Developing Sync Applications using the Funambol C++ API

Last modified: September 11, 2009

Table of Contents

1. Overview

This tutorial is a practical guide for programmers who want to use the Funambol C++ API to create sync applications. It will guide you through the steps necessary to write a sync client, using a concrete example as guideline.

1.1. Who should read this document

This document is an introduction to application development using the Funambol C++ API. The target audience of this tutorial is developers approaching client development based on the Funambol framework. The only prerequisites requested are familiarity with C++ programming and the basics of the synchronization protocol SyncML [3].

Note that this guide is intended to be a starting point, it does not cover protocol details or implementation details of the Funambol API.

1.2. Goals of the tutorial

The main purpose of this document is to describe how to create a C++ sync application using the Funambol C++ API.

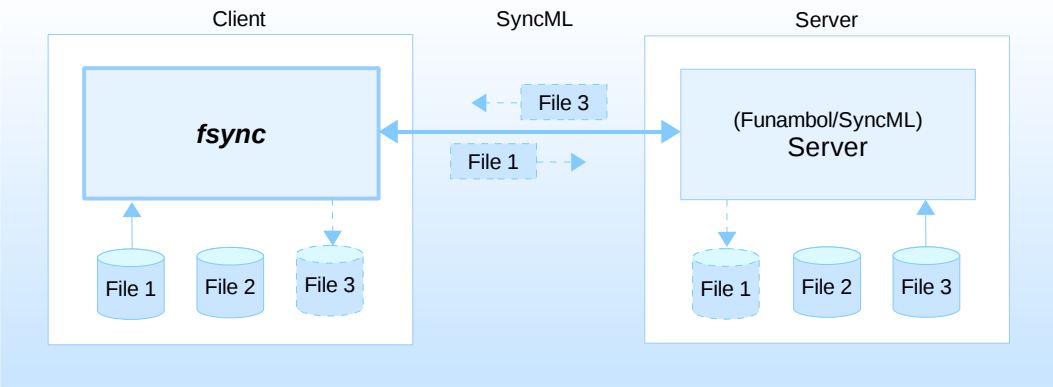

Figure 1: fsync overview

The explanation will be accompanied with an example of a real implementation, the *fsync* program, a simple sync application which is able to synchronize files in a local directory with the Funambol (or other SyncML compliant) server. Figure 1 illustrates an overview of what we are going to do.

The *fsync* application retrieves local files and exchanges modifications with the server (e.g. it sends *File1* which does not exist on the server, and receives *File3* which does not exist on the client's file system).

The first step will be to develop a minimal *fsync* application which contains the essential instructions needed by any sync application. We will see how a generic sync application works and the main components involved in the sync process. In particular it shows how the complexity of the SyncML protocol is hidden to the developer, who can focus on how to deal with the underlying data store (the file system in the case of *fsync*) and on how to interact with the user.

2. Getting started

Before going through the tutorial, it is necessary to set up the development environment.

2.1. Installing the Funambol server

First of all, you will have to install a local Funambol server to be used while testing the application examples. You can download it from https://www.forge.funambol.org/download/ (choose the correct version depending on your platform). Please follow the installation guide which you can find here: https://www.forge.funambol.org/download/documentation.html.

2.2. Obtaining the Funambol C++ API source code

In order to develop a sync application, you will need the source code of the Funambol C++ API. You can download the source package from the project's webpage [2].

Once you have downloaded the source code, check for the build requirements depending on your target platform as explained in the following sections.

2.3. Build requirements

The Funambol C++ API is available on many mobile and desktop platforms; the tutorial covers two of the most common: Windows and Posix. The result of the build is a static library to which you shall link in order to create your sync application.

2.3.1. Windows requirements

The Windows port contains the projects for Visual Studio 2005 (also the Express Edition is supported); the project to build the library is *Funambol/sdk/cpp/build/win32/win32.vcproj.*

2.3.2. Posix requirements

The Posix port requires the standard GNU toolchain (*autotools*, *make*, *gcc*). It has been tested on various Linux distributions and Mac OS X. The makefiles are under the directory: *Funambol/sdk/cpp/build/autotools.*

3. Introduction to sync application development

This section introduces sync application development using the Funambol C++ API. After a description of the components involved and the main roles, you will start to develop your first sync application.

3.1. The sync scenario

Figure 2 illustrates the typical scenario of a sync application:

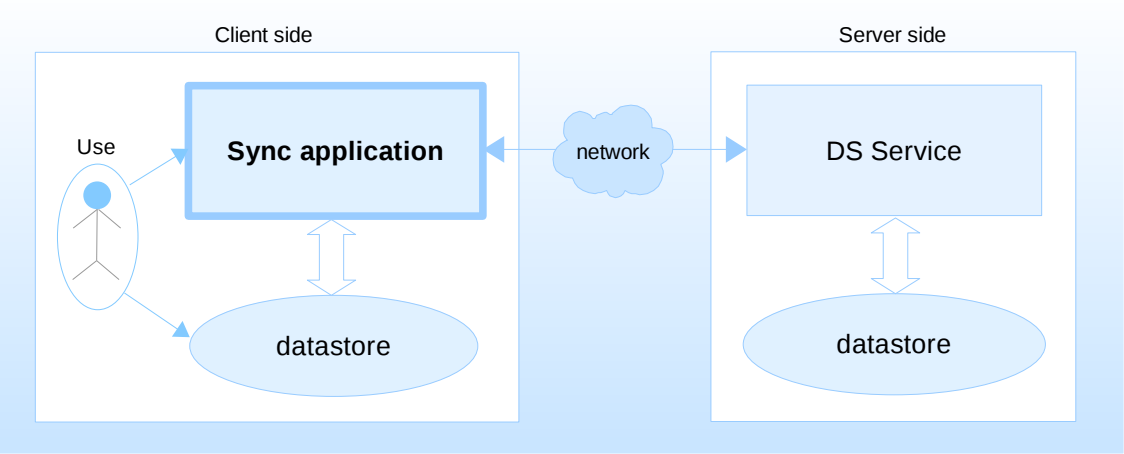

Figure 2: sync scenario

The sync application, normally invoked by the *user*, lies on the client side and is therefore commonly referred to as “client”.

It interacts with two main components:

●the DS (Data Synchronization) Service, that is the entity to which the client connects in order to exchange sync data. It can be a locally installed or remote server.

●the local datastore, that is the container/provider of the data that needs to be synchronized; a sync application that aims to sync local files (e.g. the *fsync* application) must be able to access the local file system that is handled by the user.

Note that the DS Service stores sync data on a local datastore, e.g. the file system or an SQL database.

The goal of the sync process is to keep the datastore on both sides in sync.

In this tutorial we will focus on how a sync application installed on the client side can obtain data from the local datastore and sync it with the server using the Funambol API.

4. The *fsync* application

After setting up the build environment (see section 2, Getting started) you can start developing your first sync application. This section analyzes the steps needed to create a simple sync application that synchronizes local files with a Funambol server.

4.1. fsync overview

Figure 3 illustrates the main architecture of the *fsync* application.

There are two main components used by the *fsync* application, in order to start the sync process and access to the local file system:

●the *SyncClient*, that is the entry point for the synchronization engine, which is implemented by the Funambol API. It is responsible for all the communication and protocol details of the synchronization. The *SyncClient* is already implemented so you just have to instantiate it.

●the *FileSyncSource*, which represents the glue between the *SyncClient* and the local datastore allowing the *SyncClient* to access to the file system.

The *FileSyncSource* inherits from the *SyncSource* interface which is used by the *SyncClient* to interact with a generic datastore; you should create a *SyncSource* for each datastore type (e.g. files, contacts, calendar, emails, etc.). In this case, the *FileSyncSource* is already implemented by the Funambol API.

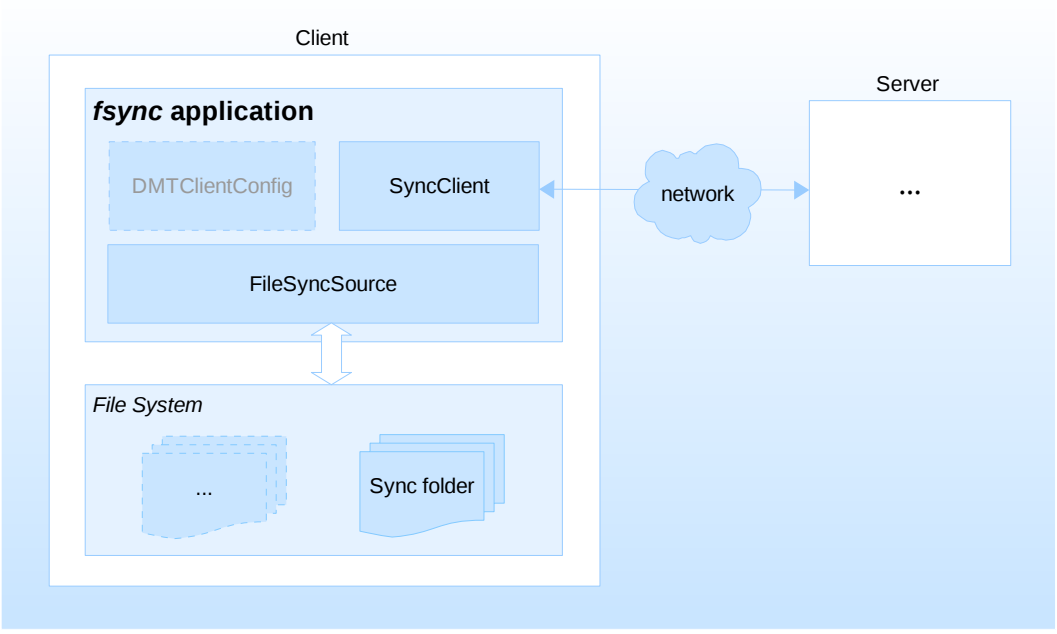

Figure 3: fsync architecture

Being *fsync* a client application, it needs to know the address of the server and the authentication credentials to be used. A generic synchronization application will also very likely have additional configuration parameters required to properly configure the data access layer; for this reason, the Funambol API provides a way to manage configuration parameters and pass them to the *SyncClient*.

The standard configuration details managed by the API are:

●access configuration, which includes the user credentials (username and password) and the server URL. It is used by the *SyncClient* when connecting to the server.

●device configuration, which is sent to the server in order to identify the client.

In addition to the information listed above, *fsync* requires further configuration parameters regarding the file system:

●the MIME type of the files (for the sake of keeping the tutorial simple, we are assuming all files are of the same MIME type, maybe a generic socket-binary)

●the remote URI, that is the datastore identifier on the server side

●the name of the directory to synchronize

The Funambol API allows to easily manage the client configuration and create a default one if needed, using the class *DMTClientConfig* (see [1]). The name of the directory to sync is an *fsync* specific parameter that is managed directly by the *FileSyncSource* class.

Being *fsync* a command line application, we will specify the access configuration and the directory name through command line parameters. You can invoke *fsync* using the following command:

```
$ fsync -d <directory to sync> -s <server url> -u <username> -p <password>
```

The flags that can be used to customize the sync configuration are:

●*-d*, used to specify the name of the directory to synchronize

●*-s*, used to specify the address of the server

●-u, used to specify the username;

●-p, used to specify the password.

Now you are ready to create the *fsync* application.

4.2. Creating the project

First of all, you will need to create a new project, depending on your platform.

On Windows:

1.create a new Visual Studio project: select an empty project

2.import the Funambol SDK to the current solution:

Funambol/sdk/cpp/build/win32/win32.vcproj

3.add the Funambol SDK as project dependency

4.select the release build configuration using the configuration manager

5.add the following include directories to the *fsync* project:

/path/to/Funambol/sdk/cpp/src/include/windows

/path/to/Funambol/sdk/cpp/src/include/common

6.copy the same preprocessor definitions found in the *win32* project

7.add the following library dependencies:

win32.lib (the Funambol SDK library);

wininet.lib

8.specify the following additional library folder:

/path/to/Funambol/sdk/cpp/build/win32/output/win32-rel

9.add to the project a new source file named *fsync.cpp*

On Posix:

1.build the Funambol library by running the following commands from the *Funambol/sdk/cpp/build/autotools* folder:

```
$ export LIBTOOLIZE=glibtoolize [only on Mac OSX]
$ ./autogen.sh
$ ./configure --disable-shared --prefix=<install-path>/cpp-sdk
[see –help for the parameters list]
$ make all
$ make install
```

2.create a source file named *fsync.cpp*

4.3. Writing the *fsync* source

Implementing the *fsync* application is very simple. First of all, you will have to include the Funambol API headers and specify that you are using the Funambol namespace:

```
#include "client/SyncClient.h"
#include "client/FileSyncSource.h"
#include "client/DMTClientConfig.h"
#include "client/OptionParser.h"
#include "spds/DefaultConfigFactory.h"
USE_FUNAMBOL_NAMESPACE
```

Next, define the main function:

```
int main(int argc, char** argv) {
    ...
}
```

The first thing to do is to parse the command line parameters. To do this, you can use the *OptionParser* class provided by the Funambol API:

```
OptionParser parser("fsync");
StringMap opts;
ArrayList args;
parser.addOption('s', "server", "set the server url", true);
parser.addOption('u', "user", "set the user name", true);
parser.addOption('p', "password", "set the user password", true);
parser.addOption('d', "dir", "set the local folder to sync", true);
parser.parse(argc, const_cast<const char **>(argv), opts, args);
```

The *parse()* method fills the *opts* object with all the command line options previously configured using the *addOption()* method.

Once you have read the sync options, you are ready to create the configuration object:

```
DMTClientConfig config("Funambol/fsync");
```

Then, you can fill all the configuration parameters. Device configuration and file system configuration parameters must only be set the first time the application runs:

```
if(!config.read()) {
   config.getDeviceConfig().setDevID("fsync-client");
   SyncSourceConfig* ssc = DefaultConfigFactory::getSyncSourceConfig("briefcase");
    ssc->setType("application/*");
    ssc->setURI("briefcase");
   config.setSyncSourceConfig(*ssc);
```

```
    delete ssc;  ssc = 0;
}
```

The *read()* method of the *DMTClientConfig* object returns "*false*" if the client configuration is not already set. This happens only at the first *fsync* run because after the first sync, the configuration object is persisted on the file system to keep session and global status information.

Note that the *FileSyncSource* MIME type is set to "*application/**"; this means that the raw file content is sent to the server as is.

The access configuration is set through the command line options:

```
config.getAccessConfig().setSyncURL(opts["server"]);
config.getAccessConfig().setUsername(opts["user"]);
config.getAccessConfig().setPassword(opts["password"]);
```

Now that you have the configuration parameters, you can create the *FileSyncSource* object and add it to the SyncSource array:

```
FileSyncSource fsource(TEXT("briefcase"), config.getSyncSourceConfig("briefcase"));
fsource.setDir(!opts["dir"].null()? opts["dir"]: "briefcase");
SyncSource* ssArray[] = { &fsource, NULL } ;
```

Note that the directory to sync is configured from the command line options, directly using the *FileSyncSource* object.

Finally you can start the sync process:

```
SyncClient client;
client.sync(config, ssArray);
```

The final step is needed to permanently save the current synchronization state, which contains information used by the next sync sessions:

```
config.save();
```

The *fsync* application is now implemented. To see the complete source code, please refer to Appendix A.

4.4. Building *fsync*

On Windows, select the release build configuration using the configuration manager and build the solution.

On Posix, run the following command:

```
$ g++ -O2 -I<install-path>/cpp-sdk/include/funambol/common -I<install-path>/cpp-sdk/include/funambol/posix -L<install-path>/cpp-sdk/lib -lfunambol -lcurl fsync.cpp -o fsync
```

You should now have obtained the *fsync* executable.

4.5. Testing *fsync*

In order to test *fsync*, first make sure that the Funambol server is running (see 2.1). Then, follow these steps:

1.from the *fsync* executable folder, create the directory you wish to synchronize (e.g. *briefcase*)

2.create some empty files in this directory: *file1.txt*, *file2.txt*, etc. (see Figure 4)

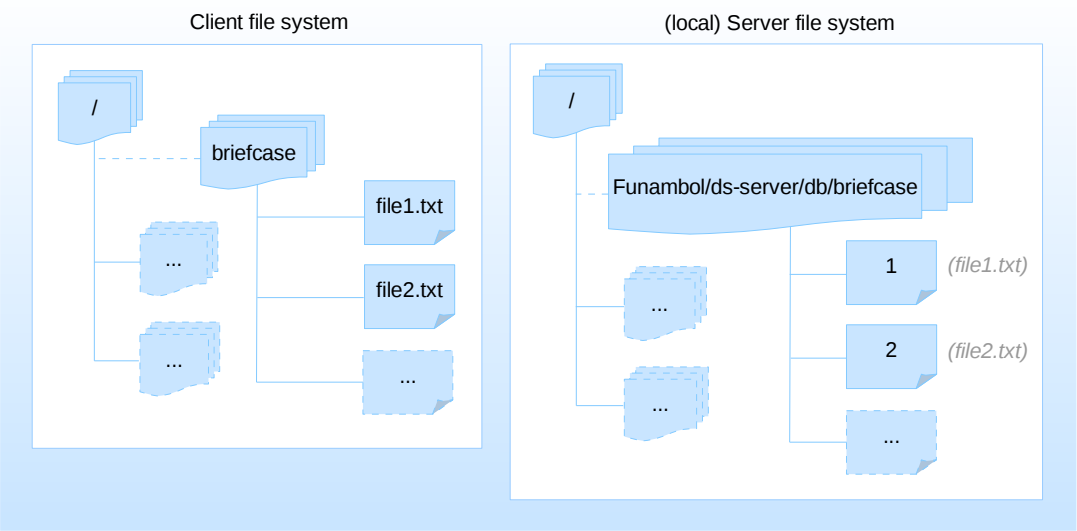

Figure 4: Client/Server file systems

3.run *fsync*:

```
$ ./fsync -d briefcase -s http://localhost:8080/funambol/ds -u guest -p guest
```

4.go to the server database path: */path/to/Funambol/ds-server/db/briefcase*. Here you will find the same list of files of the *briefcase* directory, but with different names (see Figure 4);

5.add a new file to the *briefcase* directory

6.run *fsync* again; a new file will be added to the server db

7.modify the files in the *briefcase* directory

8.run *fsync*

9.verify that the files in the server database have the same content as those stored in the client

You can run further tests to see how the sync process works; for example, delete some files from the server/client side, run *fsync*, etc.

5. Basic Types

The Funambol SDK is portable over a wide range of platforms. In order to maximize the number of platforms it can support, it does not make use of features of the C++ language such as Exceptions, Templates and the STL.

To ease the development of the library and clients, some basic types have been defined. This section describes the most commonly used types.

5.1. Enumeration

The *Enumeration* interface provides a simple, one-way iterator over a collection of items hiding how the collection is implemented.

The Funambol SDK provides a simple implementation of *Enumeration* which uses *ArrayList* as container.

5.2. ArrayList

ArrayList is a linked list container with the possibility to access an element by index and to traverse it using methods *front()*, *next()* like the STL vector.

Only objects that implement the *ArrayElement* interface can be added to an *ArrayList*.

5.3. StringBuffer

StringBuffer is a char string implementation similar to STL string, with the usual assignment, comparison and character access operators, but with some peculiarities:

- it can store empty strings or NULL strings, like a char pointer in C, and provides methods to test both conditions
- it has a *convert()* method to easily convert from *wchar_t* string using a defined encoding (default UTF8)
- it provides a *replaceAll()* method to easily substitute all occurrences of a substring

5.4. StringMap

StringMap is a simple associative array with a *StringBuffer* key and a *StringBuffer* value.

6. References

[1] The latest version of the Funambol C++ API design document, from the SVN repository:

https://client-sdk.forge.funambol.org/source/browse/client-sdk/trunk/cpp-sdk/design/

[2] Funambol C++ API source code snapshots:

https://client-sdk.forge.funambol.org/servlets/ProjectDocumentList?folderID=118&expandFolder=118&folderID=118

[3] Open Mobile Alliance SyncML specifications:

http://www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.html

Appendix A

Below is the complete source file of *fsync.cpp*:

```
#include "client/SyncClient.h"
#include "client/FileSyncSource.h"
#include "client/DMTClientConfig.h"
#include "client/OptionParser.h"
#include "spds/DefaultConfigFactory.h"
USE_NAMESPACE

int main(int argc, char** argv) {
    //Parse the commandline options
    OptionParser parser("fsync");
    StringMap opts;
    ArrayList args;
    parser.addOption('s', "server", "set the server url", true);
    parser.addOption('u', "user", "set the user name", true);
    parser.addOption('p', "password", "set the user password", true);
    parser.addOption('d', "dir", "set the local folder to sync", true);
    parser.parse(argc, const_cast<const char **>(argv), opts, args);

    // Create the config
    DMTClientConfig config("Funambol/fsync");

    // Read the configuration
    if(!config.read()) {
        config.getDeviceConfig().setDevID("fsync-client");
       SyncSourceConfig* ssc = DefaultConfigFactory::getSyncSourceConfig("briefcase");
        ssc->setType("application/*");
        ssc->setURI("briefcase");
        config.setSyncSourceConfig(*ssc);
        delete ssc;  ssc = 0;
    }
    config.getAccessConfig().setSyncURL(opts["server"]);
    config.getAccessConfig().setUsername(opts["user"]);
    config.getAccessConfig().setPassword(opts["password"]);

    // Create the FileSyncSource
   FileSyncSource fsource(TEXT("briefcase"), config.getSyncSourceConfig("briefcase"));
    // Specify the folder which needs to be synced
    fsource.setDir(!opts["dir"].null()? opts["dir"]: "briefcase");
    // Initialize the SyncSource array to sync
    SyncSource* ssArray[] = { &fsource, NULL } ;
    // Create the SyncClient
```

```
    SyncClient client;
    // SYNC!
    client.sync(config, ssArray);
    config.save();
}
```