Table of Contents

1. Overview

1.1 Abstract

This document will give you a overview about the how OpenSync and Multisync work and how they are related. It will also explain give some explanations about the technical details of the syncengine and how OpenSync and Multisync can be used.

1.2 History

Multisync was originally started by Bo Lincoln. It was intended as a matter to synchronize various handhelds, cellulars and PIM application. Its focus was entirely on PIM data (contacts, calendar and todo). Multisync was a single application with dependencies on various gnome and gtk libraries. The connection to the devices was handled by plugins, which were loaded at the start of multisync.

However it became apperant soon that this approach was not flexible enough, since some people wanted to use multisync without a gui, synchronize data besides PIM data etc. At this time a new branch of multisync was created, labeled internally multisync-0.9 (as opposed to the currently stable 0.8X branch) which is supposed to offer these new features.

1.3 The Big Picture

Synchronization consist of several different parts: Access to the data on the devices/applications, which can be seperated into connecting/disconnecting, reading the data (only the changes since the last sync and all objects for slow-sync), writing data and some more “maintenance” stuff.

The next part is the LUID mapping, which describes which object on device A belongs to which object on device B, so that if one object gets changed/deleted, the correct corresponding object gets updated.

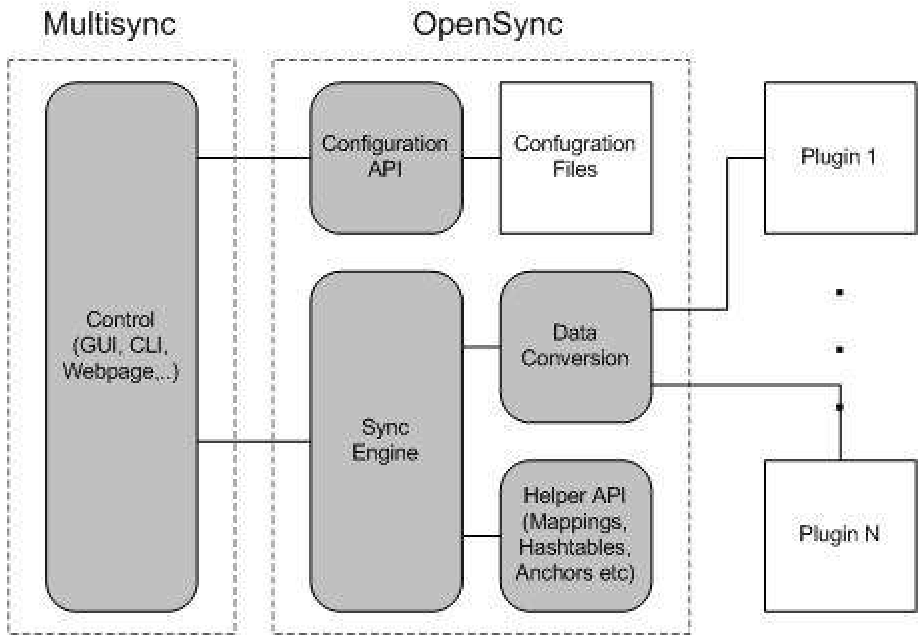
Sometimes the devices report objects in different formats (a contact could be stored in a vcard as well as some other format). To be able to synchronize, these formats have to be converted.

There is of course the syncengine itself, which takes care of the synchronization, conflict handling etc.

And there needs to some sort of UI (user interface) that presents everything to the user.

These parts are seperated among OpenSync and Multisync:

- OpenSync implements to low level functions, like sync plugins that can be used for connecting to devices, format conversion, then syncengine etc. It also provides some helper functions.
- Multisync which implements the User Interfaces like a normal GUI, a CLI etc



1.3.1 OpenSync

OpenSync implements the low level functions of the synchronization, which include the syncengine, conversion system, mapping tables, hashtables, anchor storage, configuration API etc.
The idea behind providing this framework is to make it possible for other developers of applications that are in need of synchronization to reuse the OpenSync framework and save work and get instant access to the available plugins. Another advantage for developers is that the can use the OpenSync Plugin API standard to use different functions from plugins to access devices and applications in a uniform way.
The advantage for the user is that once more applications start to use the framework he will be able to reuse SyncGroups he configured between these different applications since the configuration is stored in OpenSync.

1.3.2 Multisync

Multisync is one application that uses OpenSync to provide device synchronization to the user. It includes a Graphical UI and a Command Line Client.

2. Technical Details

2.1 Sync Plugins

A Sync Plugin is a module that provides access to a certain device / application / protocol. The basic functions that it needs to provide are:

- A "initialize" function. In this function, the plugin has to malloc a struct it needs to track its internal state, load its configuration and start the listening server if it has one. The return value is the pointer to the struct.
  Example: `static void *fs_initialize(OSyncMember *member)`
- A "finalize" function which stops a listening server and frees the malloced struct
  Example: `static void fs_finalize(void *data)`

These 2 functions are the only ones that are called synchronously on the plugins. All the other functions are called asynchronous. To be able to track them correctly, they get passed a OsyncContext struct, which they can use to answer. It is not important at what time the following functions return, the only important thing is that they use one of the osync_context_report_* functions. Since each plugin runs in its own thread, they may block as long as they want.

- The "connect" function which is called in the beginning of the synchronization. Here the plugin should connect to the device, open anything it needs etc.
  Example: `static void fs_connect(OSyncContext *ctx)`
- The "disconnect" function which is called to disconnect.
  Example: `static void fs_disconnect(OSyncContext *ctx)`
- The "get_changes" function. It is used by the syncengine to request the changes or all (in the case of slow-sync) objects from a device.
  Example: `static void fs_get_changeinfo(OSyncContext *ctx)`
- The "commit_change" function. This function is called once for each object that the engine want to write to the plugin (the second parameter).
  Example: `static osync_bool fs_commit_change(OSyncContext *ctx, OSyncChange *change)`
- The "sync_done" function which is called once all objects have been sent to the plugin. It is only called if the sync was successful (some commits may still have failed due to access or conversion errors)
  Example: `static void fs_sync_done(OSyncContext *ctx)`

There is another special function on each plugin: The “get_info” function. This is the only function on the plugin that is actually read via dlsym. It will get passed a OsyncPluginInfo struct which the plugin has to fill with values. The things it has to set there are (among others): its name, version, the pointers to the above mentioned functions and the objecttype and formats it accepts.

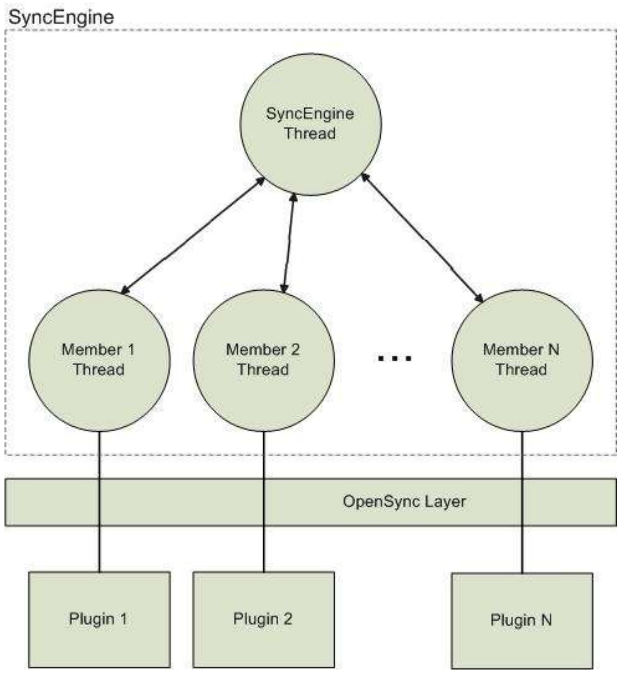
```
void get_info(OSyncPluginInfo *info) {
        info->name = "file-sync";
        ...
        info->functions.initialize = fs_initialize;
        info->functions.connect = fs_connect;
        info->functions.sync_done = fs_sync_done;
        info->functions.disconnect = fs_disconnect;
        info->functions.finalize = fs_finalize;
        info->functions.get_changeinfo = fs_get_changeinfo;
        ...
        osync_plugin_accept_objtype(info, "data");
        osync_plugin_accept_objformat(info, "data", "file");
        osync_plugin_set_commit_objformat(info, "data", "file",
fs_commit_change);
        ...
}
```

2.2 The Sync Engine

The SyncEngine is responsible for deciding what exactly needs to be done to synchronize the connected device. This includes initialization, connection, reading and writing changes, LUID mapping and keeping the log. It utilizes the helper functions provided by OpenSync.

2.2.1 Thread Model

The SyncEngine is completely multithreaded. One thread is responsible for the synchronization itself. For each member in the SyncGroup, a new thread is spawned also, so that all member may access their devices at the same time and block. The communication between the different threads is handled via asynchronous message queue on which a message bus has been implemented that supports messages with answers, payloads and timeouts.



2.2.2 Conflict Detection

There are different types of conflicts that might occur:

- Differences in the changetype. For example, a contact could have been deleted on one side, but modified on the other side.
- Differences in data. The data of 2 sides could have been altered in different ways.

Each time a conflict is detected, the syncengine calls a callback handler set by the controling application. This function gets passed a pointer the mapping with the conflict.

```
void (* conflict_function) (MSyncEngine *, OSyncMapping *)
```

Each time this callback function gets called, the user has to resolve the conflict.

2.2.3 Resolving Conflict

The user can decide in different ways how this conflict should be handled. The first solution is to pick a winning side, whose change overwrites the other sides.

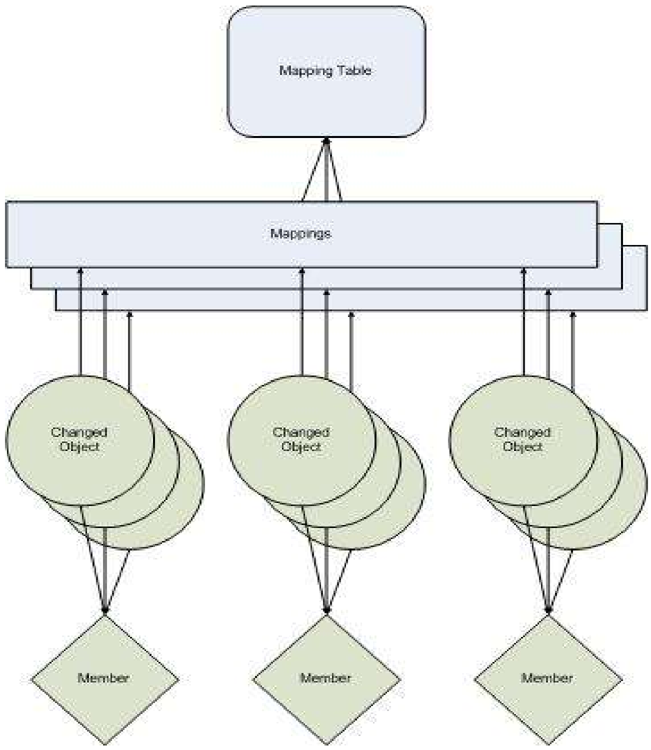
```
osync_mapping_set_masterentry(mapping, change);
```

The second solution is to keep all changes as seperate entries (Duplication).
Another possibility is that the user wants his choice to be valid for all future conflicts (e.g. “Duplicate all conflicts”).

```
msync_mapping_duplicate(engine, mapping);
```

2.3 LUID Mapping

The LUID mapping is one functions that is provided optionally by OpenSync. It can be used to save / load information about objects and their mappings. The changes are saved in a database in the currently set configuration directory of the syncgroup. The things that are stored away are: the LUID, the objtype, the format, the id of the member which reported this object, the id of the mapping.
This stored information is read in the beginnning and can be used to map objects from different sides.
The functions provided are: Functions for creating/deleting mappings, adding/removing objects from a mapping, searching for objects in a mapping by the member that reported it, and searching for a object on a member by uid.



2.4 Format Conversion

A lot of times it happens that different devices report the same object in different formats (a contact could be encoded in a vcard, a palm format etc). To be able to synchronize one needs to be able to convert the formats.
Each object has 2 attribute types assigned to it:

- The objecttype. This attribute classifies the abstract information the object carries. It does not say anything about the format in which the object is represented. One example for a objecttype would be "Contact".
- The objectformat. This attribute shows in which format the objecttype is

currently encoded. One example for such a format for the objecttype “Contact” would be “vcard”. The formats are stackable which means that you can wrap one format into another format (which is referred to as “capsulation”). The “Contact” object can have its information formated in a “vcard” format, which is then wraped up into the “file” format (which adds information about the modes, owners etc of the file).

2.4.1 Plugins

To be able to support almost any objecttype / format we choose to implemt the conversion system with plugins. These plugins work similar to SyncPlugins. They have a “get_info” function, in which they register the objecttypes / formats they support. This function gets a pointer to a “OsyncFormatEnv” struct which represent the environment in which to register.

```
void get_info(OSyncFormatEnv *env)
{
	OSyncObjType *type = osync_conv_register_objtype(env, "data");
	...
	OSyncObjFormat *format = osync_conv_register_objformat(type, "file");
	osync_conv_format_set_compare_func(format, compare_file);
	osync_conv_format_set_detect_func(format, detect_file);
	osync_conv_format_set_duplicate_func(format, duplicate_file);
	...
	osync_conv_register_converter(type, CONVERTER_CONV, "file", "vcard", 
conv_file_to_vcard);
	osync_conv_register_converter(type, CONVERTER_CONV, "vcard", "file", 
conv_vcard_to_file);
}
```

2.4.2 Converters

One thing a plugin can register is a converter. A converter converts the data of a object to another format, therefore replacing the current format with another. The converter gets passed a pointer to some data. It then has to parse this data and return a pointer to a newly alloced struct with the data in the new format.

2.4.3 Encapsulators / Desencapsulators

A desencapsulator takes the data it gets passed and just removes the current format layer, therefore only return the data of the object. The file desencapsulator for example removes all information about the mode, owner etc of the file and just returns the content of the file.

2.4.4 Detectors

At some point during the conversion, someone has to label the object has belonging to a certain objecttype / format. This can be done during reading the change (when you connect to the evolution addressbook you know that you will always get “Contacts” of format “vcard”. But some plugins cannot do this (the file-sync plugin for example has no idea of the file is actually some data or a saved “vcard”). Therefore we have data detectors that try to parse and identify the input they get. (The “vcard” detector for example looks of it finds the “BEGIN:VCARD” in the data).

2.4.5 Conversion

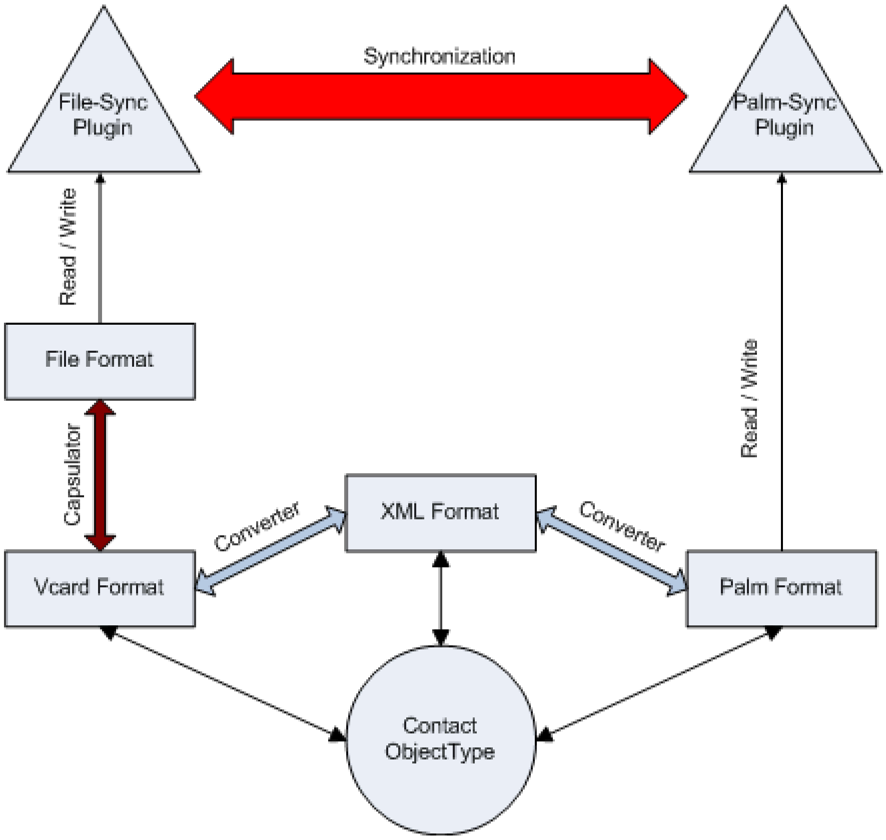
This section will now explain how a format conversion would look like.

There are 2 occasions where a conversion might be necessery:
- To compare the data of objects to find conflicts
- To commit the data to another side

For the first case the data has to be converted to a “common” format to compare, which might or might not be the format that the member understands. For the second case the data always has to be converted to a format that the member can accept. These formats are set via the accept_objtype and accept_objformat functions in the “get_info” function of the plugin.

Sometimes it not directly possible to convert to a given format since no direct converter exists but a detour over another format has to be taken. This is why the conversion “path” is detected using a Shortest Path algorithm. It will always try to convert with as little conversions as possible. One special thing in detecting this conversion path are the encapsulators. Since desencapsulating always is a loss of information, the path is search without the desencapsulators first. If no path is found the desencapsulators are used.

The next example shows how a conversion environment for a synchronization between a palm and a file-sync plugin could look like. Here also a intermediate XML Format is used.



2.5 Hashtables

Hashtables are a optional feature of OpenSync. They can be used by the plugins to detect changes in the objects. A lot of devices do not implement a change database which is used to track if a object has been synchronized. For these devices the hashtable can be used.

The plugin has to compute a hash for each object on the plugin (which might be a timestamp or something like a md5). Each time the engine queries for the changes, the plugin uses the hashtables function to compare the old hash with the new hash. If they are not the same, the plugin knows what has changed and can report this change to the syncengine.

A example usage of the hashtable could look like this:

- Initial Sync:
  Hashtable compares new timestamp "1103060445-110306044" with old timestamp NULL and reports the change as being "Added".
  Hashtable updates to current hash in the database to "1103060445-110306044"
- Now the object gets modified
- Second Sync:

Hashtable compares new timestamp “1103061332-110306123” with old timestamp “1103060445-110306044” and reports the change as being “Modified”.
Hashtable updates to current hash in the database to “1103061332-110306123”

2.6 Anchors

OpenSync also provides a “anchor” storage. A anchor is some kind of data that is stored on the device and is updated during each synchronization. The anchor is also stored locally using the functions provided by OpenSync. If the anchors do not match the next time a sync is initiated, a SlowSync is requested-

2.7 SlowSync

If a client detects that its database has been reset or the last sync was not successful, a SlowSync is performed, where each client sends all its objects (as opposed to just the changed objects) to the syncengine. This is also used for the initial sync.

2.8 Initial Mapping

If a SyncGroup has never been synchronized before, it needs to detect the initial mappings on the first sync. It might happen that the devices where synchronized before using another application and therefore might contain the same objects on both sides (or the user added the same object to both sides by hand). The intial mapping happens like this:
All unmapped objects from both sides are compared. The comparison might return 3 answers:

- MISMATCH: The objects were different. The search continues until there are no unmapped objects left on the other side. In this case the object is being reported as “Added” and will propagate to the other sync during the sync.
- SIMILAR: The objects were not exactly the same but some key properties are the same (like the name for a vcard). In this case a mapping is generated for the objects and a conflict is raised. If the connection between these objects where not correct (there might be contacts with the same name but different persons), the user can still choose to keep both object as seperate entries (Duplication).
- SAME: The objects were the same. A mapping is made for the objects and no conflict is raised.

2.9 Sync Alerts

Some plugins support SyncAlerts. A SyncAlert is when the plugin has the capability to detect that a object has just being changed and reports this to the engine which can then decide to initiate a synchronization.

2.10 Filters

OpenSync supports filtering objects based on certain criterias. The criterias currently supported are filtering on the“objecttype” and on the member.

If you filter a objecttype you can request to enable or disable a certain object completely. If it is disabled no object of this objecttype will be read or written.

You can filter the object from a certain member by enabling “read-only” or “write-only” on this member.

More filters will be supported in the future (also custom filters).

3. Usage

This section will give short examples on how to use OpenSync and Multisync with commented code pieces.

3.1 OpenSync

3.1.1 Listing Plugins

```
int i;
OSyncPlugin *plugin;

/* This will malloc a new OsyncEnv (for
OsyncEnvironment) struct, which holds the
information about available groups, plugins etc
*/
OSyncEnv *osync = osync_env_new();
/* The next command will init the environment
(load the sync plugins and format plugin) */
osync_init(osync);

/* Now we can iterate over the available plugins
and print their names */
for (i = 0; i < osync_env_num_plugins(osync);
i++)
{
     plugin = osync_env_get_nth_plugin(osync, i);
     printf("%s\n", osync_plugin_get_name
(plugin));
}
```

3.1.2 Listing Groups

```
int i;
OSyncGroup *group;

OSyncEnv *osync = osync_env_new();
osync_init(osync);

/* The next command will load the available
SyncGroups */
osync_env_load_groups_dir(osync);

/* Now we can iterate over the available groups
and print their names */
for (i = 0; i < osync_num_groups(osync); i++)
{
     group = osync_get_nth_group(osync, i);
     printf("%s\n", osync_group_get_name(group));
}
```

3.1.3 Configuring a member

```
char *data = NULL;
int size = 0;
/*The next command will read the config data for
this data and return a pointer and size */
if (!osync_member_get_config(member, &data,
&size))
{
     printf("Unable to get configdata\n”);
     return;
}

/* Now we can manipulate the config data, display
it to the user etc */

/* After it has been changed, we can write the
changed config data back to the member */
osync_member_set_config(member, data, size);
g_free(data);
```

3.2 Multisync

3.2.1 Synchronous Synchronization

```
/* This will create a new syncengine that can be
used to sync a group. The group must have been
added / loaded before */
MSyncEngine *engine = msync_engine_new(group);

/* This function will init the engine (and call
the “initialize” function on all members). */
msync_engine_init(engine);

/* Start the synchronization! */
msync_engine_synchronize(engine);

/* This function will return when the
synchronization finishes */
msync_engine_wait_sync_end(engine);

/* After we are done we finalize the engine to
clean it up and stop all threads */
msync_engine_finalize(engine);
```

3.2.2 Asynchronous Synchronization

```
MSyncEngine *engine = msync_engine_new(group);

/* Now we set the callback functions that the
engine uses to report to us about its status */
msync_engine_set_memberstatus_callback(engine,
member_status);
msync_engine_set_changestatus_callback(engine,
entry_status);
msync_engine_set_enginestatus_callback(engine,
engine_status);
msync_engine_set_mappingstatus_callback(engine,
mapping_status);
msync_engine_set_conflict_callback(engine,
conflict_handler);

msync_engine_init(engine);

...
/*When the user presses synchronize */
msync_engine_synchronize(engine);

/* Now the callback functions get the information
about reported objects, written objects, errors,
sync end etc */

...
/* When the application quits */
msync_engine_finalize(engine);
```

4. Glossary

LUID

The LUID (Local unique identifier) uniquely identifies a object on a member. Most of the time it is only valid and unique on one member.

SyncGroup

A SyncGroup is a group of members which are supposed to be synchronized. This group can contain any number of members.

Member

A member is part of a SyncGroup. It is a configured instance of a certain

SyncPlugin.

SyncPlugin

A SyncPlugin has the code that is needed to connect to a certain device / application /server.

FormatPlugin

A FormatPlugin has the code that describes a certain format ("vcard" for example) and provides functions for manipulating this format.