames or**
1189 **uninteresting frames in general ?**

1190 A: From [Thumbnail management](#): “Video thumbnails can be generated using
1191 the GStreamer thumbnailing plugin. This plugin already provides an heuristic
1192 method to extract the thumbnail from a video stream, by selecting a frame
1193 with a wide distribution of colors (to avoid presenting a title screen or other
1194 essentially-blank frame). Other ways could be implemented if required, just by
1195 implementing a thumbnail plugin.

1196 **Q: How document thumbnailing works to avoid thumbnails of blank**
1197 **pages ?**

1198 A: The existing Tumbler plugins used to extract thumbnails from Open/LibreOffice,
1199 PDF and Microsoft Office documents gets the thumbnail stored inside the
1200 file. It is responsibility of the office applications to write a proper thumbnail.
1201 Typically it is just the thumbnail of the first page of the document, which
1202 usually is the best option since the first page contains the title in bigger font
1203 sizes, cover of the document and logos. Any other approach is debatable,
1204 so Collabora does not recommend to make thumbnails from only text pages
1205 since they are less likely to be useful, thumbnailing normal text would become
1206 unreadable.

1207 **Q: How the applications can store and retrieve the last time a media**
1208 **file was played ?**

1209 A: This functionality can be provided by the Grilo metadata store plugin.
1210 The application must query the last values and set new values through
1211 Grilo API. The media file is identified via the file URI. The metadata

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

1212 store plugin stores these values in a Tracker database. It currently sup-
1213 ports the following values: last position where media item was played
1214 (GRL_METADATA_KEY_LAST_POSITION), number of times a media
1215 item has been played (GRL_METADATA_KEY_PLAY_COUNT) and last
1216 date a media item was played (GRL_METADATA_KEY_LAST_PLAYED).
1217 Grilo is making use of the properties already defined on the Tracker ontologies
1218 like nfo:lastPlayedPosition, nie:usageCounter and nie:contentAccessed. A ben-
1219 efit of using Grilo is that Tracker details are not exposed to the applications,
1220 for example alternatively Grilo has another plugin to store these fields in a
1221 separate SQLite database in case Tracker was not used, but the API to set and
1222 get these properties would be the same.

1223 **Q: How a thumbnail is retrieved ?**

1224 A: Thumbnails can be retrieved through different ways depending on what
1225 specific APIs the application is using. The best way for media applica-
1226 tions would be through the Grilo API, see `grl_media_get_thumbnail` and
1227 `grl_media_get_thumbnail_binary_nth` (in case several thumbnails are avail-
1228 able for a media item). Grilo API is internally using glib library to retrieve this
1229 through `g_file_query_info`, `G_FILE_ATTRIBUTE_THUMBNAIL_PATH`
1230 and `g_file_info_get_attribute_byte_string`. Grilo API will need to be
1231 modified in case more thumbnails need to be stored on the USB flash devices.

1232 **Q: How the system behaves on robustness on power loss ?**

1233 A: This and other questions on system robustness will be answered on a separate
1234 document focused on system robustness. Anyway, please see chapter [Indexing](#)
1235 [database on removable device](#) for an advance of some issues regarding USB flash
1236 devices.

1237 **Q: How a media file from a USB Flash device is identified ?**

1238 A: It is identified by its complete URI, e.g. `/media/D8C0-024E/Joaquin`
1239 `Sabina/Joaquin Sabina & Fito Paez - Llave sobre mojado.mp3`". In some
1240 systems, USB flash devices are mounted on a directory with a hex identifier
1241 (depending on system configuration). This identifier is the UUID (Universally
1242 Unique Identifiers), not the label of the USB flash device. It is generated when
1243 the filesystem is created, and it is very. Generally it is a 128 bit identifier, but
1244 some filesystems like VFAT have smaller resolution (32 bits).

1245 **Q: Is it configurable the timeout for Tracker extract operations ?**

1246 A: No, they are not currently, but it would be simple to make them configurable
1247 for example through GSettings. There are two timeouts. A watchdog timeout
1248 which checks that the tracker extract process does not hang during metadata
1249 extraction (by default set to 20 seconds). There is an additional idle timeout,

1250 which stops a tracker extract process if it has been idle for some time (30 seconds
1251 by default).

1252 **Q: Does Tracker retry in case Tracker Extract fails due to the watch-**
1253 **dog timer ?**

1254 A: By default, Tracker retries up to two times if a tracker extract process fails.
1255 It will also retry in case the file is modified or the USB flash where it is located
1256 is reinserted.

1257 **Q: Does Tracker store marks for the corrupted files ?**

1258 A: Currently, there is no property to identify corrupted files in Tracker. A file
1259 whose extract process has failed due to corruption in the file, it would just have
1260 properties from the nfo ontology (nepomuk file object), but it would not have
1261 properties from other subclasses like nmm (nepomuk multimedia).

1262 **Q: There are reports of performance of page queries on Tracker**
1263 **databases is negatively affected by the number of rows in the**
1264 **database. Collabora to double check.**

1265 A: Some tests running SPARQL queries have been done with databases near
1266 6000 items and the mentioned problem was not reproducible (no performance
1267 problems found). Please provide data set and application code reproducing this
1268 problem for further investigation.

1269 **Q: Should Tracker be used for Radio Stations information ?**

1270 A: Tracker has already ontologies to store radio station information. So, it
1271 would be possible to use it to store and retrieve the user favorite radio stations.
1272 However, the interface to access and update this information would be through
1273 plain SPARQL, which has a steep learning curve for developers. Additionally,
1274 the radio station information is not shared with other applications. The only
1275 advantage of using Tracker would be that the global search would automatically
1276 work for radio station information, so it would not be necessary to implement
1277 an extra global search plugin to look for this info in another database. The final
1278 decision must take into consideration how well the existing ontology for radio
1279 stations ([nmm:RadioStation](#)²⁶) is suited to Apertis' roadmap.

1280 **Q: What happens when a USB flash device is inserted in a USB port**
1281 **?**

1282 A: When the user inserts an USB flash device, there are three main components
1283 participating in the action:

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

- 1284 • **Linux kernel** (including device drivers). The kernel will be able to com-
1285 municate with the device as soon as it is powered up, initialized and
1286 announced through the USB Bus.
- 1287 • **Udev**²⁷ is the device manager for the Linux kernel. Primarily, it manages
1288 device nodes in /dev. It is the successor of devfs and hotplug, which
1289 means that it handles the /dev directory and all user space actions when
1290 adding/removing devices, including firmware load. The Udev daemon
1291 listens to the netlink socket used by the kernel to communicate with user
1292 space applications. The kernel will send a bunch of data through the
1293 netlink socket when a device is added to, or removed from a system. The
1294 Udev daemon catches all this data and will do the rest, i.e., device node
1295 creation, module loading etc.
- 1296 • **UDisks**²⁸ (formerly known as DeviceKit-disks) lies on top of udev, and it
1297 is an abstraction for enumerating disk and storage devices and performing
1298 operations on them. It is a replacement for part of the functionality
1299 which used to be provided by the now deprecated HAL (Hardware Abstrac-
1300 tion Layer). UDisks is a user daemon with D-Bus interface which gets
1301 notifications from udev.

1302 See the following table for an idea of what happens when a USB flash device
1303 is inserted. The table provides a general idea about the timings for different
1304 operations in the system. Note, although the timings are based on real mea-
1305 sures, are not guaranteed since all the software components have not been
1306 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 round |
| 46 | 40 | (5b) Tracker Extract finishes gathering metadata for all files in the USB flash device |

1307 **Q: How does the monitoring of filesystem changes work in Tracker ?**

1308

1309 A: The monitoring of changes in files and directories of the filesystem is handled
1310 internally by Tracker Miner via the [GFileMonitor](http://www.gnome.org/gio/unstable/GFileMonitor.html)²⁹ API. Note GFileMonitor

²⁷http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf

²⁸<http://www.freedesktop.org/wiki/Software/udisks>

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

1311 is just an abstraction in glib, which abstracts the file monitoring functional-
1312 ity, since there are several backends available implementing such functionality
1313 depending on the specific operating system. Note, this mechanism is a very
1314 efficient way to get notified about changes on the filesystem, since it is directly
1315 provided by the kernel, instead of doing active polling. Linux uses the **inotify**
1316 backend. For a more detailed view of the inotify API see the tutorial “*Monitor*
1317 *filesystem activity with inotify*³⁰”.

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