



Global search

1 Contents

2	Information Classification	2
3	Information Sources	2
4	Content Categories	2
5	Content Flags	3
6	Auxiliary Information	4
7	Search Priority	5
8	Speech Recognition	5
9	Guidelines	6
10	Decentralized Indexing	6
11	Extendable Via Plug-ins	6
12	Easy for Application Developers	7
13	Highly Responsive	8
14	Limited System Impact	9
15	Predictable Interaction	9
16	Balance of Configuration and Heuristics	9
17	Potential Search Back-ends	10
18	Primary Sources	10
19	Auxiliary Sources	12
20	The SDK Persistence API	12
21	Example Search Flow	13
22	Implementation Examples	15
23	Applications	16
24	Preferences	16
25	Documents	16
26	Media	16
27	Contacts	16
28	Events	17
29	Communications	17
30	Definitions	17
31	Locations	18
32	Using Existing Global Search Software	18
33	New Software Development	19

34 Apertis will store several types of information – media files, documents, contacts,
35 e-mails, applications and their preferences, chat logs, and more. Much of this
36 content will be stored with the application that generates or consumes it. A file
37 manager would be very cumbersome for finding content in all these locations,
38 and some of these data types are not strictly files. A powerful search system
39 needs to be implemented to facilitate convenient access to the user’s data.

40 Not all interesting information is locally stored. Apertis may be equipped with
41 an internet connection and the user may want their search to include videos on
42 YouTube or text from Wikipedia.

43 If a GPS device is present, search results could potentially include nearby points
44 of interest like gas stations, coffee shops and museums.

45 Compiling and displaying search results from these varied sources is only a
46 partial solution. The interface should also allow interaction with the content –
47 by launching an appropriate handling application.

48 The goal of this document is to define global search in the context of Apertis
49 and establish guidelines for implementing an effective global search system.

50 **Information Classification**

51 **Information Sources**

52 There are two types of information sources available to Apertis for searching.

- 53 • Primary – Sources that are used in the generation of search results.
- 54 • Auxiliary – meta-information sources for providing further detail about a
55 primary search result.

56 The source types can be further broken down into three storage locations:

- 57 • Internal – data stored directly in the embedded Apertis device.
- 58 • External – data stored on a removable device. External devices can be
59 removed and have their data altered elsewhere, so care must be taken
60 when caching results.
- 61 • Remote – data available from the internet. Availability depends on
62 whether the Apertis device has network access, user preferences governing
63 network use, and the status of the remote service.

64 **Content Categories**

65 The results returned from primary sources may be divided into broad categories:

- 66 • Applications – Installed applications, applications in the currently running
67 application stack, and perhaps software available from the application
68 store.
- 69 • Preferences – Application or global UI settings.
- 70 • Documents – Spreadsheets, presentations and word processor files. Web
71 pages, including the web browser’s bookmarks, would also fall in this
72 category.
- 73 • Media – Photos, videos and music. This could also include radio stations,
74 both broadcast and internet.
- 75 • Contacts – E-mail, phone and chat contacts.
- 76 • Events – Important dates from a calendar application or social media sites.
- 77 • Communications - Emails, SMS and conversation logs from chat services.
- 78 • Definitions – Dictionary entries and Wikipedia articles.

- 79 • Locations – Points of interest from the navigation software and the current
80 location.

81 Applications should provide a list of categories that apply to the content they
82 handle to allow the search framework to make intelligent decisions regarding the
83 scope of a search.

84 It is likely that some applications will want to extend the available set of cat-
85 egories by providing new categories in their manifests. Collabora recommends
86 that developers wanting to add a new category are required to be approved by
87 the application store.

88 Allowing developers to specify their own content categories would reduce the
89 search front-end’s ability to combine and prioritize similar results if applications
90 chose different category names that mean the same thing. The application store
91 would be able to approve or deny any request for a new category, and suggest
92 re-use of an existing category if appropriate.

93 Even the list above isn’t completely orthogonal – definitions could be a subset
94 of documents. Special cases like this should only be considered if it’s deemed
95 that a clear benefit arises from the separation.

96 **Content Flags**

97 Search results can contain additional Boolean properties that may apply to all
98 categories. Collabora recommends a collection of flags to further qualify search
99 results in order to allow better sorting and presentation:

- 100 • Favorite – content with this tag has been selected by the user as high
101 priority – favorite radio stations, contacts, e-mail threads.
- 102 • Online – Activating some search results – such as browser bookmarks
103 – would require a data connection.
- 104 • Fee – The result leads to a service with a fee for usage. Examples could
105 include long distance phone calls, or application store software.

106 As with content categories, it may be useful to allow applications to specify new
107 flags in their manifests. The same concerns apply here as for categories, and
108 the application store should carefully consider which new flags are allowed.

109 **Auxiliary Information**

110 In many cases, auxiliary data can be added to the search results either to provide
111 useful information to the user, or to assist the search manager in prioritizing
112 results more effectively:

- 113 • Frequency/recency of usage is useful for prioritizing search results.
- 114 • Presence information can be provided for contacts in search results.

- 115 • Thumbnails can be generated for local media.
- 116 • Weather can be provided for locations (with the current location either
117 settable as a preference, or taken from a GPS device)
- 118 • Distance from current location can be determined for locations – linear
119 distance can be determined quickly, but a driving distance would take
120 significantly longer.
- 121 • More advanced auxiliary information providers could look up movie rat-
122 ings and reviews from online services.

123 In some cases, such as presence information for contacts, the auxiliary infor-
124 mation is provided by the same library (libfolks) and at the same time as the
125 primary results. In other cases, the search manager may need to query auxiliary
126 data sources as an additional step.

127 Unlike flags and categories, auxiliary information can't be extended by applica-
128 tion manifests, since it must be fully understood by the search framework to be
129 displayed or utilized for priority calculations.

130 It is possible that a system will have multiple sources for the same auxiliary in-
131 formation – perhaps a freshly installed system uses Google for querying weather
132 information. If a user then installs a third-party weather application, it may be
133 capable of providing more accurate forecasts.

134 The Google Weather API actually ceased to exist in August of 2012
135 and is mentioned only for illustrative purposes.

136 Resolving which provider to use in situations like these may be difficult. Some
137 possible resolution methods would be:

- 138 • If an application is present on the user's home screen it will be selected.
- 139 • Most recently installed applications will be selected.
- 140 • The HMI could provide an interface for selecting the preferred provider.

141 While HMI intervention is not a preferred option, it may not always be possible
142 to infer the user's preference without assistance.

143 **Search Priority**

144 Not all information is of equal importance, and if a search has too large a number
145 of matches to display, the higher priority matches should come first. Since there
146 are many primary sources with differing response times, the results must be
147 prioritized or the fastest responders will dominate the results list.

148 Having a few different priority levels to assign the different categories to should
149 be sufficient:

- 150 • Top – Contacts and recently or frequently used items of all categories.

- 151 • High – Media, Documents and nearby locations.
- 152 • Medium – Applications and application settings.
- 153 • Low – E-mails, chat logs and SMS contents.
- 154 • Bottom – Pay-for-use services.

155 Within priority levels, information can be sorted with auxiliary information.
156 For locations, distance from current location could be a reasonable sort criteria.
157 For applications, the most recently used applications should likely be higher up
158 the list.

159 **Speech Recognition**

160 Hands free operation is a necessity in an automotive user interface, and the
161 global search interface needs to be implemented with that as a primary goal.
162 Entering arbitrary words, and having the search framework update a list of
163 results while a request is being entered isn't possible with speech recognition.

164 The search framework needs to be designed to be accessed comfortably in two
165 different input modalities. By providing two search methods – a full search,
166 and a simplified keyword search, the same powerful search mechanisms can be
167 accessed easily by either voice or entered text.

168 The use of keywords for initiating and filtering searches will simplify verbal
169 interaction with the system and provide a fast and efficient interface. Category
170 names could also be recognized, allowing a quick interface to recently used items.

171 Applications should provide a list of keywords in their manifests to indicate the
172 set of keywords they may return in their search results. Allowing applications to
173 add new keywords from their manifests is likely less problematic for the search
174 interface than new categories or flags, and as such needs little or no application
175 store review. However, localization of category names and keywords is critical,
176 since Apertis may be deployed in multiple languages.

177 It may be worthwhile to hard code some response logic, such as “weather”
178 launching the preferred weather application, or having a short phrase like
179 “switch to <name of local radio station>” control the radio.

180 It would be simpler to do this than to try to fine tune the search system's
181 heuristics to cause this to naturally occur, and would prevent installation of a
182 new application (which might share keywords with installed applications) from
183 changing expected behavior.

184 **Guidelines**

185 Collabora feels the following features will help create a responsive, flexible and
186 convenient global search interface.

187 **Decentralized Indexing**

188 Trying to store all these different types of data in a single central repository for
189 searching presents some difficult problems:

- 190 • If the on-disk format of the search database changes, a lengthy re-indexing
191 of all searchable content must take place.
- 192 • Remote content has dramatically different requirements than local con-
193 tent, and may change or disappear.
- 194 • If an application's data is already in a conveniently searchable form, stor-
195 ing a second copy of it in a database wastes storage space, cache memory,
196 processing time, and, potentially, decreases user interface responsiveness.

197 Apertis has special considerations as well – the application rollback system also
198 governs the settings and data associated with an application. If a rollback is
199 performed, data in a central database would have to be purged and re-created.

200 Separating the search front-end from the database and allowing it to query multi-
201 ple sources for results will allow the use of many different available components,
202 allow searching remote content that can't be indexed, and allow for search back-
203 ends with different search strategies and response times to be compiled into a
204 single result list.

205 **Extendable Via Plug-ins**

206 Many desktop search applications aggregate data from several back-ends to
207 produce their search results. Each source has a plug-in specifically written to
208 process a certain kind of data and return standardized search results.

209 [Using existing global search software](#) provides details on some exist-
210 ing global search solutions.

211 Allowing applications to be responsible for providing search results on their own
212 data enables them to provide more appropriate results than if a general purpose
213 service naively indexed everything on the system.

214 Applications would be able to provide their own plug-in, which may commu-
215 nicate with an application agent, to create a custom search back end for the
216 application's content.

217 Agents are described in the `Software agents in Apertis` document

218 Further, application search databases can be stored with the rest of the appli-
219 cation data in a way that allows application rollback to govern them as well, so
220 in the event of an application rollback search results will still be consistent with
221 the data and no lengthy re-indexing process will be required.

222 Some back-end plug-ins may be capable of prioritizing their results. These
223 priorities should be normalized for fair comparison across plug-ins, and then
224 used by the front end to sort results within priority levels.

225 **Easy for Application Developers**

226 Many applications will work with data that should be exposed via the search
227 interface, but if integrating an application with global search is difficult then
228 developers may do it poorly or not do it at all.

229 For applications using the Apertis persistence framework to store data, it may be
230 possible to have a single search plug-in that can mine the persistence framework
231 to produce results for multiple applications.

232 Since the applications are responsible for the structure of their data in the
233 persistence framework, it's difficult for a generic plug-in to guess what data
234 should be searchable. Applications may store sensitive information, such as
235 passwords, in the framework as well.

236 Another difficult problem is that the plug-in should be able to track which
237 results were selected in order to increase their priority in future searches, but
238 this is difficult to maintain separately from the searchable data.

239 The following criteria simplify the implementation of a generic plug-in for mining
240 the persistence framework:

- 241 • The persistence framework allows applications to create special tables for
242 searchable data.
- 243 • Only the contents of these tables are searchable.
- 244 • The format for searchable data is dictated by the persistence framework
245 and contains extra fields for use by the plug-in for gathering statistics.
- 246 • The application manifest indicates whether the plug-in can search an ap-
247 plication's data – even if the application uses the searchable data format,
248 it may still provide its own search plug-in, and not wish to have its results
249 duplicated by the generic plug-in.

250 In addition to allowing applications to intentionally expose data to the search
251 framework, if the SDK provides functionality for an application to maintain a
252 list of recently used items in a standard way, a generic plug-in could use that
253 information to provide search results.

254 Activation of these search results must invoke the application in a way that the
255 appropriate data is immediately displayed. The application manager and the
256 application will have to negotiate this launch.

257 Searching the persistence API's storage is covered further in [The SDK persis-
258 tence API](#).

259 **Highly Responsive**

260 Users will expect new search results to be presented as they type, with the result
261 list becoming more refined the more text they enter. It is important that the
262 text entry always feel responsive, even if the results are slightly delayed.

263 Search results may take a noticeable amount of time to accumulate. Local
264 results should arrive quickly, but remote results could take seconds. Waiting
265 for all results to be available before presenting any to the user would result in
266 a disappointing experience.

267 In order to avoid penalizing fast responders to wait for the slowest plug-ins to
268 finish their queries, search results should be displayed to the user promptly as
269 they become available.

270 Asynchronous coupling between primary and auxiliary is also important. If a
271 search returns a contact, the user may intend to send an email or place a call to
272 that contact immediately – waiting for online status before showing that search
273 result at all might give the impression that the search system is slow.

274 An indication that search results are still being accumulated should be presented
275 to the user, as slow responding back-ends may take a significant amount of time
276 to finish, and a user may choose to wait for more search results if they know
277 more may become available.

278 It may be preferable to delay querying slow, online, or resource heavy search
279 result providers until the user signifies the end of text interaction in some way.
280 A quickly accumulated subset of potential search results could be displayed
281 during text entry with a full search only conducted if they hit “enter” instead
282 of selecting a result.

283 This would prevent sending off a large number of resource intensive requests for
284 every entered character during the time when they’re likely to be immediately
285 invalidated by more input.

286 **Limited System Impact**

287 If the search framework immediately responded to a search request by sending
288 requests to all available plug-ins concurrently, the resulting spike in I/O and
289 memory consumption would likely have detrimental effects on system interac-
290 tivity. If the search results in significant storage device access, useful data will
291 be pushed from system caches resulting in a generally sluggish system for a
292 while after a search takes place.

293 Efforts should be made to do the minimal amount of searching possible to satisfy
294 the user’s request. Since applications are required to specify in their manifests
295 what categories and keywords apply to their data, a keyword based search only
296 needs to access a subset of search plug-ins.

297 Starting with a “shallow” search and allowing a progressively deeper search
298 (perhaps by touching a “more results” button, or by speaking the word “more”)
299 will allow the search manager to query high priority plug-ins first, and only
300 query lower priority plug-ins if the user is dissatisfied with the search results.

301 The initial search will prefer plug-ins for applications on the home screen and
302 applications that are already running, as well as higher priority search content,

303 with subsequently “deeper” searches progressing to lower priority levels.

304 As the user performs searches and the system accumulates more information
305 on what plug-ins are most likely to provide the results they choose, the “more
306 results” function will be used less and less frequently.

307 **Predictable Interaction**

308 Rapid changes in already visible search results could result in the user selecting
309 an unintended item. Care should be taken to minimize movement of search
310 results after display.

311 Results should be displayed in sorted order, not displayed and then sorted. As
312 new items are added they may change the position of existing items – new high
313 priority results will push lower priority results down the list.

314 Aggressive timeouts may need to be set for online sources to help mitigate this.
315 Search results from online sources could be given a shared timeout, at which
316 point the results will be ordered and injected into the displayed list all at once.

317 If the result list can be navigated with up/down buttons or a similar physical
318 interface then the selection should stay with the currently selected item if new
319 results appear. If the selection stays with the ordinal position in the list, then
320 an unintentional activation is much more likely to occur.

321 **Balance of Configuration and Heuristics**

322 Exposing preferences to control all aspects of the search process will almost
323 certainly confuse more users than it will help. Trying to represent all the possible
324 combinations of flags to the user in a sensible way will likely not be possible.
325 The ability to turn individual search sources on and off is probably useful, and
326 this is the way search configuration is presented on some operating systems
327 (OSX, Android).

328 If the interface is too configurable it makes testing new search heuristics more
329 difficult, as they need to be tested for interactions with all possible combinations
330 of the available settings. Giving the user control over what is searched, but not
331 how it’s presented, should allow some user customization while maintaining
332 consistency for developers.

333 The system should track a user’s search history and use that information to
334 change the priority levels of content categories, and the effect of content flags.
335 This will allow the system to adapt to a user’s preferences over time. Since
336 applications can add new content categories, flags, and keywords this will also
337 allow these new types to eventually find the priority level that matches the users
338 interest In them.

339 Some system settings should affect the search system. If Apertis is equipped
340 with a wireless modem, the search system should obey the system settings for
341 wireless data usage. It might be useful to allow finer grained control over remote

342 searching. Back-ends that require network traffic to perform a search could be
343 presented as a single result (like: “Search Wikipedia for: ...”). Activating that
344 result would perform the remote search and replace the single line with the new
345 results as they become available.

346 **Potential Search Back-ends**

347 A significant body of search software already exists and would be appropriate to
348 integrate into a global search framework; some convenient libraries and protocols
349 exist for quickly creating new search back-ends.

350 The following sections provide an overview of some potential primary and aux-
351 iliary sources. For some of them indexing services are already available, others
352 don't yet have a free implementation or are Apertis specific.

353 **New software development** later in this document is intended to
354 give an overview of what suggested components would require new
355 software development.

356 **Primary Sources**

357 The following software solutions bear strong consideration for inclusion as pri-
358 mary search backends:

- 359 • [Zeitgeist](#)¹ - An activity logger that tracks frequently used content as well
360 as chat logs. While it's possible for individual apps to track recently used
361 data, Zeitgeist can track this data on a whole system level.
- 362 • [Evolution-data-server](#)² - A component allowing access to calendar, tasks,
363 and address book information.
- 364 • [Folks](#)³ - A “meta-contact” aggregator that can return information for con-
365 tacts across a wide array of services (including Evolution-data-server's
366 contact information).
- 367 • [Grilo](#)⁴ - A framework for browsing remote media.

368 New search backends could readily be built from:

- 369 • [OpenSearch](#)⁵ - A standard for internet based searching implemented by
370 many existing searchable pages – Wikipedia, Google, Bing, and IMDb to
371 name a few.
- 372 • [Lucene++](#)⁶ - A generic text search engine that can be used in applications
373 that want to implement their own search back-ends.

¹<https://zeitgeist.freedesktop.org/>

²https://wiki.gnome.org/Apps/Evolution/EDS_Architecture

³<https://live.gnome.org/Folks>

⁴<https://live.gnome.org/Grilo>

⁵<http://www.opensearch.org/>

⁶<https://github.com/luceneplusplus/LucenePlusPlus>

374 Some Apertis specific systems are good candidates for delivering search results:

- 375 • Application Manager – The application manager could provide search for
376 installed applications, and perhaps even allow searching running applica-
377 tions to allow a quick jump to recently used applications on the application
378 stack.
- 379 • Preference Manager – The preference manager has access to all application
380 and global UI settings, and could provide these settings to the search
381 framework.
- 382 • Browser – The browser application’s bookmark list should be exposed by the
383 search infrastructure.

384 There may be times when more than one primary search source returns the
385 same result - the Zeitgeist activity logger, for instance, tracks recently used
386 content. Recently played media may be returned as a search result from both
387 Zeitgeist and a media indexing service. When such a collision occurs, the two
388 results should be combined (before consulting auxiliary sources) and displayed
389 as a single search result.

390 Some care will need to be taken in selecting how the plug-ins query results.
391 For example, the application and preference managers could be queried over
392 D-bus since they’re likely to be long running services. The search plug-in for
393 browser bookmarks should directly query the bookmark database, as it would
394 be undesirable to launch an instance of the browser to service a search request.

395 **Auxiliary Sources**

396 Once a result is provided, useful additional information can be added by auxil-
397 iary sources:

- 398 • Tumbler can provide thumbnails for documents and media
- 399 • Plugins can offer related searches, eg. songs by the same artist or in the
400 same album, similar songs, places near a location,
- 401 • On-line services could be used to retrieve album art, lyrics, or movie plot
402 synopses.

403 **The SDK Persistence API**

404 The SDK will provide a persistence API to applications – as this API can be used
405 to store recently used or favorite items. The SDK Persistence will also provide
406 a plugin for the global search infrastructure, to provide useful information as
407 both a primary and an auxiliary source.

408 Several types of data could potentially be managed by the SDK persistence API:

- 409 • Favorite lists - items the user has declared to be important.

- 410 • Recently used lists – items the user has interacted with recently. This is
411 a convenience API to information stored in Zeitgeist
- 412 • Application-specific data – anything an application wants exposed to the
413 search framework.

414 Data should be stored in such a way that the search result can be easily passed
415 to the appropriate application for launching. One possible set of data for an
416 item stored by the persistence API would be:

- 417 • The information classification (as in [Information classification](#)) for the
418 stored item.
- 419 • The name of the item – the name of the web page a bookmark refers to,
420 name of a radio station, etc. This is what will be shown as the search
421 result.
- 422 • A reference to the activatable item - a local file name, a URL, or other
423 relevant data that would be passed to the application to activate it.
- 424 • The time of the last usage of this piece of data (see following comments).
- 425 • Potentially some simple keywords so proprietary data can be better inte-
426 grated with search.
- 427 • Any additional information the application wishes to attach to this item
428 - unused by the search system.
- 429 • Any additional information used by the search subsystem, not modifiable
430 by the application itself. For example, the original plugin that provided
431 the item.

432 In practice there are several ways to decide if an item is recently used. An
433 application could track the last 5 documents it has been required to open, or a
434 web browser could track all sites it has visited in the last 2 weeks.

435 Examples of favorites and recent used items for common applications

Application	Favorite	Recent used items
Web browser	Bookmarks	History
Navigation	Favorite places	Last destinations
Radio	Station list	Last station
Weather	Favorite locations	Last location
Contacts	Favorite contacts	Last contacts called or messaged (sent or received)
Documents	Files in ~/Documents	Last opened documents
Media player	Playlists	Last played
Calendar	Next events	Last opened event

436 It is recommended that regardless of the methods of determining recency, a date
437 of last usage is stored in the persistence framework for searchable items. This

438 will allow the search system to fairly prioritize results from different applications.

439 Application-specific data presents a rather big challenge to the search frame-
440 work, both in terms of implementation and UI design. While some application
441 concepts can be represented in intuitive ways by a generic search interface, that
442 will be the exception rather than the rule. Therefore, Collabora recommends
443 that search be limited to item names and keywords that the application may
444 associate with the name. More complex searches, such as searching for music
445 that is above a given media rating, should be available in the application itself,
446 otherwise the general search will be too complex to use and implement.

447 **Example Search Flow**

448 A search begins in the HMI, either by voice recognition or by in-
449 teraction with the touch screen. Before any lengthy search is per-
450 formed, hard coded response logic is checked for simple responses such
451 as changing the radio station or checking the weather at the current loca-

459 from the presented results, or request that the search engine try lower priority
460 (and potentially slower) content types to satisfy the request (dynamic data).

461 It's not unlikely that a single plug-in can return results of different content
462 types – the application manager's plug-in, for example, may return applications
463 as well as application preferences. The search system must be able to tell the
464 plug-ins that it is currently only interested in a subset of available content types
465 to control the returned results.

466 The plugins may also suggest related search items, eg. Similar songs, songs by
467 the same artist, places near a location. The UI will display these related items
468 as a subitem. If selected, the search engine will initiate a new search with the
469 selected condition, and the search will start over.

470 Once the user selects an appropriate result, the appropriate action should be
471 taken (Some examples are: launching an application or changing the radio sta-
472 tion). The search framework should use the finally selected search result to
473 assist in re-prioritizing plug-ins and categories for future searches.

474 **Implementation Examples**

475 It is not the intent of this document to dictate application design decisions,
476 such as file formats or storage methods for application data (like bookmarks,
477 calendar entries, and contact information.)

478 However, this section provides some potential ways to provide search results for
479 each of the content types from **Content categories** and some recommendations
480 that may make developing the search system easier.

481 Collabora recommends against trying to use Tracker as an indexer for any prop-
482 rietary data formats, instead preferring a plug-in for the search framework
483 instead.

484 If an application changes the format of the data it wants to store, the Tracker
485 database would need to be updated for application management operations.
486 Tracker's database is not governed by the application rollback system, so these
487 updates would not be reversible.

488 Similarly, it would be preferable to avoid using Tracker to mine any new file
489 types, or have it index application storage areas other than the general stor-
490 age area. Proprietary file types can instead be handled by agents or plug-ins
491 provided by the applications that operate on them.

492 Since Apertis will not have a file browser, some standard file types (vcards,
493 icalendar, GPX) should likely not be stored at all, and instead be consumed
494 and deleted by the appropriate application when presented to the device.

495 Allowing these formats to be stored, indexed and displayed as search results
496 would create confusion when the application responsible for that data type also

497 returned a similar search result. This problem is explained further in the follow-
498 ing sections.

499 **Applications**

500 For the purposes of global search, applications can very broadly be separated
501 into two groups:

- 502 • Installed – results can be returned by a plug-in that uses the application
503 launcher’s database of application manifest to return pertinent results.
- 504 • Available from the store – a plug-in that connects to the application store
505 could locate installable applications that match the user’s search.

506 **Preferences**

507 The Apertis Application Development document defines a system in which ap-
508 plication settings for all applications are managed by a single app-settings ap-
509 plication.

510 Under such a system, a single plug-in could be written to provide any settings
511 managed by the preference manager as search results to the global search front
512 end.

513 **Documents**

514 Document search results can be provided by several sources:

- 515 • Local documents in standard formats will be returned by the system in-
516 dexer.
- 517 • Favorite and recently used files and web pages can be returned by the
518 SDK persistence API search plug-in.
- 519 • A plug-in could perform a Google search.
- 520 • Data in proprietary formats could be searched by application specific plug-
521 ins.

522 **Media**

523 The Media Management Design deals specifically with the handling of media
524 content via a combination of Tracker, Tumbler and Grilo.

525 Radio station results could be provided by the SDK persistence API. Tracker
526 also has an ontology for radio stations, so storing station data there is an option.

527 **Contacts**

528 The Contacts design defines an approach to contact management based on a
529 libfolks front end. A plug-in using libfolks could be created for the global search

530 system to provide contacts as search results.

531 A file format – vcard (.vcf, .vcard) exists for the exchange of contact information.
532 If it’s deemed necessary to index these for some reason, it should be noted that:

- 533 • “Activating” a vcard file generally results in adding a contact to a contact
534 database – which is quite likely not what the user is trying to do via the
535 search interface.
- 536 • A vcard file may contain a subset of the information available to libfolks,
537 and will not remain in sync with it if contact information is updated.
- 538 • Activating the vcard may in fact replace more recently updated informa-
539 tion in the contact system with older data.

540 As such, a vcard file search result may be hard to distinguish from a contact
541 search result, and vcard files should probably not be returned as results at all.

542 **Events**

543 Like contacts, calendar events have a standardized file format for passing along
544 event data – iCalendar (.icf). Also like contacts, this format is probably only
545 used for synchronizing events between devices and is probably not the calendar
546 application’s native storage format.

547 Like .vcf files, .icf files should probably not be part of the returned search re-
548 sults to avoid confusing behavior. Instead, a plug-in that uses the calendar
549 applications native storage format could provide these results.

550 Depending on application design decisions, a single calendar application might
551 not be the only source of searchable “events” - a social media application might
552 also provide search results.

553 **Communications**

554 The applications responsible for handling phone, SMS, e-mail and instant mes-
555 saging data can all be responsible for searching their own logs for providing
556 search results.

557 A plug-in based on libfolks could provide auxiliary information about the con-
558 tacts involved in the communications returned by the primary results providers.

559 **Definitions**

560 A Plug-in could search Wiktionary via the OpenSearch API, or a standalone
561 dictionary application could provide a plug-in to provide results from its local
562 database.

563 **Locations**

564 Navigation and weather software can provide favorite or recent locations via the
565 persistence API's plug-in.

566 A plug-in for the navigation software could allow searching the map data to
567 return possible destinations, and a weather plug-in could be queried for current
568 conditions at those locations.

569 A weather plug-in should probably employ efficient caching, since searching for
570 nearby points of interest will almost always return a large number of locations
571 in the same weather reporting domain.

572 **Using Existing Global Search Software**

573 Many search frameworks already exist, and it may be possible to re-use some
574 of their code. [Unity lenses](#)⁷ have been singled out as a particularly interesting
575 search architecture.

576 The Unity search system consists of 3 pieces:

- 577 • The Dash – the user interface components. These are an integral part of
578 the Unity UI, which itself is a plug-in for the compiz window manager.
- 579 • A collection of Lenses – search front ends which pass up result lists to the
580 user interface components. Each data type is intended to have its own
581 lens.
- 582 • A collection of Scopes – back end plug-ins that return results to front end
583 lenses. A lens can pull data from any number of scopes.

584 Lenses and Scopes are processes launched via D-bus to service search requests
585 – though a lens may have a “local scope” built into it and not require any
586 additional scopes. Both components are written in the Vala programming lan-
587 guage using libunity, and must have D-bus .service files so they can be demand
588 launched by D-bus activation.

589 In order to leverage the Unity Lens search infrastructure in Apertis, the front
590 end components would have to be re-implemented – or the code from the Unity
591 compiz plug-in could be extracted and heavily re-factored to fit within the Aper-
592 tis UI.

593 The existing code is heavily integrated with Unity, and may be very difficult to
594 extract without having to also duplicate a lot of other Unity functionality. It
595 may be easier to mimic the dash's D-bus interfaces instead of trying to fit its
596 code into Apertis.

597 Since the lens architecture requires the user to select what kind of data they're
598 searching for, in addition to UI for displaying search results, a method of select-

⁷<https://wiki.ubuntu.com/Unity/Lenses/Guidelines>

599 ing which lens to search with would also be required. In the Unity Dash this is
600 known as the “lens bar”.

601 A set of Lenses are required, one for each type of searchable data – the list of
602 content categories from [Content categories](#) would provide a good selection of
603 lenses. Some of the lenses already available for Unity might fill these roles.

604 Scopes would need to be created for the different data sources – such as a generic
605 plug-in for mining the persistence framework. Since the persistence framework
606 might contain data that fits different categories, multiple scopes may need to
607 be written for it, each presenting only one category of information.

608 Multiple scopes can provide results to a single lens, so, for example, a “com-
609 munications” lens could have a back-end scope for e-mail, and another for SMS
610 messages.

611 The lens concept differs slightly from the search paradigm presented earlier in
612 this design. Using lenses, the user would have to pick what type of data they
613 were searching for by selecting a lens, as opposed to all types of data being
614 prioritized and combined in a single list.

615 **New Software Development**

616 To implement a global search interface like the one described in this document,
617 new software components will need to be created:

- 618 • A plug-in framework for integrating search back-ends, perhaps built on or
619 with code re-used from software from [Using existing global search software](#)
620 A similar plugin framework is also offered by Grilo. Although Grilo is
621 focused on multimedia content, the plugin framework could be reused and
622 adapted to serve general content, as needed by the SDK Persistence API.
623 Also, Grilo is already used within Apertis, avoiding new dependencies.
- 624 • Plug-ins for the framework – many of these will be thin wrappers around
625 existing search functionality such as that listed in [Potential search back-](#)
626 [ends](#), some will be Apertis specific and require more development.
- 627 • A UI for presenting and interacting with search results.
- 628 • Preference management for the search system.