Sync4j

Native SyncClient API 1.0 Programming Guide

Sync4j
http://www.sync4j.org

Table of Contents

1. Overview

The Sync4j Native SyncClient API is a C++ programming API by the mean application developers can interact with the Sync4j platform in order to take advantage of its powerful data synchronization features.

This document explains, from a developer point of view, the architecture and the use of the Sync4j Native SyncClient API 1.0.

1.1. SyncClient API Architecture

The SyncClient API is built up of two main modules: data synchronization and device management; they are layered as shown in Figure 1, where the device management layer is responsible for device and application configuration management and the data synchronization layer is responsible for everything regarding the SyncML protocol and the data synchronization process.

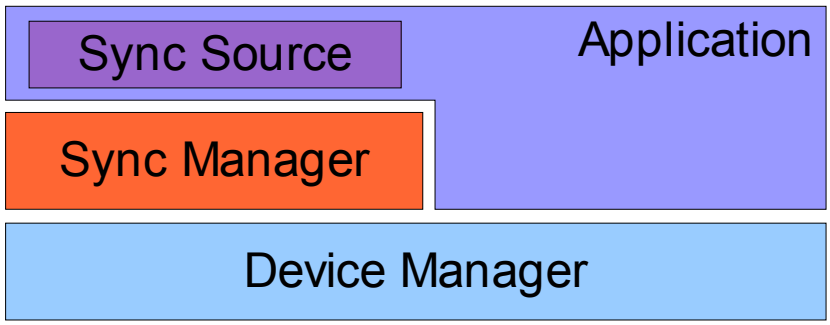

Figure 1 - SyncClient API architecture

The application can access the services provided by both modules: the *Sync Manager* to perform synchronization actions and the *Device Manager* to read, manipulate and write configuration data. In addition, the Device Manager is intended to store application configuration information, enabling the application to be transparently managed remotely with the SyncML Device Management features that will be implemented in a next release of the API.

A *Sync Source* is an application module used by Sync Manager to interact with the application data sources. The way the Sync Source access the external data source is application specific and transparent to the synchronization engine.

1.2. Synchronization Overview

The client application interacts with two objects of the SyncClient API for C/C++: the SyncManager and the SyncSource. The SyncManager is the component that handles all the communication and protocol stuffs. It hides the complexity of the synchronization process providing few simple methods to deal with. The SyncSource represents the collection of items that are exchanged between client and 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 interaction between the three entities Client, SyncManager and SyncSource is depicted in Figure 2.

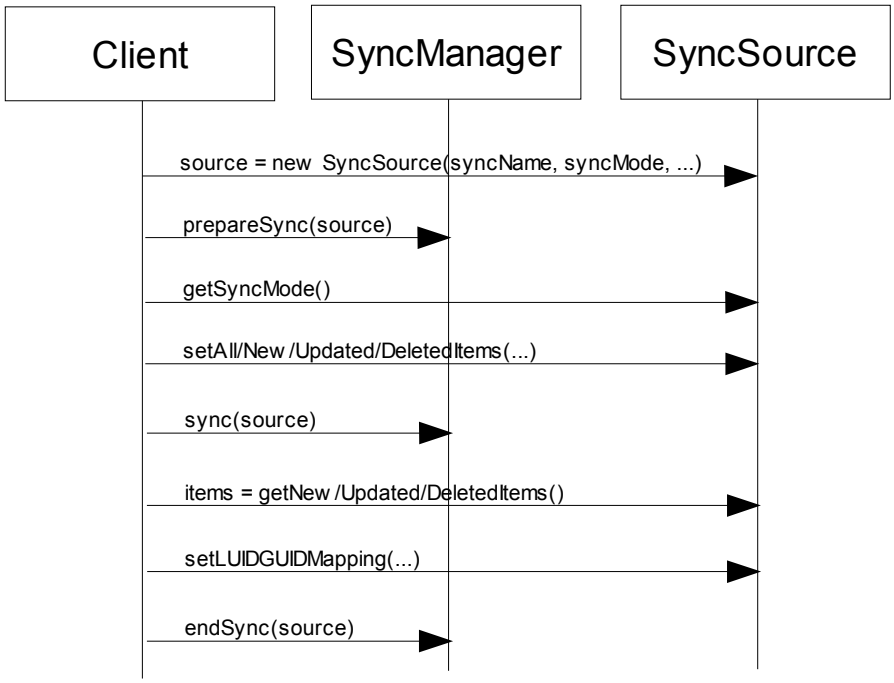

Figure 2 - Client-SyncClient API interaction diagram

First of all the client application tells the SyncManager to prepare the synchronization of a specific server database (identified by a newly create SyncSource). This starts a new synchronization session where the SyncManager negotiates with the server which type of synchronization should be performed (one-way, two-way, slow, etc). After this initialization process, the type of the synchronization that is going to be executed is known, so that the client can feed the sync source with the relevant items (usually using setAllItems() in case of a slow sync and the other setXXXItems() methods in case of one of fast sync).
Now, the SyncSource and the SyncManager are ready to perform the sync. The only thing the client has to do is to call sync(). When the method returns, if the sync process was successfully terminated, the client can read the items received from the server calling the getXXXItems() methods of the SyncSource object.
In the case the client creates its own LUID for new items, it can set the LUID-GUID mapping back to the SyncSource so that at the end of the synchronization process (which is fired by the endSync() method), the mapping is sent to the server. This step is optional.

1.3. Sync4j SyncClient API Licensing

Sync4j SyncClient API licensing options include:

- The Commercial License, which allows you to provide commercial software licenses to your customers or distribute applications based on The Sync4j SyncClient API within your organization. This is for organizations that do not want to release the source code for their applications as open source/free software; in other words they do not want to comply with the GNUGeneral Public License (GPL).
- For those developing open source applications, the Open Source License allows you to offer your software under an open source/free software license to all who wish to use, modify, and distribute it freely. The Open Source License allows you to use the software at no charge under the condition that if you use the Sync4j SyncClient API in an application you redistribute, the complete source code for your application must be available and freely redistributable under reasonable conditions.

In their simplest form, the following are general licensing guidelines:

- If your software is licensed under either the GPL-compatible Free Software License as defined by the Free Software Foundation or approved by OSI, then use our GPL licensed version.
- If you distribute a proprietary application in any way, and you are not licensing and distributing your source code under GPL, you need to purchase a commercial license of the Sync4j SyncClient API
- If you are unsure, we recommend that you buy our cost effective commercial licenses. That is the safest solution. Licensing questions can be directed to license@sync4j.org for our advice, and we encourage you to refer to the Free Software Foundation or a lawyer as appropriate.

Commercially licensed customers get commercially supported product with assurances from Funambol. Commercially licensed users are also free from the requirement of making their own application open source.

For OEM's, ISVs, corporate, and government users, a commercial license is the proper solution because it provides you with assurance from the vendor and releases you from the strict requirements of the GPL license.

Nevertheless, you can test Sync4j SyncClient API under the GPL license and inspect the source code before you purchase a commercial non-GPL license.

For more information about Sync4j SyncClient API license, please refer to Chapter 6 of this manual.

1.4. Notes and Conventions

The following notes and conventions do apply in this document:

- Wherever a char, char* or char[] type is specified in the API, it is intended to be a wide char or a char depending on the default for the platform where the API is used. These are:
    - Win32: wchar_t
    - PPC: wchar_t

2. Data Synchronization API

This section describes in more details the classes of the Data Synchronization API, which are shown in Figure 3 and Figure 4.

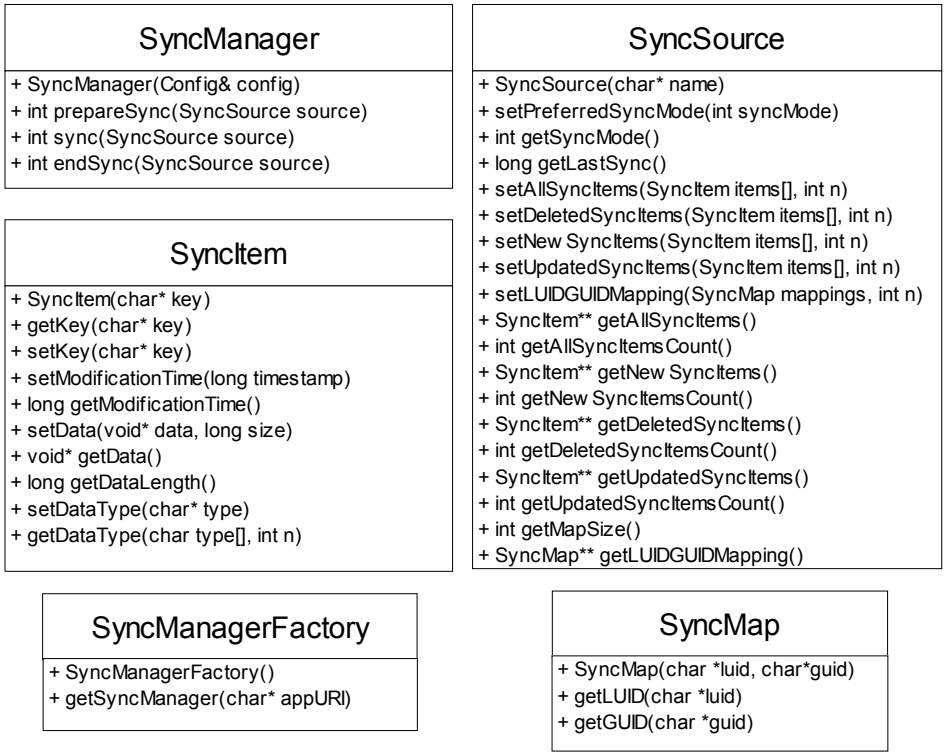

Figure 3 - SPDS classes (1)

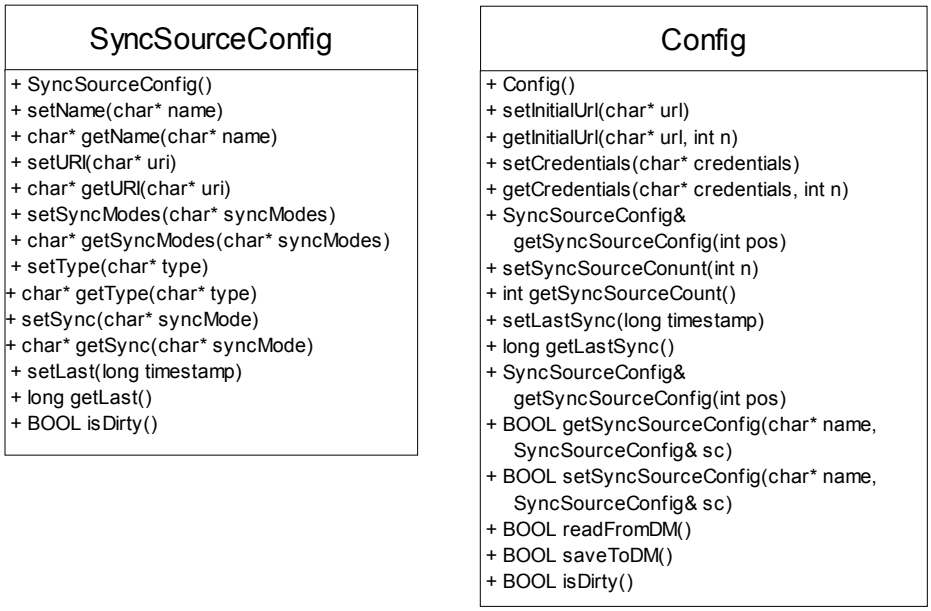

Figure 4 - SPDS classes (2)

2.1. The Sync Manager

SyncManager is the contact point between an application and the synchronization engine. It is designed to hide as much as possible to the developer the details of the synchronization logic, protocol and communication.

The way to work with the SyncManager is as follows (see Figure 2):

- Get a SyncManager instance.
- Create a SyncSource representing the server database to synchronize.
- Call prepareSync() in order to start the synchronization process.
- Fill the SyncSource with the client-side modified items.
- Call sync().
- Get the server-side modifications and apply them to the local database.
- Set the source LUID-GUID mapping.
- Call endSync().

Here is a code snippet of how the above sequence can be implemented:

```
SyncManagerFactory factory = SyncManagerFactory();
SyncManager* syncManager = factory.getSyncManager(APPLICATION_URI);
SyncSource source = SyncSource(SOURCE_NAME);

if (syncManager == NULL) {
    error();
    goto finally;
}

ret = syncManager->prepareSync(source);

if (ret != 0) {
    // error handling
    ...
}

switch (source.getSyncMode()) {
   case SYNC_SLOW:
```

```
    setAllItems(source);  // fill the allItems source property
    break;

case SYNC_TWO_WAY:
     setModifiedItems(source);  // set the client-side modified items
     break;

     default:
         break;
}

if (syncManager->sync(source) != 0) {
     error();
     goto finally;
}

// Now source contains server-side modifications
// ... Do whatever appropriate ...

setMappings(source);  // set LUID-GUID mapping

if (syncManager->endSync(source) != 0) {
     error();
     goto finally;
}
```

Method	Description
SyncManager()	Constructor
int prepareSync(SyncSource& source)	Initializes a new synchronization session for the specified sync source. It returns 0 in case of success, an error code in case of error (see SPDS Error Codes).
int sync(SyncSource& source)	Synchronizes the specified source with the server. *source* should be filled with the client-side modified items. At the end of the process *source* will be fed with the items sent by the server. It returns 0 in case of success or an error code in case of error (see SPDS Error Codes).
int endSync(SyncSource& source)	Ends the synchronization of the specified source. If source contains LUIG-GUID mapping this is sent to the server. It returns 0 in case of success or an error code in case of error (see SPDS Error Codes).

2.1.1. Configuring the Sync Manager

Sync Manager requires few configuration parameters such as the url of the SyncML server, which Sync Sources can be synchronized and so on. This information is stored in the device manager layer (see Device Management API). Configuration parameters are grouped by so called *configuration contexts* and organized in a *management tree*.
Please refer to the Deveice Management API section for details on how configuration information are stored on each platform.
Sync Manager requires the following configuration parameters (grouped by context):

Property	Description
<application uri>/spds/syncml	
begin	Timestamp of the beginning of the last synchronization
deviceId	Device identifier

Property	Description
end	Timestamp of the end of the last synchronization
password	Principal's credential
firstTimeSyncMode	Reserved
password	The user's password
proxyHost	Reserved
proxyPort	Reserved
serverName	The server uri used to address the sync server
syncUrl	Synchronization url
useProxy	Reserved
username	The user's account nickname
<application uri>/spds/sources/<source name>	
last	The timestamp of the last synchronization
name	Source display name
sync	The default sync mode to be used for the source; one of: • none • slow • two-way • one-way • refresh
syncModes	Comma separated list of supported sync modes (e.g.: none, slow, two-way)
type	Source type (e.g. text/plain, text/vcard, ...)

Note that under the *<application uri>/spds/sources* context can be put as many sources as needed.

2.2. SyncManagerFactory

This is the factory for SyncManager objects. Use its getSyncManager() method to get a new configured instance of SyncManager.

Method	Description
SyncManagerFactory()	Constructor.
SyncManager* getSyncManager(char* appURI)	Creates and initializes a new SyncManager instance. The creation process includes the reading of the relevant configuration information from the device manager layer.

2.3. SyncSource

This is the class that represents a synchronization data source. Its must be filled with the client-side items before passing it to the SyncManager in order to synchronize it. After the

SyncManager has finished a sync(), the SyncSource object contains the server-side items. Items are stored as arrays of SyncItem objects.

Method	Description
SyncSource(char* name)	Constructs a SyncSource with the given name. The name is used to retrieve the source configuration from the `<appURI>/spds/sources/` configuration context.
char* getName(char* name, int dim)	Returns the source name. If sourceName is <> NULL, the source name is copied in it. If sourceName is <> NULL the returned value is sourceName. Otherwise, the returned value is the internal buffer pointer. Note that this will be released at object automatic destruction.
setType(char* type)	Sets the items data mime type
char * getType(char *type)	Returns the items data mime type. If type is NULL, the pointer to the internal buffer is returned, otherwise the value is copied in the given buffer, which is also returned to the caller.
setPreferredSyncMode(SyncMode syncMode)	Sets the synchronization mode required for the SyncSource. It can be one of the values of the enumeration type SyncMode.
getPreferredSyncMode()	Returns the default synchronization mode.
SyncMode getSyncMode()	Returns the synchronization mode for the SyncSource. It may be different from the one set with *setPreferredSyncMode()* as the result of the negotiation with the server.
setSyncMode(SyncMode syncMode)	Sets the synchronization mode for the SyncSource.
long getLastSync()	Returns the timestamp in milliseconds of the last synchronization. The reference time of the timestamp is platform specific.
setLastSync(long timestamp)	Sets the timestamp in millisencods of the last synchronization. The reference time of the timestamp is platform specific.
setLastAnchor(char* last)	Sets the last anchor associated to the source
char* getLastAnchor(char* last)	Gets the last anchor associated to the source. If last is NULL the internal buffer address is returned, otherwise the value is copied in the given buffer and the buffer address is returned.
setNextAnchor(char* next)	Sets the next anchor associated to the source
char* getNextAnchor(char* next)	Gets the next anchor associated to the source. If next is NULL the internal buffer address is returned, otherwise the value is copied in the given buffer and the buffer address is returned.

Method	Description
setAllSyncItems(SyncItem* items[], int n)	Sets all the items stored in the data source. For performance reasons, this should only be set when required (for example in case of slow or refresh sync). The items are passed to the SyncSource as an array of SyncItem objects.
setDeletedSyncItems(SyncItem* items[], int n)	Sets the items deleted after the last synchronization. The items are passed to the SyncSource as an array of SyncItem objects.
setNewSyncItems(SyncItem* items[], int n)	Sets the items created after the last synchronization. The items are passed to the SyncSource as an array of SyncItem objects.
setUpdatedSyncItems(SyncItem* items[], int n)	Sets the items updated after the last synchronization. The items are passed to the SyncSource as an array of SyncItem objects.
setLUIDGUIDMapping(SyncMap* mappings[], int n)	Sets the LUID-GUID mapping of the last synchronization.
SyncMap** getLUIDGUIDMapping()	Returns the LUID-GUID mappings.
int getMapSize()	How many mappings.
SyncItem** getAllSyncItems()	Returns the all items buffer.
int getAllSyncItemsCount()	How many items in the all items buffer.
SyncItem** getNewSyncItems()	Returns the new items buffer.
int getNewSyncItemsCount()	How many items in the new items buffer.
SyncItem** getDeletedSyncItems()	Returns the deleted items buffer.
int getDeletedSyncItemsCount()	How many items in the deleted items buffer.
SyncItem** getUpdatedSyncItems()	Returns the updated items buffer.
int getUpdatedSyncItemsCount()	How many items in the updated items buffer.

2.4. SyncItem

An item is represented by a SyncItem, who associates an identifying key with the item content.

Method	Description
SyncItem(char* key)	Constructs a new SyncItem identified by the given key. The key must not be longer than DIM_KEY (see SPDS Constants).
char* getKey(char* key)	Returns the SyncItem's key. If key is NULL, the internal buffer is returned; if key is not NULL, the value is copied in the caller allocated buffer and the given buffer pointer is returned.
setKey(char* key)	Changes the SyncItem's key. The key must not be longer than DIM_KEY (see SPDS Constants).

Method	Description
setModificationTime(long timestamp)	Sets the SyncItem's modification timestamp. *timestamp* is a milliseconds timestamp since a reference time (which is platform specific).
long getModificationTime()	Returns the SyncItem's modeification timestamp. The returned value is a milliseconds timestamp since a reference time (which is platform specific).
setData(void* data, long size)	Sets the SyncItem's content data. The passed data are copied into an internal buffer so that the caller can release its buffer after calling setData(). NOTE: in the current implementation data MUST point to a zero terminated string.
void* getData()	Returns the SyncItem's data buffer.
long getDataSize()	Returns the SyncItem's data size.
setDataType(char* type)	Sets the SyncItem's data mime type
getDataType(char *type, int n)	Returns the SyncItem's data mime type.

2.5. Config

This class represents the entire SyncManager configuration. It groups utility methods to easily access configuration properties.

Method	Description
Config()	Constructs a new Config object
setInitialUrl(char* url)	Sets the initial url to which the server responds
getInitialUrl(char* url, int n)	Returns the configured initial url to which the server responds
setCredentials(char* credentials)	Sets the credentials to send along with the request
getCredentials(char* credentials, int n)	Returns the configured credentials
SyncSourceConfig* getSyncSourceConfig()	Returns the SyncSource configuration objects
void setSyncSourceCount(int n)	Set the number of the valid SyncSource configuration objects
int getSyncSourceCount()	Returns the number of valid SyncSource configuration objects
setLastSync(long timestamp)	Sets the last sync timestamp (the last time the SyncManager was run).
long getLastSync()	Returns the last sync timestamp (the last time the SyncManager was run).
SyncSourceConfig& getSyncSourceConfig(int pos);	Returns the pos-th sync source configuration.
BOOL getSyncSourceConfig(const wchar_t* name, SyncSourceConfig& sc);	Returns a sync source configuration given the source name.

Method	Description
BOOL setSyncSurceConfig(SyncSourceConfig& sc)	Sets a sync source configuration given the source name and the SyncSourceConfig object.
BOOL readFromDM()	Reads the configuration from the device manager.
BOOL saveToDM()	Saves the configuration to the device manager.
BOOL isDirty()	Is the dirty flag set?

2.6. SyncSourceConfig

Similar to Config but for SyncSources.

Method	Description
SyncSourceConfig()	Constructs a new SyncSourceConfig object
setName(char* name)	Sets the SyncSource name
char* getName(char* name)	Returns the SyncSource name. If name is null, the internal buffer is returned, otherwise the value is copied into the given buffer (that must be DIM_SOURCE_NAME big).
setURI(char* uri)	Sets the SyncSource URI (used in SyncML addressing).
char* getURI(char* uri)	Returns the SyncSource URI (used in SyncML addressing). If uri is null, the internal buffer is returned, otherwise the value is copied into the given buffer (that must be DIM_SOURCE_URI big).
setSyncModes(char* syncModes)	Sets the available syncModes for the SyncSource as comma separated values. Each value must be one of: • none • slow • two-way • one-way • refresh
char* getSyncModes(char* syncModes)	Returns a comma separated list of the supported syncModes for the SyncSource. If syncModes is null, the internal buffer is returned, otherwise the value is copied into the given buffer (that must be DIM_SYNC_MODES_LIST big).
setType(char* type)	Sets the mime type of the items handled by the sync source.
char* getType(char* type)	Returns the mime type of the items handled by the sync source. If type is null, the internal buffer is returned, otherwise the value is copied into the given buffer (that must be DIM_SYNC_MIME_TYPE big).
setSync(char* syncMode)	Sets the default syncMode as one of the strings above.
char* getSync(char* syncMode)	Returns the default syncMode as one of the strings above. If type is null, the internal buffer is returned, otherwise the value is copied into the given buffer (that must be DIM_SYNC_MODE big).
setLast(long timestamp)	Sets the last sync timestamp

Method	Description
long getLast()	Returns the last sync timestamp

2.7. SPDS Error Codes

Mnemonic	Value	Description
ERR_CONN_UNABLE_CONNECT	100	It was not possible to connect to the server
ERR_PROTOCOL	400	Protocol error
ERR_AUTH_NOT_AUTHORIZED	401	The user is not authorized to perform the requested sync
ERR_AUTH_EXPIRED	402	An existing account has expired
ERR_SRV_FAULT	500	The server reported an error
ERR_SOURCE_DEFINITION_NOT_FOUND	600	The source definition for a source is not found in the DM management tree
ERR_REPRESENTATION	700	Error parsing/interpreting a SyncML message

2.8. Sync Modes

Mnemonic	Value	Description
SYNC_TWO_WAY	200	Two way sync
SYNC_SLOW	201	Two way slow sync
SYNC_ONE_WAY_FROM_CLIENT	202	One way sync from client
SYNC_REFRESH_FROM_CLIENT	203	Refresh sync from client
SYNC_ONE_WAY_FROM_SERVER	204	One way sync from server
SYNC_REFRESH_FROM_SERVER	205	Refresh sync from server
SYNC_TWO_WAY_BY_SERVER	206	Two way server alerted sync
SYNC_ONE_WAY_FROM_CLIENT_BY_SERVER	207	One way server alerted sync from client
SYNC_REFRESH_FROM_CLIENT_BY_SERVER	208	Refresh server alerted sync from client
SYNC_ONE_WAY_FROM_SERVER_BY_SERVER	209	One way server alerted sync from server
SYNC_REFRESG_FROM_SERVER_BY_SERVER	210	Refresh server alerted sync from server

2.9. SPDS Constants

Mnemonic	Value	Description
DIM_ANCHOR	32	Max length of a synchronization anchor
DIM_DEVICE_ID	50	Max length of a device id
DIM_HOST	50	Max length of an host name

Mnemonic	Value	Description
DIM_KEY	256	Max length of a SyncItem key
DIM_MANAGEMENT_PATH	512	Max length of a management path
DIM_MIME_TYPE	32	Max length of a mime type
DIM_PASSWORD	100	Max length of a password
DIM_SERVERNAME	100	Max length of a server name
DIM_SOURCE_NAME	128	Max length for a source name
DIM_SOURCE_URI	64	Max length for a source URI
DIM_SYNC_MODE	16	Max length of a sync mode
DIM_SYNC_MODES_LIST	64	Max length of a comma separated list of the available sync modes
DIM_URL	2048	Max length of a url
DIM_USERNAME	100	Max length of a usernam

3. Device Management API

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 in a manner transparent to the end user.

Configuration information is logically stored in a so called *management tree*, organized in a hierarchy of *contexts* and *management nodes*. This hides the details of the physical configuration storage that could be an SQL database, a device datastore, an XML file, a file system tree or even the device memory.

NOTE: the current version of the Funamambol SyncClient API does not support remote device management yet. This functionality will be added in a future release.

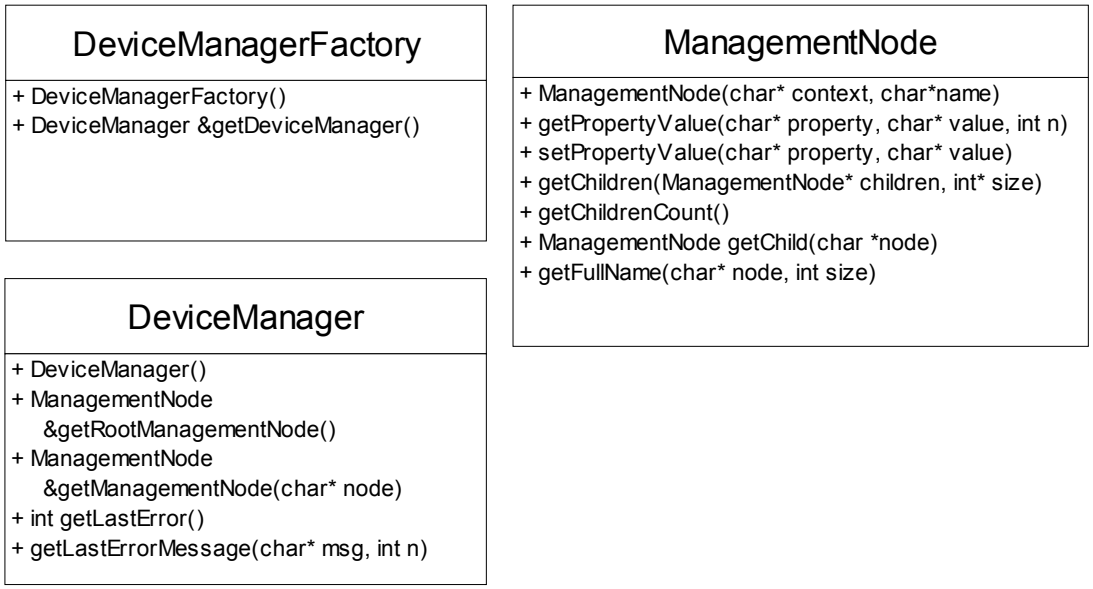

Figure 5 - SPDM classes

3.1. The Device Manager

The classes of the Device Management API are shown in Figure 5. The entrypoint is represented by the *DeviceManagerFactory* who acts as a factory for concrete implementations. In addition, concrete DeviceManager implementations can return the management tree root relative to a base configuration context. The management tree is represented by a hierarchical

structure of *ManagementNode* objects. ManagementNode provides accessing methods for the manageable properties stored in the node and additional methods to retrieve children nodes and values. Children and parent nodes can also be accessed through the given utility methods.

The physical implementation of the management tree repository may vary from simple properties files stored on a file system to configuration tables stored in a database.

3.2. DeviceManagerFactory

This class is simply a factory for DeviceManager concrete implementations. It is platform specific so that it returns a concrete DeviceManager implementation for the target platform.

Method	Description
DeviceManagerFactory()	Constructor.
DeviceManager getDeviveManager()	Creates and returns a new DeviceManager. The DeviceManager object is created with the new operator and must be deleted by the caller with the operator delete.

3.3. DeviceManager

This is an abstract class that defines the interface of a device manager object so that the caller can use a generic “DeviceManager” without the need to know platform and implementation specific details.

Method	Description
DeviceManager()	Constructor
ManagementNode* getRootManagementNode()	Returns the root management node for the DeviceManager.
ManagementNode* getManagementNode (char* node)	Returns the management node identified by the given node pathname (relative to the root management node). If the node is not found NULL is returned.

3.4. ManagementNode

This is another abstract class that acts as an interface for concrete implementations. A management node associates a context path (such as applications/app1/source/spds/sources/source1) to a set of properties. A property is a key-value pair.

Method	Description
ManagementNode(char* context, char* name)	Constructor.
getPropertyValue(char* property, char* value, int n)	Returns the value of the given property.
setPropertyValue(char* property, char* value)	Sets the value of the given property.
getChildren(ManagementNode** children, int* size)	Returns the children node of the ManagementNode.
getChildrenCount()	Returns how many children belong to the node.
ManagementNode* getChild(char *node)	Returns a child node of the ManagementNode.
getFullName(char *buf)	Returns the full node name.

3.5. SPDM Concrete Implementations

As said before, the physical media where the device manager layer stores and reads configuration contexts and properties is platform dependent. The class DeviceManagerFactory creates DeviceManager objects suitable for the implementation used.
In details, the platform-media implementation table is the following:

Platform	Media
Win32	Windows registry. The root context is relative to HKEY_LOCAL_MACHINE\Software.
PocketPC 2002-2003-.NET	PocketPC registry. The root context is relative to HKEY_LOCAL_MACHINE\Software.

3.6. SPDM Error Codes

Mnemonic	Value	Description
ERR_INVALID_CONTEXT	10000	A specified context is invalid.
ERR_SOURCE_DEFINITION_NOT_FOUND	10001	The definition of the SyncSource was not found.

3.7. SPDM Constants

Mnemonic	Value	Description
DIM_MANAGEMENT_PATH	512	Max length of a management path.
DIM_ERROR_MESSAGE	256	Max length of an error message.
DIM_PROPERTY_NAME	64	Max length of a property name.

3.8. Examples

3.8.1. Getting a DeviceManager Instance

```
#include "spdm/common/DeviceManager.h"
#include "spdm/common/DeviceManagerFactory.h"

...

DeviceManagerFactory factory = DeviceManagerFactory();
DeviceManager* dm = factory.getDeviceManager();
...
```

3.8.2. Getting a Management Tree

```
#include "spdm/common/DeviceManager.h"
#include "spdm/common/DeviceManagerFactory.h"

...

DeviceManager* dm;
wchar_t context[DIM_MANAGEMENT_PATH];
...
```

```
ManagementNode* node = dm->getManagementNode(L"/Sync4j/spdm");
...
```

3.8.3. Reading Management Node Configurable Properties

```
DeviceManagerFactory factory = DeviceManagerFactory();
DeviceManager* dm = factory.getDeviceManager();

wsprintf(context, TEXT("%s/%s"), rootContext, CONTEXT_SPDS_SYNCML);
ManagementNode* spdsConfig = dm->getManagementNode(context);
if (spdsConfig == NULL) {
    lastErrorCode = ERR_INVALID_CONTEXT;
    wsprintf(lastErrorMsg, TEXT("SyncManager configuration not found: %s"), context);
    goto finally;
}

wchar_t buf[VALUESIZE]; // big enough for everything

//
// Reads the configuration from the management node and fill
// the config object. This will be passed to the SyncManager
// constructor
//
spdsConfig->getPropertyValue(PROPERTY_USERNAME, buf, VALUESIZE);
...
```

3.8.4. Reading Node Children

```
...

int nc = node->getChildrenCount();
if (nc > 0) {
    ManagementNode** children = new ManagementNode*[nc];

    node->getChildren(children, &nc);
}
```

3.8.5. Update Configurable Properties

```
wchar_t buf[DIM_VALUE];
wsprintf(buf, TEXT("%s"), ... something here ...);
spdsConfig->setPropertyValue(PROPERTY_SYNC_BEGIN, buf);
```

4. Installation

The Sync4j Native SyncClient API is delivered as a zip archive named scapi-native-<major>.<minor>.<build>.zip, where <major>, <minor> and <build> are, respectively, the major, minor and build version numbers.

To install the API, just unzip the archive in a directory of choice. You will get the directory structure of Figure 6.

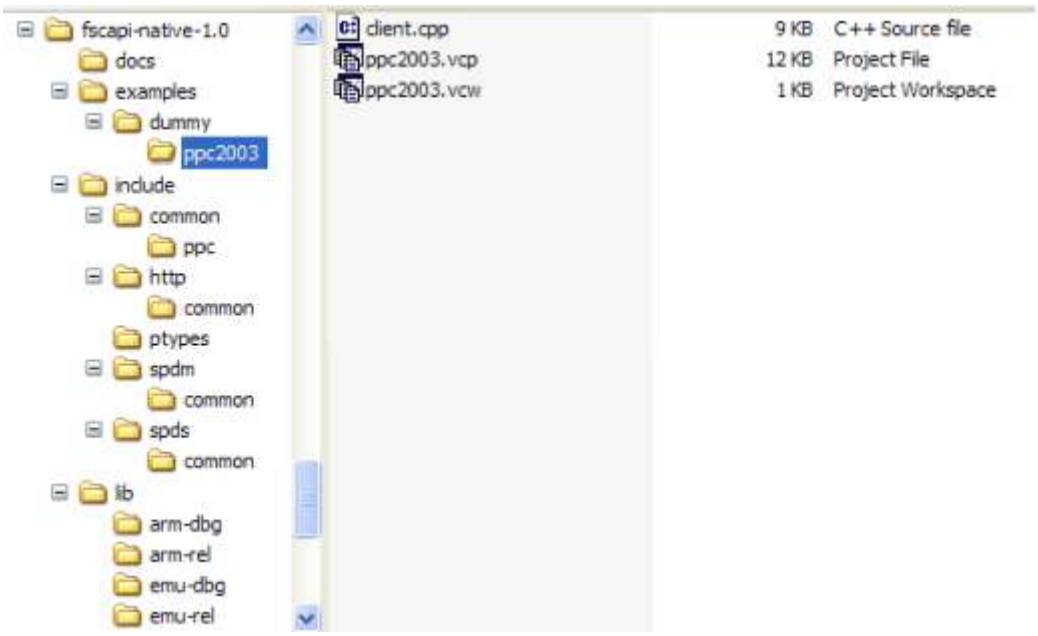

Figure 6 - Native SyncClient API package structure

Here is the meaning of the main directories:

Directory	*Description*
docs	Documentation directory.
examples	Sample programs.
include	Header files directory.
lib	Library directory containing the libraries to be used for the different devices and configurations.

5. Developing a Test Application for Pocket PC 2002-2003

In this section, we are going to describe a test application from scratch. Our test application is a C++ application, composed of the following files:

- client.cpp: the main test application.
- The Sync4j SyncClient API lib files (scapippc2003.lib)
- The Sync4j SyncClient API header files (*.h)

You can find the example source files in the 'examples' directory of the SyncClient API installation directory. The following sections explain the client application logic and the steps needed to build and run it.

The software requirements to build and run the test application are:

- Sync4j Native SyncClient API 1.0
- Microsoft Embedded VC++ 3.0 (for Pocket PC 2002) or 4.0 (for Pocket PC 2003)

5.1. client.cpp

This is the C++ program that implements a very simple client based on the Sync4j SyncClient API.

Scope of this client is to show the process of starting, carry on and completing a synchronization session, displaying the exchanged modification on the screen (or on a log file); no database will be physically updated.

Open client.cpp in your preferred editor and refer to the following annotated code for details.

First of all, there are some includes and definitions:

```
#include "common/fscapi.h"
#include "common/Log.h"
#include "spds/common/Config.h"
#include "spds/common/Utils.h"
#include "spds/common/Constants.h"
#include "spds/common/SyncMap.h"
#include "spds/common/SyncItem.h"
#include "spds/common/SyncManager.h"
#include "spds/common/SyncManagerFactory.h"

#define APPLICATION_URI TEXT("Sync4j/examples/dummy")

#define MAP_SIZE            10
#define ALL_ITEMS_COUNT      4
#define DELETED_ITEMS_COUNT  1
```

```
#define UPDATED_ITEMS_COUNT  2
#define NEW_ITEMS_COUNT      1

static SyncMap*  mappings     [MAP_SIZE           ];
static SyncItem* newItems     [NEW_ITEMS_COUNT    ];
static SyncItem* updatedItems [UPDATED_ITEMS_COUNT];
static SyncItem* deletedItems [DELETED_ITEMS_COUNT];
static SyncItem* allItems     [ALL_ITEMS_COUNT    ];
```

One of the most important piece of code is the creation of the *SyncManager* object using *SyncManagerFactory*. *getSyncManager()* creates a new *SyncManager* instance reading the configuration information it requires from the device management layer. The base of the root configuration context is specified giving the application URI:

```
SyncManagerFactory factory = SyncManagerFactory();
SyncManager* syncManager = factory.getSyncManager(APPLICATION_URI);
```

When we have the *SyncManager* instance, we need something to tell it what we want to synchronize and with which data. This is done by a *SyncSource* object.

```
SyncSource source = SyncSource(TEXT("dummy"));
```

Please note that a sync source requires a bunch of configuration properties (see Configuring the Sync Manager) that must be stored under the configuration context Sync4j/examples/dummy/spds/sources/dummy (which corresponds to <application URI>/spds/sources/<source name>).

To start a new synchronization session for the wanted *SyncSource*, we first prepare it:

```
ret = syncManager->prepareSync(source);
```

In this way, we send information about the server database to synchronize (it will be the one whose URI corresponds to the sync source name), authentication credentials, required synchronization mode and so on. If something goes wrong, the error code specified by the server is returned and is processed accordingly:

```
if (ret != 0) {
     switch (ret) {
         case ERR_PROTOCOL_ERROR:
             LOG.error(TEXT("Protocol error"));
             break;

         case ERR_AUTH_NOT_AUTHORIZED:
         case ERR_AUTH_REQUIRED:
             LOG.error(TEXT("Not authorized"));
             break;

         case ERR_AUTH_EXPIRED:
             LOG.error(TEXT("Account expired; required payment"));
             break;

         case ERR_SRV_FAULT:
             LOG.error(TEXT("Server error"));
             break;

         case ERR_NOT_FOUND:
             wsprintf(logmsg, TEXT("Server returned NOT FOUND for SyncSource %s"),
source.getName(NULL, 0));
             LOG.error(logmsg);
             break;

         default:
             error();
             break;
     }

     goto finally;
```

```
    }
```

If the initialization succeeded, the source object gets the server requested synchronization mode. If it corresponds to a slow sync, we have to feed the *SyncSource* instance with all the items of a hypothetic data store, otherwise we can feed the sync source with only the last modifications.

```
    switch (source.getSyncMode()) {
        case SYNC_SLOW:
            setAllItems(source);
            break;

        case SYNC_TWO_WAY:
            setModifiedItems(source);
            break;

        default:
            break;
    }
```

setAllItems() and *setModifiedItems()* are utility procedures provided by the client application to accomplish the feeding operation and are reported later on.

Please note that the server returned sync mode can be different from the sync source preferred sync mode. This happens, for example when the client and the server cannot trust each other state and a slow sync is imposed by the server.

We are now ready to synchronize the source.

```
    if (syncManager->sync(source) != 0) {
        error();
        goto finally;
    }
```

During this process, the client side modifications stored in *source* are sent to the server; if everything goes well, the synchronization engine reads server-side modified items and stores them, again, in *source*. At the end of the process, *source* contains the items sent by the server.

This simple application does not interact with a final database, so the only thing we do with the received items is to display them on the screen:

```
    displayItems(source);
```

Before finishing the process, we have to send the mapping between LUID and GUID to the server. This is done setting the mappings in the source object:

```
    setMappings(source);
```

We can now end the synchronization process.

```
    if (syncManager->endSync(source) != 0) {
        error();
        goto finally;
    }
```

The way the *SyncSource* object is filled is shown by the implementation o the following procedures:

```
void setAllItems(SyncSource& s) {
    allItems[0] = new SyncItem(TEXT("item1"));
    allItems[1] = new SyncItem(TEXT("item2"));
    allItems[2] = new SyncItem(TEXT("item3"));
    allItems[3] = new SyncItem(TEXT("item4"));
```

```
    //
    // NOTE: keep into account the terminator
    //
    allItems[0]->setData(TEXT("This is item One")  , 17*sizeof(wchar_t));
    allItems[1]->setData(TEXT("This is item Two")  , 17*sizeof(wchar_t));
    allItems[2]->setData(TEXT("This is item Three"), 19*sizeof(wchar_t));
    allItems[3]->setData(TEXT("This is item Four") , 18*sizeof(wchar_t));

    s.setAllSyncItems(allItems, 4);
}

void setModifiedItems(SyncSource& s) {
    newItems    [0] = new SyncItem(TEXT("item4"));
    deletedItems[0] = new SyncItem(TEXT("item5"));
    updatedItems[0] = new SyncItem(TEXT("item1"));
    updatedItems[1] = new SyncItem(TEXT("item3"));

    newItems[0]->setData(TEXT("This is a new item Four"), 24*sizeof(wchar_t));
    updatedItems[0]->setData(TEXT("This is the updated item One"),29*sizeof(wchar_t));
    updatedItems[1]->setData(TEXT("This is the updated item Three"),31*sizeof(wchar_t));

    s.setNewSyncItems(newItems,         NEW_ITEMS_COUNT    );
    s.setDeletedSyncItems(deletedItems, DELETED_ITEMS_COUNT);
    s.setUpdatedSyncItems(updatedItems, UPDATED_ITEMS_COUNT);
}

void setMappings(SyncSource& s) {
    //
    // For the purpose of this example, LUIDs are created prepending
    // the string "C - " to the GUIDs
    //
    unsigned int n;
    wchar_t luid[DIM_KEY];
    SyncItem** items;

    n = s.getNewSyncItemsCount();
    items = s.getNewSyncItems();

    for (unsigned int i = 0; i<n; ++i) {
        wsprintf(luid, TEXT("C - %s"), items[i]->getKey(NULL));
        mappings[i] = new SyncMap(items[i]->getKey(NULL), luid);
    }

    s.setLUIDGUIDMapping(mappings, n);
}
```

5.2. Configuration properties

On PocketPC devices, the device management module handles configuration information in the PocketPC registry. Therefore, configuration contexts correspond to registry keys and configuration properties to registry values.

Edit the device registry using the Remote Registry Editor available with Microsoft eMbedded Visual C++ 4.0 (or any other registry editor) and create the keys and values shown in Figure 7 and Figure 8.

Note that the management tree starts at the HKEY_LOCALMACHINE\SOFTWARE key and the application URI is Sync4j/examples/dummy.

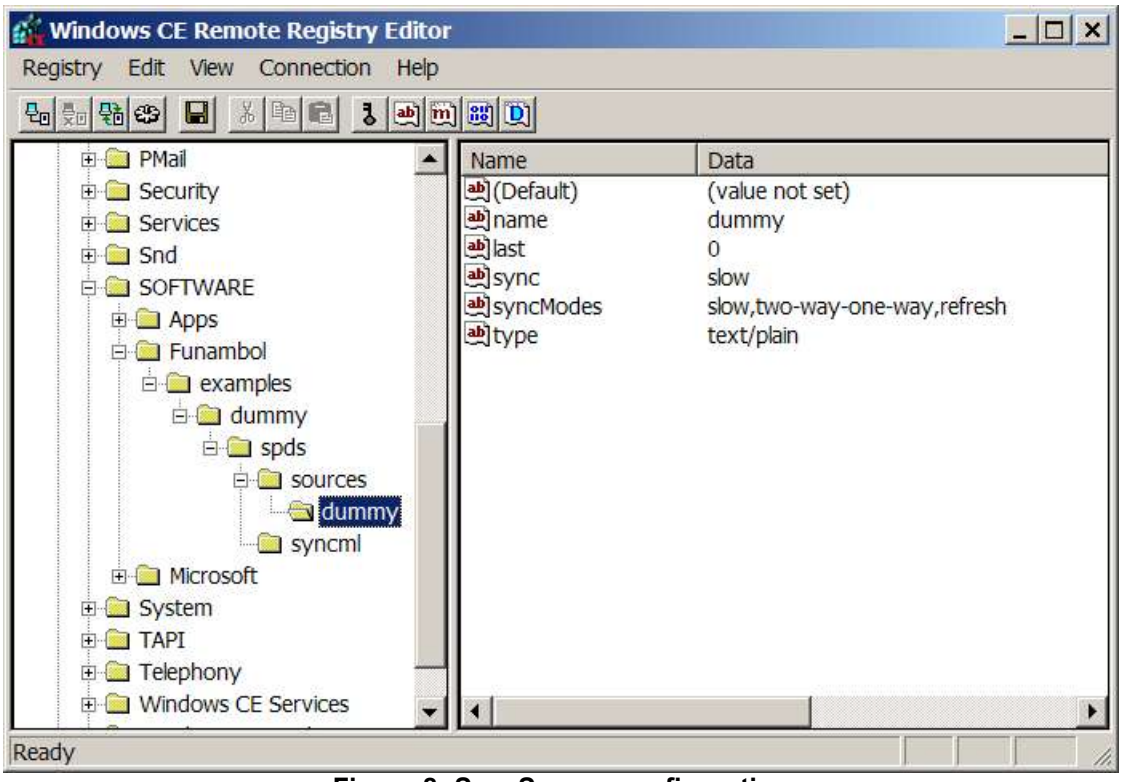

Figure 8- SyncSource configuration

5.3. Building and Running

To compile and run the example, open the project file “ppc2003.vcw” with Microsoft eMbedded Visual C++ 4.0, choose “Win32 (WCE ARMV4) Release” active configuration and “POKET PC 2003 Device” default device. Now build the project (press F7).
The file “ppc2003.exe” will be generated and transferred to the device under the root file system (\). Locate it in the File Explorer and run it.
Since a PocketPC application has no console, no output is immediately displayed. Instead, a log file “sync.txt” is generated under the Pocket PC 2003 root.
Opening the log file you have something similar to the following:

```
INFO - Preparing sync...
INFO - Starting sync...
INFO - New items
INFO - =========
INFO - key: 1, data: BEGIN:VCARD
VERSION:2.1
N:John;Doe;;;
TITLE:Marketing Director
EMAIL;WORK;PREF;INTERNET:jhon_doe@somewhere.com
TEL;WORK;PREF:+1 650 50504040
END:VCARD
INFO - Updated items
INFO - =============
INFO - Deleted items
INFO - =============
INFO - Mappings
INFO - ========
INFO - luid: C - 1, guid: 1
```

```
INFO - Ending sync...
INFO - Finalyzing...
INFO - Sync ended.
```

Choosing “Win32 (WCE ARMV4) Debug” as active configuration and repeating the steps above, you can see a more verbose log; it will look like the following:

```
INFO - Preparing sync...
DEBUG - Initialization message:
DEBUG - <SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>1</MsgID>
<Target><LocURI>http://localhost:8080</LocURI></Target>
<Source><LocURI>test</LocURI></Source>
<Cred>
<Meta><Type xmlns='syncml:metinf'>syncml:auth-clear</Type></Meta>
<Data>guest:guest</Data>
</Cred>
</SyncHdr>
<SyncBody>
<Alert>
<CmdID>1</CmdID>
<Data>201</Data>
<Item>
<Target><LocURI>dummy</LocURI></Target>
<Source><LocURI>dummy</LocURI></Source>
<Meta><Anchor xmlns="syncml:metinf">
<Last>29615689</Last>
<Next>29615689</Next>
</Anchor></Meta>
</Item>
</Alert>

<Final/>
</SyncBody>
</SyncML>
DEBUG - Initializing Winsock...
DEBUG - 3
DEBUG - Winsock initialized
DEBUG - Getting http://localhost:8080/sync4j/sync
DEBUG - Response message:
DEBUG - <SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>1</MsgID>
<Target>
<LocURI>test</LocURI>
</Target>
<Source>
<LocURI>http://localhost:8080</LocURI>
</Source>
<RespURI>http://192.168.0.34:8080/sync4j/sync?sid=W0JAYTlhMzJjLTEwNzUzNjgxNTczMT</RespUR
I>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>1</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://localhost:8080</TargetRef>
<SourceRef>test</SourceRef>
<Data>212</Data>
</Status>

<Status>
<CmdID>2</CmdID>
```

```
<MsgRef>1</MsgRef>
<CmdRef>1</CmdRef>
<Cmd>Alert</Cmd>
<TargetRef>dummy</TargetRef>
<SourceRef>dummy</SourceRef>
<Data>200</Data>
<Item>
<Data><Anchor xmlns='syncml:metinf'>
<Next>29615689</Next>
</Anchor>
</Data>
</Item>
</Status>

<Alert><CmdID>3</CmdID>
<Data>201</Data>
<Item>
<Target>
<LocURI>dummy</LocURI>
</Target>
<Source>
<LocURI>dummy</LocURI>
</Source>
<Meta><Anchor xmlns='syncml:metinf'>
<Last>1075368157493</Last>
<Next>1075368157493</Next>
</Anchor>
</Meta>
</Item>
</Alert>

<Final/>
</SyncBody>
</SyncML>

INFO - Starting sync...
DEBUG - Preparing slow sync for dummy
DEBUG - Synchronization message:
DEBUG - <SyncML><SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>2</MsgID>
<Target><LocURI>http://localhost:8080
</LocURI></Target>
<Source><LocURI>test</LocURI></Source>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>1</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://localhost:8080</TargetRef>
<SourceRef>test</SourceRef>
<Data>200</Data>
</Status>
<Status>
<CmdID>2</CmdID>
<MsgRef>1</MsgRef><CmdRef>1</CmdRef><Cmd>Alert</Cmd>
<TargetRef>dummy</TargetRef>
<SourceRef>dummy</SourceRef>
<Data>200</Data>
<Item>
<Data>
<Anchor xmlns='syncml:metinf'><Next>29615689</Next></Anchor>
</Data>
</Item>
</Status>
<Sync><CmdID>3</CmdID>
<Target><LocURI>dummy</LocURI></Target>
<Source><LocURI>dummy</LocURI></Source>

<Replace>
```

```
<CmdID>4</CmdID>
<Meta><Type xmlns='syncml:metinf'>text/plain</Type></Meta>
<Item>
<Source><LocURI>item1</LocURI></Source>
<Data>
This is item One</Data>
</Item>
<Item>
<Source><LocURI>item2</LocURI></Source>
<Data>
This is item Two</Data>
</Item>
<Item>
<Source><LocURI>item3</LocURI></Source>
<Data>
This is item Three</Data>
</Item>
<Item>
<Source><LocURI>item4</LocURI></Source>
<Data>
This is item Four</Data>
</Item>
</Replace>
</Sync><Final/>
</SyncBody>
</SyncML>
DEBUG - Getting http://192.168.0.34:8080/sync4j/sync?sid=W0JAYTlhMzJjLTEwNzUzNjgxNTczMT
DEBUG - Response message:
DEBUG - <SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>2</MsgID>
<Target>
<LocURI>test</LocURI>
</Target>
<Source>
<LocURI>http://localhost:8080
</LocURI>
</Source>
<RespURI>http://192.168.0.34:8080/sync4j/sync?sid=W0JAYTlhMzJjLTEwNzUzNjgxNTczMT</RespUR
I>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>4</CmdID>
<MsgRef>2</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://localhost:8080
</TargetRef>
<SourceRef>test</SourceRef>
<Data>200</Data>
</Status>

<Status>
<CmdID>1</CmdID>
<MsgRef>2</MsgRef>
<CmdRef>3</CmdRef>
<Cmd>Sync</Cmd>
<TargetRef>dummy</TargetRef>
<SourceRef>dummy</SourceRef>
<Data>200</Data>
</Status>

<Sync><CmdID>3</CmdID>
<Target>
<LocURI>dummy</LocURI>
</Target>
<Source>
<LocURI>dummy</LocURI>
</Source>
<Add><CmdID>2</CmdID>
```

```
<Meta><Type xmlns='syncml:metinf'>text/plain</Type></Meta>
<Item>
<Source>
<LocURI>1</LocURI>
</Source>
<Data>BEGIN:VCARD
VERSION:2.1
N:John;Doe;;;
TITLE:Marketing Director
EMAIL;WORK;PREF;INTERNET:jhon_doe@somewhere.com
TEL;WORK;PREF:+1 650 50504040
END:VCARD</Data>
</Item>
</Add>
</Sync>

<Final/>
</SyncBody>
</SyncML>

INFO - New items
INFO - =========
INFO - key: 1, data: BEGIN:VCARD
VERSION:2.1
N:John;Doe;;;
TITLE:Marketing Director
EMAIL;WORK;PREF;INTERNET:jhon_doe@somewhere.com
TEL;WORK;PREF:+1 650 50504040
END:VCARD
INFO - Updated items
INFO - =============
INFO - Deleted items
INFO - =============
INFO - Mappings
INFO - ========
INFO - luid: C - 1, guid: 1
INFO - Ending sync...
DEBUG - Mapping message:
DEBUG - <SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>2</MsgID>
<Target><LocURI>http://localhost:8080</LocURI></Target>
<Source><LocURI>test</LocURI></Source>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>1</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://localhost:8080</TargetRef>
<SourceRef>test</SourceRef>
<Data>200</Data>
</Status><Map>
<CmdID>3</CmdID>
<Target><LocURI>dummy</LocURI></Target>
<Source><LocURI>dummy</LocURI></Source>
<MapItem>
<Target><LocURI>1</LocURI></Target
><Source><LocURI>C - 1</LocURI></Source>
</MapItem>
</Map>
<Final/>
</SyncBody>
</SyncML>
DEBUG - Getting http://192.168.0.34:8080/sync4j/sync?sid=W0JAYTlhMzJjLTEwNzUzNjgxNTczMT
DEBUG - Response message:
DEBUG - <SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
```

```
<SessionID>1</SessionID>
<MsgID>2</MsgID>
<Target>
<LocURI>test</LocURI>
</Target>
<Source>
<LocURI>http://localhost:8080</LocURI>
</Source>
<RespURI>http://192.168.0.34:8080/sync4j/sync?sid=W0JAYTlhMzJjLTEwNzUzNjgxNTczMT</RespUR
I>
<NoResp/>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>2</MsgRef>
<CmdRef>0</CmdRef>
<Cmd>SyncHdr</Cmd>
<TargetRef>http://localhost:8080</TargetRef>
<SourceRef>test</SourceRef>
<Data>200</Data>
</Status>

<Status>
<CmdID>2</CmdID>
<MsgRef>2</MsgRef>
<CmdRef>3</CmdRef>
<Cmd>Map</Cmd>
<TargetRef>dummy</TargetRef>
<SourceRef>dummy</SourceRef>
<Data>200</Data>
</Status>

<Final/>
</SyncBody>
</SyncML>

INFO - Finalyzing...
INFO - Sync ended.
```

6. Sync4j SyncClient API Licensing

The Sync4j SyncClient API is released under the `GNU General Public License' (`GPL'). This means that the Sync4j SyncClient API can be used free of charge under the `GPL'. If you do not want to be bound by the `GPL' terms (such as the requirement that your application must also be `GPL', you may purchase a commercial license for the same product from Funambol; contact Funambol at sales@funambol.com .

Since Funambol owns the copyright to the Sync4j SyncClient API source code, we are able to employ `Dual Licensing', which means that the same product is available under `GPL' and under a commercial license. This does not in any way affect the `Open Source' commitment of Funambol. For details about when a commercial license is required, please see the Copyrights and Licensing section of this manual.

6.1. Copyrights and Licenses Used by Funambol

Funambol owns the copyright to the 'SyncClient API' source code.

All the source code in the SyncClient API is covered by the 'GNU General Public License'. The text of this license can be found as the file 'COPYING' in the distribution.

The *GNU General Public License* (GPL) is probably the best known *Open Source* license. The formal terms of the 'GPL' license can be found at http://www.fsf.org/licenses/. See also http://www.fsf.org/licenses/gpl-faq.html and http://www.gnu.org/philosophy/enforcing-gpl.html.

Since the 'Sync4j SyncClient API' software is released under the 'GPL', it may often be used for free, but for certain uses you may want or need to buy commercial licenses from 'Funambol'. Contact Funambol at sales@funambol.com for more information.

6.2. Using the Sync4j SyncClient API Software Under a Commercial License

The GPL license is contagious in the sense that when a program is linked to a 'GPL' program all the source code for all the parts of the resulting product must also be released under the 'GPL'. If you do not follow this 'GPL' requirement, you break the license terms and forfeit your right to use the 'GPL' program altogether. You also risk damages.

You need a commercial license:

- When you link a program with any 'GPL' code from the 'Sync4j SyncClient API' software and don't want the resulting product to be licensed under 'GPL', perhaps because you want to build a commercial product or keep the added non-'GPL' code closed source for other reasons.
  When purchasing commercial licenses, you are not using the 'Sync4j SyncClient API' software under 'GPL' even though it's the same code.

- When you distribute a non-'GPL' application that *only* works with the 'Sync4j SyncClient API' software and ship it with the 'Sync4j SyncClient API' software. This type of solution is considered to be linking even if it's done over a network.

- When you distribute copies of the 'Sync4j SyncClient API' software without providing the source code as required under the 'GPL' license.

- When you want to support the further development of the 'Sync4j SyncClient API' even if you don't formally need a commercial license.
  Purchasing support directly from 'Funambol' is another good way of contributing to the development of the 'Sync4j SyncClient API' software, with immediate advantages for you.

For commercial licenses, please contact Funambol at sales@funambol.com.

6.3. Using the Sync4j SyncClient API Software for Free Under GPL

You can use the 'Sync4j SyncClient API software' for free under the 'GPL' if you adhere to the conditions of the 'GPL'. Common uses of the 'GPL' include:

- When you distribute both your own application and the 'Sync4j SyncClient API' source code under the 'GPL' with your product.
- When you distribute the 'Sync4j SyncClient API' source code bundled with other programs that are not linked to or dependent on the 'Sync4j SyncClient API' system for their functionality even if you sell the distribution commercially.
  This is called mere aggregation in the 'GPL' license.
- When you are not distributing *any* part of the 'Sync4j SyncClient API' system, you can use it for free.

If your use of 'Sync4j SyncClient API' software does not require a commercial license, we encourage you to purchase support from 'Funambol' anyway.
This way you contribute toward 'Sync4j SyncClient API' development and also gain immediate advantages for yourself.

If you use the 'Sync4j SyncClient API' software in a commercial context such that you profit by its use, we ask that you further the development of the 'Sync4j SyncClient API' software by purchasing some level of support. We feel that if the 'Sync4j SyncClient API' helps your business, it is reasonable to ask that you help 'Funambol'. (Otherwise, if you ask us support questions, you are not only using for free something into which we've put a lot a work, you're asking us to provide free support, too.)