Funambol Client API C++ Design Document

Last modified: January 3, 2011

Revision History

Name	Date	Reason for Change
Teo	06/03/09	Added FolderData and MailAccount
Toccy	10/16/09	Added LargeObject Handling (section 3.4)
Toccy	10/29/09	Updated File/MediaSyncSource chapter

Table of Contents

1. Overview

The Funambol SyncML Client API is a C++ programming interface that application developers use in order to take advantage of the powerful data synchronization and device management features provided by the Funambol platform.

This document presents the design principles and patterns behind the Funambol Client API. The intended audience of this document is the Funambol development team and anyone interested in knowing the ins and outs of the Funambol Client API.

1.1. Document Conventions

The diagrams used in this document are inspired to the UML sequence and class diagrams, but with some simplification. The conventions used by the diagrams are described in the following sections.

Sequence Diagrams

- Each entity is represented as a box;
- a box can represent a class, an instance, an interface or even a conceptual entity; the real meaning depends by the context;
- solid arrows represents method or function calls;
- dashed arrows represent some sort of communication between two entities; it is intended that the communication mechanism is left unspecified or is not important or it is at a different abstraction layer.

Class Diagrams

- Each class is represented as a box;
- data members and methods are separated by an horizontal line;
- plain titles represent classes, italicized titles represent interface (abstract classes);
- a + next to a method or data member name means “public”
- a - next to a method or data member name means “private”
- a * next to a method or data member name means “protected”;
- a > next to a data member name means it is a property with get/set accessors;
- inheritance is represented by an arrow pointing to the base class;
- italicized methods names represent abstract method.

1.2. Funambol Client API Architecture

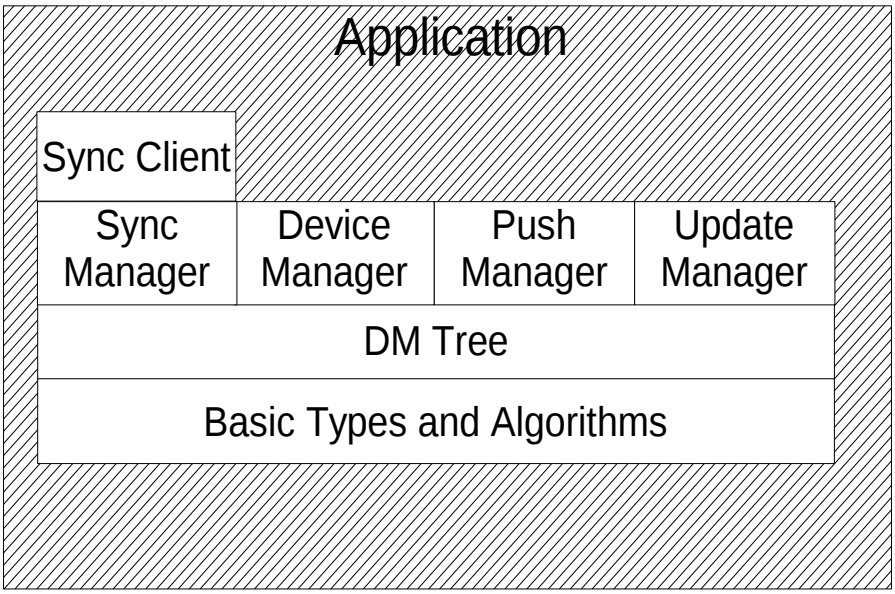


Figure 1: Funambol Client API architecture

The Funambol Client API has different modules:

- "Sync" which implements the OMA DS protocol stack and data synchronization
- "Device manager" which is responsible for device and application configuration management. It is foreseen as the implementation OMA-DM protocol.
- "Push" which implements push protocols and is capable of notifying the upper layers on push notification events
- "Update" which implements a protocol for checking the availability of new software versions
- "Sync Client" is a convenient module that can be used to easily implement standard synchronization clients.
- "DMTree" represents a tree of configuration information organized as key/value stored in nodes where the node name is the key. The DMTree can be used by a DM server for remote configuration management.
- "Basic Types and Algorithms" is mainly an abstraction layer used to make the library portable. This module also includes utility functions that are used across the library.

A final application is developed on top of the Funambol Client API and can access the services provided by all modules (and not only the top level ones). The Device Manager can also be used to transparently manage the device remotely by a OMA DM server.

The device manager abstracts the access to a *configuration repository*, represented in the picture with the DM Tree block. How the configuration repository is implemented on a particular device is platform specific and the API shall provide a concrete implementation of DeviceManager for any supported platform. However, the DeviceManager exposes such configuration parameters as a tree-like structure, where nodes represent containers for other

nodes and leafs, and leafs represent the actual configuration parameters. This tree is called “Management Tree” and follows the description found in [2].
The final goal of this module is to provide the remote management functionality using the OMA DM protocol, even if this is currently not implemented yet in the library.

The SyncManager, the PushManager and the UpdateManager need several configuration parameters to perform their job, which are passed using interface classes called SyncManagerConfig, PushConfig and UpdateConfig. These parameters can be stored using the DM, which will also allow the remote configuration once implemented, or using a client-specific method. See par. 3.6 for more details.

The final application will also likely need to access a local data repository. This is done through a so called *Sync Source;* this is an application module used by the Sync Manager to interact with the application data. The way the Sync Source accesses the local data repository is application or device specific and hidden to the synchronization engine by the Sync Source interface.

1.3. Synchronization Overview

A client application interacts mainly with two entities of the Funambol Client API: the SyncClient and the SyncSource. The SyncClient is the component that handles all the communication and protocol stuffs. It hides the complexity of the synchronization process providing a simple interface to the client application. A SyncSource represents the collection of items stored in the local repository. It contains the client logic to discover the items to send to the server and to store the ones obtained from the server. The client feeds a SyncSource with the items changed on the client side, whilst the SyncManager feeds it with the items received by the server.

The synchronization process is logically a sequence of three phases (see [4]):

1. Initialization
2. Modifications exchange
3. Ending

In the initialization phase the client sends its credentials and which database to synchronize (along with the desired synchronization type) to the server. The server responds with the authentication status and the synchronization type to perform.

After the initialization phase, first the client sends all client-side modifications and receives the status of the execution of the commands on the server; then, the server sends server-side modifications. The client applies the changes and sends the proper status to the server. In the case the server issued a new item to the client, the latter will create a new local id for it and therefore needs to communicate such new key to the server (see LUID-GUID mapping in [4]).

The ending phase is actually just sending the last status and/or mapping to the server and saving the anchors needed to perform a fast sync the next time.

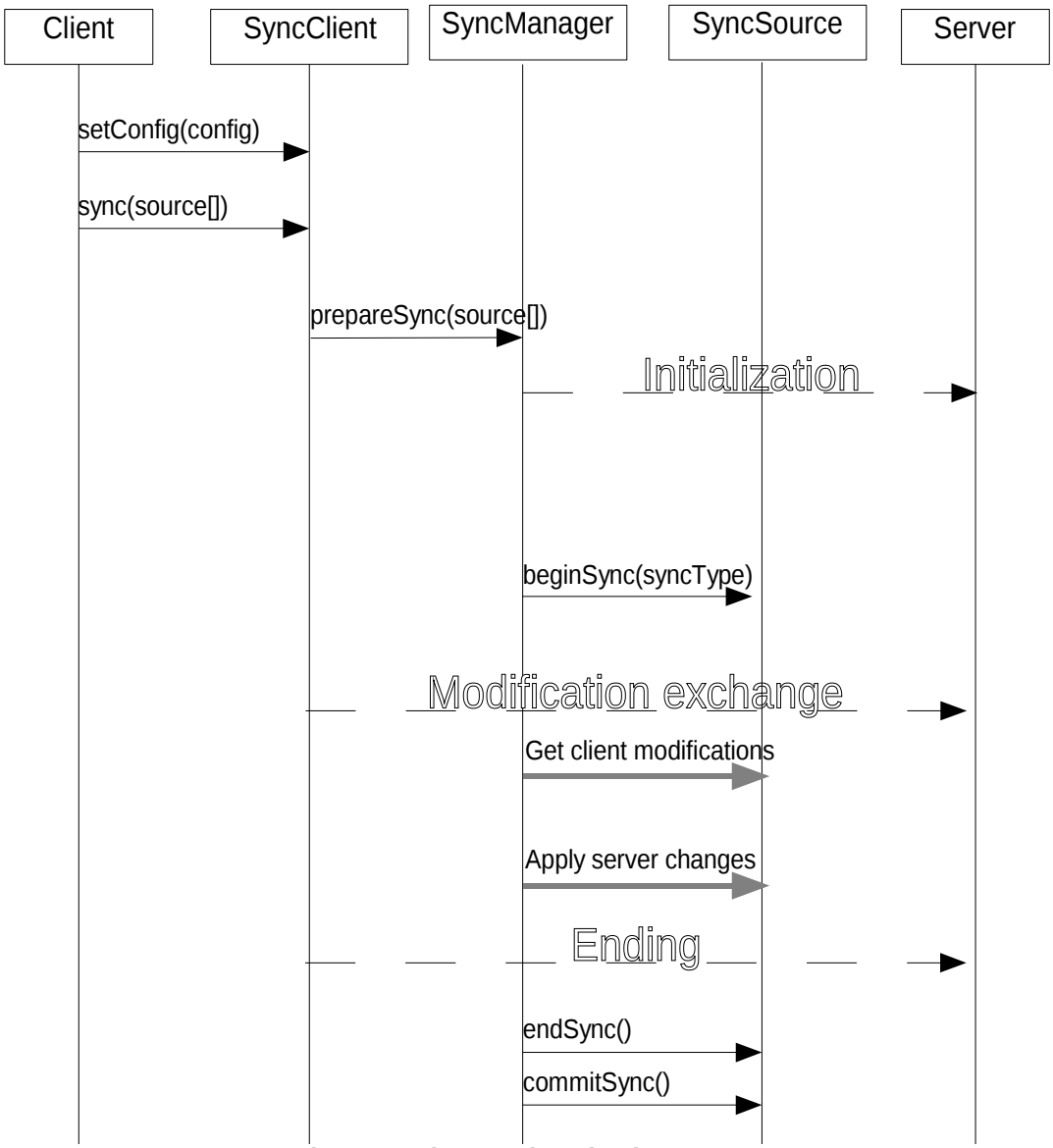


Figure 2: The synchronization process

The Synchronization Process

The entities in the sequence diagram of Figure 2 have the following meaning:

- Client: a final client application using the Funambol SDK;
- SyncClient: the SDK client kernel that the developer extends in order to interact with the client data sources;
- SyncManager: the SDK synchronization engine;
- SyncSource: the client abstraction of a client data source;
- Server: the SyncML server

As shown in the figure, the client application just kicks off the synchronization process giving an array of syncsources to synchronize. SyncClient takes control of the synchronization

process, coordinating the SyncManager and handling its return values. The macro synchronization initialization, modification exchange and mapping involve the interaction with the server and include the required SyncML messages exchange. They will be described later in dedicated sections.

1.4. Device Manager and DMTree

Goal of the device management module is to allow an easy management of the device by a remote operator. This means that a remote agent can navigate, view and change device and applications configuration and that those changes are picked up by the application, possibly requiring minimal or no user action.

In order to provide device management functionality, the Funambol client framework provides a single, shared configuration tree, called *Device Management Tree* or *DMTree*. This registry is easily accessible by client applications and hides the details of the physical storage where configuration settings are stored (it could be an SQL database, a device datastore, an XML file or even the device memory for run-time configuration values). For a detailed explanation of the role and structure of the DMTree see [2].

The device management module is abstracted in the Funambol client API by the DeviceManager interface. This is an abstract interface that must be implemented for each supported device. The current design of the Funambol client API provides the following implementations:

- Win32 – registry
- WindowsMobile – registry
- Posix (including MacOS) and Symbian– plain files tree

The DM subsystem is fully described in chapter 3.

2. Basic Types and Algorithms

This chapter describes the layer used by the API to guarantee portability across different platforms. Beside this main goal, this layer also provides common functions and algorithm used in different parts of the APIs.

The module provides the following abstractions:

- basic containers (such as ArrayList and KeyValueStore)
- strings
- OS concepts (such as time, thread, socket, network connection)
- cryptography algorithms (such as base64, des, md5)
- logger

The following sections provide a brief description of each component.

2.1. Basic Containers

This component defines basic type to store/retrieve items in ordered way. Its purpose is to provide a common set of objects that can be used on any platform. STL could be an alternative to this component, but the library followed a different design. In particular the library works on platforms (such as Symbian) where STL is not officially supported. Therefore it was decided to implement a Funambol containers library.
At the moment this library does not use templates and parts of its implementation are common to all platforms. A couple of possible changes are foreseen:

- introduce the use of templates
- make it a light weight implementation. On platform where STL is available, this component can simply wrap STL objects and be really simple (thus there would be less common components and more platform specific ones).

In terms of functionalities the library provide the following abstractions:

- ArrayList: is a list that can be accessed in random way. Since the library is not template based, a type can be stored in an ArrayList if it implements ArrayElement interface.
- KeyValueStore: is a container for pairs of key/value. This interface is intended to be sub-classed and implemented in different ways. For example there can be an implementation which is based on files and ArrayList (PropertyFile) or implementations based on SQLite and so on.
- KeyValuePair: a simple pair of key and value StrinigBuffer.
- StringMap: is a simple associative array to store pair of StringBuffer, one used as key and unique in the container, the other as value.

2.2. Strings

Strings are implemented by a single class “StringBuffer” (*TODO: this is true as long as we get rid of WCHAR*). StringBuffer allows the manipulation of an internal buffer which is manipulated by the class itself. The buffer is not externally modifiable, but it is made available as a constant buffer. Various operations are supported to modify the string, compare and so on. The implementation of this class is common and it is not intended to rely on system specific strings.

2.3. OS abstraction

The library must be portable to a wide variety of platforms, including standard desktop environments (such as Windows and *nix) and more exotic embedded systems (such as Symbian or the iPhone).
Because of this it is important to abstract some important OS concepts, in a way that they can be mapped on any system.

Platform Adapter

To have a portable way to access platform specific resources and pathnames, in the release 7.1 of the API has been introduced the concept of the PlatformAdapter, which must be initialized prior to any use of the library, with the **application context**, which is a unique string identifying the application. This string is used to compose the config path used by DMTClientConfig and other classes. For example, if you call:

PlatformAdapter::init("aVendor/theApp");

you will have the config stored under: ~/.config/aVendor/theApp on posix, or under Software/aVendor/theApp key in the Windows registry, and so on.

The same path is used for the different temporary files created during the sync, stored in the same subtree of the config for posix, or under Application Data/aVendor/theApp on Windows.

Now, for clients is important that this values is set once and never changed during the application lifetime, otherwise the results of having the config path changing is not predictable.

For this reason, the init() method is protected against a double call, and just logs an error if called twice.
However it is possible to force multiple initializations by setting the *force* flag to true. For instance it's used by the unit test framework, which needs to create multiple contexts in the same application. This option is disabled by default.

Time abstraction

Time is abstracted the same way *nix systems do. There must be a “time” function which returns the current time in Unix format (milliseconds elapsed since Jan 1st 1970).

Http Connections

Http connections are handled by a ***HttpConnection*** class, which is capable of performing http requests on secure/insecure channels.
The class *AbstractHttpConnection* is the abstract interface, concrete implementations of *HttpConnection* exist for each platform.

The platform dependent implementation of this class must reimplement the *open*, *request* and *close* methods, with the system calls provided by the underlying system to send HTTP requests.
The classes that use *HttpConnection* (like the *TransportAgent* and the *HttpUploader*) should first open the connection via the method *open*, specify the request method, and then send the request calling *request* method. The response is returned in the outputStream passed. If the request method is not specified, the default is POST.

The ***TranportAgent*** class uses the *HttpConnection* (**TODO!**) and is at the basis of the synchronization process with the C++ APIs. It's used by *SyncManager* to send SyncML messages over the wire(less) network and to retrieve the server responses.
The *TranportAgent* is a common class for all platforms, since the http connection abstraction is wrapped in the *HttpConnection* class. It defines a retry mechanism (in case of network error) and proper behavior for the status/errors returned.
Its interface is the following:

GPRS connection abstraction (TODO)

At a lower level than the *HttpConnection*, the ***GPRSConnection*** has the role of a generic interface to estabilish GPRS/UMTS connection, check if the client is connected, get connection informations, and eventually drop the connection.
It does not need an implementation on all platforms (i.e. on win32 it's useless), but it has a key role on other platforms, like Symbian, where the GPRS connection is handled at a client level, and so it's really important to make sure all the threads/processes share the same GPRS connection.

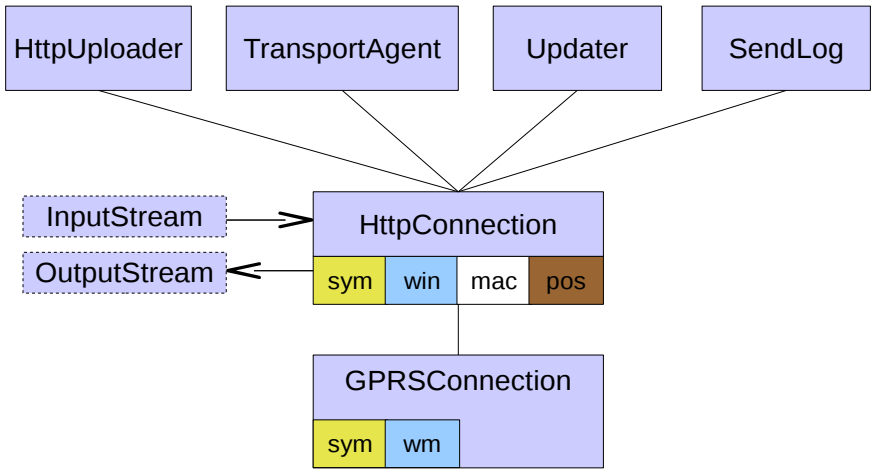


Figure 3: Http and GPRS Connection abstractions

Thread abstraction

This class is not intended as a general replacement for threads everywhere. Is is meant to provide the set of functionalities that are needed by the library. Clients should still use the platform specific threads whenever they do not need to be portable.
A Thread is abstracted by FThread, a class whose most important methods are listed in the following table.
FThread allows the creation and manipulation of threads. One controversial feature that is not offered by FThread is the ability to terminate a thread. On some platforms this feature is

either unavailable or deprecated. Because of this, FThread has a method for soft termination that does not guarantee the thread is actually terminated. It is just a suggestion for the thread to terminate. Each thread must then implement its own mechanism for cooperative termination. This is similar the the Java approach to the problem.
FThread is an abstract interface, on each platform we will have an implementation that maps abstract threads on platform threads. This implies that we have as many FThread implementation as supported platforms. The class is abstract because the user is forced to define its own `run` method.

Socket abstraction

This class is not intended as a general replacement for sockets everywhere. Is is meant to provide the set of functionalities that are needed by the library. Clients should still use the platform specific sockets whenever they do not need to be portable.
The design philosophy behind Fsocket and FServerSocket is to keep it as simple as possible, providing only the functionalities which are necessary to the library.

FSocket provides methods for creating client sockets, while FServerSocket provides a method to create server sockets (TO BE IMPLEMENTED). These methods act like a sockets factory. The abstraction of socket is very simple. It provides a pipe where data can be written and read. All read/write operations are synchronous and block the caller.

Cryptography

This component provides functions to encode/decode data. In particular it supports the following formats:

- base64 (encoding/decoding)
- des (encoding/decoding)
- md5 (encoding)

Logging

Logging is performed by a single class with the interface specified by the table below.

Method	*Description*
Log(BOOL reset = FALSE)	Creates a new Log instance. If reset is TRUE, the existing content is discarded, otherwise new messages are appended.
~Log()	Destructor.
void error(const wchar_t* msg)	Appends the given error message.
void info(const wchar_t* msg)	Appends the given info message.
void debug(const wchar_t* msg)	Appends the given debug message.
void trace(const wchar_t* msg)	Traces the given message. Tracing is different from logging since tracing messages usually go to a dedicated device provided by the device application toolkit or development environment.
void reset()	Reset the current log.
void setLevel(LogLevel level)	Sets the current logging level to the given level.
LogLevel getLogLevel()	Returns the current logging level.
BOOL isLoggable(LogLevel level)	Checks if the given logging level is currently visible in the log.

LogLevel is an enumeration define as follows:

```
typedef enum {
   LOG_LEVEL_NONE  = 0,
   LOG_LEVEL_INFO  = 1,
   LOG_LEVEL_DEBUG = 2
} LogLevel;
```

Logging levels are in a hierarchical relationship: only the messages logged at a lower or equal level than the current logging level really will go into the log. For example, if the logging level is set to none, no messages but errors will go in the log; if the logging level is DEBUG, all messages will go in the log because debug and info messages are at a level <= DEBUG. Finally, if the logging level is INFO, only info messages will go into the log.

The Log is accessed by a globally scoped instance exported to all modules. Plus, a wchar_t logmsg[] is provided to make it easier the creation of messages.

The Log class is platform specif since different devices may use different logging facility. For example a Win32 based API can use just a file as logging media, while a Palm device will use a palm database.

3. Data Synchronization Layer

This chapter describe the design of the Funambol Client API Data Synchronization Layer. The Data Synchronization process implemented in the Funambol API follows the OMA DS 1.2 specification (see [4] and related documents).

3.1. Synchronization Process

The synchronization process has been briefly introduced in the synchronization overview. In the following sections, the synchronization process is described and designed in more details.

Initialization

As per the SyncML specification, the initialization phase can be performed in two ways:

1. As a separate package
2. Together with the modifications exchange package

The Funambol Client API implements “separate initialization” only, which is the option that optimizes at best network usage in the most common cases. In fact, with a synchronization without a separate initialization, there is the risk to start a potentially long synchronization, while the server would refuse the sync (for example because it does not authorize the client). Performing separate initialization avoids this issue with a minimal impact on network traffic.

During initialization three important tasks are performed:

1. client authentication
2. server authentication
3. database alerting

Client Authentication

This section covers how the client sends its authentication credentials to the server and how the authentication process goes.

The SyncML specifications mandate that client implementations must support at least *basic* and *MD5 Digest* authentication. The Funambol Client API implements both.
Client authentication is delivered in the Cred element of the SyncML message header. Plus, even if the client sends in the first message its credentials in one of the supported types, the server can refuse it and challenge the client for a different kind of credentials. For example, if the client starts sending Basic credentials and the server requires MD5 authentication, the server responds with a 401 status to the client credentials and provides a Chal element with the requested authentication.

For additional information on Basic and MD5 authentication see section 2.5 of [4].

Basic authentication is identified by the URI syncml:auth-basic; it is pretty simple and it is very similar to what happens in WEB applications.
In this case, credential data represent are encoded as follows:

```
B64(username':'password)
```

Where B64() is a function that encodes the given string in Base 64. *username* and *password* represent the account's authentication information.

The MD5 Digest scheme is identified by the URI syncml:auth-md5. Let MD5(data) denote the result of applying the MD5 hash algorithm to “data”, the result is a 128-bit binary quantity. Let A be the concatenation of an authentication identifier as the originator’s userid, followed by the COLON (i.e., “:”) separator character, followed by some secret known by the originator and recipient such as the originator's password for the corresponding userid, for instance:

```
A=”Bruce1:OhBehave”
```

Let AD be defined as:

```
AD = MD5(A)
```

Let B64(data) denote the result of the base64 encoding algorithm applied to “data”. This authentication scheme is the MD5 digest form of the concatenation of B64(AD), followed by the COLON (i.e. “:”) separator character, followed by the recipient specified nonce string. The maximum duration that the nonce string can be used by the originator is the current SyncML session. Note that issuing a nonce does not constitute use – a nonce may be issued for use in the next session. More frequent changes to the nonce string can be specified with the NextNonce element type within the Meta element type of the Chal element type. The MD5 credential, a 128-bit binary digest value, MUST be Base64 character encoded when transferred as clear-text XML. For WBXML representation, the additional Base64 character encoding is not necessary (but still allowed).

Computation of the MD-5 Digest

The digest is computed as follows:

```
Let H = the MD5 Hashing function.
Let Digest = the output of the MD5 Hashing function.
Let B64 = the base64 encoding function.
Digest = H(B64(H(username':'password))':'nonce)
```

This computation allows the authenticator to authenticate without having knowledge of the password. The password is neither sent as part of the credentials, nor is it required to be known explicitly by the authenticator, since the authenticator need only store a pre-computed hash of the username:password string.

Password and Nonce Usage

The nonce value is recommended to be at least 128 bits (16 random octets) in length.

The nonce value is issued in a challenge from either the device or the server. In the case of the credentials being sent prior to a challenge being issued, then the last nonce used shall be reused. The authenticator must be aware that the issuer of the credentials may be using a stale nonce (that is to say, a nonce that is invalid due to some previous communications failure or a loss of data). Because of this, if authentication fails, one more challenge, along with the supply of a new nonce, must be made.

A new nonce is generated for each new session.

After being successfully used, a nonce obtained from the server must be stored persistently for a future use. Not that this must be done only if used successfully (to prevent nonce hacking).

Nonce Generation

It is important that nonces are generated randomically, so that it is difficult to try to hack a next nonce from a stolen nonce.
We will generate the nonce with the following algorithm:

```
void generateNonce(char nonce[16]) {
  srand ( time(NULL) ); // random number generation initialization

  for (unsigned int i = 0; i < 16; ++i) {
    nonce[i] = ((rand()%100) * (rand()%100))%100;

    if (nonce[i] < 32) {
        nonce[i] += 96;
    }
  }

  //
  // Note that the nonce won't be 0 terminated
  //
}
```

Server Authentication

This section covers how the client can optionally authenticate the server. Server authentication is mainly requested in the case of OMA DM more than OMA DS. However, since the C++ API is the base for both DM and DS, server authentication must be implemented too.

The mechanism behind server authentication is not far from what explained early. The only difference is that instead of authenticating a user id, the client authenticates a server id.

For basic authentication, the client expects that the server sends a Cred element with proper credential data. If the Cred element is missing, the client challenges the server for the requested authentication mechanism.

The given credentials must be in the form:

Basic	`B64(serverid':'password)`
MD5 Digest	`H(B64(H(serverid':'password))':'nonce)`

In order for the client to be able to check that the given credentials are correct, it needs to retrieve the expected server id and server password (and maybe nonce) from a permanent store. It is decided to use still the client DMTree as persistent store of such information, and in particular, we will use the SyncML DM management object OMA DM specification ([3]).

Authentication Algorithm

Client and server authentication are performed concurrently during initialization. The algorithm to do so is shown in Figure 4.

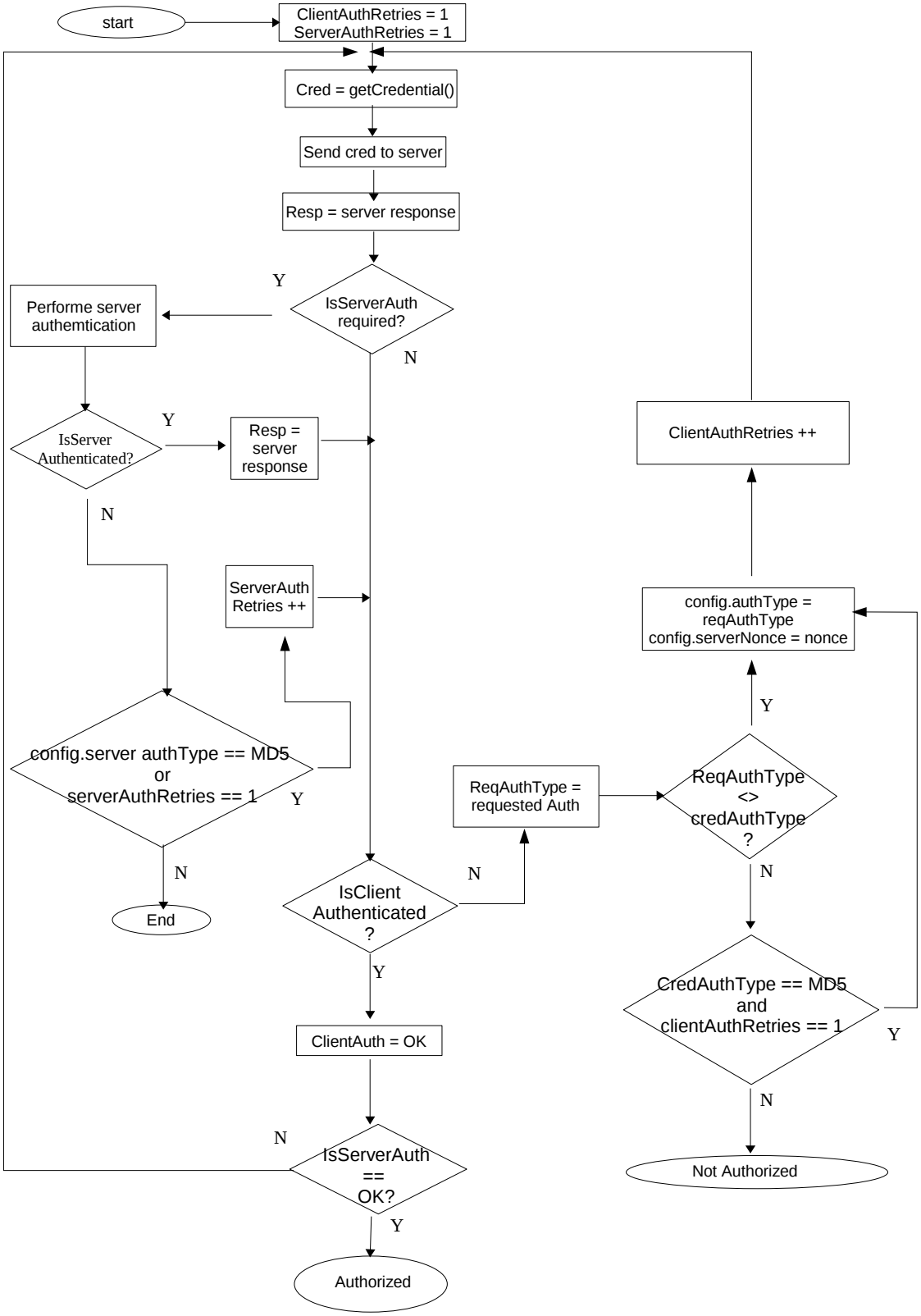


Figure 4: Authentication algorithm flowchart

Database Alerting

The second task performed during initialization is database alerting. Database alerting is the means the client requests to synchronize a particular database; plus, it specifies which type of synchronization should be performed. The SyncML specifications define the following synchronization types:

Sync Type	Description	Alert Code
Two-way	A normal sync type in which the client and the server exchange information about modified data in these devices. The client sends the modifications first.	200
Slow	A form of two-way sync in which all items are compared with each other on a field-by-field basis. In practise, this means that the client sends all its data from a database to the server and the server does the sync analysis (field-by-field) for this data and the data in the server.	201
One-way from client only	A sync type in which the client sends its modifications to the server but the server does not send its modifications back to the client.	202
Refresh from client only	A sync type in which the client sends all its data from a database to the server (i.e., exports). The server is expected to replace all data in the target database with the data sent by the client.	203
One-way from server only	A sync type in which the client gets all modifications from the server but the client does not send its modifications to the server.	204
Refresh from server only	A sync type in which the server sends all its data from a database to the client. The client is expected to replace all data in the target database with the data sent by the server.	205
Server alerted	A sync type in which the server to alerts the client to perform sync. That is, the server informs the client to starts a specific type of sync with the server.	207
Smart one-way from client only	Funambol extension. Like “One-way from client only”, with optimizations to avoid sending the same (big) item two times. It's used for Media items from mobile Clients to Server (see section 3.10) .	250
Smart one-way from server only	Funambol extension. Like “One-way from server only”, with optimizations to avoid sending the same (big) item two times.	251
Incremental smart one-way from client only	Funambol extension. Like “One-way from client only”, with optimizations to avoid sending the same (big) item two times. No deletes are sent.	252
Incremental smart one-way from server only	Funambol extension. Like “One-way from server only”, with optimizations to avoid sending the same (big) item two times. No deletes are sent. It's used for Media items from Server to desktop clients.	253

From the specifications perspective only Slow and Two-way sync types are mandatory. However, the Funambol C++ API supports issuing all the defined synchronization types.

The expected behavior for each synchronization type is faced more deeply in chapter ... .

Clients specifies that a local database shall be synchronized with a remote database with the SyncML Alert command. It is used like in the following example:

```
<Alert>
  <CmdID>1</CmdID>
  <Data>200</Data> <!-- 200 = TWO_WAY_ALERT -->
  <Item>
    <Target><LocURI>./rmtcontacts</LocURI></Target>
    <Source><LocURI>./lclcontacts</LocURI></Source>
    <Meta>
      <Anchor xmlns='syncml:metinf'>
        <Last>234</Last>
        <Next>276</Next>
      </Anchor>
    </Meta>
  </Item>
</Alert>
```

Target specifies the remote database; Source is the URI of the local database. The Anchor element is used to determine if the local and server database start from a well known state or if a slow sync must be performed instead. Refer to the SyncML specifications for details.

Note that the server can respond with a different Alert type for a given database. The client must always performs the synchronization type as given by the server.

Note that it is responsibility of the SyncManager to select the local databases to synchronize and thus to alert the server accordingly. This is done using the configuration information retrieved by the DMTree as explained in section 2.6.

Modifications Exchange

The modification phase is when client modifications are detected on the client and sent to the server who replies with the server side updates. These updates are applied to the local database. Accordingly to the SyncML specs, updates are exchanged with commands like Add, Replace, Delete, Copy, Move, etc.
However only Add, Replace and Delete are mandatory to SyncML implementations; the others are optional (see the specs for details).

Since the initialization phase is done separately, the modification exchange phase is carried on only if initialization completed successfully.

The real synchronization process is driven by the SyncManager. It uses the SyncSource interface in order to interact with a specific data source. The development of the SyncSource is delegated to the application developer.

The role of a SyncSource is manifold:

- provide information about the SyncSource to synchronize (preferred synchronization type, name, local and target URIs, mime type of the data...)
- keep the synchronization state from the datasource point of view
- retrieve all items in the datasource regardless their state
- retrieve the modified items only from the datasource
- apply server side changes
- commit changes

The interaction between the SyncManager and the SyncSource is the core of the synchronization process. Note that since the number of items in the local data source may be relevant, the SyncSource works like a cursor: it provides getFirstItem()/ getNextItem()/getFirstModifiedItem()/getNextModifiedItem() to retrieve the first and next items in the data store. Plus, in this way the SyncManager has the opportunity to choose to add the item in the current message or in the next one.

In order to keep the client implementation simple, the decision on when to break the message is made with an heuristic rule: the SyncManager is configured with a “Max Data Size” parameter determining the maximum amount of data (without any overhead due to the protocol) are allowed to be sent in a single message. The developer must make sure that this amount is undersized of the possible overhead added by the protocol.

When the server receives a modification from the client, it applies the change in the server database and returns the status. The SyncManager communicates the status of the performed command by calling setItemStatus() so that the SyncSource can take appropriate actions (for example, in the case of a success code, it can reset the item dirty flag or permanently delete a soft deleted item).

On the opposite direction, when modifications are received from the server, the SyncManager decodes each command and call add/update/deleteItem(). These calls apply the change and return a status code (the same value that will be sent in the SyncML message).
Note that addItem() returns also the new local ID generated by the datastore; this is stored by the SyncManager in a persistent storage along with the GUID received by the server so that the mapping will be communicated back to the server in a Map command (see next paragraph for details.

Finalization

After the modifications exchange phase, the client sends the last Status and Map commands (if any are requested depending on the commands received), the server commits the final status and stores the next anchor received in the initialization phase, sending back a final status message; the client must then store its anchor too.

3.2. Item handling

This chapter describes in deeper details how the items exchange is handled by the engine and what a client should do to implement a reliable exchange of the data in any situation. In particular, the focus is on the interrupted synchronization and on how they can be recovered.

Change detection and sending of client modifications

The first task of the client is to extract the changes from the local datastore. This can be done in different ways:

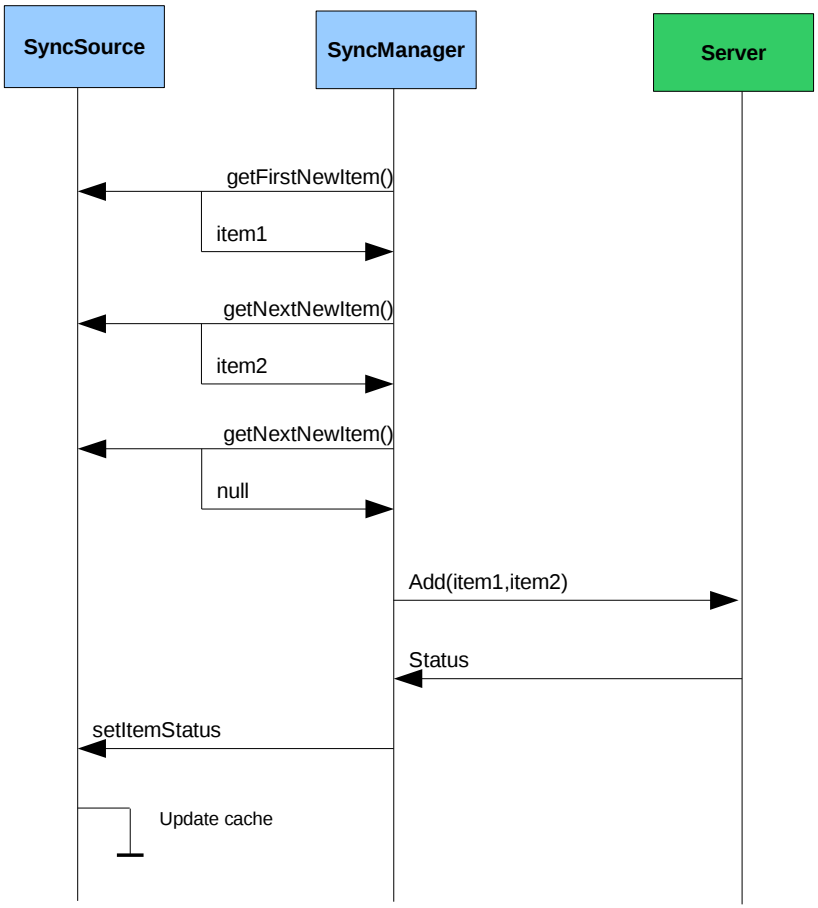
1. keeping a snapshot of the datastore at the end of the last sync, and comparing it with the current status at the beginning of the new sync.
2. listening for events generated on some systems when an item is added/modified/deleted.

The first method is less efficient but is more reliable (no risk to loose a change because an event is lost), and it's the only options for systems with no notification system (for instance, in a generic file sync client).
When a notification system is available (on Outlook for example), using the listeners can speedup a lot the sync, specially the fast sync with few items to exchange.

At the end of this phase, the client SyncSource implementation has three lists of added, updated and deleted items from the last sync available, and will use them to respond to the getNextXXXItem() requests from the SyncManager.

For each item sent to the server, the server responds with a status, where 200/299 and 418 codes means success (for instance, 201 is ITEM_ADDED, see [4]).

The OMA DS specs say that, in case of an interrupted sync, the client must send the modification command (Add, Update or Delete) until it has received a status code for the item. So, when the SyncSource implementation is notified of the status received with the method setItemStatus, it must update its change log to not send the same command again. Note that even in case of error usually some action should be taken; for example, if the server returns a 500 status, meaning that it is not able to handle a particular item, the client should not try to resend the same item again.



Receiving server modifications

After the client has finished to send changes (from a client perspective, this will happen after the three item queues are empty and the SyncSource methods had returned NULL to the SyncManager requests), the server can start sending its changes.
The diagram shows the Add command because it's the one where the client has more responsibilities in making the process works fine.
The added items are sent by the server with the GUID, which is the only know id at that point.
The SyncManager will call the SyncSource addItem() method; this method must:

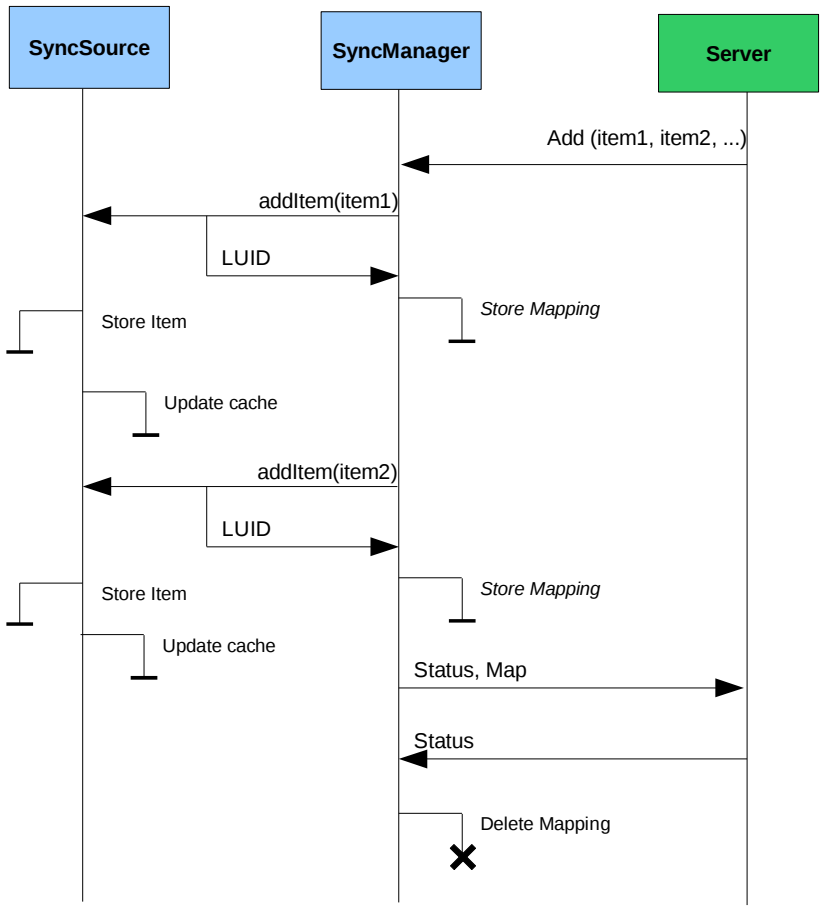
- add the item to the local datastore
- if the add is successful and the id used by the client is different from the GUID, the local id (LUID) must be returned to the SyncManager by changing the id in the SyncItem reference passed as parameter. In this way, the engine can get the new id and create the Map command to send back to the server. If the client uses the same id of the server, the Map command is not requested
- return a status code to the SyncManager to be sent to the server.
  Common success codes are:
  - 200: generic success
  - 201: item added
  - 211: item already deleted
  - 418: item already present (in case the client can do twin detection)

  Common error codes are:

  - 415: unsupported media type
  - 500: generic error

Caching of Map Operations

To correctly support the recover of an interrupted sync, caused by a network error or a manual stop, the SyncManager must store the mapping information in a persistent storage just before sending it. If the server responds with the status on the mapping, the SyncManager should reset the storage; otherwise it should send again the mapping at the next sync. Note that sending twice the same mapping info (in the case the connection drops after the server received it) is not an error. Follows the flow of the mappings cache.



The SyncManager is able to instantiate its own MappingsManager, one for source, that contains methods to write the mappings in the storage, read from it and reset. The MappingsManager has a constructor parameter that is the name of the source it has to handle: the source name is used as an identifier for the KeyValueStore that handles the storage.
The KeyValueStore is created through a static method getMappingStore that asks the MappingStoreBuilder to create a new instance of the proper KeyValueStore.

The default implementation of the MappingStoreBuilder returns a PropertyFile that is able to store in the filesystem.

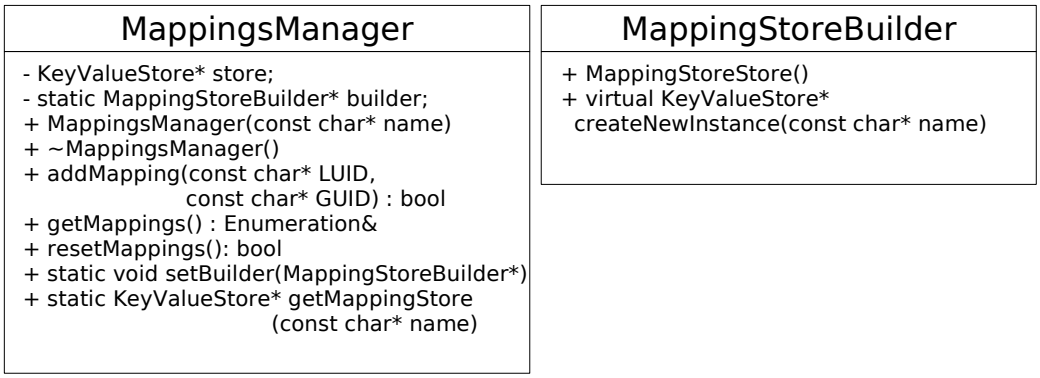


Figure 5: The classes involved in caching map operation

If the client wants to provide its own KeyValueStore implementation, it may create a new derived class from MappingStoreBuilder and set it in the MappingsManager through the setBuilder method.
Since the MappingsManager is created by the SyncManager, the SyncManager itself is responsible to delete the object. On the other hand the MappingsManager has its own KeyValueStore object inside that can be the default one or created by the MappingStoreBuilder passed by the client: in both cases the MappingsManager is responsible to delete the instance and the client doesn't care about it.
By the way, the client is responsible for the lifecycle of the MappingStoreBuilder that sets in the MappingsManager.

3.3. Multiple Messages In One Package

The Funambol Client API C++ supports multiple messages per package, which means the client can send its modified items in more messages and get the server items in the same way. This process is commonly called “multi-message” and it is used to meet the limited resources requirements of mobile devices.

The SyncML protocol specifies how multi-message shall be performed by the synchronizing entities; the process can be summarized as follows:

1. Client sends the first n items in the first message of PKG #3, without the <Final/> element;
2. Server applies the changes and replies with a message containing only status commands; the server is not allowed to send its own changes until client finished to send client side changes (which means unless the client ends the package with the <Final/> element);
3. Client sends the next message with the next items; if the message contains the last items, the package is terminated with the <Final/> tag;
4. Server applies the client changes and returns a message with only status; if the client sent its last massage, Server goes to the next step; otherwise, the process goes back to step 2;
5. Client replies to Server with simply a status command (optionally a 222 Alert code);
6. Server starts sending server side updates; if the message is not the last one, the <Final/> element is not added to the message; otherwise, <Final/> indicates the end of the package;
7. Client applies the received changes and replies with status command only; when the message obtained by Server contains the <Final/> element, the process ends.

3.4. Large Object handling

The Funambol Client API C++ supports the large object defined by the OMA DS standard, which means that it's possible to exchange objects that are bigger than the message size by splitting them into chunks. Each chunk is send to the peer agent with a tag <MoreData/> indicating that the object it's not finished yet.
The receiving agent (either on the server side or on the client side) it's responsible for reassembling the object.

Client to Server

[since v.8.2.2]
The sync engine is able to split large objects into small chunks to be sent to the Server, this is done with the help of ***InputStream*** and ***ItemReader*** classes.

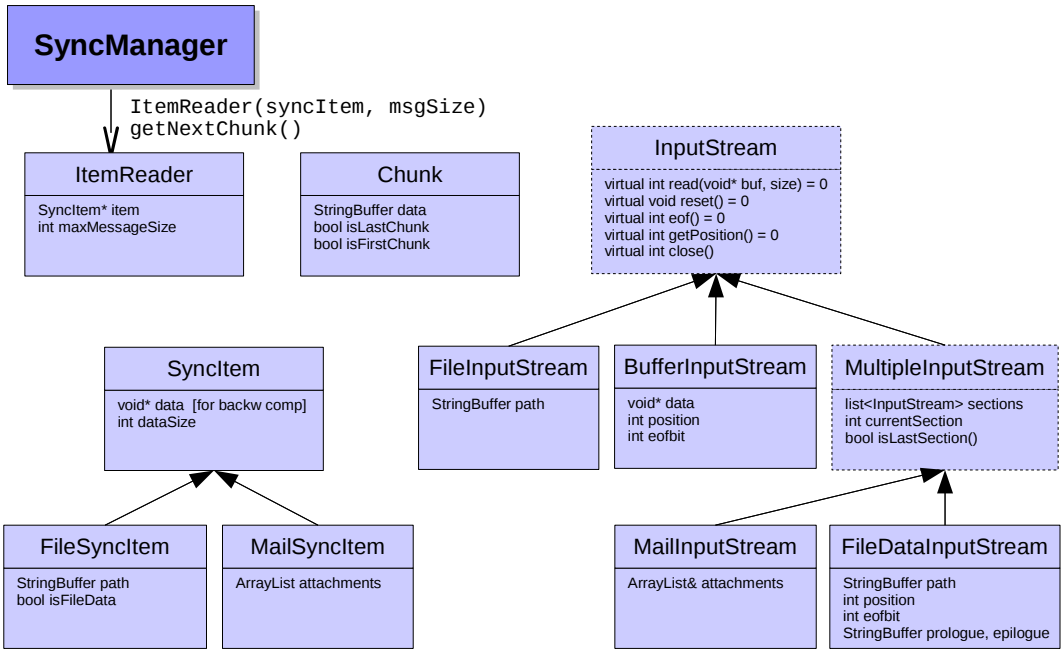


Figure 6: ItemReader and InputStream classes

The *SyncManager* asks the first/next syncItem to the *SyncSource* and passes it to the *ItemReader* object along with the maxMessageSize information, which is the maximum amount of bytes to be sent in a single SyncML message.
The *ItemReader* is responsible to read chunks of data from the *SyncItem* (see *InputStream::read(void* buf, int size)* method), calculating the right read number of bytes to be read with respect of the SyncItem's encoding type.
For instance, if the encoding desired is base64 the chunk size is calculated as follows:

```
size = int(chunk size / 4) * 3
```

the result size must be divisible by 3, in order to obtain a valid base64 string when joining back the chunks on Server side.

The *ItemReader* is also responsible to transform the chunk data with the right encoding / encryption, then a ***Chunk*** object is created and returned back to the *SyncManager*. The *Chunk* contains the data ready to be sent and a boolean '*isLastChunk*' which is set to true by

the *ItemReader* if the chunk returned is the last one. The data transformation is executed with the help of ***EncodingHelper*** class, which has the ability to convert the data chuncks given the type of encoding required and the user's credentials.
The sync engine will just need to ask for the next chunk to the *ItemReader* until the last one is returned, then the next *SyncItem* will be processed.

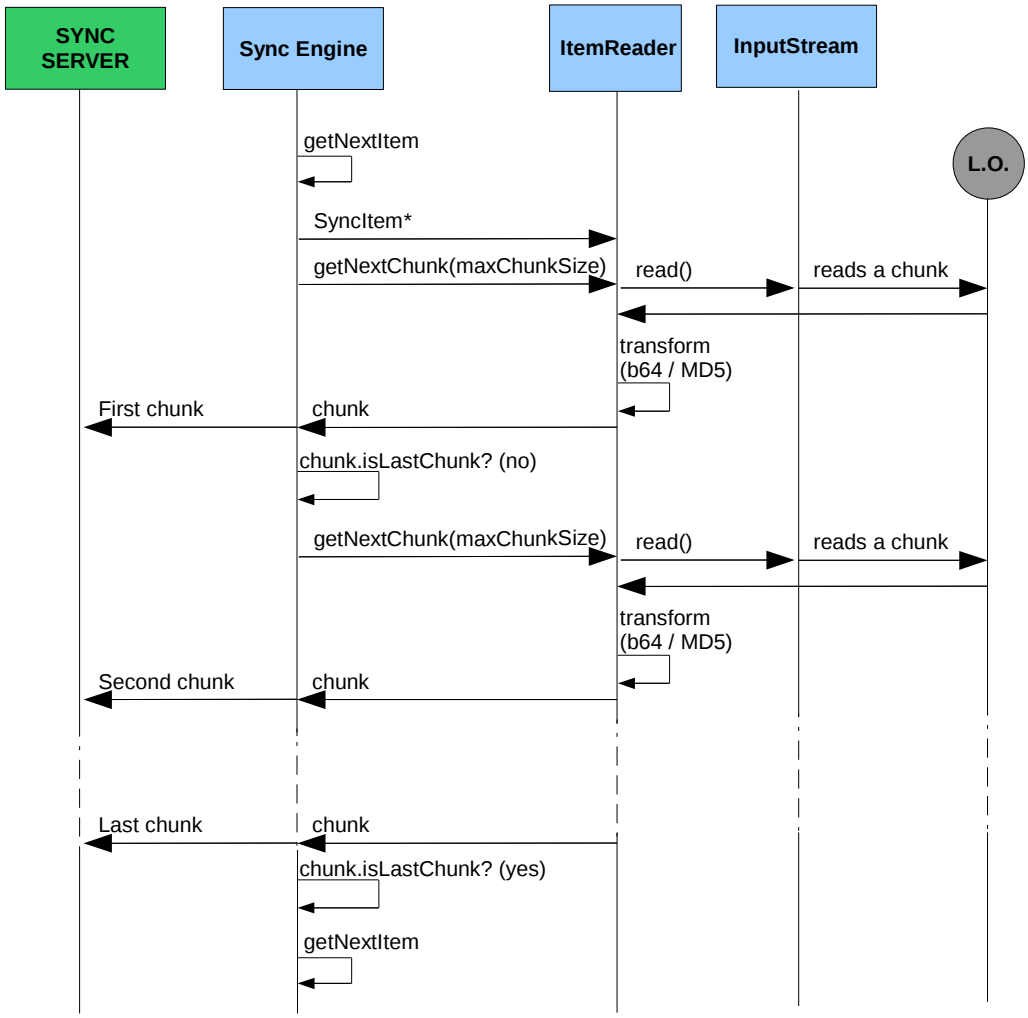


Figure 7: Large Object support for outgoing items

The *InputStream* class is responsible to split and return the data of the correct size as requested by the *ItemReader*, until the end of data. It's an abstract class, it defines the interface methods to access the stream (*read*, *reset*, *close*). Specific implementations of InputStream can read data directly from a stream in order to avoid loading a large object in memory.
In the default implementation, the *SyncItem* uses a ***BufferInputStream***, which reads directly from the '*void* data*' buffer. This way, the *BufferInputStream* reads chunks from that buffer loaded in memory, which is set as usual calling *SyncItem.setData()*.

The *BufferInputStream* is created passing the SyncItem's '*data*' buffer to read from in the constructor, so it's properly recreated new if the *SyncItem::setData()* method is called.

In the *FileSyncSource*, the *FileSyncItem* is used. *FileSyncItem* uses a ***FileInputStream*** to directly read file's content from the filesystem, given its path. In case a file data object is required, the ***FileDataInputStream*** is used instead of *FileInputStream*: the data returned from the *read()* method is not just the raw file content, but it's an XML data object following the OMA File Object specs.

So the formatting operation of the file data object is done inside the *InputStream::read()* method, with different implementations for each type of data.
For more structured data, this operation can be complex: the abstract class ***MultipleInputStream*** defines a useful inteface for data that can be rapresented as a sequence of different input streams. So each stream is defined as a section, and there's an array of sections that are accessed in order. This way, the *read()* operation is nothing more than calling *read()* on each section defined, and the eof bit is set when the last section is ended.
For instance the *FileDataInputStream* extends the *MultipleInputStream*, defining 3 sections:
1.an xml prologue (a *BufferInputStream* on a fixed string)
2.the file content (a *FileInputStream*)
3.an xml epilogue (a *BufferInputStream* on a fixed string)

For emails object type, *MailSyncItem* and *MailInputStream* are defined.
MailInputStream::read() method will format the email data content, including email attachment that are read directly from the streams where they are stored, and will return the data chunked as usual. *MailInputStream* is also extending *MultipleInputStream*, as email data is a generic sequence of text data (*BufferInputStream*) and attachments (*FileInputStream*). The list of attachments is passed in the constructor (it's a list of file paths where the attachments can be retrieved).

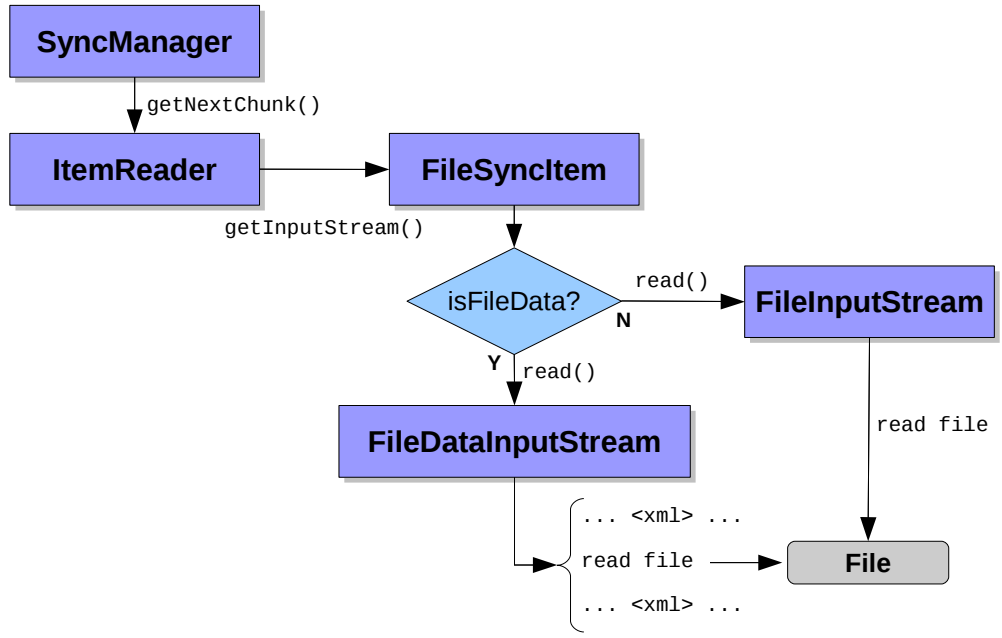
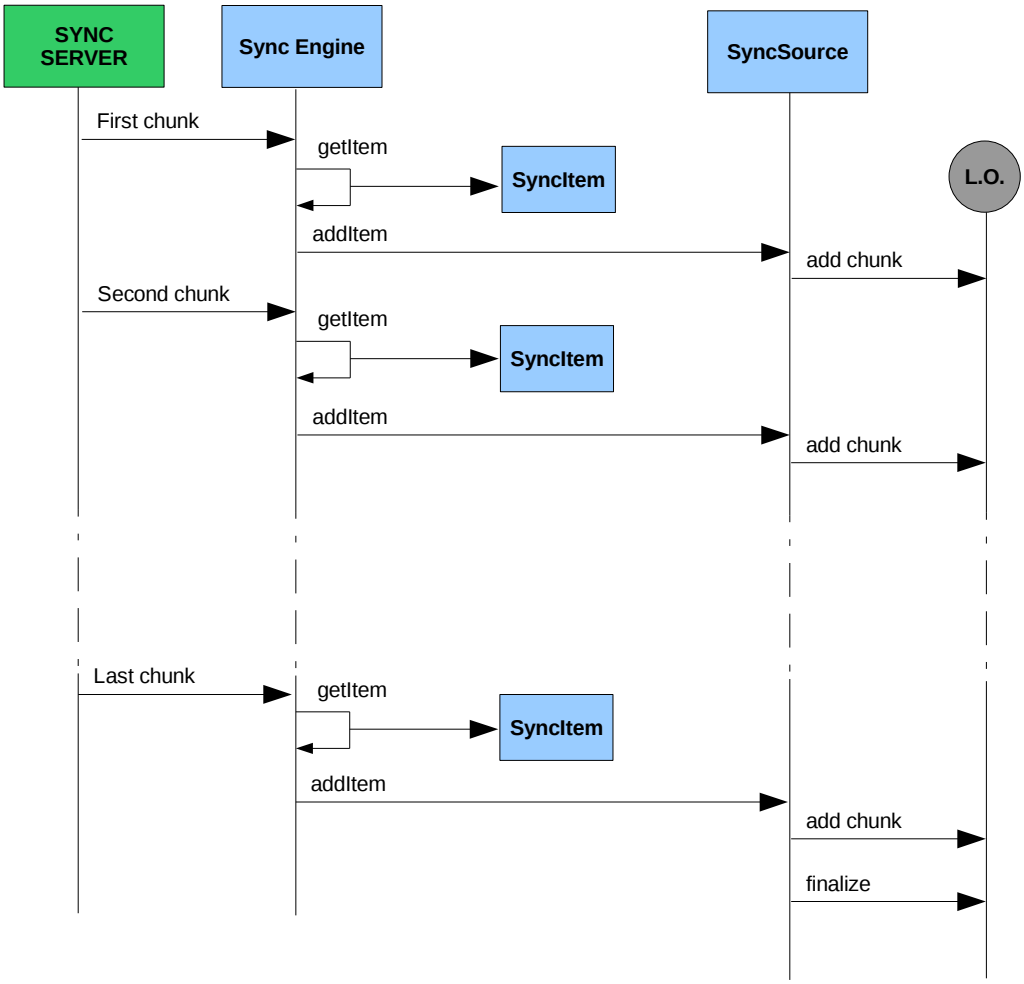


Figure 8: FileInputStream example

Server to Client

The engine is able to do the reassembling work and give the complete object to the SyncSource. Since this may not be possible on limited devices due to the RAM limitations, it's possible for the client to tell the engine to leave the reassembling work to the SyncSource.
In this case, the SyncItem is marked with a MoreData flag and sent to the client, which is responsible for the reassembling. The advantage is that this can be done on the file system directly, for instance, with a less RAM usage. (to be implemented)



3.5. Data Synchronization Layer Design

This section defines the design of the data synchronization layer accordingly to the sections so far presented. More on specific features of the API will be detailed in dedicated chapters. As depicted in Figure 1, we already identified some fundamental components of the API: SyncClient, SyncManager, TransportAgent, SyncSource. Roles and responsibilities of these components are summarized in the table below.

Component	Implementing class	Roles and responsibilities
SyncClient	spds/common/SyncClient	This component is the main interface to the application using the API. It represents the client SyncMLAgent as defined in the SyncML specification. The role of the SyncClient class is to provide an high level interface to a client application so that the developer can enable its own application to use SyncML simply calling a method of this class (see Figure 2). This is also where the synchronization process is implemented, from initialization to finalization, including multi message and so on. This class can be derived to overload some methods to perform custom operations (e.g. prepareSync, beginSync, endSync)
SyncManager	spds/common/SyncManager	This is responsible for the implementation of each single phase of the synchronization. It is also responsible of driving the process of building the proper SyncML messages to send to the server through the TransportAgent and to interpret the server response and take the consequent actions.
TransportAgent	http/common/TransportAgent	This low-level component is responsible of transporting the SyncML messages to and from the server. It accesses the system network API to perform the needed HTTP calls.
SyncSource	spds/common/SyncSource	This component represents a client database. It is used by the SyncManager when it needs to read on store data from and to the local database. SyncSource is purely an interface, which client developers must implement in order to access the most disparate data sources.

SyncClient

The class diagram of SyncClient is illustrated in Figure 9.
SyncClient is the interface class between the client and the engine. It can be overridden to implement the virtual protected methods to perform client specific action at certain points of the sync. See the methods description below.

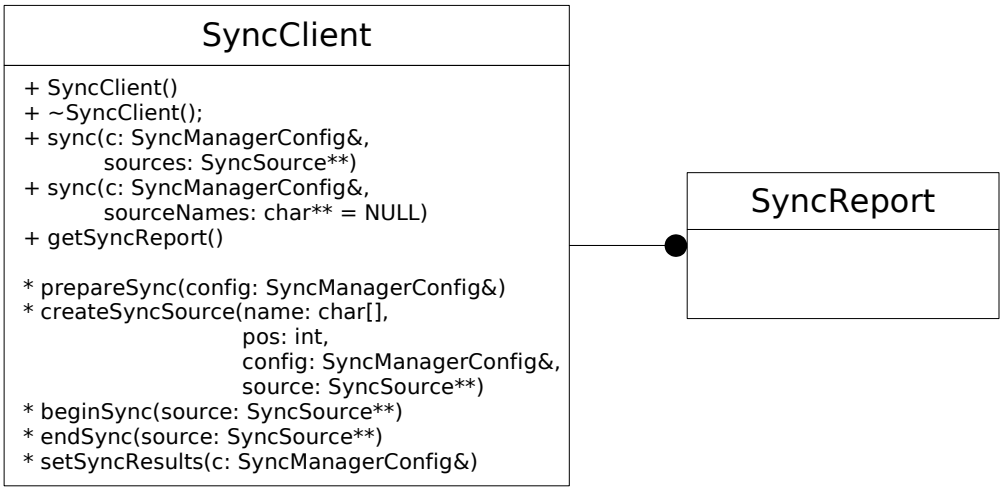


Figure 9: SyncClient class diagram

As a design choice, we want that a SyncClient could potentially be configured from many sources (the DM subsystem is the preferred one, but other systems may be all right), therefore to properly configure the SyncClient instance the reference to the configuration object (an instance of SyncManagerConfig – for details, see the dedicated section about the configuration objects in later in this document) is passed when calling one of the two methods sync().

At the end of the sync, it is possible to retrieve from the SyncClient a report of the sync (number of items exchanged, error code and error message), using the getSyncReport() method. Please refer to the SyncReport paragraph for more datails.

At the end of the sync, the sync results are also stored in the configuration, so the last sync error codes can be available even after the *SyncClient* is destroyed. It is duty of the Clients to eventually persist in memory the configuration after the sync ended, in order to access the last error codes even after a client restart.
The sync result codes are copied from the *SyncReport* into the config parameters:
–*SyncManagerConfig::lastGlobalError*
–*SyncSourceConfig::lastSourceError*
The source's sync results are set only for the sources effectively synchronizated.

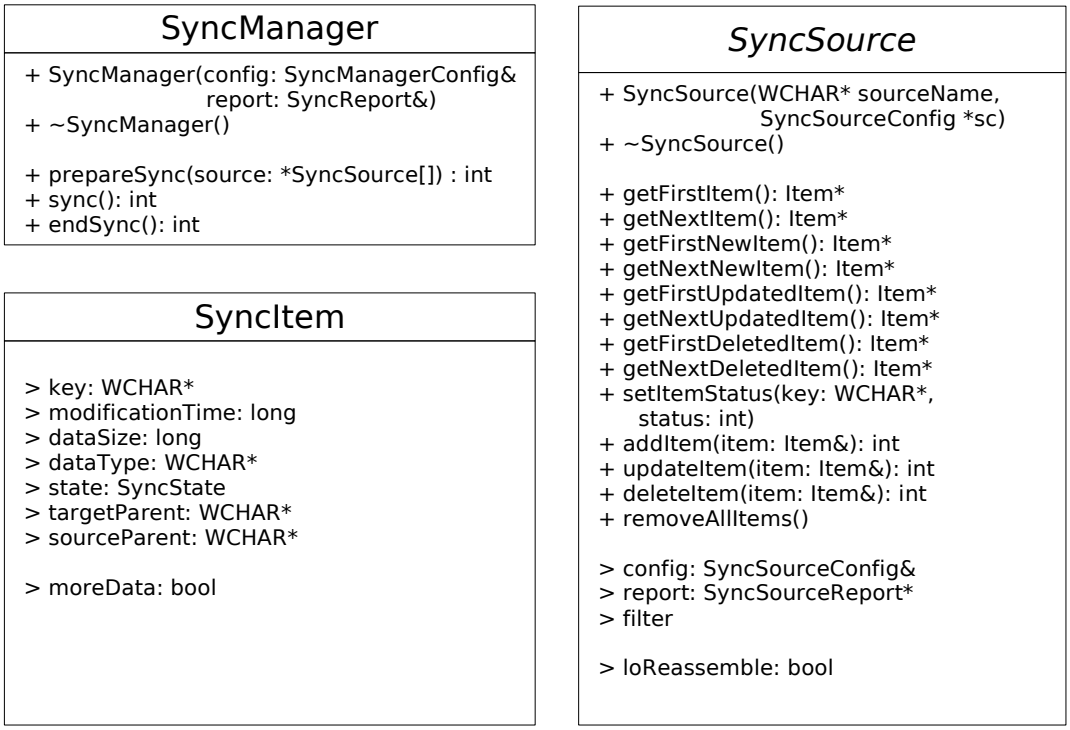


Figure 10: SyncManager class diagram

SyncManager

The class diagram of the most important classes related to the SyncManager is shown in Figure 10.

SyncManager represents the core of the data synchronization API. As the prepareSync(), sync() and endSync() methods suggest, it handles all the phases of the synchronization process.

Along with SyncManager, two other important classes/interfaces are defined: SyncSource is an interface that represent a local data store; SyncItem represent a single item to synchronize.

prepareSync(source: *SyncSource[]) : int

This method takes an array of SyncSource to synchronize and handles the SyncML initialization phase, which means authenticating to the server and alerting it with the database that the client wants to synchronize and with which synchronization mode.

The passed in SyncSource array will be kept for reference in the subsequent calls. This means that the caller must make sure they stay allocated for the entire lifetime of the SyncManager instance.

As previously seen, the authentication process may take some communication between client and server and may require session information to be kept at least until the counterpart is authenticated (i.e. Nonce). This information includes:

- the default authentication type (the one the client will first try to use)
- the requested authentication type (the one requested by the server)
- username
- password
- server id
- server password
- nonce

Plus, there are different algorithm to create the actual authentication data to send in the message. All this work is encapsulated in the class CredentialHandler, whose role is to perform any credential calculation related task (see Figure 11 for the class diagram).

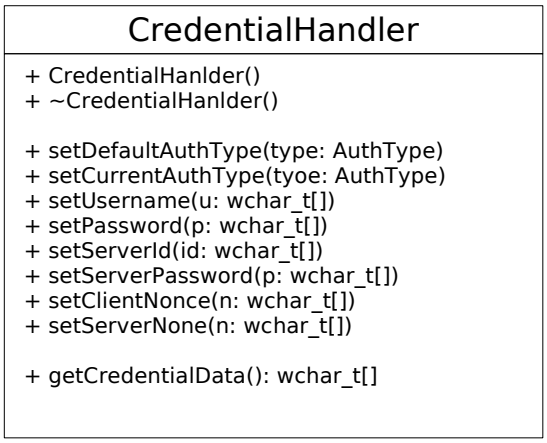


Figure 11: CredentialHandler class diagram

SyncManager instantiates a CredentialHandler at the beginning of the initialization process and keeps using it until the end; it feels the authentication properties (auth types, username and server id, passwords and nonces) from the configuration object or from the messages returned by the server and then calls getCredentialData() to retrieve the data block to insert in the SyncML message. Such block is the base64 encoding of the credentials data. Note that such wchar_t string is dynamically allocated with new C++ operator and must be deleted by the caller.

SyncManager also provides client capabilities. This information combine device specific information with data source related information, therefore properties from many configuration objects will be used to build the DevInf command to send to the server. DevInf properties are retrieved from the configuration (SyncManagerConfig object) passed to SyncManager by reference (see SyncManager and SyncSource configuration).
If it is the first time SyncManager connects to a server, client capabilities are sent in a Put command (see the client capabilities handling session).
In the same way, if during initialization the server requests client capabilities sending, SyncManager will send them in a Results command.

In the initialization phase it is not foreseen to be necessary to split the package in multiple messages, therefore no multi-message mechanism is implemented at this stage.

sync()

When sync() is invoked the SyncManager performs the following tasks:

- for each sync source in the array passed in in prepareSync():
    - detecting client modifications;
    - sending client modifications (in multiple messages if required);
    - processing return messages committing successfully completed commands (based on the status code returned by the server);
- for each sync source in the array passed in in prepareSync():
    - receiving server modifications;
    - processing and applying server modifications, returning proper status codes;
    - sending modification status codes back to the server.

The API supports the Large Objects defined by the OMA standard, and takes care of the object segmentation and reassembling when sending/receiving data by default.

Detecting Sending and Committing Client Modifications

Note that the way changes are detected is left to the SyncSource implementor. It may be based on flag modification, change log handling, versioning... this is hidden to the SyncManager who just asks for modifications calling get[First/Next] [New/Updated/Deleted]Item() on a SyncSource instance.

getFirstItem() and getNextItem() are used in the case of a slow sync and retrieve the entire data source content.

get[First/Next][New/Updated/Deleted]Item() are used in the case of fast sync and retrieve the items in the data store that are in the corresponding state.

From the SyncManager perspective, a SyncSource is like a database cursor: the cursor is initialized when getFirstXXX() is called and each getNextXXX() call returns an item or NULL if the cursor reached the end of the result set.

Client modifications so retrieved are embedded in the SyncML message and sent to the server. Note that not necessarily all modifications go in one single message. SyncManager determines if an item must go in the current message or in the next message based on the number of items in the current message and their size (see the multi message section earlier in this document).

When the SyncManager receives a status for a modification command previously sent, it calls the method setItemStatus() of the corresponding SyncSource. This gives the SyncSource the opportunity to commit the change, for example resetting the item modification flag.

Applying Client Modifications

After the client has done with its modifications the server starts sending its own. These are interpreted by the SyncManager who eventually calls the SyncSource's add/update/deleteItem() methods. They return a status code as defined by the SyncML specifications.

Note that the item parameter passed to addItem() on return may have a a changed key. This represents the local id (LUID) that will be returned to the server in a Map command.

The SyncSource Interface

SyncSource is an abstract class that implementors must extend in order to access specific data sources. It contains a SyncSourceConfig reference to the correspondent object owned by SyncManagerConfig (passed by constructor). It also contains a SyncSourceReport

pointer, which is owned by SyncReport and passed using setReport() method. So the 'report' member is a pointer to an external object, it MUST NOT be deleted by SyncSource.
The methods defined by this interface are listed in the following table.

Method	Description
SyncSource(const WCHAR* sourceName, const SyncSourceConfig *sc)	Constructor: creates a SyncSource from the specified sourceName, set the internal configuration reference from the passed SyncSourceConfig.
SyncSourceConfig& getConfig()	Returns the internal reference of SyncSourceConfig.
SyncSourceReport* getReport()	Returns the internal pointer to the SyncSourceReport.
void setReport (SyncSourceReport* sr)	Sets the internal pointer to the passed SyncSourceReport. The pointer is copied (no space allocated), so SyncSource does NOT own the SyncSourceReport.
Item* getFirstItem()	Returns the first item regardless its modification state, or NULL if there are no more items. If the SyncSource keeps an underlying cursor (i.e. a database cursor), the call shall reset it if already in use. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item* getNextItem()	Returns the next item regardless its modification state, or NULL if the source contains no items. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item* getFirstNewItem()	Returns the first item flagged as new or NULL if there are no new items. If the SyncSource keeps an underlying cursor (i.e. A database cursor), the call shall reset it if already open and in use. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item* getNextNewItem()	Return the next new item or NULL if there are no more new items. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item* getFirstUpdatedItem()	Returns the first item flagged as updated or NULL if there are no updated items. If the SyncSource keeps an underlying cursor (i.e. a database cursor), the call shall reset it if already in use. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item* getNextUpdatedItem()	Return the next updated item or NULL if there are no more updated items. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item* getFirstDeletedtem()	Returns the first item flagged as deleted or NULL if there are no deleted items. If the SyncSource keeps an

Method	Description
	underlying cursor (i.e. A database cursor), the call shall reset it if already in use. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
Item& getNextDeletedItem()	Return the next deleted item or NULL if there are no more new items. The returned item is allocated with the C++ operator new and it is managed inside API (**caller MUST NOT free it**)
void setItemStatus(wcar_t* key, int status)	Called when a Status for a modification command previously sent is returned by the server. This gives the opportunity to the SyncSource to know if the modification has succeeded and to reset the related modification flag.
int addItem(Item& item)	Adds a new item to the underlying data source. The call returns the status code that SyncManager shall return the server in the Status command. If the SyncSource generates a new local id for the added item, it must set the new key in the Item object so that the SyncManager will be able to send it back to server in a Map command. This method shall not modify the status flag of the item (the item shall not be flagged as new).
int updateItem(Item& item)	Updates an item into the underlying data source. The call returns the status code that SyncManager shall return the server in the Status command. This method shall not modify the status flag of the item (the item shall not be flagged as modified).
int deleteItem(Item& item)	Deletes an item from the underlying data source. The call returns the status code that SyncManager shall return the server in the Status command. This method shall not modify the status flag of the item (the item shall not be flagged as deleted and it should be permanently deleted).
int removeAllItems()	Called by the SyncManager when the sync mode is refresh-from-server to remove all the items by the client storage. The SyncSource can stop the sync by returning an error code to the SyncManager (for instance, a client may ask the user before doing this operation).
bool getReassembly()	Return true is the Large Object Reassembly is done by the engine, false if it's done by the client.
setReassembly(bool)	Set if the reassembly is done by the engine (true), or by the client (false). This property must be set to true by clients that wants to do the reassembly by theirselves. The default is to leave it to the engine.
Clause* getFilterClause()	Returns the filter clause that should be used in preparation for the synchronization of this sync source.

Method	Description
void setFilterClause(Clause& clause)	Sets the filter clause that should be used in preparation for the synchronization of this sync source.

Synchronization Events Notification

Many of the actions performed at the SyncManager level are notified to the registered listeners. This is described in a dedicated section and the details are delegated at the development level.

endSync()

Role of this method is to conclude the synchronization session. This includes sending a final Status command or the last Map command, if during the synchronization process any item was created on the client and new local ids where generated.
If there are any errors in this phase, as a network error in sending/receiving the latest message or the lastest server message is wrong or it contains syncml errors (i.e. header status 500), the anchor are not updated and the client is prevented to store them wrongly.

Also, in this phase the SyncSource endSync method is called, and the SyncSource implementation has the chance to perform ending/cleanup operations too.

3.6. SyncManager and SyncSource Configuration

The SyncManager object requires a considerable amount of configuration information; plus, such information may vary as the API improves or may require flexibility due to particular client needs. For these reasons, it makes sense to design an overall configuration architecture, with the responsibility of implementing the details of such flexibility and complexity giving an easy to access configuration framework to the other modules of the API and to client developers.

The configuration is handled by the client, which passes a reference of SyncManagerConfig to the SyncClient constructor.
Note: the config object passed to SyncClient must be already filled with all desired properties, so it's a client duty to read or create the SyncSourceConfig before the sync process, and eventually save it at the end. This is done as a design choice to leave the client all the freedom to manage the configuration, based on its necessity.
If a DMTClientConfig implementation is used, standard methods to read and save all config properties are provided (see Figure 12).
The class DefaultConfigFactory gives the opportunity to generate default config objects with all standard properties; a client should use a inherited class and override these methods in order to have a factory for specific config objects.

The class diagram of such configuration framework is illustrated in Figure 12.
The classes of the configuration framework are described in the following sections.

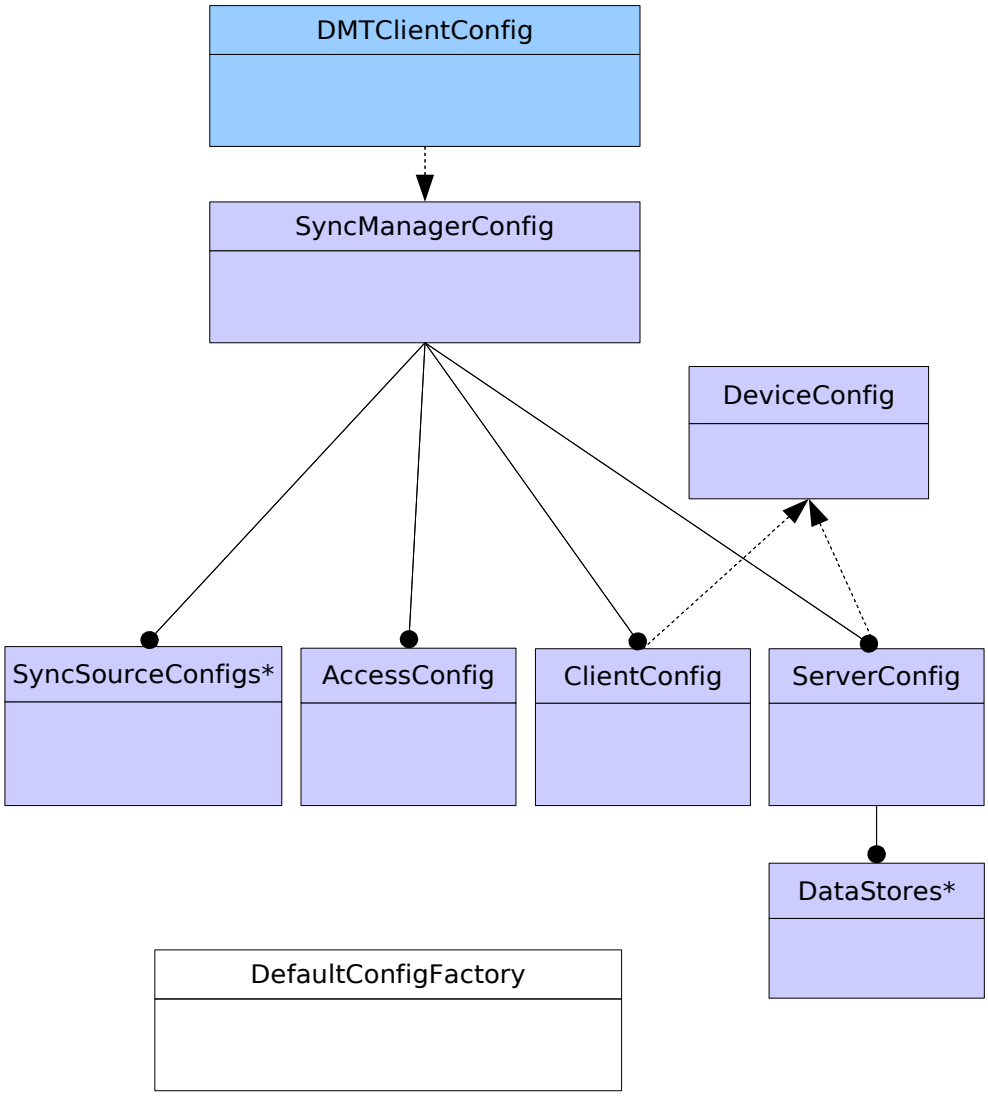


Figure 12: Configuration framework class diagram

SyncManagerConfig

SyncManagerConfig groups the configuration information needed by the SyncManager. This implementation is just a transient repository; persisting configuration settings is delegated to subclasses. The framework provides also a subclass able to read and store configuration information from/to the DM tree.

A client developer can choose to use the DM subsystem to handle the config persistency, or to derive from the SyncManagerConfig interface providing another way to persist the data on the device storage.

SyncManagerConfig methods are described in the following table.

Method	Description
SyncSourceConfig* getSyncSourceConfigs()	Returns a NULL terminated array of SyncSourceConfig objects, where each object contains the configuration of a SyncSource. It is used to enumerate all the SyncSources handle by the clients.
SyncSourceConfig* getSyncSourceConfig(const char* name)	Returns a pointer to the internal SyncSourceConfig with the configuration of the SyncSource with the given name. If such SyncSource does not exist, remains untouched and the method returns NULL.
SyncSourceConfig* getSyncSourceConfig(unsigned int i)	Returns a pointer to the internal SyncSourceConfig with the configuration of the SyncSource in position 'i' of the SyncSourceConfig array. If SyncSourceConfig[i] does not exist, the method returns NULL.
unsigned int getSyncSourceConfigsCount()	Returns the number of SyncSourceConfig.
AccessConfig& getAccessConfig()	Returns the access settings configuration currently selected to be used by the SyncManager.
DeviceConfig getClientConfig()	Returns the client settings.
DeviceConfig getServerConfig()	Returns the server settings.
BOOL setSyncSourceConfig (SyncSourceConfig& sc)	Set the internal SyncSourceConfig with the given name from sc.getName(). SyncSource settings are copied from sc object. If such SyncSource does not exist it is created new and appended in the SyncSourceConfig array.
setAccessConfig(AccessConfig& ac)	Set the internal AccessConfig. Access settings are copied from ac object.
setClientConfig(DeviceConfig& dc)	Set the internal DeviceConfig for client. Client settings are copied from dc object.
setServerConfig(DeviceConfig& dc)	Set the internal DeviceConfig for server. Server settings are copied from dc object.

SyncManagerConfig parameters are listed in the following table:

Property	Description
abortSyncProcess	Should be set by the client to abort the sync process smoothly

Property	Description
	The *SyncManager* check this flag periodically (see *isToAbort* method)
lastGlobalError	This is the global error code of the last synchronization done (global, not related to a specific source: for sources errors see *SyncSourceConfig::lastSourceError*). It is read/stored in the configuration, so it is accessible at any time, even after the sync ended. If using the *SyncClient* object to trigger the sync, its value is the SyncReport's *lastErrorCode*, set at the end of each sync session. Code 0 means "no error occurred".

DMTClientConfig

This is a concrete implementation of a SyncManagerConfig object. It reads/stores configuration information using the DM tree.

DMTClientConfig public methods are described in the following table.

Method	Description
DMTClientConfig(char* root)	Creates a new DMConfig object that will use the given root as the DMTree root from which the configuration will be accessed.
~DMConfig()	Destructor.
SyncSourceConfig* getSyncSourceConfig(char* name, BOOL refresh =FALSE)	This method is overloaded to implement the refresh of the config from DM when the second parameter is TRUE.
SyncSourceConfig* getSyncSourceConfig(unsigned int i, BOOL refresh =FALSE)	This method is overloaded to implement the refresh of the config from DM when the second parameter is TRUE.
ManagementNode* getSyncMLNode()	Provides access to the "syncml" configuration node, can be used to read/write custom configuration options. Config must have been opened before. Returns the node pointer owned by config and valid while the config is open.
ManagementNode* getSyncSourceNode(int index)	Get the specified SyncSource configuration, with the index specified. Returns the node pointer owned by config and valid while the config is open.
BOOL read()	Fills the object with the settings read from the DMTree. It returns TRUE in case of success, FALSE otherwise.
BOOL save()	Stores the object values to the DMTree. Returns TRUE in case of success, FALSE otherwise.
BOOL open()	Opens the configuration backend associated with the root context. Calling on an open config does nothing. Return TRUE for success.
void close()	Closes the configuration backend. Frees all resources associated with and invalidates all

Method	Description
	ManagementNode pointers returned by this config.

In addition to these ones, DMTClientConfig has also the methods to read and write the components classes from and to the DMTree. These methods are defined protected and are called by read() and write() only, and are independent from the underlying implementation of the DM (they uses only the methods getPropertyValue and setPropertyValue defined by the interface).

AccessConfig

The methods of AccessConfig are fundamentally getters and setters of the properties listed in the class diagram of Figure 12. By general rule, setters methods copy the passed value to the internal one, while getter methods return the internal pointer to the desired property. The less obvious properties are described below.

Property	Description
useProxy	Boolean. Should the sync engine use a HTTP proxy?
beginTimestamp	The beginSync timestamp.
endTimestamp	The endSync timestamp.
firstTimeSyncMode	The SyncMode that the sync engine should use the first time a source is synced.
syncURL	The syncURL value. If the URL does not start with http:// (or HTTP://) or https:// (or HTTPS://), http:// is prepended to the given string.
serverNonce	The server nonce value: from client to server.
clientNonce	The client nonce value: from server to client.
isServerAuthRequired	Boolean. Does the server require authentication?
maxMsgSize	The maximum message size (Byte) accepted for XML messages received from server (server to client).
maxModPerMsg	The maximum number of modifications sent in each XML message from client to server.
readBufferSize	Specifies the value for the size of the buffer used to store the incoming stream from server. It is expressed in byte. If set = 0, a default value = 4096 is used.
encryption	Boolean. Do we use ciphering?
userAgent	The user agent string, will be attached to http messages to identify the client on server side. It should be a short description with the client name plus its version.
responseTimeout	The number of seconds of waiting response timeout. If set = 0, a default value = 300 is used.
dirty	The dirty flag, used to select which properties have been modified. Not used by now (T.B.D).

Notes:

'userAgent' value should always be specified. If the property is left empty, the user agent will be derived from 'devID' plus 'swv' fields from DeviceConfig. If also devID is empty, a default user agent value will be used: “Funambol SyncML Client”.

DeviceConfig

SyncManagerConfig contains two instances of DeviceConfig:

1. ClientConfig: used to store Client related configuration settings
2. ServerConfig: used to store Server configuration settings (obtained via devInfo)

ClientConfig

The methods of ClientConfig are fundamentally getters and setters of the properties listed in the class diagram of Figure 12. By general rule, setters methods copy the passed value to the internal one, while getter methods return the internal pointer to the desired property. Most of ClientConfig properties are used to generate the <DevInf> element for client capabilities (see Client Capabilities Handling chapter). ClientConfig properties are described below.

Property	*Description*
verDTD	Specifies the major and minor version identifier of the Device Information DTD used in the representation of the Device Information. The value MUST be “1.1”. This property is mandatory.
man	Specifies the name of the manufacturer of the device. This property is optional.
mod	Specifies the model name or model number of the device. This property is optional.
oem	Specifies the OEM (Original Equipment Manufacturer) of the device. This property is optional.
fwv	Specifies the firmware version of the device. This property is optional.
swv	Specifies the software version of the device. This property is optional.
hwv	Specifies the hardware version of the device. This property is optional.
devID	Specifies the identifier of the source synchronization device. The content information MUST specify a theoretically, globally unique identifier. This property is mandatory.
devType	Specifies the type of the source synchronization device. Type values for this element type can be e.g. “pager”, “handheld”, “pda”, “phone”, “smartphone”, “server”, “workstation”. Other values can also be specified. This property is mandatory.
dsV	Specifies the implemented DS version. This property is optional.
utc	Boolean. Specifies that the device supports UTC based time. If utc = TRUE, the server SHOULD send time in

Property	Description
	UTC format, else MUST send in local time. Client MAY send time in local or UTC format. Default value = TRUE.
loSupport	Boolean. Specifies that the device supports handling of large objects. Default value = FALSE.
nocSupport	Boolean. Specifies that the device supports number of changes. Default value = FALSE.
logLevel	Specifies the logging level on the device. It can be one of 0 – 1 – 2 (none, info, debug). Default value = 1 (info).
maxObjSize	Specifies the maximum object size allowed by the device. Default value = 0 (no maxObjSize set).
devInfHash	This is a hash value generated from all properties that are used for the <DevInf> element, plus the syncURL property from AccessConfig. Initial value = “0”.
sendDevInfo	Boolean. Specifies if the <DevInf> element should be sent by the client. Default value = TRUE
forceServerDevInfo	Boolean. The Client can force to ask the Server capabilities during sync, setting this parameter to true. Default=FALSE

Notes:
If there is no firmware/software/hardware version of the device available (fwv, swv, hwv), then their content information can also be a date, for example, 19980114 or 19990714T133000Z. Only hours, minutes and second MUST be specified in the time component.
“devInfHash” property is used to verify if any DevInf element (or syncURL) has changed since last sync; in that case the devInf is sent to the server.

“sendDevInfo” parameter has been added to enable clients to avoid sending device capabilities. If a client doesn't want to send its capabilities should explicitly set the “sendDevInfo” value to false in the configuration with the method “setSendDevInfo()”.

If using DMTClientConfig implementation to read/save configuration, the parameter “forceServerDevInfo" is not persisted in memory because it should just be set to true everytime the Client needs to force it (otherwise the Server caps would always be requested).
Clients can set it by calling the method setForceServerDevInfo(true).

ServerConfig

Server configuration parameters are set during the synchronization when the Server sends its capabilities, and are available to the Clients under the ServerConfig object (see 3.16). Most of the properties are the same of ClientConfig, just referred to the Server instead of the Client. The following table describes additional properties used by *ServerConfig*:

Property	Description
smartSlowSync	Specifies if the Server supports the Smart Slow sync (for emails sync). Values are: 0 = true, 1 = false, 2 = cannot determinate. Default value = 2.
lastSyncURL	Specifies the Server URL corresponding to the Server

Property	Description
	capabilities currently stored (the URL of the last successful sync). If the Server URL changes, the capabilities are no more valid.
dataStores	ArrayList of DataStore objects. It's a list of datastores supported by the Server, as sent within the Server capabilites.
mediaHttpUpload	Boolean flag. If true, the server supports Media HTTP upload.

SyncSourceConfig

The methods of SyncSourceConfig are fundamentally getters and setters of the source configuration properties, with the exception of the CTCap handling methods, that will be better explained in the par. 3.15).
The following table describes the less obvious properties:

Property	Description
encoding	Specifies how the content of an item should be encoded. The form of this parameter is a semi-column separated list of formats that must be applied in sequence from the leftmost to the rightmost. For example, if format is “des;b64”, when the item will be output in the message, the content must be first transformed with the “des” encoder and than with the b64 encoder.
encryption	Specifies if the content of an outgoing item should be ecrypted. If this property is not empty and valid, the 'encodings' value is ignored for outgoing items. Actually the only valid value is “des”. Default value is an empty string (no encoding).
syncModes	Specifies all supported sync modes for the given source. The form is a comma separated list of modes. Sync modes can be one of “slow”, “two-way”, “one-way-server”, “one-way-client”, “refresh-from-server”, “refresh-from-client”, “addrchange”, “smart-one-way-client”, “smart-one-way-server”, “incremental-smart-one-way-client”, “ncremental-smart-one-way-client”
sync	This is the current sync mode used by the client. The parameter MUST be one of the sync modes specified in property “syncModes”.
supportedTypes	A string rapresenting the source types (with versions) supported by the SyncSource. The string must be formatted as a sequence of 'type:version' separated by commas ','. For example: “text/x-vcard:2.1,text/vcard:3.0”. The version can be left empty, for example: “text/x-s4j-sifc:” Supported types will be sent used for the DevInf (see Client Capabilities chapter).
type	The source type used by client. This is one of types

Property	Description
	specified in “supportedTypes” property.
version	The version of the source type used by client.
last	Long value that specifies the last timestamp for this source.
fieldLevel	True if the SyncSource is able to apply field-level replaces (see 3.15 for more details).
ctCaps	ArrayList of CTCap objects, each one representing the content capability information for one type supported by the source.
lastSourceError	The last error code, for this source (0 means "last sync successfull"). If using the SyncClient object to trigger the sync, the last sync result code of the corresponding source is set at the end of the sync. This way the information will be available even after the sync ended.

For the CTCap handling, it is also available the following utility method to add a content type capability information:

Method	Description
int addCtCap(properties, type, version, fieldLevel)	this method creates and adds to the ctCaps list a CTCap object using the given information. Only the properties parameter is mandatory, the others are take by default from the config (see above).

DefaultConfigFactory

The methods of DefaultConfigFactory are factories for config objects. They are provided to help the client developer in the creation of the initial configuration.
For the SyncSourceConfig, the right default values for a standard Funambol server installation is provided based on the name for the basic source (contact, calendar, task, note).

Method	Description
AccessConfig* getAccessConfig()	Returns a default generated AccessConfig. Returns an AccessConfig pointer allocated new, so it must be freed by the caller.
DeviceConfig* getDeviceConfig()	Returns a default generated DeviceConfig. Returns a DeviceConfig pointer allocated new, so it must be freed by the caller.
SyncSourceConfig* getSyncSourceConfig(const char* name)	Returns a default generated SyncSourceConfig, based on the name. Returns a SyncSourceConfig pointer allocated new, so it must be freed by the caller.

3.7. CacheSyncSource

This abstract class implements the SyncSource interface, adding a method to detect the changes in the local store since the last sync based on cache files to make easier the implementation of new sync sources.
It requires an instance of a class implementing the KeyValueStore interface to store the sync cache, made by pairs of LUID (the local id of the item) and a fingerprint (default method is CRC of the content, but can be a timestamp or any other way to detect a change on the item). By default, CacheSyncSource is able to obtain a PropertyFile (which implements KeyValueStore as a file), but if a more efficient way to store it is available for the platform, the developer can create anoher store and pass it in the CacheSyncSource constructor (see also the SQLKeyValueStore abstract class).

The methods to implement are:

- getAllItemList: returns a list of StringBuffer with all the keys of the items in the data store
- insertItem: adds a new item into the data store
- modifyItem: modifies an item in the data store
- removeItem: removes an item from the data store
- removeAllItems: removes all the items from the data store
- getItemContent: get the content of an item given the key
- getItemSignature: [OPTIONAL] get a fingerprint of the item, which is any string which allows to detect changes in the item. As a default implementation, the CRC is used.

3.8. ConfigSyncSource

The ConfigSyncSource is a special SyncSource that gives the possibility to synchronize a certain number of parameters stored in the DMTree with the server.
To achieve this, the developer of the client has to create an ArrayList of properties to sync and set into the ConfigSyncSource by the setConfigProperties() method.
The methods of the SyncSource read and write the items into the DMTree that depends on the platform implementation: i.e. it is on windows registry for windows, on file for posix...
The server and client must know which properties will be exchanged to handle correctly the items.
An example of property that the client can exchange is
`./Email/Address`
with value
`name@address.com`

In the DMTree will exist a node `Email` that starts from the root of dmtree. Inside the node there is a property `Address` with value `name@address.com`

The ArrayList of properties will contains only the key `./Email/Address` and the sync source will retrieve the value in the dmt.

The sync source implements the methods to retrieves new and updates parameters. It handles the parameters from server that can be added and modified. The deletion of the properties is not admitted and can be achieved erasing the value of the property.

Method	***Description***
ConfigSyncSource(const WCHAR* sourceName, const StringBuffer& applicationUri, AbstractSyncSourceConfig* sc, KeyValueStore* cache =	Constructor: creates a SyncSource from the specified sourceName, set the internal configuration reference from the passed AbstractSyncSourceConfig. The applicationUri refers to the root of the DMTree.

Method	Description
NULL);	If KeyValueStore is null it uses the default one created by the CacheSyncSource
void setConfigProperties(ArrayList properties)	This method allows the developers to set into the SyncSource an ArrayList with all the properties to sync with the server. The Arraylist contains StringBuffer keys.

3.9. FileSyncSource

This class extends the CacheSyncSource abstract class, implementing a plain file datastore. The FileSyncSource is intended to easily synchronize generic files contained inside a given directory on the Client (class member 'dir'). The directory location (full path) must be passed in the constructor.

By default, only the files under 'dir' folder are synchronized (no subfolders). To synchronize files under subfolders, the Client must set the 'recursive' boolean to true, calling the method 'setRecursive'.

For outgoing items, FileSyncSource uses the FileSyncItem class instead of standard SyncItem. This way the large objects are automatically managed by the sync engine, loading the file's content chunck by chunk directly from the file stream (see chapter 3.4 for details).

Depending on the MIME type defined for the source (see SyncSourceConfig::getType()), this class can work in two ways:

1. if the type is "application/vnd.omads-file+xml", the files are wrapped into the OMA File Object representation, to preserve the file name and attributes.
   The item's content will be retrieved using the *FileDataInputStream*.
   The attributes currently supported are:
   - name: the file's name
   - body: the file's content, automatically encoded in base64
   - size: the file size, in bytes
   - modified: the last modification time, in UTC
   The source encoding can be left in plain text (“bin”), as the file's content is already encoded in base64.
   Note: file's creation time is not supported as it is not available on Symbian platform.

2. otherwise, the file is sent as it is to the server (raw file data)
   The item's content will be retrieved using the *FileInputStream*. The source encoding must be set to base64 (“b64”), because the file's content is retrieved as it is.

For incoming items, the format of the file is detected by the content.
Here is an example of the OMA File Data Object, as it will be sent using FileSyncSource:

```
<File>
<name>background.jpg</name>
<modified>20091029T163000Z</modified>
<body enc="base64">/9j/4RDuRXhpZgAASUkqAAgA[...]gYUlABSZigBM0UAf/2Q==</body>
<size>26302</size>
</File>
```

The protected method 'filterOutgoingItem' can be reimplemented by a derived class to eventually filter outgoing items (from Client to Server). Default implementation is empty (no filtering, all files are sent).

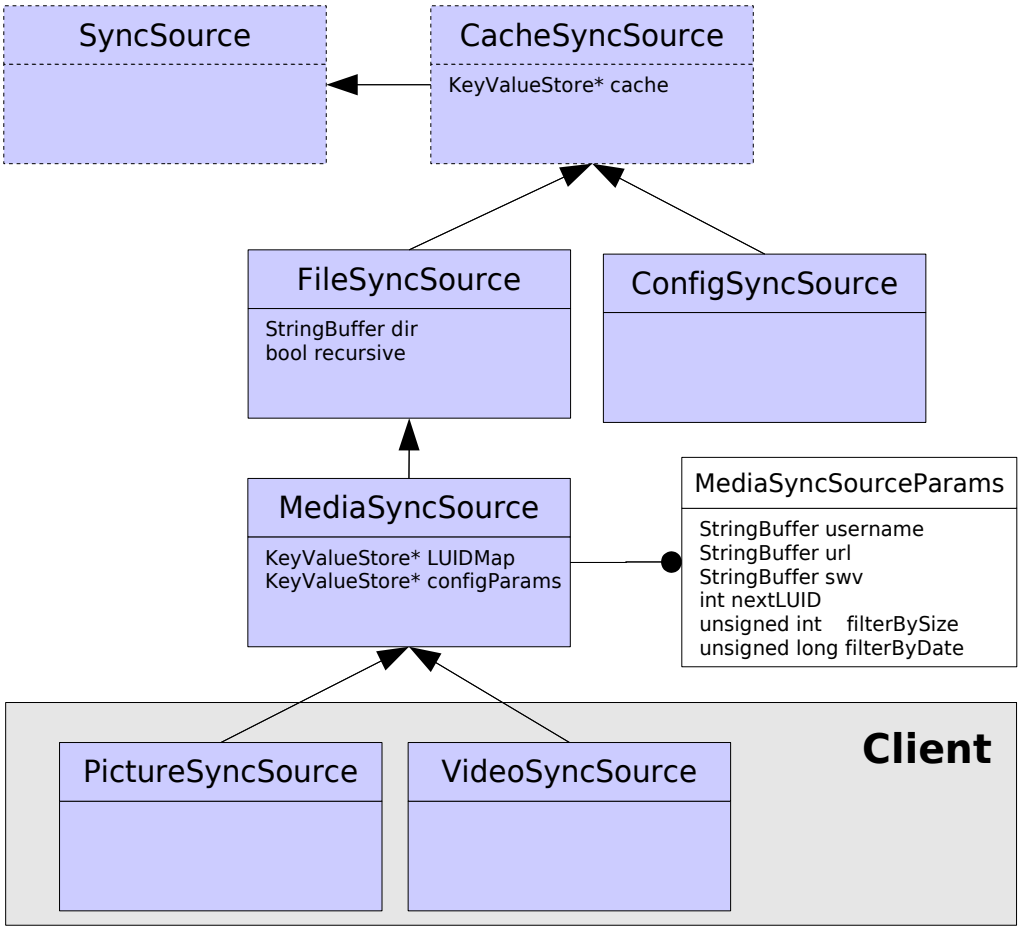


Figure 13: SyncSource classes

3.10. MediaSyncSource

This class extends the FileSyncSource class, to define a special behavior for generic “media files” like pictures or video files, to be synchronized between a mobile Client and a Server. Such files can be really huge, and so the main purpose of this SyncSource is to avoid sending the same file twice, as much as possible. In order to do that, some solutions have been applied to move around constraints set by classic SyncSource implementations.

The goal of the MediaSyncSource is to share the solutions detailed below, so that mobile Clients (like WindowsMobile or Symbian) can take advantage of this class.
A Client just needs to:

– set the 'dir' folder to sync, and optionally the 'recursive' flag
– fill a MediaSyncSourceParams class with the username, ServerURL and swv parameters, and pass it to the MediaSyncSource constructor
– optionally set the parameters *filterBySize* and *filterByDate*, to filter outgoing items

– optionally extend the MediaSyncSource, to define more specific filters (implementing *filterOutgoingItem* and *dynamicFilterItem* methods)

Upload file's content via HTTP

Since v.8.7, the MediaSycSource synchronizes only the file's metadata via syncML, while the file's content is uploaded at the end of the sync using an HTTP POST.
This is an improvement that speeds up a lot the upload of big files (between 2X and 5X, depending on the platform). Also, the data can be sent as it is, with no base64 encoding, which results in no encoding overhead (~30% of data) and less processing, both on client and on server side.
The upload is done in the method *endSync()*, using the *HttpUploader* module, which has a specific implementation on each platform.
The destination URL for the upload is fixed, it's composed based on the params set:
`http://<serverURL>[:port]/upload/<sourceURI>?LUID=<luid>`
The params username and password are used for the HTTP Basic Authentication:
`Authorization: Basic <b64(username:password)>`
The deviceID is added in the http headers:
`deviceId: <deviceID>`
These parameters are passed by the client to the MediaSycSource using the MediaSycSourceParams.
Please see [6] for details about the http upload protocol specifications.

Sync direction: "smart-one-way-from-client" (code 250)

The MediaSyncSource is designed to be used only on mobile clients: no media files are expected from Server to Client.
In case the Server sends files back to the Client, they will be simply rejected (return status code 405 = command-not-allowed) and not inserted.
The Server will not check anchors with this syncmode, so slow-syncs are never executed.

Memory optimization for Large Objects

Extending the FileSyncSource class, MediaSyncSource taks advantage of the Large Object support for outgoing items. Media files are tipically big, so the sync engine will load the file's content chunck by chunk directly from the files stream, instead of loading the whole content in memory (see chapter 3.4 for details).

Cache file inside the folder under sync

The MediaSyncSource cache file is stored inside the folder under sync, as a file "funambol_cache.dat".
The cache is associated with that folder, so if the user goes back to a folder already synchronized some time ago he will be already in sync (with classic implementation, the files found would be sent again to the Server as new items).
This is done to make sure a media file is not sent twice, even if the folder to sync changes (i.e. switching between Phone's memory and Memory Card on a mobile device).

Special items inside the cache

If the Server URL or the username change, the Server will request a slow-sync and in this case the Client MUST send all the items: in this situation the cache file is no more valid and should be cleared. But the cache file is now stored inside the folder to sync and the user could have changed the folder to sync, so we can't just clear it.
The solution is to keep track of the 'username' and 'Server URL' parameters inside the cache, as special items that are used just to check the cache validity before the sync starts (see private method checkCacheValidity()). In case they are different, the cache is cleared.
Additionally, the software version parameter is stored as well (swv), to handle backward compatibility actions (for future use).

These 3 parameters must be passed by the Client, and this is done filling the MediaSycSourceParams class. It will be passed to the class constructor.

Note: these parameters are not removed from the cache during the sync, because if the sync crashes we want to find them in the cache next time. So we just need to keep them in the cache during the sync, and exclude them from thew sync logic (see fillItemModifications() reimplemented).

Outgoing files filtering

Outgoing files can be filtered, in order to avoid sending too much data during a single sync session. Filtering is of two types:

Static filtering
This filtering does not change over time, so it is done at the beginning while scanning the folders involved in sync. This kind of filtering includes:
1.filter by type: files that are not media files are filtered out (*PicturesSyncSource* will send only files with this extensions: .jpg, .jpeg, .gif, .png)
2.filter out the Funambol cache files (they are now stored inside the folder to sync)
See method *MediaSycSource::filterOutgoingItems().*

Dynamic filtering
This filtering may be enabled/disabled by the user, or may change over time. It is executed after the media cache creation, to avoid sending new/delete items in case the filter is enabled/disabled (we MUST not send a media file more than once!).
This kind of filtering includes:
1.filter by size: files with size bigger than *params::filterBySize* are filtered out
2.filter by date: files with a modification time earlier than *params::filterBySize* are filtered out
Clients can just set the two params filterByDate and filterBySize when calling the constructor of *MediaSycSource*, to specify custom filtering (default values is 0 = filter disabled).
See method *MediaSycSource::dynamicFilterItem().*

Some considerations on the implementation:
- during a slow sync the filtering is done on allItems list
- during normal sync (smart one way from client), the filtering must be executed on the new items list, as well as the updated and deleted items list
- files filtered will always be in the newItems list (the cache does not have them) and will be excluded in every sync. So the filters are executed every time, as they are dynamic and may change.
- a file previously included in sync may be modified so that it will be now filtered out. So the list of updatedItems must be checked as well.
- a file previously included in sync may be deleted by the user, and a filter may be changed so that file should be filtered out now. So the list of deletedItems must be checked as well.
- in the case of deleted items, the file is no longer available so most of the filters cannot be executed: however the filterByDate is still valid, as the file's last modification time is available from the MediaSycSource's cache (it's the local file's signature, see below).

Local files signature

Implements *CacheSyncSource::getItemSignature()* method: the signature is the file last modification time (CRC computing can be too expensive for big files such as media files).

Unique item's key (LUID)

The item's key used by FileSyncSource is the name of the file. Switching the folder to sync, it may be that the same key is associated to different files with the same name.
To distinguish between pictures with the same name and path, a mapping between LUID and full path of each file synchronized is used: the LUID is a unique item's key sent to the Server:

it's an incremental number, different for every file (so first file sent will have the key = “1”, then “2” and so on).

The mapping is stored in a PropertyFile "funambol_luid.dat", containing [file name;LUID] pairs for each item in cache. It's accessed while sending items to the Server, to get the LUID value from the file name (implements 'fillSyncItem'), and when receiving status codes back from the Server, to retrieve the file name from the LUID (implements 'setItemStatus'). For incoming items, the 'getKeyAndSignature' method is reimplemented to retrieve the file name from the LUID sent by the Server.
To avoid this mapping PropertyFile to grow indefinitely, it's refreshed every time the cache is saved (see 'refreshLUIDMap' method): all entries that have no correspondence in the cache are removed.

The LUID map is stored inside the folder to sync, so that we keep the mappings for the current directory even if the folder was changed. Everytime a new LUID is generated, the parameter 'nextLUID' is updated, so we are sure we never use a LUID already taken. The 'nextLUID' value must be stored in a indipendent place from the folder to sync, so it's saved under the platform config dir inside a PropertyFile.

Handle file renames (TODO)

If a media file is just renamed, on the next sync the Client would send a new item and a delete item. This is not efficient, to avoid this we need to check new and deleted items before every sync, to find 2 items with the same signature (so it's the same file renamed).

Manage “quota exceeded on Server” error

In case the storage limit is reached on the Server, an error status code 420 (means “device full”) is sent to the Client. MediaSyncSource is able to react in case of this status code, and stops sending new items to the Server (see fillSyncItem() method).
An internal error is raised in this situation, so the sync process will continue till the end but no more items are exchanged with the Server. Then, the error code 420 is returned to the Client by the endSync() method, so Clients can execute specific actions to handle this situation (like displaying an error message, or stop the push/schedule services for this source).

Note that lastErrorCode is immediatly set when the wrong status is received, in order to stop sending the current item even if it's a large object one (and so it's automatically read from the proper input stream). The sync is not aborted but it continues till the end, so the MediaSyncSource's cache files can be updated regularly.

Resume upload of a single file (future)

As a consequence of “Upload file's content via HTTP” improvement, in case of HTTP errors for an item the next time the upload of that file can be resumed from the point that was interrupted. The client should always do a first POST with no data, so the Server can return the status code “resume” for this item, specifying the range of data still missing in the http headers. The client will read the headers, and post only the interested bytes for that item (To be better defined).
This will avoid to send again the data for the interrupted item, which can be very large, improving the performance.

3.11. HttpUploader

The *HttpUploader* module is used to upload a given amount of data via an HTTP POST request. This class is used by *MediaSyncSource*, to upload media files at the end of the sync. There are a number of parameters, used to properly set the destination URL and the HTTP headers. For a detailed description of the HTTP upload protocol, please refer to the pictures sync design document [6].

The destination url is composed like:
“http://<serverURL>[:port]/upload/<sourceURI>?LUID=<luid>[;jsession=<sessionId>]”
Where the “luid” is the local UID of the file to send, and the “sourceURI” is the server's source name, both passed by the syncsource using the module. The “sessionId” is optional and not currently used, though supported (please see below).
It uses a basic http authentication (“Authorization” http header), using the proper BasicHttpAuthentication class. The “deviceId” is also set as a custom request header, it is used by the Server to match the client once the http request is submitted.
The HttpUploader uses the HttpConnection class to handle the http request, so its implementation is common on all OS (HttpConnection has an implementation on each platform).

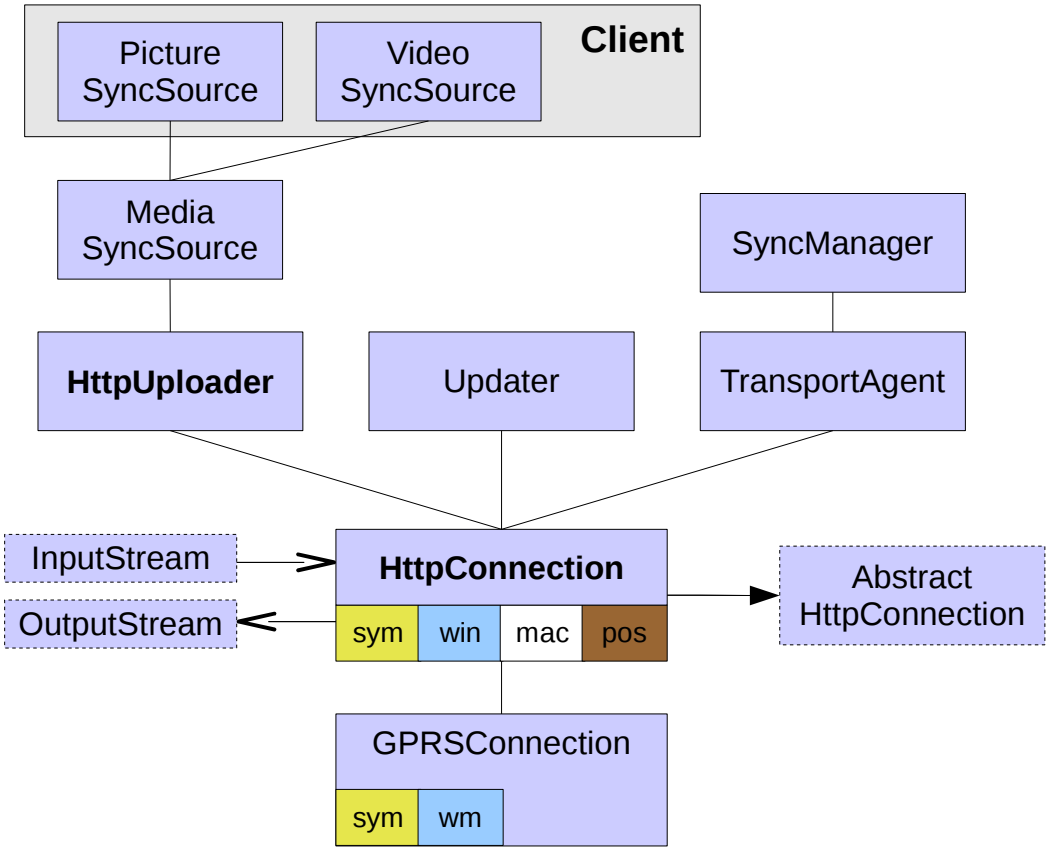


Figure 14: HttpUploader and HttpConnection

The method upload() is the main function to make the HTTP POST request: an *InputStream* is passed as a parameter, so the data is directly read from the stream (to avoid allocation of big files in memory: the stream passed by MediaSyncSource is typically a FileInputStream). The implementation of *HttpConnection* allows to send the data chunk by chunk, the size of chunks can be specified setting the HttpUploader parameter *maxRequestChunkSize*. The http response returned by the Server is just ignored (the HttpConnection's OutputStream). The HttpUploader is also able to manage the JSessionId parameter returned by the Server, in order to execute subsequent uploads without the need of a basic http authentication everytime (please see [6] for details). Anyway it is disabled by default, so the authentication is always added to the http requests, if not differently specified.

3.12. Synchronization Report

A SyncReport is used to summarize all results of a single synchronization. During the synchronization process, all results about different operations are stored in a SyncReport object, so the client will be able to get these informations at the end. Accessing this object a client can easily know for example the outcome of each source synchronized, retrieve the number of items modified on both sides, and the status code of each one. The class diagram of SyncReport architecture is illustrated in Figure 15.

The SyncClient owns an instance of SyncReport, so each synchronization process managed by a SyncClient is associated to a unique SyncReport. It is passed by reference to the SyncManager, so it can be filled during the sync session. On client side, after calling the SyncClient.sync() method to start a synchronization, the correspondent report can be retrieved calling SyncClient.getSyncReport() method.

Note: the *SyncReport* is available only during the sync process, it will be destroyed at the end of the sync if the *SyncClient* is deleted (since it's owned by the *SyncClient* itself). At the end of the *SyncClient::sync()* method, the sync results (the global *lastErrorCode* and also the last error codes of each source synced) are copied to the *SyncManagerConfig*, so it will be persisted in the configuration. So it will be available even after the sync process ended.

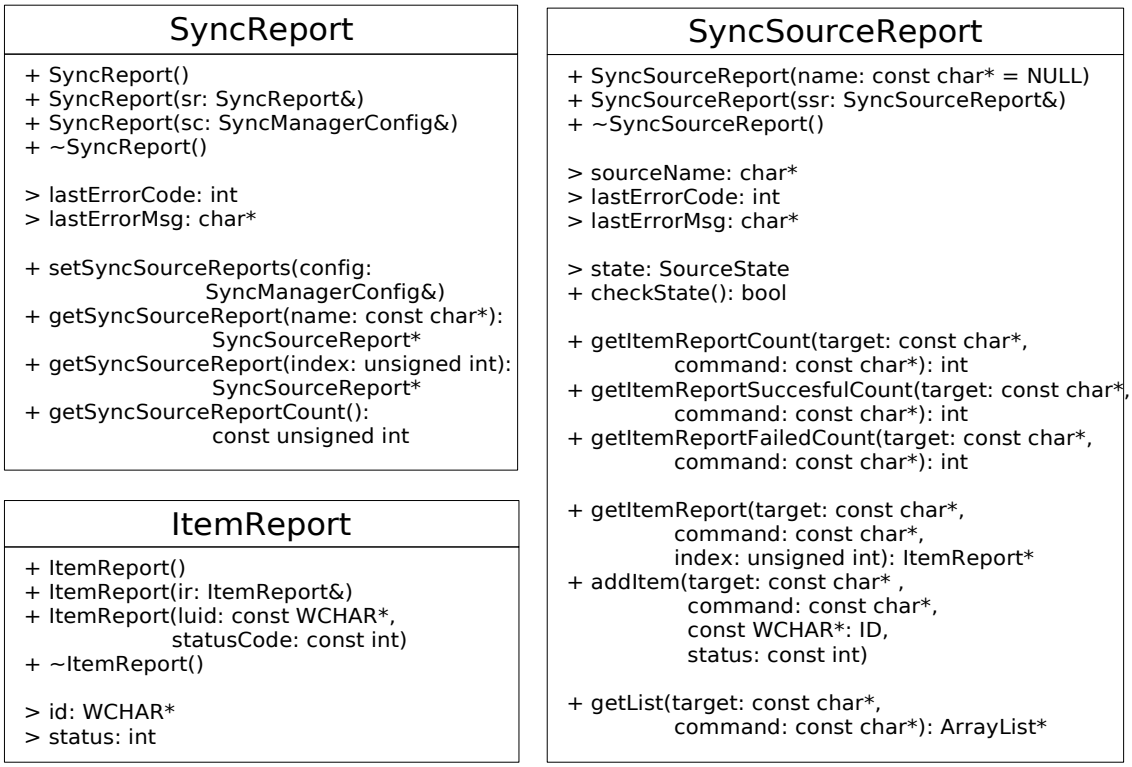


Figure 15: SyncReport class diagram

SyncReport

This class contains informations about the global sync process, and an array of SyncSourceReport objects: each one rapresent the report of each SyncSource synchronized.
SyncReport public methods are described in the following table.

Method	Description
SyncReport()	Constructor, creates a new empty SyncReport object.
SyncReport(SyncManagerConfig& sc)	Constructor, creates a new SyncReport object and call setSyncSourceReports().
~SyncReport()	Destructor.
setSyncSourceReports (SyncManagerConfig& config)	Creates the array of syncSourceReport based on configuration passed.
SyncSourceReport* getSyncSourceReport (const char* name)	Returns a pointer to the internal SyncSourceReport object, selected by its name.
SyncSourceReport* getSyncSourceReport (const int index)	Returns a pointer to the internal SyncSourceReport object, selected by its index in the array.
unsigned int getSyncSourceReportCount()	Returns the number of SyncSourceReport objects.

SyncSourceReport

SyncSourceReport class rapresent the report of each SyncSource synchronized. It also contains six lists of ItemReport (list for new, modified, deleted items on server and on client), and methods to get the number of items of each list. It contains also 2 additional lists for items uploaded/downloaded via HTTP: these are used only by MediaSycSource for the sync of media items, since now the items data is transferred at the end of the sync via http (and so the total number of items transferred via http can be different from the number of metadata synced via syncML). For all other sources, these 2 lists are just ignored.
The member “state” is used to know if the SyncSource is currently active (used in synchronization), inactive (ignored) or if some errors occurred (so will be skipped in sync).
SyncSourceReport public methods are described in the following table.

Method	Description
SyncSourceReport(const char* name = NULL)	Constructor, creates a new SyncSourceReport object. Set sourceName if name is passed.
~SyncSourceReport()	Destructor.
bool checkState()	Returns true if source is active (current state = SOURCE_ACTIVE)
int getItemReportCount (const char* target, const char* command)	Return the number of ItemReport for a specific list (based on target and command).
int getItemReportSuccessfulCount (const char* target, const char* command)	Return the number of ItemReport with status code successful for a specific list (based on target and command).
int getItemReportFailedCount (const char* target, const char* command)	Return the number of ItemReport with status code NOT successful for a specific list (based on target and command).
addItem (const char* target, const char* command, const WCHAR* ID, const int status)	Add a single ItemReport to the correct list (based on target and command). The ItemReport is created from ID (luid) and status.

Method	Description
	This method is called by API each time an item status is obtained (both client and server status that come from a modification operation).
ItemReport* getItemReport(const char* target, const char* command, unsigned int index)	Returns the internal pointer to the ItemReport from its index.
ArrayList* getList(const char* target, const char* command)	Used to get access on the right list, based on target and command (see notes below for correct values).

Notes:
“target” values are: CLIENT - SERVER
“command” values are: COMMAND_ADD – COMMAND_REPLACE – COMMAND_DELETE – HTTP_UPLOAD – HTTP_DOWNLOAD
"state" member can be one of: SOURCE_ACTIVE - SOURCE_INACTIVE - SOURCE_ERROR

ItemReport

ItemReport class rapresents the result information on a single item synchronized, such as the luid of the item and its status code (200/201/500...).

3.13. Item Content Transformation

In order for the DS Agent to be able to transform the data obtained from a SyncSource in some other format, the concept of DataTransformer is introduced. A DataTransformer is associated to a syncsource. The class diagram of the DataTransformer architecture is illustrated in Figure 16.

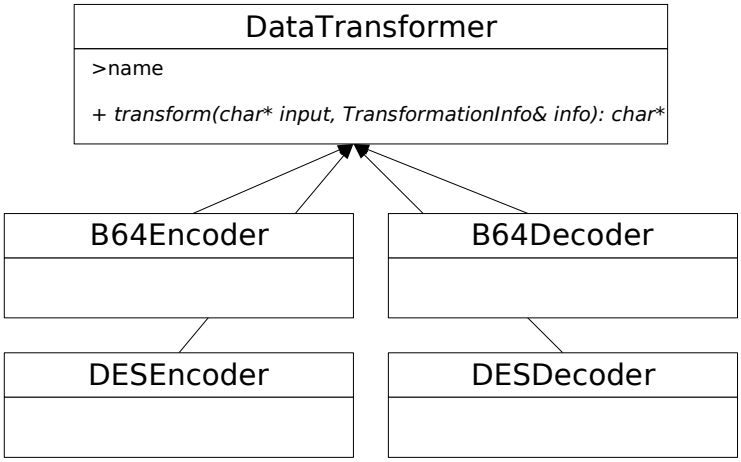


Figure 16: Data transformation class diagram

A DataTransformer is identified by a name and it may act as an encoder or decoder. The following names are defined (case sensitive!):

- b64 : base64 encoder/decoder

- des : DES cipher/decipher

In addition to the classes shown in the figure, a factory class is provided to create encoders and decoders given the transformer name. Such factory has the following interface:

Method	Description
DataTransformer* getEncoder(wchar_t* name)	Creates and returns the DataTransformer able to encode data to the format given by name; such instance is allocated with the new operator and must be deleted by the caller with the delete operator.
DataTransformer* getDecoder(wchar_t* name)	Creates and returns the DataTransformer able to decode data from the format given by name; such instance is allocated with the new operator and must be deleted by the caller with the delete operator.

Scope of this architecture is that at the syncsource level, the encoding/decoding of the content of each single item is hidden. The engine will take care of such conversions. The syncsource will only be influenced (or will only influence) the content type.

The transformation is implemented in the *transform()* method, which takes the input data plus a pointer to an optional parameter that the transformer may need to perform the processing. Note that a transformer may or may not be required to allocate the memory needed for the transformed data. See the implementation notes for instruction on how this is done and, therefore, how the allocated memory should be freed. In case of any error a NULL pointer will be returned.

TransformationInfo is a structure with the following fields:

Field	IN/OUT	Description
long size	IN/OUT	Size in bytes of the data to transform in input; the size in bytes of the transformed data in output.
wchar_t* username	IN	Pointer to a string containing the username of the user performing the current synchronization.
wchar_t* password	IN	Pointer to a string containing the password of the user performing the current synchronization.
wchar_t* sourceName	IN	Pointer to a string containing the name of the source being encoded/decoded.
BOOL newReturnedData	OUT	TRUE if the transformer has allocated new memory for the returned data. If so, the transformer is required to use the new operator so that the caller can release the allocated memory with the delete operator.

The following error codes are defined:

ERR_DT_UNKNOWN: the requested data transformer is unknown

ERR_DT_FAILURE: the transformer was not able to perform the transformation; see the transformer implementation description for additional information

Basic algorithms for encoding and decoding are provided by the basic component of the library (see section 2).

3.14. Configuration DMTree

This section describes how the configuration information of the data synchronization layer are stored in the DM tree. DM and DM tree concepts are described later in section 3. The DM Tree structure that keeps the configuration information needed by the framework classes is represented below.

```
./spds/sources
  + source
    + name
      - <source name>
    + uri
      - <source uri>
    + syncModes
      - <sync modes>
    + type
      - <content type>
    + version
      - <content type version>
    + sync
      - <sync type: “none”, “slow”, “two-way”...>
    + encodings
      - <b64, plain/text>
    + last
      - <integer: last time stamp>
    + supportedTypes
      - <all supported source types: “type1:ver1,type2:ver2,...”>
    + fieldLevel
—     - <bool: true if source suports field level replace>

  + mailSource
    + name
       - <source name>
    + uri
      - <source uri>
    + syncModes
      - <sync modes>
    + type
      - <type>
    + sync
      - <sync type: “none”, “slow”, “two-way”...>
    + encodings
      - <b64, plain/text>
    + last
      - <integer>
    + downloadAge
      - <integer>
    + bodySize
      - <integer>
    + attachSize
      - <integer>
    + mailMaxMsgSize
      - <integer>
    + Inbox
      - <integer>
    + Outbox
      - <integer>
    + Sent
      - <integer>
    + Trash
      - <integer>
    + Draft
      - <integer>
```

```
./spds/syncml

  + Conn
    + syncURL
      - <syncurl>
    + proxyHost
      - <proxy host>
    + proxyPort
      - <proxy port>
    + proxyUsername
      - <proxy username>
    + proxyPassword
      - <proxy password>
    + useProxy
      - <1 | 0>
    + checkConn
      - <if connection needs to be checked: 1 | 0>
    + responseTimeout
      - <integer value>
    + readBufferSize
      - <integer value>
    + userAgent
      - <user agent>

  + Auth
    + username
       - <username>
    + password
       - <password>
    + serverID
       - <serverID>
    + serverPWD
       - <serverPWD>
    + clientNonce
       - <serverNonce>
    + serverNonce
       - <serverNonce>
    + isServerAuthRequired
       - <1 | 0>
    + clientAuthType
       - <authentication type>
    + serverAuthType
       - <authentication type>

  + DevInfo
    + devID
      - <device id>
    + man
      - <manufacturer>
    + mod
      - <model>
    + dsV
      - <implemented DS version>

  + DevDetail
    + devType
      - <device type>
    + oem
      - <original device manufacturer of the device>
    + fwv
      - <firmware version of the device>
    + swv
      - <software version of the device>
    + hwv
      - <hardware version of the device>
    + loSupport
      - <Large Object support: 1 | 0>

  + Ext
    + begin
```

```
  - <last synchronization begin timestamp>
+ end
  - <last synchronization end timestamp>
+ firstTimeSyncMode
  - <first time sync mode>
+ maxMsgSize
  - <max message size>
+ maxModPerMsg
  - <max # of items in a message from client>
+ devInfHash
  - <the hash for devinf>
+ logLevel
  - <log level>
+ encryption
  - <1 | 0>
+ maxObjSize
  - <max size of an object>
+ utc
  - <if client handle UTC timestamps: 1 | 0>
+ nocSupport
  - <Number of changes support: 1 | 0>
+ verDTD
  - <DTD version in use>
```

Notes:

1. If proxy host is an empty string, it is intended as “do not use proxy”.
2. <preferred authentication type> is one of: “syncml:auth-md5”, “syncml:auth-basic”.
3. <log level> is one of: 0 – 1 – 2 (none, info, debug).
4. <Integer value> means 0 get the default value, otherwise the value set

3.15. Client Capabilities Handling

The SyncML specifications states:

The sync client MUST send its device information to the server when the first synchronization is done with a server or when the static device information has been updated in the client. The client MUST also be able to transmit its device information if it is asked by the server. The client SHOULD also support the receiving of the server device information.

Therefore, we have two scenario:

1. The client connects to a new server;
2. The server issues a Get command requesting for client capabilities.

In both cases the client shall send its capabilities. Plus, if the client has already sent its capabilities to a server in a previous synchronization, the DevInf is not sent anymore.

For case 1, the SyncManager determines if it has to include client capabilities looking at the value of the devInfHash: if its value is changed it means that some DevInf parameter is changed, or the syncUrl property has been modified.
Note: the syncUrl parameter has been added to the generation of devInfHash, to ensure the client sends device informations when connecting to a different server.
However a client application can decide to reset devInfHash when any other relevant configuration settings is changed.

In case 2, the SyncManager determines if it has to send the client capabilities inspecting the messages received by the server during initialization. If any contains a Get command, in the reply the SyncManager will send a Results command containing the DevInf build from the various configuration objects.
The device informations contains details about the device: (manifacturer, firmware version, etc.), about the DS capabilities (support of LargeObjects, NumberOfChanges, etc.) and about the DataStore (aka, the SyncSource).

The DataStore is a complex item, containing for each source informations about::

- the source name and remote uri
- the supported mime types and version, with the preferred one (for rx and tx) e.g.: text/x-vcard, 2.1
- if the DataStore supports field level update:
    - if true, the server is allowed to send only the modified properties of an item, and the client is able to change them keeping the rest of the item untouched
    - if false, the server must send all the item to the client.
- the list of supported properties.

The informations are taken from the configuration data (see par. 3.6), except the DataStore properties, which are not known by the API and must be set by the client.
The client has to provide an ArrayList of CTCap object for each type of data supported by that SyncSource. Each CTCap object contains the informations above (please see the library documentation for more info).
Besides the get/set methods for the ctCap property, which is the full ArrayList of contant-type capabilities set for the source, the SyncSourceConfig provides the utility method addCtCap() which allows to easily add them.

The minimal operation a client must perform to add the mandatory informations for a source supporting only one content type is to call addCtCap() passing the list of properties supported.
The other parameters are optionals, and defaults to the values stored in the config itself (type, version and fieldLevel, see above).

If the source supports more types, the subsequent calls must specify also the values different from the preferred ones.

3.16. Server Capabilities Handling

Server device informations (Server capabilites) SHOULD be sent by the Server to the Client. These capabilites are very important, they're the primary way to know which Server are we synchronizing to.

The Client can ask for Server capabilites via a “Get” SyncML command, and the Server SHOULD always reply with a “Results” command containing a “DevInf” tag with all its capabilites.
Alternatively, the Server could send its capabilites via a “Put” command (for example if some have changed and need to be notified to the Client (**since Funambol Server v.8.2**).

The Server capabilities handling is managed autonomously by Funambol SDK C++ APIs, during synchronization: the parameters are stored inside the configuration (ServerConfig object, see 3.6) and so they're available to Clients.
The Server device info are requested if at least one of the following situation happens:

1. Mandatory Server information is missing
   Actually only the Server version (swv) is considered a required parameter, so the Server caps are requested if the Server version is empty.

2. The Server address has changed
   To check if the Server address has changed, the last Server URL is stored in the ServerConfig object. If it's changed, the Server caps currently stored are also cleared because they're no more valid.

3. The Client forced it, using the config param “forceServerDevInfo”
   Clients can force to ask for Server capabilites, by calling the method setForceServerDevInfo(true). This way, the capabilites are always requested to the Server, even if not necessary.

Note: if using DMTClientConfig implementation to read/save configuration, the “forceServerDevInfo” parameter is not persisted in memory because it should just be set to true everytime the Client needs to force it (otherwise the Server caps would always be requested, and this is not the general behavior).

Server Capabilites are stored in the Client's configuration DMTree as shown in figure 17:

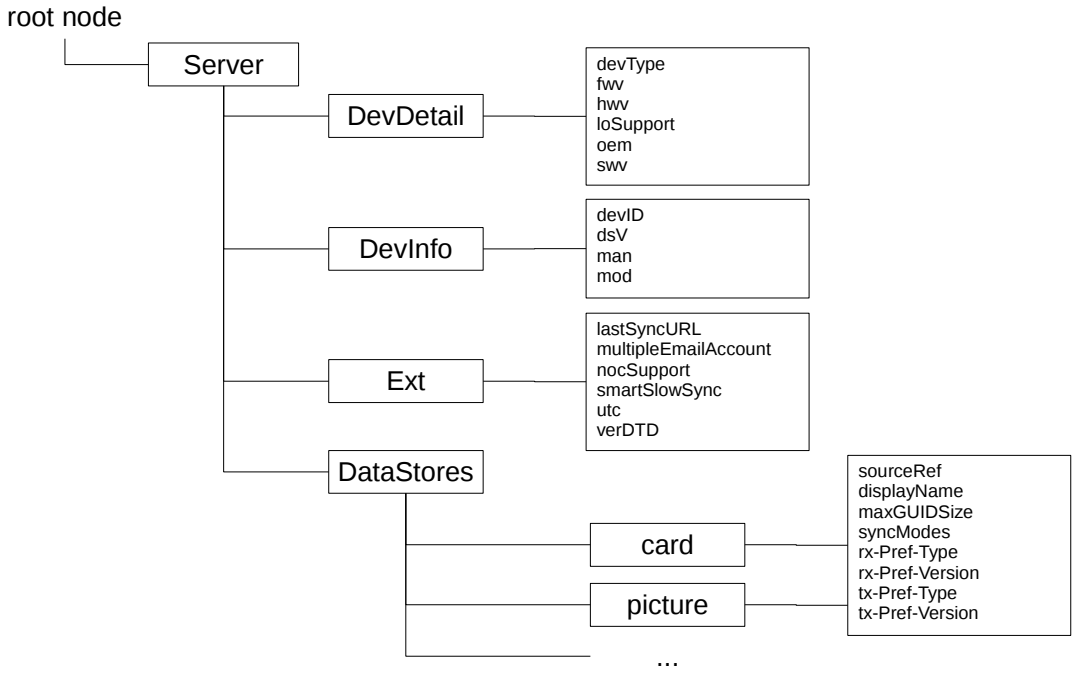


Figure 17: DMTree of Server capabilities

Server dataStores rapresents the available Server syncSources and their preferences. Everytime the Server dataStores are received we cleanup the existing DataStores config tree and completely replace it with the new data, in order to reflect the current Server dataStores. Funambol Server sends the dataStores for all the supported syncsources (and not just for the sources under sync) since v.8.2.
The dataStores parameters *rx-Pref-type* and *rx-Pref-version* are particularly important, since they rapresent the Server's preferred format for received data fof that syncsource: Clients may want to use those informations to select the right data format for each syncsource.

3.17. Synchronization Events Notification

As illustrated in Figure 1 the architecture of an application based on the Funambol API is layered; in order to let the application level know about what happens in the layers below, an event based notification mechanism is implemented. This is based on events and listeners.

Synchronization Events

The events described in the table below are defined.

Event	Event Type	Description
SyncEvent	SYNC_BEGIN	Fired to notify that the synchronization process started. It shall bring the following info: • datetime: a Date object representing the date and time of the beginning of the sync
SyncEvent	SYNC_END	Fired to notify that the synchronization process ended. It shall bring the following info: • datetime: a Date object representing the date and time of the end of the sync
SyncEvent	SYNC_ERROR	Fired to notify that the engine encountered a not blocking error (the engine does not throw an exception for it). It shall bring the following info: • datetime: a Date object representing date and time when error occurring • message: the error message
SyncEvent	SEND_INITIALIZATION	Fired to notify that the initialization package was correctly set and processed. It brings the following info: • datetime: a Date object representing the date and time of the end of the initialization processing
SyncEvent	SEND_MODIFICATION	Fired to notify that the modifications package was correctly set and processed. It brings the following info: • datetime: a Date object representing the date and time of the end of the modifications processing
SyncEvent	SEND_FINALIZATION	Fired to notify that the final package was correctly set and processed. It brings the following info: • datetime: a Date object representing the date and time of the end of the finalization processing
SyncTransportEvent	SEND_DATA_BEGIN	Fired to notify that the engine started to send data to the server.

Event	Event Type	Description
		This event shall notify the following events: • going to start sending data It shall bring the following info: • data length: the total amount of data to send
SyncTransportEvent	SEND_DATA_END	Fired to notify that the engine has sent all data to the server. It shall bring the following info: • data length: the total amount of data sent
SyncTransportEvent	RECEIVE_DATA_BEGIN	Fired to notify that the engine started to receive data from the server. This event shall notify the following events: • going to start reading data It shall bring the following info: • data length: the total amount of data to receive when it is first fired;
SyncTransportEvent	DATA_RECEIVED	Fired to notify that the engine is receiving data from the server. This event shall notify the following events: • received a certain amount of data It shall bring the following info: • data length: the bytes received
SyncTransportEvent	RECEIVE_DATA_END	Fired to notify that the engine has received all data from the server. It shall bring the following info: • data length: the total amount of data received
SyncSourceEvent	SYNC_SOURCE_BEGIN	Fired to notify the beginning of the synchronization of a particular syncsource. It shall bring the following info: • source URI: the source being synchronized • sourcename: the source name • mode: type of the performed sync

Event	Event Type	Description
		• date: a Date object representing the date and time of the beginning of the synchronization
SyncSourceEvent	SYNC_SOURCE_END	Fired to notify the end of the synchronization of a particular syncsource. It shall bring the following info: • source URI: the source being synchronized • sourcename: the source name • mode: type of the performed sync • date: a Date object representing the date and time of the end of the synchronization
SyncSourceEvent	SYNC_SOURCE_SYNCMODE_REQUESTED	Fired to notify a SyncMode change, requested by server for a particular syncsource. It shall bring the following info: • source URI: the source being synchronized • sourcename: the source name • mode: type of sync requested • date: a Date object representing the date and time of the end of the synchronization
SyncSourceEvent	SYNC_SOURCE_TOTAL_CLIENT_ITEMS	Fired to notify the total number of items that client will transfer for a particular syncsource (number of changes). It shall bring the following info: • source URI: the source being synchronized • sourcename: the source name • mode: type of the performed sync • data: the number of items. • date: a Date object representing the date and time of the end of the synchronization
SyncSourceEvent	SYNC_SOURCE_TOTAL_SERVER_ITEMS	Fired to notify the total number of items that server will transfer for a particular syncsource (number of changes).

Event	Event Type	Description
		It shall bring the following info: • source URI: the source being synchronized • sourcename: the source name • mode: type of the performed sync • data: the number of items. • date: a Date object representing the date and time of the end of the synchronization
SyncItemEvent	ITEM_ADDED_BY_SERVER	Fired to notify that an item addition has been received. It shall bring the following info: • source URI: the source the item belongs to • item key: the item key (GUID from the server)
SyncItemEvent	ITEM_DELETED_BY_SERVER	Fired to notify that an item deletion has been received. It shall bring the following info: • source URI: the source the item belongs to • item key: the item key (GUID from the server)
SyncItemEvent	ITEM_UPDATED_BY_SERVER	Fired to notify that an item update has been received. It shall bring the following info: • source URI: the source the item belongs to • item key: the item key (GUID from the server)
SyncItemEvent	ITEM_ADDED_BY_CLIENT	Fired to notify that an item addition has been sent. It shall bring the following info: • source URI: the source the item belongs to • item key: the item key (LUID)
SyncItemEvent	ITEM_DELETED_BY_CLIENT	Fired to notify that an item deletion has been sent. It shall bring the following info: • source URI: the source the item belongs to • item key: the item key (LUID)
SyncItemEvent	ITEM_UPDATED_BY_CLIENT	Fired to notify that an item update

Event	Event Type	Description
		has been sent. It shall bring the following info: • source URI: the source the item belongs to • item key: the item key (LUID)
SyncStatusEvent	CLIENT_STATUS	Fired to notify a status to send. It shall bring the following info: • command: the command the status relates to • status code: the status code • item key: the key of the item this status relates to if it is in response of a modification command. • SourceUri: the source uri
SyncStatusEvent	SERVER_STATUS	Fired to notify a received status command. It shall bring the following info: • command: the command the status relates to • status code: the status code • item key: the key of the item this status relates to if it is in response of a modification command. • SourceUri: the source uri

Those events are represented by the classes of Figure 18.

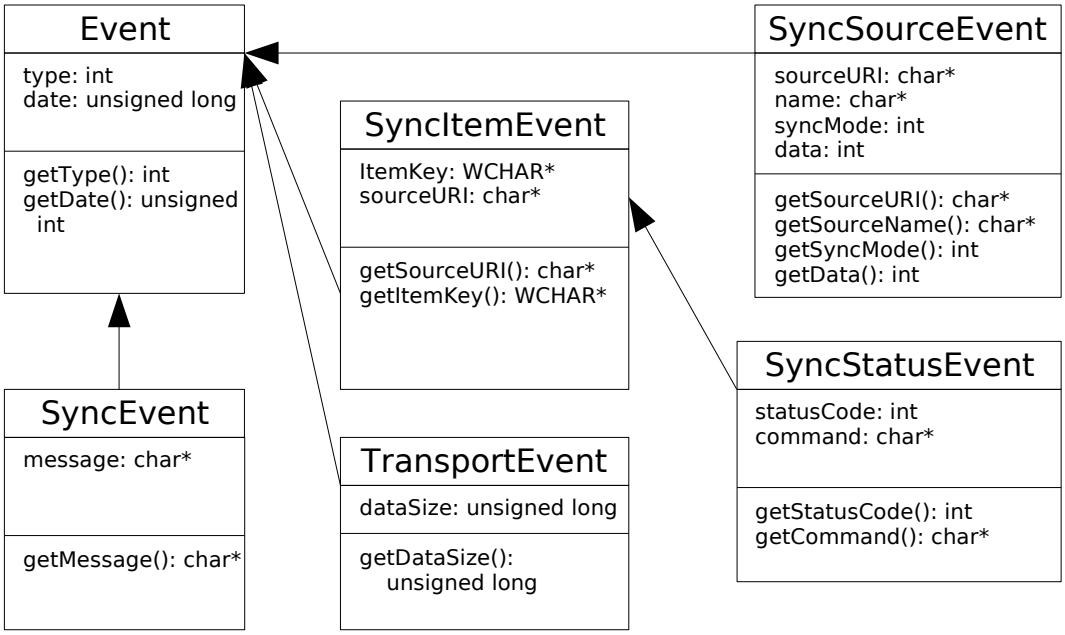


Figure 18: Events class diagram

Event

Is the abstract base class for any other event subtype. It is defined by an event code and the timestamp of when the event is generated. The timestamp is in the numeric form of a datetime.

SyncEvent

This event is used to notify the listeners for the principal states of the synchronization process.

The following event types are defined:

Event Type	Value	Description
SYNC_BEGIN	0x0001	Fired when the synchronization process start. It's created and fired by the SyncManager when a client the sync() method is invoked.
SYNC_END	0x0002	Fired when the synchronization process ends. This type of event is created by the SyncManager and is fired when the synchronization process ends.
SYNC_ERROR	0x0003	Fired by the SyncManager when an error occurs.
SEND_INITIALIZATION	0x0004	Fired before sending the initialization message. It is created and fired by the SyncManager before calling syncInitialization()
SEND_MODIFICATION	0x0005	Fired before sending the modification message. It's created and fired by the SyncManager before calling syncModifications()
SEND_FINALIZATION	0x0006	Fired before sending the finalization message. It's created and fired by the SyncManager before calling

Event Type	Value	Description
		syncMapping()

These events bring the following information:
- message: only if the event is a SYNC_ERROR, null otherwise

TrasportEvent

This event is used to notify the listener of the sending or receiving of data to/from the server.

The following event types are defined:

Event Type	Code	Description
SEND_DATA_BEGIN	0x0011	This type is fired before sending a message to the server; it brings the length of the message.
SEND_DATA_END	0x0012	This type is fired when the client has sent all data to the server. It brings the length of the message.
RECEIVE_DATA_BEGIN	0x0013	This type is fired when the client receives the content length of the response message from the server. It brings the content length of the message.
DATA_RECEIVED	0x0014	This type is fired for each block of data received from the server. It brings the length of the data read.
RECEIVE_DATA_END	0x0015	This type is fired when the client has received all data from the server. It brings the length of the message.

SyncSourceEvent

This event is used to notify the listener of the beginning and the end of the synchronization for a particular SyncSource.
The following event types are defined:

Event Type	Code	Description
SYNC_SOURCE_BEGIN	0x0021	This type is fired 2 times for each syncsource: before sending client modifications and before receiving server modifications for the syncsource.
SYNC_SOURCE_END	0x0022	This type is fired 2 times for each syncsource: after sending all client modifications and after receiving all server modifications for the syncsource.
SYNC_SOURCE_SYNC MODE_REQUESTED	0x0023	This type is fired when a syncMode Alert is received by server.
SYNC_SOURCE_TOTAL _CLIENT_ITEMS	0x0024	This type is fired by SyncSource before sending items, when the total number of items is detected. It brings the number of items (data).
SYNC_SOURCE_TOTAL _SERVER_ITEMS	0x0025	This type is fired when NOC (number of changes) tag is received from server. It brings the number of items (data).

These events bring the following additional information:
- sourceUri: uri of the syncSource
- syncMode: type of the performed sync

SyncItemEvent

This event is used to notify the listener when a item is added/updated/deleted by a command received from the server or when an item has to be added/updated/deleted on the server.

The following event types are defined:

Event Type	Code	Description
ITEM_ADDED_BY_SERVER	0x0031	This type is fired when a new item is added on the client.
ITEM_DELETED_BY_SERVER	0x0032	This type is fired when an item is deleted on the client.
ITEM_UPDATED_BY_SERVER	0x0033	This type is fired when an item is updated on the client.
ITEM_ADDED_BY_CLIENT	0x0034	This type is fired for each new item detected on the server (as soon as it is detected).
ITEM_DELETED_BY_CLIENT	0x0035	This type is fired for each item deleted on the server (as soon as it is detected).
ITEM_UPDATED_BY_CLIENT	0x0036	This type is fired for each item updated on the server (as soon as it is detected).
ITEM_UPLOADED_BY_CLIENT	0x0037	This type is fired for each item uploaded via http from the client to the server (it is used in the new implementation of the media http upload)

These events bring the following additional information:
- source URI: the source the item belongs to
- item key: the item key

SyncStatusEvent

Fired to notify a sending or a received status command.

The following event types are defined:

Event Type	Code	Description
CLIENT_STATUS	0x0041	This type is fired for all status command that the client sends to the server
SERVER_STATUS	0x0042	This type is fired for all status command received from the server.

These events brings the following additional information information:
- command: the command the status relates to
- status code: the status code

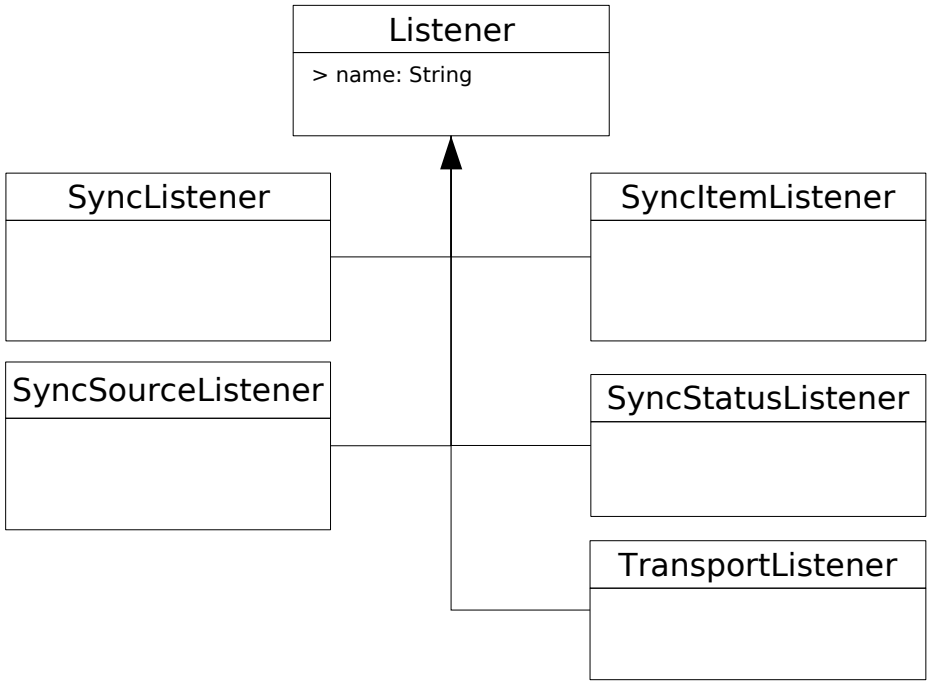


Figure 19: Listeners class diagram

Listeners

For each event described in the above sections a corresponding listener is defined. The class diagram of such objects is shown in Figure 19. All the listeners derive from a base Listener, from which they inherit a name property, that is used in the registration mechanism described in the next section. The name is optional and the default is the empty string, to keep the compatibility with pre 7.1 library version.

Note that the event instances are passed by reference. This reference must be considered valid only for the scope of the call. Outside such scope no assumption can be made, therefore if some of the event information is needed elsewhere, it must be copied.

The listener classes are to be considered abstract classes, but they also provide a standard empty implementation for each method. In this way developers can simply overwrite only the methods they are interested to.

Listener registration

Starting from version 7.1 of the Funambol SDK, it is possible to register multiple listeners instances for each class, using different names.

In order to allow the application developer to register/unregister a listener, the singleton class ManageListeners is available, with methods to set, unset or get the listeners.
The multiple listener mechanisms is based on the listener name: setting a new listener with a different name, adds a listener to the chain (all the registered listener are called for each event fired, see next paragraph).
To set a listener, call the proper set method with a newly allocated Llistener. The pointer will be owned by the ManageListener and released when another listener with the same name is set, or when the unset method is called. For example:

set a new listener:

```
ManageListener::getInstance().setSyncListener(
                                new CustomSyncLlistener(“myname”));
```

re-set the same listener:

```
ManageListener::getInstance().setSyncListener(
                                new OtherSyncLlistener(“myname”));
```

unset the same listener:

```
ManageListener::getInstance().unsetSyncListener(“myname”);
```

Firing events

To fire an event from inside the API, the following global scoped functions are provided.

Method	*Description*
fireSyncEvent(msg: const char*)	Fires a SyncEvent
fireTransportEvent(size: unsigned long)	Fires a TransportEvent
fireSyncSourceEvent(sourceURI: const char*, sourcename: const char*, mode: SyncMode, data: int, type: int)	Fires a SyncSourceEvent
fireSyncItemEvent(sourceURI: const char*, itemKey: const WCHAR*)	Fires a SyncItemEvent
fireSyncStatusEvent(command: const char*, statusCode: int, uri: const char*, itemKey: const WCHAR*)	Fires a SyncStatusEvent

They create an Event object and pass it to all the registered listeners in the chain.
Note that, in a multi-threaded application, the listeners callbacks are called in the thread where the sync is running. For this reason, the listener should return quickly the control to the sync engine and just fire an action for the UI thread, otherwise the whole sync is slown down by the UI update.

3.18. Filtering

Filtering is a new feature introduced in the SyncML DS 1.2 specifications that allow a SyncML client to synchronize a database restricting the set of data returned by the server to the items that fall into a given filter. See [4] for details.
The C++ SDK contains objects and methods to allow a client to specify a SyncML filter, for servers supporting SyncML 1.2 filtering: a client can build a filter using the classes described below, and the the engine will format the SyncML filter in the initialization phase.

A filter is represented by a Clause object which can have the forms of the classes represented in the class diagram of Figure 20.
Clause is the base class of all clause objects. It also specifies the type of the clause subclasses can represent.

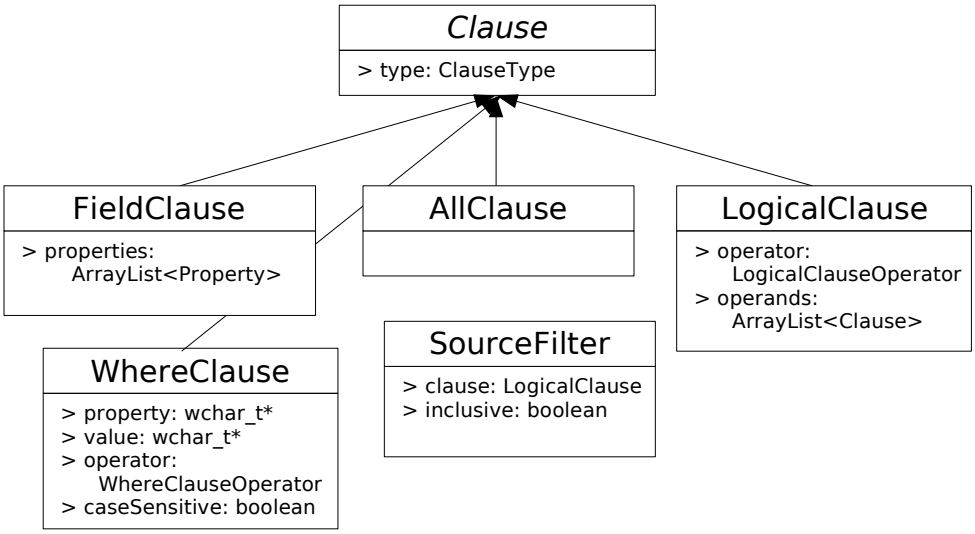


Figure 20: Clauses class diagram

A FieldClause represents a constraint on a property, where the latter is of type Property as defined in the syncml/core classes.

A WhereClause is a used to represents a record filter, which is in turn a query string following the CGI syntax (see the specs). WhereClause supports the following operators: EQ, NE, GT, LT, GE, LE, CONTAIN and NCONTAIN.

LogicalClause combines one or more Clause object in a logical relationship. The following logical operators are supported: NOT, AND, OR.

AllClause represents a “non filter” and is used when no particular selection must be done.

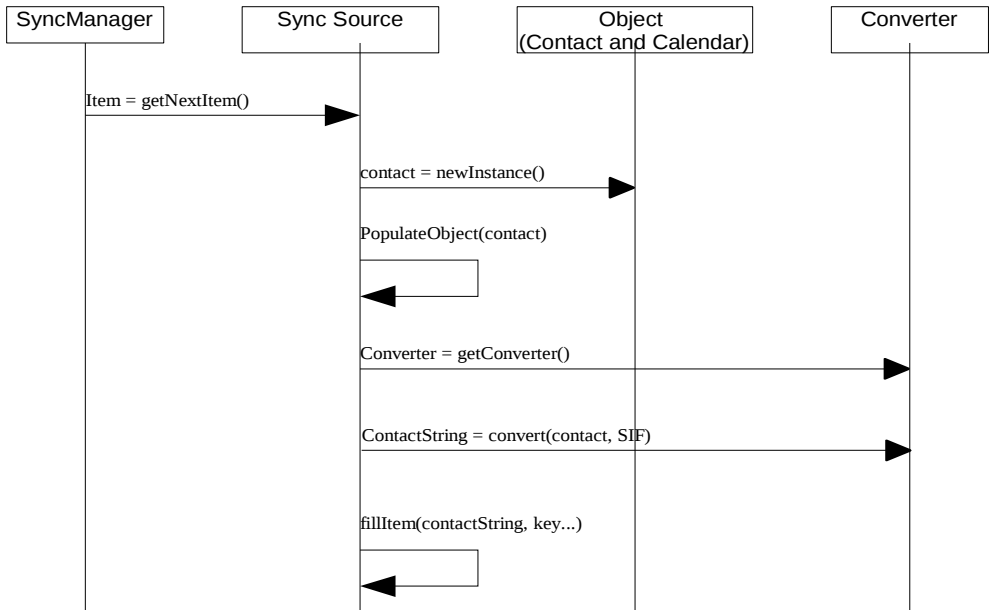
SourceFilter represents a filter to be applied to a SyncSource, such as a logical AND combination of record-level and field level predicates. In particular, the first operator is a LogicalClause containing the record filter, whilst the second operator is a FieldClause or a containing the record level filters. Both expressions can be an AllClause if the particular piece of the filter is missing (i.e. record-only or field-only filter).
As dictated by the specifications, a filter can be inclusive or exclusive, influencing how client and server handle the items outside the filter. This is specified by the property inclusive.

In addition to the Clause classes, a ClauseFormatter class is developed to convert a Clause hierarchy into a syncml/core/Filter object.

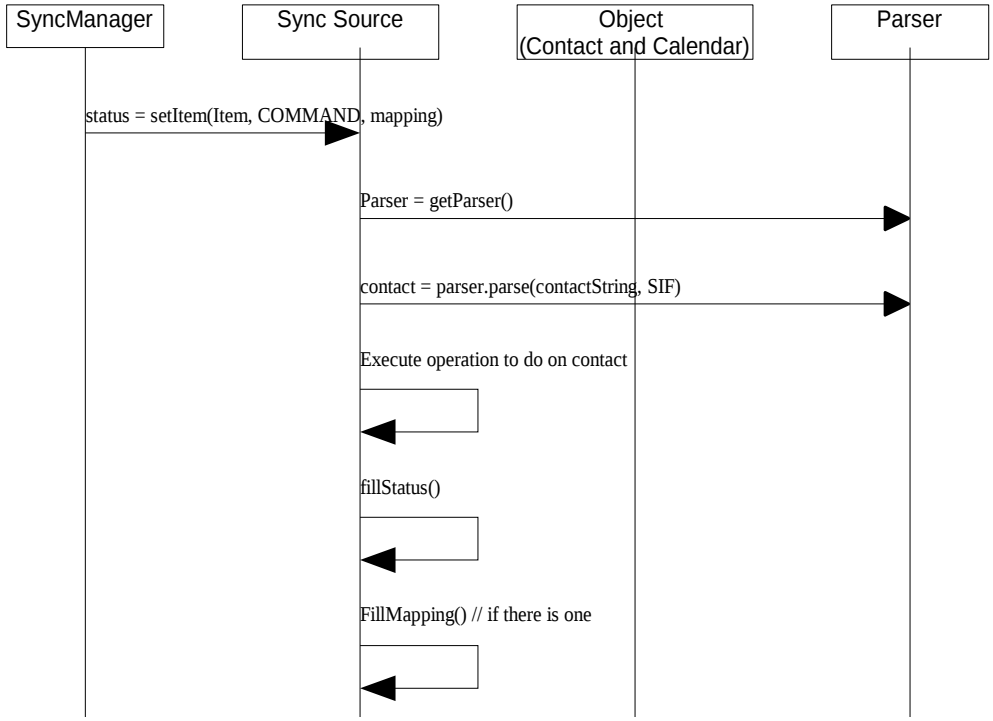
3.19. Converter and Parser for Contact and Calendar objects

In order to provide facilities to the developer the Funambol Client API C++ contains classes to represent Contact and Calendar object and function to convert it to SIF-C or SIF-E format (Funambol Interchange Format) and cCcard or iCcalendar.
On the other hand there are function to parse the SIF format or the vCard and iCal format and create the correspondent Contact or Calendar object.
These utilities, used into SyncSource, aim to have a simple way to manage the object and to convert their structure in xml or vCard before set into the item object. The structure want to be as flexible as possible to be extended with other useful object as task, note...

The converter Diagram is as follow



Converter is the superclass of ContactConverter and CalendarConverter. The getConverter function retrieves the specific converter class (i.e. ContactConverter) and applies the proper “convert” method. The parameter SIF or VCARD are constants to choose the action to perform. The same is for Calendar object.
Object is the superclass of Contact and Calendar object. They represents the objects that can be filled and used by the SyncSource



Parser is the superclass of ContactParser and CalendarParser. The getParser function retrieves the specific parser class (i.e. ContactParser) and applies the proper “parse” method. The parameter SIF or VCARD are constants to choose the action to perform. The same is for Calendar object.

Item Content Encoding/Decoding

Items can be embedded into the SyncML message in a format different from plain text. In this case, the SyncML specs imposes that the <Item> element specifies a <Format> element which tells how the content has been encoded. The SyncML specs limits (even if not explicitly) to just one value as format. However, we may have the need to specify that a particular item content was transformed with many stages and different algorithms. In order to be able to apply the appropriate processing, the counterpart must be aware of which algorithms were applied and in which order.
To support that, the content of the format element is interpreted in the Funambol API as a semi-column separated list of formats, applied in sequence from the leftmost (the first encoding applied) to the rightmost (the last encoding applied).

For example, when receiving an item, if the specified format is “des;b64”, the API expects the content being b64; therefore, to obtain the clear content, the received data must be 64 decoded first and than deciphered with DES.
On the other side, when the API creates the SyncML message, if the item's SyncSourec is configured with format “des;b64”, the API will apply DES ciphering first and than it will base64 encode the encrypted data. The server will apply the decoder in the reversed order.

3.20. MailAccount and FolderData handling

FolderData and MailAccount are two classes implemented to handle an hierarchical synchronization process used for the Multiple Email Account.

FolderData implements every value specified into the syncml specifications and a Parent value that helps the process to store the correspondent parent sent by the server into the SyncItem value.
MailAccount extend FolderData to have specific accessors to values sent by the server into the Ext tags into the Folder tag.

Value	Description
VisibleName	The name of the user shown into the email.
EmailAddress	The email address
Protocol	The protocol value. Usually is SyncML used by the clients.
Username	The username
Password	The Password
InServer	In case of non syncml accounts this value is the Incoming Server
OutServer	In case of non syncml accounts this value is the Outcoming Server
InPort	In case of non syncml accounts this value is the Incoming Server Port
OutPort	In case of non syncml accounts this value is the Outcoming Server Port
InSSL	This value enables the SSL encryption for the Incoming Server
OutSSL	This value enables the SSL encryption for the Outcoming Server
Signature	The user signature
DomainName	The (optional) domain name of the account

It is also another value called Deleted used by the client to mark that account to be deleted. The client will then iterate on all the mailaccounts to find the one to be removed by the list.

The handling of the mail accounts is implemented into the MailAccountManager class.
It's used to add/modify/delete email accounts and folders.
All settings for each email account are stored in the config, passed into the constructor, in order to be able to check for any local change in the account settings.
Clients should extend this class and implement virtual methods to create/update/delete accounts and folders in the specific platform.

Method	Description
createAccount(MailAccount& account)	Creates a new email account
updateAccount(const MailAccount& account)	Updates an email account.
deleteAccount(const WCHAR* accountID)	Deletes an email account.
createFolder(FolderData& folder)	Creates a new folder under the parent account.
updateFolder(const FolderData& folder)	Updates an email folder under the parent account.
deleteFolder(const WCHAR* folderID)	Deletes an email folder under the parent account.
GetAccountNumber()	Returns the number of

Method	Description
	existing email accounts.
accountExists(const StringBuffer& accountID)	Checks the config, returns true if the account exists.

4. Device Manager Layer

Goal of the device management layer is to allow an easy management of a remote device, usually by remote administration or help-desk staff. This means that a remote or local agent can navigate, view and change device and applications configuration attributes and that those changes are seamlessly applied, possibly with minimal or no user actions.

In order to provide device management functionality, the Funambol API provides a data model for system and applications configuration parameters based on tree view of them. This data model is mainly borrowed by the OMA DM specifications, even if it may differ in many ways.
Such tree-based repository is called DM Tree. The DM Tree must be easily accessible by client applications and must hide the details of how and where configuration information is physically stored.

4.1. Terminology

This section defines terminology used throughout the document and is based on *SyncML Device Management Tree and Description* [1].

ACL
Access Control List. A list of identifiers and access rights associated with each identifier.

Description Framework
A specification for how to describe the management syntax and semantics for a particular device type.

Dynamic node
A node is dynamic if the DDF property Scope is set to Dynamic, or if the Scope property
is unspecified.

Interior node
A node that may have child nodes, but cannot store any value. The Format property of an interior node is node.

Leaf node
A node that can store a value, but cannot have child nodes. The Format property of a leaf node is not node.

Management object
A management object is a subtree of the management tree which is intended to be a (possibly singleton) collection of nodes which are related in some way. For example,

the ./DevInfo nodes form a management object. A simple management object may consist of one single node.

Management client
A software component in a managed device that correctly interprets SyncML DM commands, executes appropriate actions in the device and sends back relevant responses to the issuing management server.

Management server
A network based entity that issues SyncML DM commands to devices and correctly interprets responses sent from the devices.

Management tree
The mechanism by which the management client interacts with the device, e.g. by storing and retrieving values from it and by manipulating the properties of it, for example the access control lists.

Node
A node is a single element in a management tree. There can be two kinds of nodes in a management tree: interior nodes and leaf nodes. The Format property of a node provides information about whether a node is a leaf or an interior node.

Permanent node
A node is permanent if the DDF property Scope is set to Permanent. If a node is not permanent, it is dynamic. A permanent node can never be deleted.

Server identifier
The SyncML DM internal name for a management server. A management server is associated with an existing server identifier in a device through SyncML DM authentication.

4.2. Architecture

The Device Management architecture is shown in Figure 21.

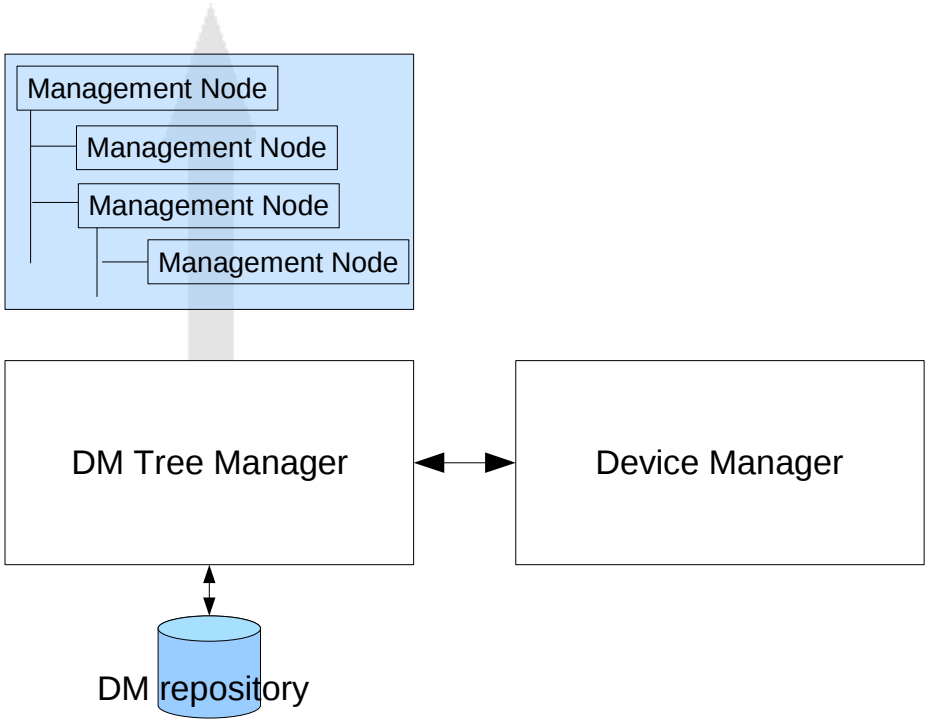


Figure 21: Device Manager layer architecture

Device Manager: this component is responsible for the abstraction of the management operation that can be performed on a device. It will be the place where a device management protocol such as OMA DM will be implemented.

Device Mamagement Tree: this component represents device configuration parameters in a tree data structure where each node is represented by a *Management Node*. An application that wants to store its configuration in a place where the device manager will be able to expose it to a remote server (when a remote device management protocol such as OMA DM will be implemented), should use this component.

Management Node: a management node represents a container of configuration properties and, optionally, of other nodes. Configuration properties are represented with the form of key-value pairs.

4.3. Class Diagram

The components described in the section above, are implemented with the classes illustrated in Figure 22.

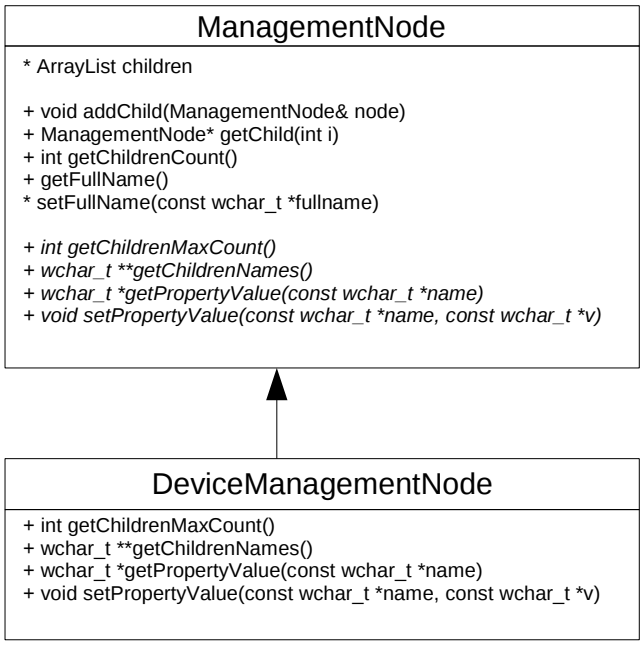


Figure 22: ManagementNode class diagram

The class DMTree is a generic implementation of the Device Management Tree, and provide a method to retrieve a ManagementNode. Note that the ManagementNode returned is created with the standard C++ **new** operator and must be deleted by the caller with the standard C++ **delete** operator.
The methods of the class are virtual, so that if a device-specific implementation is requested, it can be implemented overriding the base class implementation. For this reason, a DMTree instance must be always retrieved using the device-specific DMTreeFactory, that can return a specific subclass if needed.

ManagementNode is the key class that represents a node of the management tree. A node contains properties and maybe other nodes. This is an abstract class, and a device-specific implementation, named DeviceManagementNode, must be provided for each platform. The ManagementNode interface is described below.

Method	***Description***
ManagementNode(wchar_t* context, wchar_t* name)	Creates a ManagementNode from its context path and its name. context represents the parent node, name the node name.
~ManagementNode()	Virtual destructor.
wchar_t getPropertyValue(const wchar_t* property)	Returns the value of the given property.
void setPropertyValue(const wchar_t*	Sets a property value.

Method	Description
property, const wchar_t* value)	
AddChild(ManagementNode & child)	Add a new child to the node
ManagementNode *getChild(int index)	Returns this node's child at the specified index
int getChildrenCount()	Returns how many children belong to this node (already added with addChild).
int getChildrenMaxCount()	Returns how many children are present on the underlying storage system for this node
ManagementNode* clone()	Creates a new ManagementNode with the exact content of this object. The new instance MUST be created with the C++ new operator.
void getFullName(wchar_t* buf, int size)	Returns the full node name

The class DeviceManagementNode must implement the pure virtual methods (see class diagram), to give access to property values and children nodes stored on the underlying storage system.

5. Push Manager

The push manager is responsible for connecting to a Funambol Push server and handle notifications of push events. The push manager is an agent running in the background which connects to a server and waits for notifications. On notifications it is capable of propagating the notification to upper layers. Funambol supports three different push mechanisms:

- SMS
- STP (Server TCP Push)
- CTP (Client TCP Push)

[5] covers these aspects in more detail. In this document we present the architecture of the push manager within the C++ API.

5.1. Push manager architecture

The push manager is intended to offer TCP push via STP and/or CTP. SMS push is not handled by the library as its implementation is largely system specific.
The functionalities the library shall deliver are:

- handles the STP protocol when a STP connection is initiated by the server and handle STP messages
- handles the CTP protocol when STP is not available
- handles local store modification to implement the client push mechanism

Server push mechanisms (CTP and STP) are handled by a manager that switches between STP and CTP when needed. This manager implements a policy and it tries to use STP when this is possible or CTP otherwise. Two separate components implement STP and CTP.

STP is a fairly simple protocol, but it requires some OS support. The protocol requires that the device opens a server socket and listens for incoming notifications. The device must also inform the server about its public IP. It must therefore be possible to discover any IP change that may occur.

CTP is a protocol in which the device opens a connection to the CTP server and waits for notification. More details on STP and CTP can be found in [5].
The architecture of the push manager is represented in figure 2.

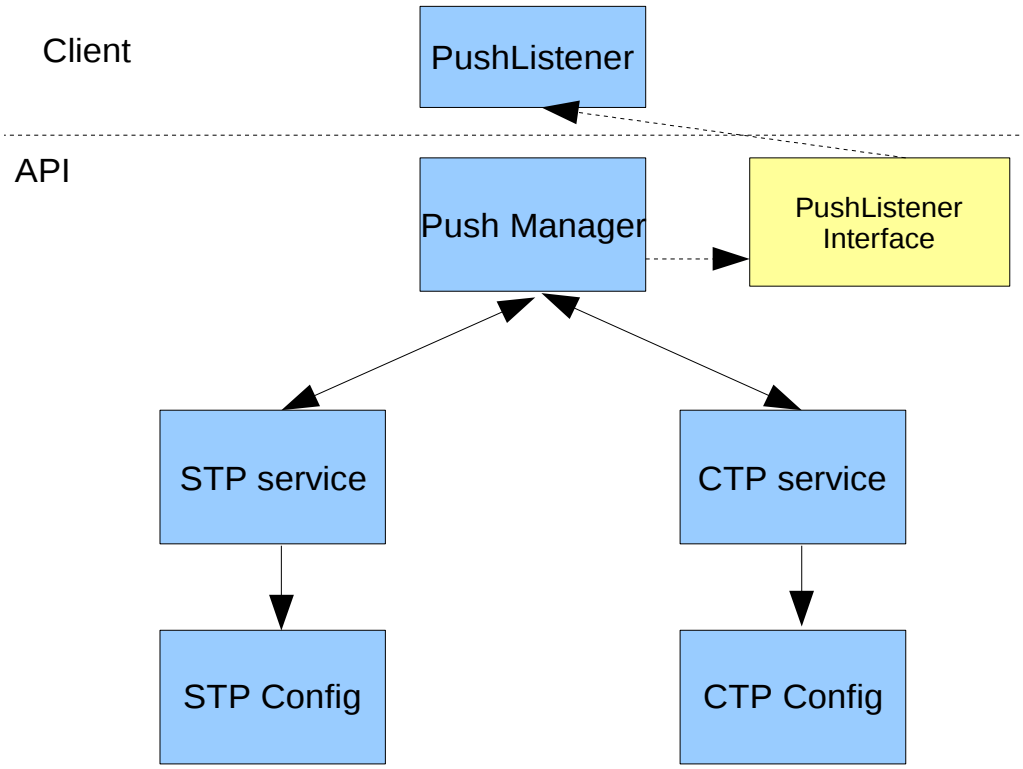


Figure 23: Device Manager layer architecture

A client may decide to use the PushManager and get the advantage of using a high level component which implements policies to choose the best available push mechanism. But it may also decide to use directly the low level services. The all export convenient interfaces to be used.
All these components are intended to be instantiated only once, therefore they are singleton.

The PushManager by default makes an attempt of starting the STP service. If it succeeds then it waits for STP notifications. If it fails it tries to start the CTP service and then waits for notifications. In the future this policy may be made more sophisticated and the client will have the possibility of forcing a specific strategy (such as, use only STP or only CTP).
The following tables list the main methods of the classes in the Push Component.

Method	***Description***
static PushManager* getInstance();	Get the sole instance of PushManager
void start();	Start push service. Configuration is fetched from the DM (push node). The call is non blocking and the service is started in a separated thread.
void stop();	Terminates the push service.
void registerPushListener(PushListener& listener);	Set the given listener to send notifications on pushed events. The given listener must be kept alive for all the Push Manager lifetime. Push Manager keeps a pointer to this listener.
void setPushPolicy(enum Policy);	Sets the policy. Initially only three values are supported:

Method	Description
	• STPFirst (prefer STP over CTP) • STPOnly • CTPOnly
enum ServiceType getServiceType()	Returns the currently used push method (CTPServiceType or STPServiceType)
bool getConnected()	Returns true if connected to a push server. In other words this method returns true iff we are ready to be pushed.

The following list describes the CTPService main methods. It is interesting to observe that CTPService has it own registerPushListener. When used through the PushManager, the PushManager register itself as listener and then forward any notification received (if worth to be propagated). Figure 15 is not really accurate in this regard, as for easiness, it does not show this information flow.
Another interesting aspect is that the CTPService exposes its main thread when started. This is necessary so that client can wait on it if necessary.

Method	Description
static CTPService* getInstance();	Get the sole instance of CTPService
FThread* startCTP();	Start CTP service. Configuration is fetched from the DM (push/ctp node). The thread running the service is returned (see below for FThread description)
int32_t stopCTP();	Terminates the CTP service.
void registerPushListener(PushListener& listener);	Set the given listener to send notifications on pushed events. The given listener must be kept alive for all the CTPService lifetime. CTPService keeps a pointer to this listener.

The STP service has an interface very similar to the CTPService one.
PushListener is a very simple interface with the following main methods:

Method	Description
virtual void onNotificationReceived(const ArrayList& serverURIList);	This method is called when a push notification is received for one or more sources.
virtual void onCTPError(const int errorCode, const int additionalInfo = 0);	Method called when a push error occurs. errorCode is a specific push error code (as defined by CTPErrors, STPErrors or PushErrors) additionalInfo [optional] further information about the error

All the configurations (CTPConfig, STPConfig and PushConfig) are stored into the DM, under the push node. If the client changes any parameter it should stop and restart the push service for the changes to take effect.

5.2. Note on the implementation

As of Funambol V7 only a subset of the library is actually implemented. In particular the CTP service is implemented, but there is no STP and no high level manager. Clients can start and stop CTPService directly.
The CTPService needs some cleanup to comply to the design specification. In particular it exposes more public methods than it should.

6. Appendix A: compatibility reference.

This chapter describes the changes required for a client when upgrading to a new version of the API. The interface of the API aims to be as stable as possible, but adding new features or cleaning up old stuffs may result in changes in the client interface.

6.1. Migrating from Funambol V6x to V7.0.

SyncSource interface:

- new method removeAllItems():
- deprecated methods:
    - getFirstItemKeys()
    - getNextItemKeys()
- SyncSource does not inherit anymore from ArrayElement. This means that the clone() method is not mandatory anymore, unless you want your SyncSource to be inserted in an ArrayList. In the latter case, you have to make it inherit both from SyncSource and from ArrayList.
- [optional] setItemStatus is now available also in the version with the command [Add, Update or Delete]. This can be used by the SyncSource implementation to act differently depending on the command the status is referring to.

Posix port:

- the name of the library is changed to libfunambol.a
- the path of the include file is changed to funambol

6.2. Migrating from Funambol V7.0 to V7.1.

PlatformAdapter:

- call PlatformAdapter::init(context) is required before using the library
- the call DMTClientConfig(context) is replaced by DMTClientconfig() ; the former is still available for backward compatibility, but it's deprecated.

SyncSource interface:

- removed methods:
    - getFirstItemKeys()
    - getNextItemKeys()

KeyValueStore interface:

- method save() renamed to close()

Interface ErrorHandler removed:

- it's been introduced in v3.0 but never been used by the sync engine. Now removed.

References

[1] SyncML Device Management Protocol, version 1.1.2, Open Mobile Alliance
[2] SyncML Device Management Tree and Description, version 1.1.2, Open Mobile Alliance
[3] SyncML Device Management Standardized Objects, version 1.1.2, Open Mobile Alliance
[4] SyncML Data Synchronization Protocol, version 1.2, Open Mobile Alliance
[5] Funambol Client API C++ Design Document
[6] Pictures sync design document (please note: internal link)
https://svn.funambol.com/svn/funambol-carrier-edition/trunk/portal/doc/picture-sync-design-document.odt