GeeXboX Valhalla

Version 2

Mathieu SCHROETER

libvalhalla.GeeXboX.org

October 5, 2013

Contents

CHAPTER 1

Introduction

GeeXboX Valhalla is a library initially written for the GeeXboX[1] project. Its first goal is to interact with the graphical user interface Enna[2]. Valhalla can search audio, video and image files recursively from one or more directories. Several mechanisms extract metadata and properties to save them in a SQLite database. Other informations can be retrieved from some web services, like covers, lyrics, etc, ...

This documenation is destinated to the developers which are interested by hacking libvalhalla and to learn how to use it. In all cases it is highly recommended to look at the source code (at the same time that this document). The public API has a *Doxygen* documentation and many special internal features are commented in the code. Because libvalhalla uses many threads, the developer must take care of all tasks on the pointers which are sent from one thread to an other. In some exceptional conditions, a pointer can be available in two threads at a time (for writing too). It is not a bug, many attributes are considered to be read-only (most of time). A tool like *Valgrind* will return many possible race-conditions (see the *Helgrind* tool). But all of these are only false-positives. This documentation tries to explain how are view the operations between the threads and why *Valgrind* is "wrong" (in our case).

Note: if you are thinking that the "threads are evil", maybe you should re-consider the question. It is like thinking that the object languages are better that imperative languages. There is no good or bad answer. But for sure, the C language and the multi-threading programming need to be more strict in the ways which are used in the architecture of the library. To think that an imperative language can not be used to develop programs with an object approach, it is only a misunderstanding of what is an object.

1.1 Why Valhalla?

> « In Norse mythology, Valhalla (from Old Norse Valhöll "hall of the slain") is a majestic, enormous hall located in Asgard, ruled over by the god Odin. Chosen by Odin, half of those that die in combat travel to Valhalla upon death, led by valkyries, while the other half go to the goddess Freyja's field Fólkvangr. In Valhalla, the dead join the masses of those who have died in combat known as Einherjar, as well as various legendary Germanic heroes and kings, as they prepare to aid Odin during the events of Ragnarök. » [3]

In Valhalla, the heroes are happy: they fight, kill, are reborn to assail again in a closed field.

You can imagine by analogy, that the scanner (the Valkyries) finds the media files (the legendary heroes) in order to insert them into the database (to Valhalla). And each file (hero) is checked in loop (is reborn to assail again). The grabbers can be seen as the Valkyries too.

[1] An embedded oriented Linux distribution `http://www.geexbox.org`

[2] `http://enna.geexbox.org`

[3] Valhalla. (2009, February 11). In Wikipedia, The Free Encyclopedia. Retrieved 14:49, February 14, 2009, from `http://en.wikipedia.org/w/index.php?title=Valhalla&oldid=270025459`

CHAPTER 2

General overview

Valhalla is heavily based on threads to parallelize all tasks. FFmpeg/libavformat is used to fetch the metadata. The data are saved in a database managed by SQLite. This library can be extended with many grabbers. The library is highly configurable, but the grabbers are only static.

2.1 Software architecture

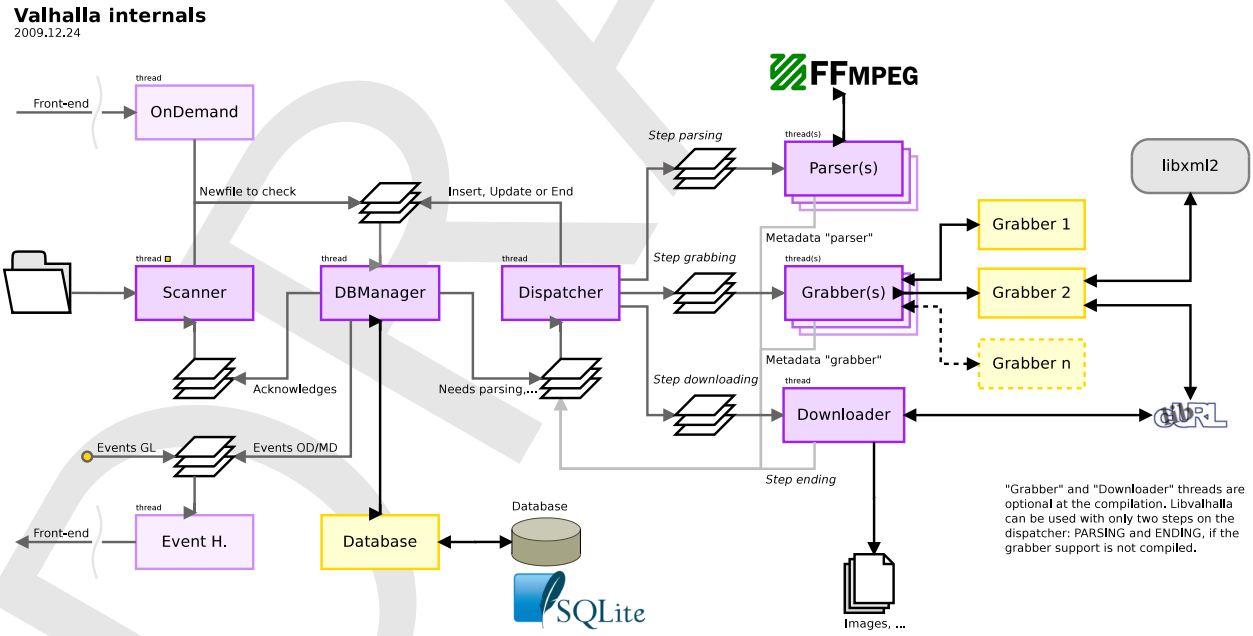

Figure 2.1: General architecture

There are height threads (or more) to handle all functionalities of the library. At the left, two inputs are available, the *Scanner* and the *OnDemand*. The *Scanner* searches the files from at least one directory (recursively or not). The *OnDemand* is a file specifically pointed by the frontend. When a file must be checked for metadata fetching, it is sent to the *DBManager*. Its goal is to check the validity of the demand by searching this one in the database, and accordingly to some rules the file is sent to the *Dispatcher* to be proceeded.

The *Dispatcher* is like a switching device. It sends a file to the appropriate part like the *Parser*, the *Grabber* or the *Downloader*. When a step is ended, the file is sent to the *Dispatcher*, until all steps are done then it is sent to the *DBManager*. So, an acknowledge is sent to the *Scanner* or an event is sent to the *EventHandler* (if the file comes from the *OnDemand*). Note that many parts can be disabled at the compilation. And some threads are not loaded accordingly to the configuration at the initialization of Valhalla, like the *Scanner*, the *Grabbers* and the *EventHandler*.

The steps with the *Dispatcher* can be seen as follow:

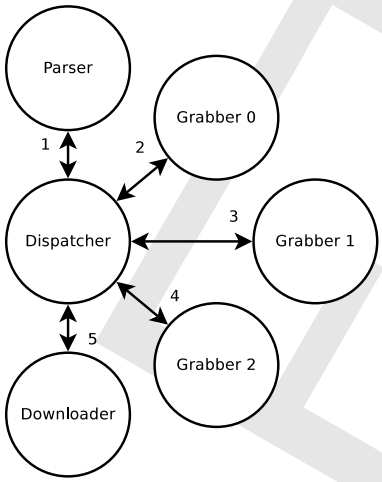

Figure 2.2: General view around the *Dispatcher*

The file jumps from component to component but every time by the *Dispatcher*. The number of grabbers depend of the configuration (at the compilation) of libvalhalla, the type of file and if the user has disabled explicitly one or more grabber. As the grabbers are optional, it is possible to have only the *Parser* and nothing more. A grabber fetches always and only textual metadata. When a data looks like an image (a blob), then only the local path on the binary is saved in the database. The binaries are downloaded by the *Downloader* when all grabbers have ended they job. In other words, many references on binaries can exist in the database before that the binaries will be available.

2.2 Organisation

TODO: more informations

All components use its own file. It is organized as follow:

- `valhalla.c`
  like the factory class, it creates objects and all user inputs are sent to it.
- `dbmanager.c, dispatcher.c, downloader.c, event_handler.c, grabber.c, ondemand.c, parser.c`
  are the components, each one can have one or more threads.
- `grabber_*.c`
  are the grabbers which are called by grabber.c. The main header is `grabber_common.h` with the necessary *Doxygen* documentation.
- The other files are different features and functionalities which are used by many parts.

The public header is only `valhalla.h`.

2.2.1 Versioning policy

The version number is composed of three values (as usual). The major version means no backward compatibility to previous versions (e.g. removal of a function from the public API). The minor version means backward compatible change (e.g. addition of a function to the public API or extension of an existing data structure). The micro version means a noteworthy binary compatible change (e.g. new grabber).

2.3 Database

The database uses a relational model. The design is think in order to be as generic as possible. All sort (expect blob) of metadata can be saved and retrieved easily.

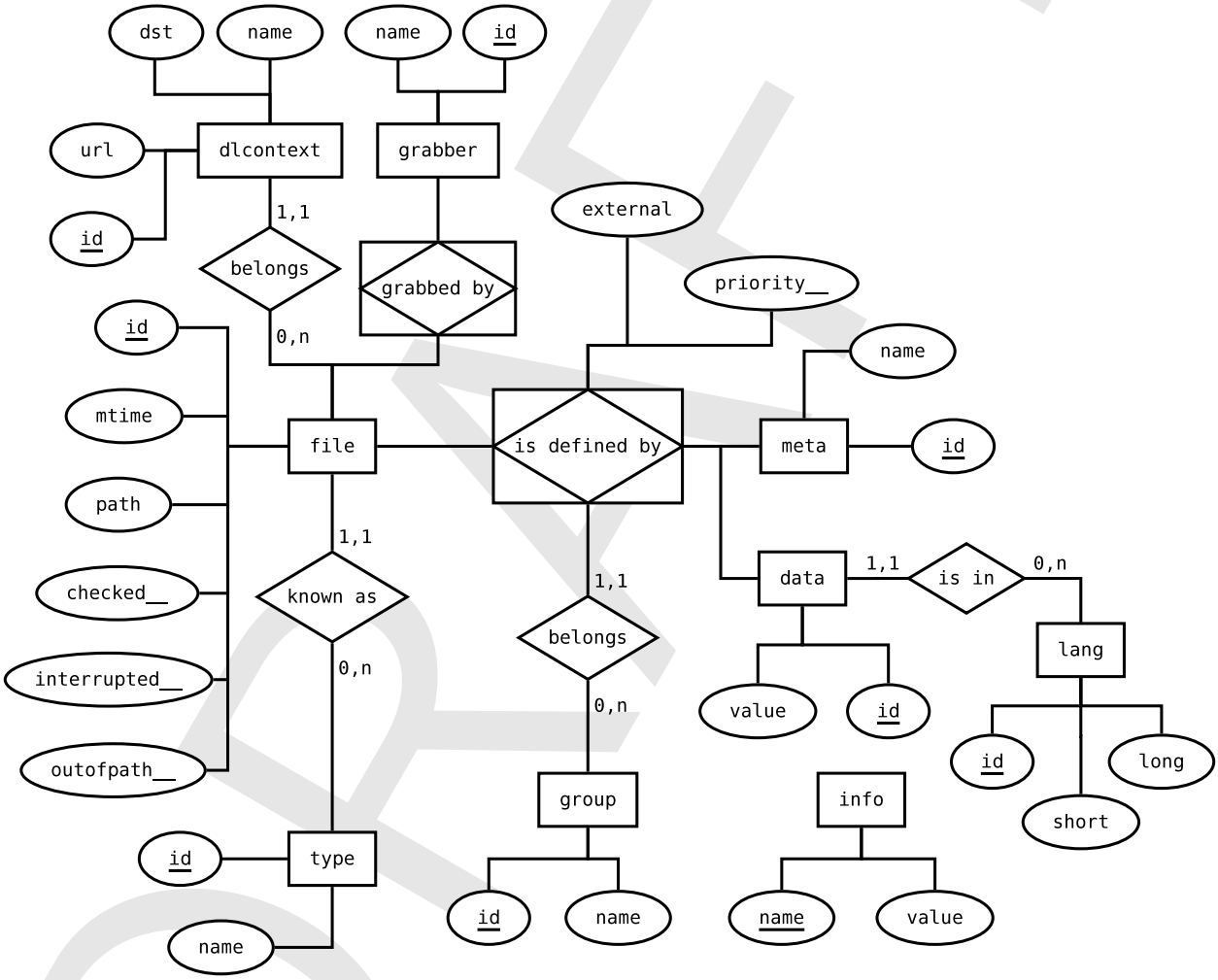

Figure 2.3: Entities/Relations diagram

The table "file" can be considered as the main table. It contents the list of all files which are been retrieved by the *Scanner* and by the *OnDemand*. A file has some public properties like the type (audio, video, image, unknown), the path (which can be absolute or relative) and the time of the last modification. About the properties like checked___, interrupted___ and outofpath___, please refer to the section ###SECTION###.

The tables "dlcontext" and "grabber" are only for internal purposes. The first one is used in order to save the contexts with the *Downloader* when the library is forced to stop as soon as possible (see the section 3.3.2). The second one is the list of grabbers which have already been completed for the file. Here too, the goal is to avoid a useless regrabbing for a file which was interrupted.

The metadata are defined with three tables. Every meta has a name in the "meta" table, and every meta can have one or more data with the "data" table. The relations between the meta and the data are done with a relation table (is defined by) with a joint on a file. This relation table has some properties like the priority and a group. The groups are used in order to prevent misunderstands between the meta which can have the same name but not the same meaning. For example, you can imagine a meta "genre" which is used to define a style of music, and the genre for a person. Maybe you can have both meta for the same file. Then the group can be used for the distinction.

The "info" table is used only for internal purposes. More informations are available in the section ###SECTION###.

CHAPTER 3

Internals

3.1 Components

Even if libvalhalla is written in C, the code is structured with an oriented object approach. Here we are speaking of "components". A component is seen like an object, a file source and one (or more) thread(s). All users inputs are sent via valhalla.c which can be seen like the factory class. This one is used in order to create and delete all components and to handle all actions with the public API.

For example, the actions dedicated to the database are always sent to the *DBManager*. In this case, some functions are just bridges between the public API and the functions in database.c. It looks like:

`valhalla.c` → `dbmanager.c` → `database.c` **instead of (**

`valhalla.c` → `database.c` **or even** `database.c` (directly) with a bad design

)

Where valhalla.c is like the factory class, dbmanager.c a class to handle all actions on the database and database.c the low level handling with SQLite.

3.1.1 Scanner

The *Scanner* searches for files in the paths provided by the user. It is one of the components in the library to send files to the *DBManager*. The *Scanner* can be always alive, or can be used for only some loops. A loop is one pass of all directories configured for the scan. The scan can be recursive or not. When a file is found, it is tested against some conditions and accordingly to these, then it is sent to the *DBManager*, or not. If no path are set before the run, then the *Scanner* is not loaded.

You can see below, the list of all interesting options which can be configured:

- Paths addition
- File suffixes addition
- Number of loops
- Time to wait between the end and the start of a new loop
- Time to wait before the first scan (delay)

To enable the *Scanner*, at least one path must be added. If several paths are added, then each one is scanned one by one in the same order that the additions. Each path can be configured for a recursive or simple scan. If the scan is recursive, there is a security to limit the depth. The limit is set to 42 sub-directories and it can't be changed dynamically. Note that the symbolic links are ignored by the *Scanner* (and the *OnDemand*).

The file suffixes are a simple way to ignore the non-media files. It is possible to don't add a suffix, in this case all files found are sent to the *DBManager*. If at least one (or more) file suffix(es) is/are defined, only these suffixes are considered for the scan.

When one loop is finished, there are several possibilities. It depends how the user is using the library. An optional function provided by the public API can be used in order to wait on the end of the *Scanner*. But in many cases, it can be useful to loop for an undetermined number because the end of a program is not always depending to the end of the scanning. Then, the number of loops can be an integer higher than 0 or infinite. The time to wait is useful to don't rescan the paths immediately after a loop. At the end of one loop, if the time is defined then it starts a timer, and begins the new loop when the timer is reached. A public function gives the possibility to wake up explicitly the *Scanner* in any time.

Note: instead of a recursive scan on the directories, we should use some mechanisms like inotify with Linux. But in this case, it needs to re-implement the way for every operating system. Maybe this functionality will be available in the future.

Next loop event

When a full scan is finished, the *Scanner* waits for the file acknowledges and goes in waiting list. The acks come from the *DBManager* when a file has been defined for the last step which is STEP_ENDED. The ack is sent to the *Scanner*, and this one waits for all acks, even for the files which were not handled correctly. In other words, all files sent to the *DBManager* return a ack. After that, the *Scanner* uses an interruptible sleep in order to go in waiting list. Because this sleep is interruptible, it is possible to wake up the *Scanner* with the public API. Or it will wait the time which was configured by the frontend.

Between the last ack received and the sleep, the scanner sends a NEXT_LOOP event. This one is useful for the other threads. For example some grabbers wait on this event in order to do some cleanups for their internal structure. For the *DBManager* this event is very important. It is used for many major cleanups in the database. These cleanups are done only between the loops, because it takes a significantly time (many deletions and updates in the whole tables, ...).

Note: when libvalhalla is forced to stop, the database is not cleaned. In fact there is no effect for the user. But if this action was done at the shutdown, it can increase the time and especially when the files (scanned) are available through a network share.

3.1.2 OnDemand

The *OnDemand* is the second component which is able to send files to the *DBManager*. Its goal is to retrieve with a top priority the metadata for one file, over all other files which are coming from the *Scanner*. *OnDemand* is asynchronous, and events are sent for each step to the frontend.

A demand can not be sent to the *DBManager* without several checks for the file.

1. Is the file available? If the file does not exist, we can go out immediately. The file must not be a symbolic link.
2. Is the file already and fully available in the database?
   In this case, we can send the ENDED event because there is nothing more to do.
3. If the file is not available, then we must search if this one was already sent in Valhalla by pausing all threads (*Grabber*, *Downloader*, *Parser*, *Dispatcher* and *DBManager*).
   The threads must be sent in waiting list because the file can exist in a thread but not in a queue. When all threads are paused, we are sure that all files are somewhere in the queues. Note that the *Scanner* is not interrupted by the *OnDemand*.
4. For each queue, the file is searched. Note that a file can exist several times, but the same pointer is used. We can stop the search when the first occurrence is found.

(a) If a file is found, then we must change its priority to HIGH. This file is moved to the top of the queue, then the thread will handle this one immediately.

(b) If nothing is found, a new file is sent to the *DBManager* with the HIGH priority.

5. Only here, all threads are woken.

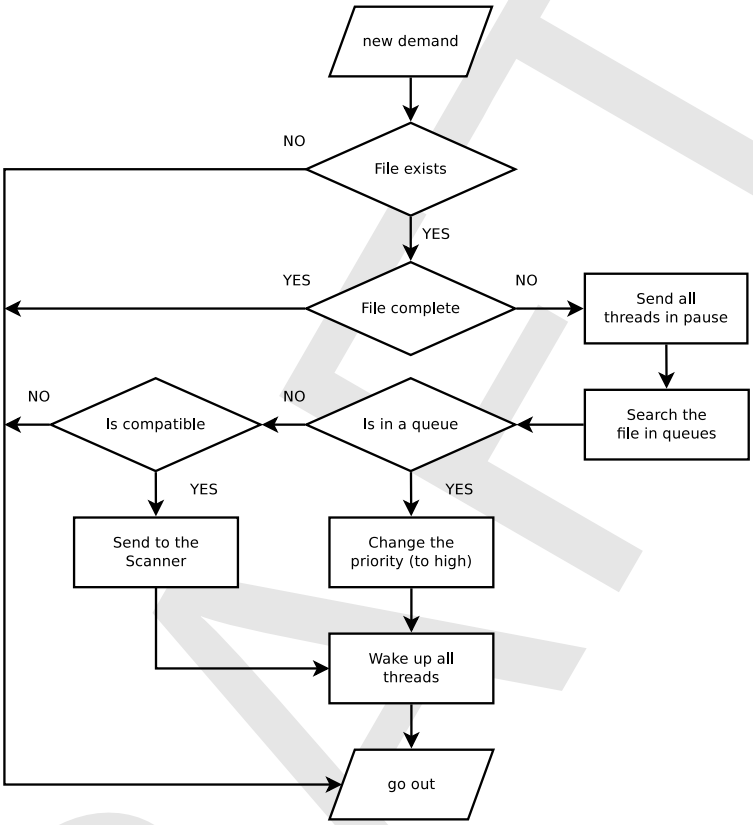

Figure 3.1: Demand checking

Note: the queues are named fifo_queue for historical reason. The code comes from libplayer[1] where only the FIFO mechanism is used. LIFO was used only with libvalhalla when the OnDemand component was added. But the name for the queues was not changed.

Pause

The pauses are synchronious. It sends a message to the thread and waits on the answer. A thread finishes always its last operation before the pause. It is not like the force-stop mechanism which is able to break an operation in the middle. According to these facts, the pause is a bit slower. It depends mostly of the grabbers, but it can use several seconds by demand.

Note: a future improvement will be to use an asynchronious call on all components (like the force-stop mechanism). It should be a bit faster. At the moment, the threads are sent in waiting list one by one.

Priorities

The priorities are handled by the position in the queues. A file with an high priority is a file which is alway put at the beginning of the queue. In other words, the queue is used as a LIFO. The normal priority uses the queue as a FIFO. Note that too much demands are not very efficient because all files with an high priority will be always resent to the top of the queues. Of course, these files will finished before all other files.

[1] A multimedia framework `http://libplayer.geexbox.org`

3.1.3 DBManager

All actions on the database are handled by the *DBManager*. The selections can be parallelized, but the writes on the database are always serialized in the thread.

Note: in the case where several writes are done in concurrency, even if SQLite is fully thread-safe it can have many side effects for libvalhalla. The database file will be always intact (SQLite is very robust even against hardware problems). But it can be possible to have some broken associations, wrong informations, etc,... Then only this thread is able to perform the writing.

The *DBManager* is the second step after the *Scanner* and the *OnDemand*. The *DBManager* does many checks on the file to be sure that it must be sent to the next step (or not). And in some cases, informations are retrieved from the database in order to restore the previous context for this file. The reason is that libvalhalla can be interrupted in all parts of its component (see force-stop in the section 3.3.2). Before that the whole library is uninitialized, the states are checked and the contexts (for the *Downloader*) are saved in the database if necessary. These contexts (and the list of grabbers already completed) are loaded by the *DBManager* (or not). It depends if the file is still available (for example).

Steps for a file

There are 6 distinct steps for one file. The first one is the ACTION_DB_NEWFILE when the action comes from the *Scanner* or the *OnDemand*. The date of the last modification is searched in the database, then the *DBManager* asks for the interrupted__ flag (see section ###SECTION###). It is used to know if the file is complete or not (force-stop). According to these informations, the context for the downloader and the list of grabbers (to ignore for this file) are got from the database. If the date of the last modification has changed, then the file is deleted from the database in order to perform a new parsing, grabbing, etc,... like with a new file. If the file is new, it is inserted at this step (even if no metadata are still available).

In all cases the file is sent to the *Dispatcher* for the *Parser*. Because the *Parser* is necessary in order to retrieve the main metadata. The next step is the *Grabber* a number of times which depends of the number of grabbers and some conditions. The file is sent to the *Dispatcher* between every grabber. When all grabbers have done their job, the step changes for the *Downloader*. When all distant files were downloaded, the file is in the ending step and the *Dispatcher* sends this one to the *DBManager*.

The *DBManager* detroys the file with the ending step and sends a acknowledge to the *Scanner* (if the file comes from it) or it sends an event specifically for the *OnDemand*.

Lock for the grabber

When a file was sent from the *Dispatcher* to the *DBManager* and this file is still in the grabbing step, while that the metadata are inserted in the database, the *Grabber* is locked. This lock is needed in order to prevent a race condition between the *DBmanager* and the *Grabber* on the meta_grabber attribute of the file structure. As soon as the metadata are inserted and flushed, the semaphore is released.

External metadata

TODO

3.1.4 Dispatcher

The *Dispatcher* sends every file to the right component according to the step defined in they attributes. The step is never changed by the *Dispatcher*. But in one case, the priority can be changed for a lower. More details at the end of this section.

Splitting between components

The *Dispatcher* is intended to split a file into two components where appropriate. There is currently only one case where it is necessary. When a file has finished a PARSING or GRABBING step, the metadata must be inserted in the database. But if there is at least one grabber, the file is sent to this one too. So, the file is split and available in both components (*DBManager* and *Grabber*). The grabber must wait on the *DBManager* until that a semaphore is released, then it can continue with the new grab.

Last step (ending)

For the last step, the *Dispatcher* must change the priority for the file. We can think that it will be a problem for the *OnDemand*, but it is not the case. When this step appears, all metadata are available in the database. Only the acknowledge (or the last event) is not sent. The last step can not be done with the high priority because the file is sent two times to the *DBManager*. The first one is used in order to insert the metadata (the last grabber or the parser), and the second one is used to destroy the file. Of course, if the second is sent with an high priority, then the file will be destroy before the insert.

3.1.5 Parser

The *Parser* is the first step in order to retrieve the metadata about a file. It is the most important step, because most of the *Grabbers* are unusable with at least the "title", "author", "artist" or "album" metadata. The *Parser* is based on libavformat for the metadata parsing. But many files has no metadata, and the "title" can not be found. In this case a second functionality (optional) can be used to be sure that a "title" is always available. A decrapifier tries to found a "title" with the filename. This one is able to clean the name and to strip some keywords (and more) with the use of a blacklist.

libavformat

FFmpeg is very powerful, but in the case of libvalhalla the speed for the *Parsers* is very important. When libavformat tries to guess the format, it uses many functions for probing. More formats will use more call on the probing functions. A trick is used in libvalhalla to increase significantly the speed. For more details about that, please refer to the section 3.3.5.

Decrapifier

The *Decrapifier* performs many operations on the filename in order to retrieve some useful keywords (especially for the grabbers). Note that this "title" is useful for the frontend too. There are two types of keywords. The first one is the simple keyword where the *Decrapifier* searches and tries to remove it from the filename. These keywords are case-insensitive. The second type is a special keyword with a pattern. Only three patterns are available where two are only related to TV show and this type of pattern is case-sensitive.

Joker	Description	Examples (patterns)
`NUM`	trim a number	`"CDNUM"`
`SE`	trim and retrieve a season	`"SExEP", "SeasonSE"`
`EP`	trim and retrieve an episode	`"SSEEEP", "sSEeEP", "EpisodeEP"`

Table 3.1: Decrapifier

When an EP or SE is found, a new metadata is inserted with the name "season" and "episode". These metadata can be used by the grabbers too.

Note: it is possible to use several times the same joker in a pattern. In the case of SE and EP, several metadata will be inserted. With NUM, it can be useful when these metadata are not interesting. For example, something like "NUMxNUM", "SNUMENUM", "sNUMeNUM", etc,... Remember that with a joker a pattern is always case-sensitive.

3.1.6 Grabber

Several grabbers can exist in parallel. Every grabber will have its own thread and can handle only one file at a time. The word "grabber" is used for two distinctive features. There are the threads for grabbing, and there are the different grabbers (what and where to grab).

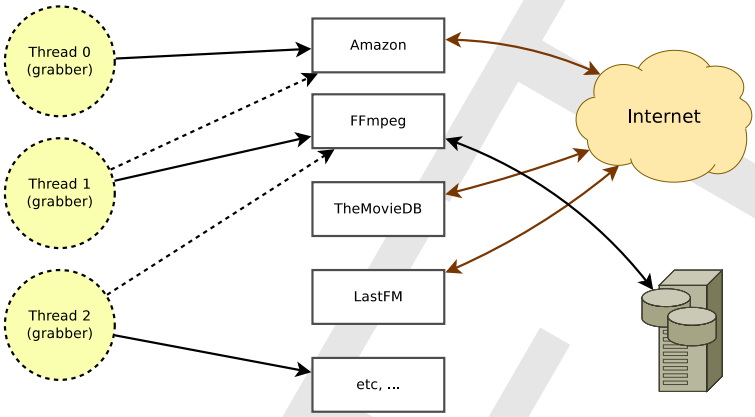

Figure 3.2: Grabbers (general)

Every thread tries to lock the first available grabber. In the figure 3.2, you can see three threads and "five" grabbers. The thread 0 has locked the grabber Amazon, the thread 1 can not lock Amazon, then it locks the next (FFmpeg). Thread 2 can not lock Amazon and FFmpeg, then it locks an other somewhere.

All threads don't try to lock every grabber. The reason is that it depends of several conditions, like the type of file (a grabber for movies can not be locked for an audio file), the state of the grabber (is the grabber available for a new grab?) and if the grabber is already locked or not. A grabber can not be available for a new grab when this one needs some time between the grabs. With some webservices, it is impossible to grab too fast. So, a delay is forced in the configuration of the grabber.

The locking

The next figure is a simplification of the way that a thread is using in order to lock a grabber. The goal is to try the locking of every grabber until that one will be available. But if after two passes of the grabber list, none of them are available, the thread stops the tries and re-sends the file to the *Dispatcher*. So, an other file can have a chance to lock a grabber. This functionality ensure that a thread is never blocked (for nothing).

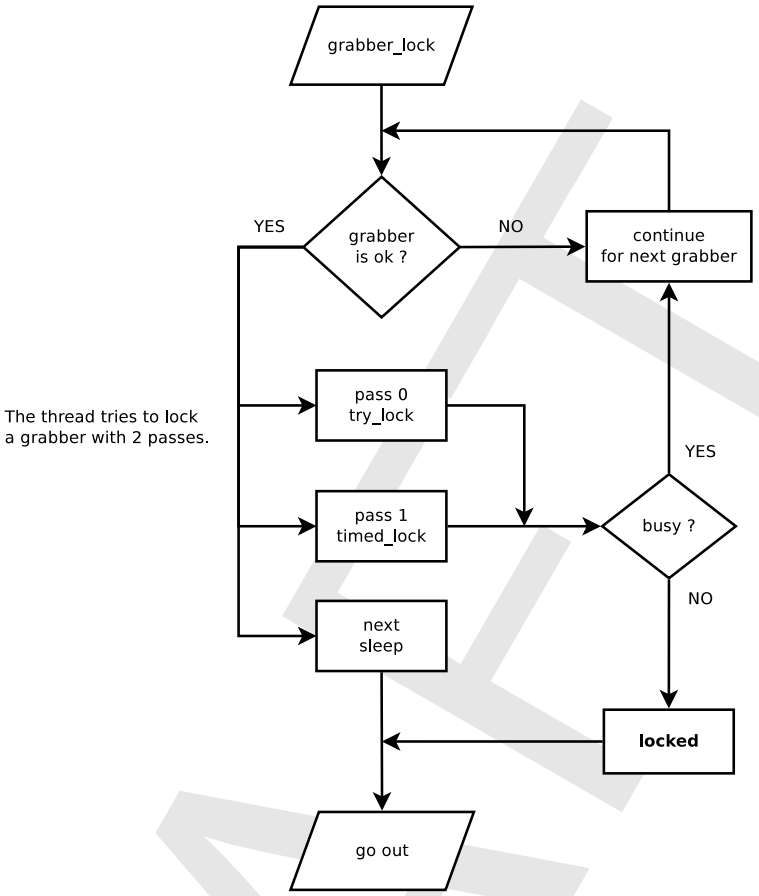

Figure 3.3: Grabbers locking

The first pass uses a trylock and the second pass uses a timedlock set to $200\,\mathrm{ms}$. Of course, the lock is done only on the grabber which are compatible with the file. A second check is done after the lock. A grabber can have a minimum time between two grabs. The reason is that some web services block a user against too much accesses. Then the grabber must sleep more time even if it seems not busy.

Waiting on the DBManager

See 3.1.4 for some general informations. The *Grabber* must wait on the file until that the *DBManager* has done its work. The pointer on the metadata grabbed is the same in both components. In this case, until that the *DBManager* has not done its work (to save all metadata in the database), the *Grabber* can not use the pointer for the new metadata. Then a semaphore is locked in the *Grabber* until that the *DBManager* will release this one.

Grabber API

The API for the grabbers is available (mostly) through grabber_common.h. For some special features, XML, lists, MD5, HMAC-SHA256, metadata, utils, etc, ... more headers are available. But in all cases, a grabber has no access on the internals components like the *DBManager*, the *Parser*, etc, ... This API uses many documentation based on the *Doxygen* syntax.

- **Metadata additions:**
  When a grabber found new metadata, it must use the functions provided by metadata.h in order to populate the meta_grabber attribute of the file.
- **Files for the *Downloader*:**
  A metadata can be a name (hashed with MD5) for a blob. Mostly for an image like a cover for example. In this case, the list_downloader attribute must be populated with the functions provided by utils.h.

Only these attributes are safe for writing. All other attributes of the file must be considered as read-only.

3.1.7 Downloader

The *Downloader* tries to download all files which were added to the list_downloader attribute of a file. The download is done only after all grabs for one file. In other words, even if a metadata for a cover (for example) is already available in the database, the image is not still downloaded until the *Downloader* step is ended. Because the download can take a significant time, this component can be interrupted in any time. The files which were not downloaded are saved in the dlcontext table at the shutdown of libvalhalla. These contexts are re-loaded when necessary by the *DBManager*.

3.1.8 EventHandler

The *EventHandler* is only destinated to the user of libvalhalla. Internal events are handled in an other way and they are not available for the public use. The public events are a bit different. There are three types of events accordingly to the functionalities which are used.

Global events

These events are very general. The goal is to prevent the user of some global actions in libvalhalla, mostly accordingly to the state of the scanner. For example, when the scanner is beginning a new scan, when it is sleeping, etc,... look at the public API (or *Doxygen*) for more informations.

Metadata events

TODO

OnDemand events

When an on-demand request is generated, the frontend can follow all useful steps by these events. There are four events; whe the file is parsed, grabbed (for each grabber with their name), and when the request is ended (implicitly it is the same event that when the downloader has ended). In order to know for what file the event is destinated, the path is sent with the event. In other words, only one callback is set by the frontend at the initialization. The path must be used for the comparaison.

3.2 Other parts

TODO

Lists

TODO

Statistics

While libvalhalla is running, many values (counters and timers) are recorded for some statistics. A counter is just one (or more) values incremented accordingly to some actions. A timer is used in order to retrieve the difference of time between two actions. Most of components use the statistics. All values can be read with the public API. Note that the statistics are printed to the terminal in a more friendly way when the verbosity is set to INFO.

The statistics are grouped with an ID. The frontend can retrieve all IDs and for each one, the counters and the timers can be read with their raw values and a name. The timers use a time based on the nanosecond and all values are returned with an unsigned int64.

3.3 Mechanisms

This section describes the mechanisms to handle correctly the communications between the components and some special features.

3.3.1 Waiting on the scanner

TODO

3.3.2 Force-stop

TODO

cURL

TODO

3.3.3 Concurrent demands for DBManager

TODO

3.3.4 Fifo queues priorities

TODO

3.3.5 FFmpeg and parsing tricks to be faster

When a file must be open by libavformat, several ways are possible. The more common way is to pass only the essential arguments, then libavformat will detect the format automatically. A second way is to pass the right format to FFmpeg. The second way is faster but in this case, the format must be the right or unexpected behaviours can appear.

The idea is using the functions for probing. When FFmpeg tries to discover the format, it probes the file with every demuxer from the first to the last and several times if necessary. It can use a significant time. In order to be faster, libvalhalla will use the probing functions only with the demuxer which looks like the best choice. This decision is done accordingly to the file suffix. For example, if the file is something.mp3, the "mp3" demuxer will be used for the probe. If the score for this demuxer is bad, then nothing is passed to FFmpeg and the full autodetection will be engaged.

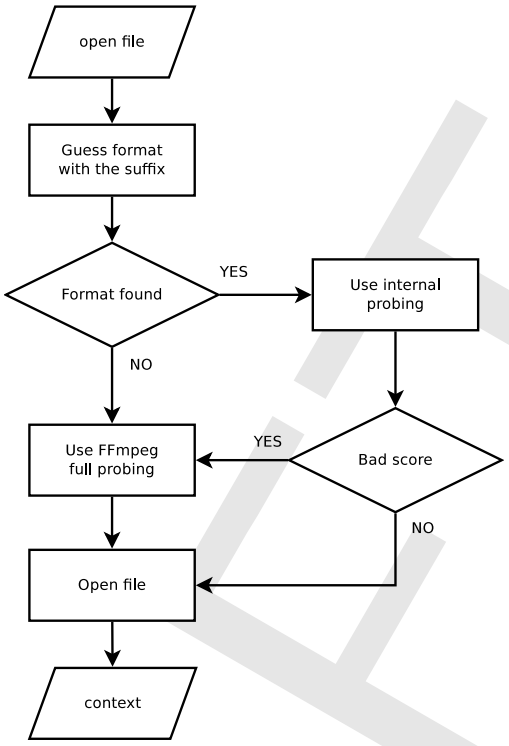

Figure 3.4: FFmpeg file probe

Note: the internal "probe" function uses the same way that FFmpeg. But it depends of some behaviours of FFmpeg and this functionality can be broken between the versions of libavformat. Look in lavf_utils.c for more informations about this trick.

Indexes

About images

libavcodec and multi-threading

Most of parts in FFmpeg are thread-safe. Only libavcodec must be used with locking in order to prevent race conditions with the codecs. Libavcodec should not be a dependency of libvalhalla, but according to the fact that the FFmpeg grabber is threaded and this one uses a function of libavformat which depends of libavcodec "av_find_stream_info()", libvalhalla must provide the necessary locks for libavcodec. A lock manager is registered by valhalla.c when libavcodec is available. Note that the FFmpeg grabber is always disabled if the libavcodec headers are not found by the "configure" script.

3.3.6 Cleanup

There are two types of cleanup. The first one is related to the database and the second one is related to in-memory structures for the files.

Database

The tables "meta", "data", "grabber", and the association tables are cleaned only at the end of a loop by the *DBManager*. The goal is to delete only the essential data in the database in order te be as fast as possible. The idea is to delete the "noise" in a table with one query instead of many triggers. This way is very efficient and works by searching all IDs which are no longer available in the table. For example, if a file is deleted from the "file" table, all associations on this file are broken, but they are not deleted immediatly. At the end of the loop, a query deletes all associations where the file_id does not exist in the "file" table.

There are two considerations. First, the foreign key mechanism available with SQLite can not be used. The second is that the database is not cleaned when libvalhalla is forced to stop (see the section 3.3.2).

Garbage

A mechanism in libvalhalla does many work in order to prevent garbage at the shutdown. This one is called mrproper() and it is very important in the case where libvalhalla is forced to stop. Because in this case, many files exist in memory with many metadata. The function mrproper() checks all files and does the right action accordingly to its state. And all files are finally released from the memory. Note that libvalhalla is leak-free (otherwise there is a bug somewhere, please report). Look at valhalla.c for more informations.

3.3.7 OS dependent codes

The library supports many platforms with the help of some specific codes. The supported OS are:

- Linux
- *BSD (FreeBSD, OpenBSD, NetBSD, ...)
- GNU Hurd
- Darwin (MacOSX)
- Windows (Windows NT and Windows CE)

The main problems of portability between the OS are related to the high precision timer functions and the priorities for the processes (threads). There are two types of processes. The process with its own memory space and the thread where the memory space is shared. In order to change the priority for a thread, it needs (in a general way) to configure the type of scheduler in order to have the possibility to use the real-time priorities. These schedulers are SCHED_FIFO and SCHED_RR. A thread by default uses SCHED_OTHER (like the processes). But the major constraint is regarding the rights needed for the real-time schedulers. Only the root can use them. In this case, the only way is the use of SCHED_OTHER with the non-real-time priorities. To proceed we need the process ID corresponding to the thread.

Note: with many kernels a thread is seen like a process. Then it is possible to change the priority of the time-shared scheduler with the common functions "setpriority()".

It is useful to look at thread_utils.c file. For Linux, we should use the gettid() function which is able to retrieve the ID for the thread. We can not use getpid() which is intended to return the TGID (thread group ID, or shared PID). The main problem here is that the glibc has no function called gettid(), then it is necessary to use a specific syscall(). For *BSD (and Darwin) it uses an other way. The kernel creates a PID for every thread, then we must use getpid(). The last particular case is for Windows. POSIX functions are not available, we must use the Windows API. For GNU Hurd, we can not change the priority because getpid() returns always the same PID and gettid() seems unavailable. It is the only regression in libvalhalla with this kernel.

The second major portability problem concerns the high precision timer. All POSIX kernel provides the functions for this purpose. But for Windows, it is a bit more difficult and needs an emulation.

Windows emulation for clock_gettime()

The timer functions of Windows are not very efficient. The granularity is about $15.625\,\mathrm{ms}$. libvalhalla uses the POSIX function clock_gettime() which work with the nanosecond. Linux is able to be very precise in the microsecond and retrieves the time in the nanosecond. The idea is to emulate the precision with the high precision timer available with the processor (or the RTC). This timer is readable with the functions provided by the Windows API. QueryPerformanceFrequency() and QueryPerformanceCounter() are needed and return the raw values provided by the precision timer. From these values, it is necessary to compute the time and to sychronize this time with the current time. Please, look at osdep.c for the code related to this section.

The current implementation is not perfect. The main problem concerns the divergence between the real clock and the precision timer. A re-synchronization is needed sometimes in order to prevent side-effects when libvalhalla is used on a very long time. At the moment, the synchronization is done only at the initialization.

Darwin and clock_gettime()

Darwin does not provide the POSIX version of clock_gettime. But with the help of the Mach headers, a functions nearly the same is available.

CHAPTER 4

How to use libvalhalla

This chapter concerns only some particular use of the library. Most of the documentation (*Doxygen*) is available in the public header "valhalla.h".

4.1 Initialization

TODO

4.2 Running

TODO

4.3 Events

TODO

4.4 Selections on the database

TODO

4.5 External metadata

TODO

4.6 Uninitialization

TODO