Global search

Contents

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

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

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

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

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

Information Classification

Information Sources

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

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

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

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

Content Categories

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

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

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

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

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

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

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

Content Flags

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

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

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

Auxiliary Information

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

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

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

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

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

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

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

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

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

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

Search Priority

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

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

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

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

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

Speech Recognition

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

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

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

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

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

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

Guidelines

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

Decentralized Indexing

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

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

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

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

Extendable Via Plug-ins

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

> Using existing global search software provides details on some existing global search solutions.

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

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

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

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

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

Easy for Application Developers

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

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

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

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

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

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

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

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

Searching the persistence API's storage is covered further in The SDK persistence API.

Highly Responsive

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

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

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

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

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

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

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

Limited System Impact

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

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

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

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

with subsequently "deeper" searches progressing to lower priority levels.

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

Predictable Interaction

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

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

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

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

Balance of Configuration and Heuristics

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

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

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

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

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

Potential Search Back-ends

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

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

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

Primary Sources

The following software solutions bear strong consideration for inclusion as primary search backends:

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

New search backends could readily be built from:

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

[1] https://zeitgeist.freedesktop.org/
[2] https://wiki.gnome.org/Apps/Evolution/EDS_Architecture
[3] https://wiki.gnome.org/Projects/Folks
[4] https://wiki.gnome.org/Projects/Grilo
[5] http://www.opensearch.org/
[6] https://github.com/luceneplusplus/LucenePlusPlus

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

- Application Manager – The application manager could provide search for installed applications, and perhaps even allow searching running applications to allow a quick jump to recently used applications on the application stack.
- Preference Manager – The preference manager has access to all application and global UI settings, and could provide these settings to the search framework.
- Browser – The browser application's bookmark list should exposed by the search infrastructure.

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

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

Auxiliary Sources

Once a result is provided, useful additional information can be added by auxiliary sources:

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

The SDK Persistence API

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

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

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

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

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

- The information classification (as in Information classification) for the stored item.
- The name of the item – the name of the web page a bookmark refers to, name of a radio station, etc. This is what will be shown as the search result.
- A reference to the activatable item - a local file name, a URL, or other relevant data that would be passed to the application to activate it.
- The time of the last usage of this piece of data (see following comments).
- Potentially some simple keywords so proprietary data can be better integrated with search.
- Any additional information the application wishes to attach to this item - unused by the search system.
- Any additional information used by the search subsystem, not modifiable by the application itself. For example, the original plugin that provided the item.

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

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

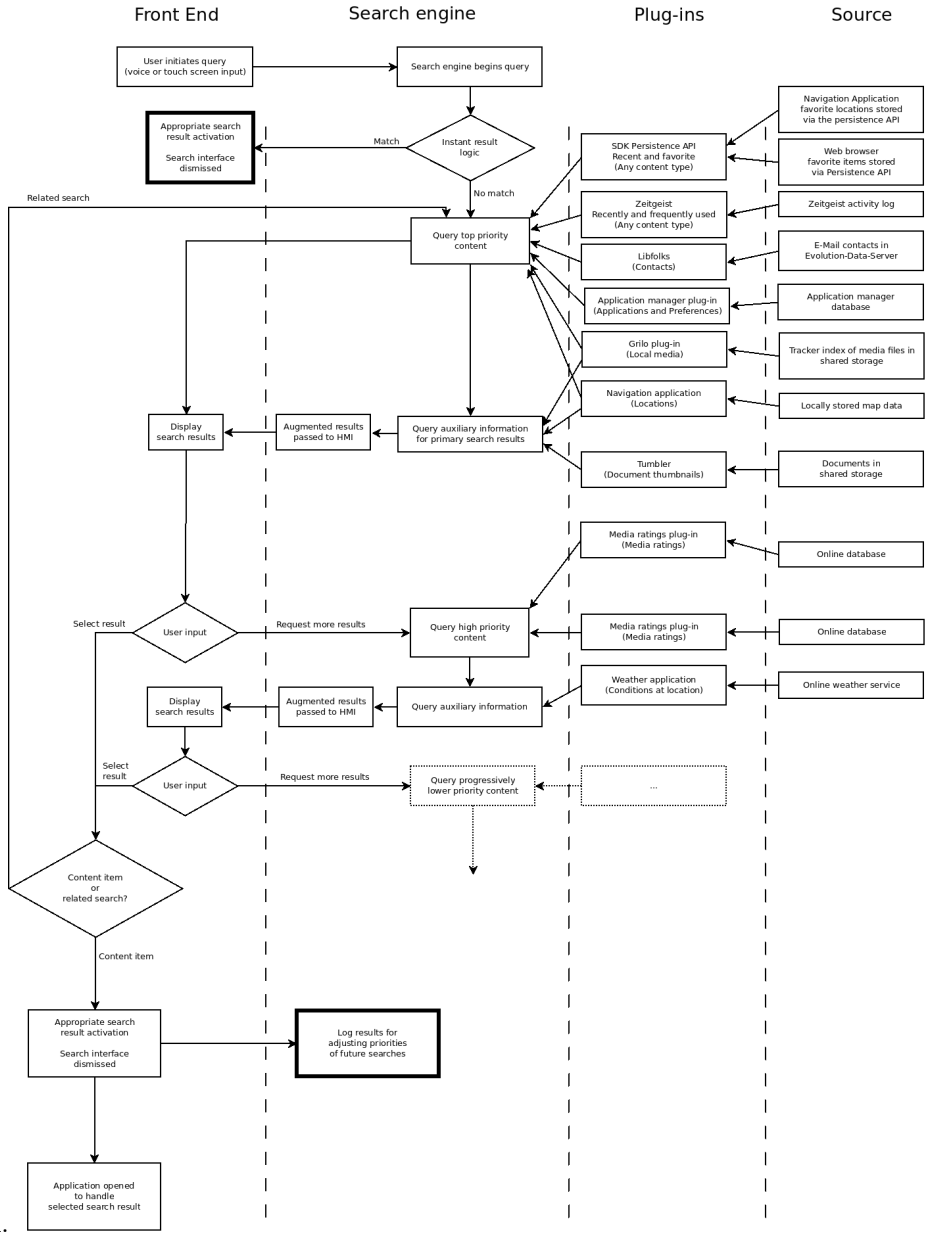
It is recommended that regardless of the methods of determining recency, a date of last usage is stored in the persistence framework for searchable items. This

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

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

Example Search Flow

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



tion.

If this logic completes the search, the appropriate action is taken and the interface is dismissed. If there is no user selection within a configurable time, the search engine begins performing queries of the back end plug-ins, starting with the highest priority information categories (static data content), including auxiliary information.

As search results are accumulated and displayed, the user is able to either select

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

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

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

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

Implementation Examples

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

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

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

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

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

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

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

returned a similar search result. This problem is explained further in the following sections.

Applications

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

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

Preferences

The Apertis Application Development document defines a system in which application settings for all applications are managed by a single app-settings application.

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

Documents

Document search results can be provided by several sources:

- Local documents in standard formats will be returned by the system indexer.
- Favorite and recently used files and web pages can be returned by the SDK persistence API search plug-in.
- A plug-in could perform a Google search.
- Data in proprietary formats could be searched by application specific plug-ins.

Media

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

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

Contacts

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

system to provide contacts as search results.

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

- "Activating" a vcard file generally results in adding a contact to a contact database – which is quite likely not what the user is trying to do via the search interface.
- A vcard file may contain a subset of the information available to libfolks, and will not remain in sync with it if contact information is updated.
- Activating the vcard may in fact replace more recently updated information in the contact system with older data.

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

Events

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

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

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

Communications

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

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

Definitions

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

Locations

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

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

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

Using Existing Global Search Software

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

The Unity search system consists of 3 pieces:

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

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

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

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

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

[7] https://wiki.ubuntu.com/Unity/Lenses/Guidelines

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

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

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

Multiple scopes can provide results to a single lens, so, for example, a "communications" lens could have a back-end scope for e-mail, and another for SMS messages.

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

New Software Development

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

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