SyncClient API 2.6 for Java Programmer Guide

Sync4j
http://www.sync4j.org

Funambol
http://www.funambol.com

Revision History

Name	*Date*	*Reason for Change*	*Ver./Rev.*
Stefano Fornari		Original draft	1.0

Table of Contents

1. Overview

The Sync4j SyncClient API is the means application developers can embed and 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 SyncClient API 2.6 for Java.

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.

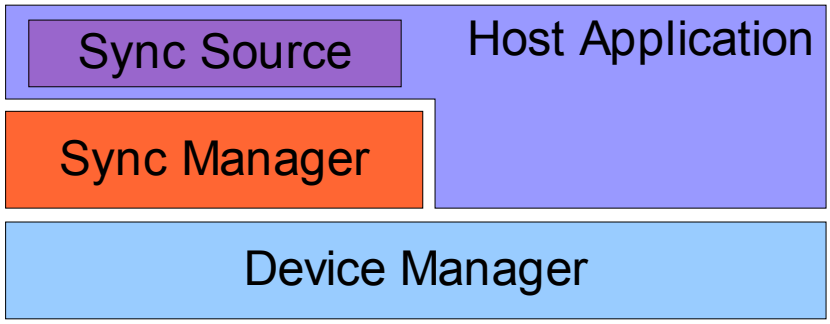

Figure 1 - SyncClient API architecture

The host application can access the services provided by both modules: the *Sync Manager* when a synchronization has to be performed and the *Device Manager* when the configuration must be read, manipulated or written. In addition, the Device Manager is intended to store host 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 a host application module that groups callback functions called 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.

2. Data Synchronization API

The synchronization API is grouped under the packages *sync4j.syncclient.spds* and *sync4j.syncclient.spds.engine*. The most important classes for a quick start are the Sync Manager itself, implemented in *sync4j.syncclient.spds.SyncManager* and the Sync Source interface, defined by *sync4j.syncclient.spds.engine.SyncSource*. The former is the driver of any synchronization operation, whilst the latter is the interface the host application has to implement to access external data sources. They are described in the following sections.

2.1. The Sync Manager

sync4j.syncclient.spds.SyncManager is the contact point between a host application and the synchronization engine. It is designed to hidden as much as possible the details of the synchronization logic, protocol and communication to the host application developer.

The simplest way to use SyncManager is to get a new instance and call its *sync()* method, as in the example below:

```
SyncManager syncManager = SyncManager.getSyncManager("test");
syncManager.sync();
```

getSyncManager() is a factory method that creates a new SyncManager bound to the given application URI. The application URI is an application identifier that must be unique amongst all the SyncPlatform-enabled applications running on the device.

The information required by the synchronization engine to initialize and to kick off a data synchronization session is stored in the device management configuration tree (see later) and can be manipulated by the means of the SyncPlatform Device Management API.

Error conditions are signalled throwing *sync4j.syncclient.spds.SyncException* for synchronization problems or *sync4j.syncclient.spds.DMException* for configuration problems.

See the javadoc documentation for the published API.

2.2. Sync Sources

A *Sync Source* is responsible for storing and retrieving data from/to an external data source.

A Sync Source is modeled by the interface *sync4j.syncclient.spds.engine.SyncSource*, which defines the following methods:

Name	Description
getName()	Returns the source display name.
getSourceURI()	Returns the source identifying target URI.
getType()	Returns the encoding mime type for items content.

Name	Description
beginSync()	Called just before the source synchronization takes place. If this method throws a SyncException, the synchronization won't be performed.
commitSync()	Called after the source synchronization has been committed. If this method throws a SyncException, the synchronization will not be committed.
getAllSyncItems(principal)	Returns all items belonging to the given principal.
getDeletedSyncItems(principal, since)	Returns deleted items belonging to the given user since the given point in time.
getNewSyncItems(principal, since)	Returns new items belonging to the given principal since the given point in time.
getUpdatedSyncItems(principal, since)	Returns updated items belonging to the given principal since the given point in time.
removeSyncItem(principal, key)	Removes the item belonging to the given principal and corresponding to the given key.
removeSyncItems(principal, keys)	Removes the items belonging to the given principal and corresponding to the given keys.
setSyncItem(principal, key)	Adds or updates the item belonging to the given principal and corresponding to the given key.
setSyncItems(principal, keys)	Adds or updates the items belonging to the given principal and corresponding to the given keys.

A *SyncSource* handles the items it contains in term of SyncItem (*sync4j.syncclient.engine.SyncItem*), another interface that models the smallest piece of information that can be synchronized. Exampled of content delivered by a SyncItem are database records or a vCard contacts.
A SyncItem is uniquely identified by its SyncItemKey (*sync4j.syncclient.engine.SyncItemKey*), which can be any application defined object (even if usually a simple String object is good enough).
Data are stored in one or more SyncProperty objects (*sync4j.syncclient.engine.SyncProperty*), of which two are standard properties and must be set in each SyncItem:

- BINARY_CONTENT: the binary version of the content as created by the source or read from the SyncML message. The format must be the one intended to be transported over the SyncML message. This property is a mandatory property because used by the synchronization engine.
- TIMESTAMP: contains the timestamp of the last modification (creation, update, deletion). This property is a mandatory property because used by the synchronization engine.

A SyncSource can create its own concrete implementation of SyncItem, even if this is usually not required. The SyncClient API provides a generic SyncItem implementation (*sync4j.syncclient.engine.SyncItemImpl)* that meets the most common needs.

A *SyncSource* is not used directly by the host application, instead its methods are called by the synchronization engine during modifications analysis (see the synchronization process section).

SyncSource methods are designed to perform an efficient synchronization process, letting the source selecting the changed items instead of doing more complex field by field comparison. It is source developer responsibility to make sure that the *getNew/Updated/DeletedSyncItems()* methods return the correct and values.

2.3. The Sync Process

From the host application developer perspective, the interaction with the synchronization engine is limited to firing the synchronization process calling *sync()*. However, under the covers, a lot of work happens. The main tasks performed during a sync execution are:

- synchronization initialization
- client modifications detection
- SyncML synchronization with the server
- server modifications execution

In order to make it possible, the synchronization engine interacts with the host application in two of the above tasks: *client modifications detection* and server *modifications execution* where the methods of the synchronizing sync source are called.

An important aspect of the synchronization process is the concept of fast and slow synchronization.
Fast synchronization can be performed when client and server rely on their respective state, because, for example, they have synchronized recently. In this case only the differences (the modifications) since the last synchronization are exchanged.
When for any reason, client and server are not confident about their respective state, fast synchronization cannot be done and *slow synchronization* is performed. In this case, the client sends its database content to the server, who compares the received information with its local database and then sends back the operations the client has to apply in order to be again up to date and in sync.

The synchronization process tasks are briefly described in the following.

Synchronization initialization
In this phase the synchronization engine prepares a new synchronization session, communicating to the server which sources it wants to synchronize and for which user. The server evaluates the request and responds a status message in which it allows or denies the request.

The Sync Manager synchronized the sources registered in the way described in the Sync Sources section.

Client modifications detection
Here there are two possibilities: in the case of fast sync, the Sync Manager asks the registered Sync Sources which items have changed since the last synchronization; in the case of slow sync, the Sync Manager asks for all items in the data store. As said, in this phase, the Sync Manager calls back the SyncSource's methods *getXXXSyncItem()*, which return the modified (or all) items.

SyncML synchronization with the server
This is the process of exchanging database modifications through the SyncML protocol. This task is hidden to the host application developer.

Server modification execution
This is the phase where server side modifications must be applied to the locale data store. Again, the Sync Manager delegates the SyncSources to execute the changes.

The synchronization process flow looks like Figure 2.

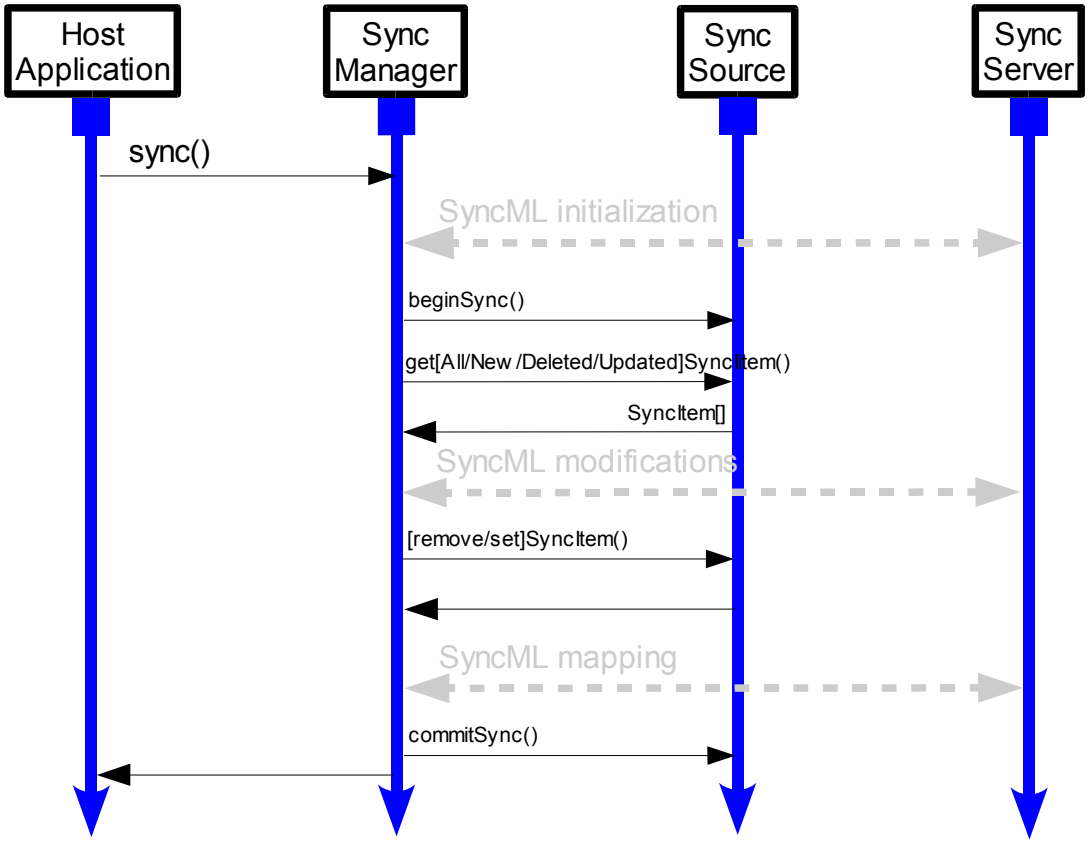

Figure 2 - Synchronization process flow

2.4. Configuring the Sync Manager

Sync Manager requires few configuration parameters such as the url of the SyncML server, which Sync Sources must be synchronized and so on. This information is stored in the device manager layer (see Device Management API). Configuration parameters are grouped by *configuration contexts* and organized in a *management tree*.
Sync Manager makes use of the following configuration parameters, divided by context:

Property	*Description*
<application uri>/spds/syncml	
syncml-url	The initial URL for the SyncML request
targetLocalUri	Server target URI
username	The principal to present to the server along with the SyncML request
password	Principal's credential
device-id	The device unique id
<application uri>/spds/sources/<source name>	
name	Source display name
type	Source type (e.g. text/plain, text/vcard, ...)
sourceClass	Source class name
sourceURI	Source URI

Property	Description
...	Other implementation specific parameters

Note that multiple sources can be put under the *<application uri>/spds/sources* context so that many source will be synchronized in sequence during the synchronization process.

2.4.1. Configuring a Sync Source

When the Sync Manager synchronizes a Sync Source, it first of all needs an instance of the implementation class. This is obtained thanks to the value specified for the *sourceClass* configuration property. After the instance is created, the other configuration properties under the <application uri>/spds/sources/<source name> management node are set to the corresponding properties in the implementation class (as soon as they have a setter method).

For example, suppose the following SyncSource configuration is stored in the device management:

- <application uri>/spds/sources/OrderSyncSource
  - sourceClass=sync4j.example.DBSyncSource
  - uri=OrderSyncSource
  - type=xml/recordset
  - database=orderdb
  - tableName=order
  - param=value

The DBSyncSource would look like:

```
public class sync4j.example.DBSyncSource
implements SyncSource
{
  private String uri;
  private String type;
  private String database;
  private String tableName;

  public DBSyncSource() {
    this.database = "default";   // setting a default database
  }

  public String getUri() {
    return uri;
  }

  public void setUri(String uri) {
    this.uri = uri;
  }

  public String getType() {
    return type;
  }

  public void setType(String type) {
    this.type = type;
  }

  public String getDatabase() {
    return database;
  }

  public void setDatabase(String database) {
    this.database = database;
  }

  public String getTableName() {
    return tableName;
  }
```

```
  public void setTableName(String tableName) {
    this.tableName = tableName;
  }

  ...
  // other methods
  ...
}
```

Note that the configuration parameters *sourceClass* and *param* have no corresponding *setXXX ()*, therefore they will not be set.

3. Device Management API

Goal of the device management module is to allow an easily 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.

3.1. The Device Manager

The main classes of the Device Management API are shown in Figure 3. The entrypoint is represented by the class *DeviceManager* 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.

3.1.1. Management Nodes

A management node can be considered a map that associates parameters names to values. This is a generic view that covers almost all possible requirements. However, a management node can have an optional special property: class. If class is specified, it is assumed that the properties of the node are properties of that class so that an instance of the class can be created and initialized with the node properties when *getValue()* is called. To do so, the class specified with the class property must adhere to the standard JavaBeans conventions. In particular:

- The standard empty constructor must be provided with public visibility
- For each property that is read/write from/to the ManagementNode the corresponding get/setXXX() methods must be provided with public visibility

There is no need to implement a particular interface or extend a particular base class.

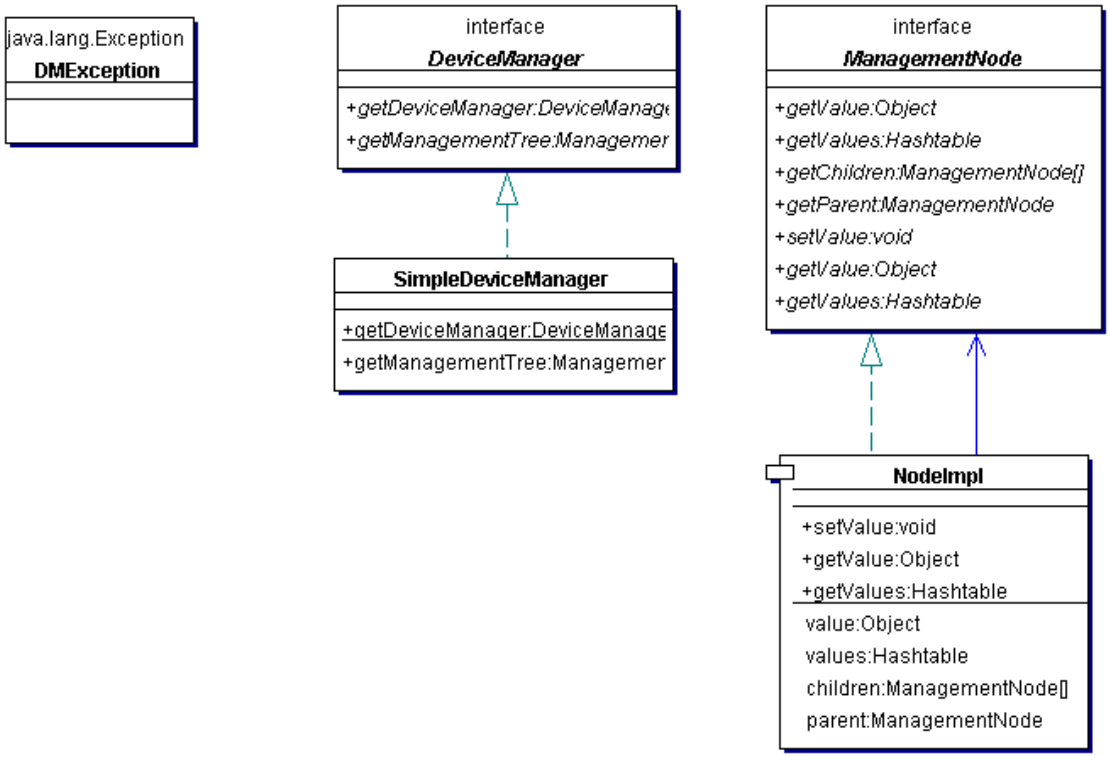

Figure 3 - SyncPlatform DM class diagram

3.1.2. SimpleDeviceManagement

The Sync4j SyncClient API 2.6 provides a simple implementation of a device manager that uses the file system to store the management tree. It is implemented by the class *sync4j.spdm.SimpleDeviceManager*, which uses directories to represent configuration contexts and properties files to store configurable properties.
The management tree is relative to a base directory specified with the system property *spdm.dir.base*. If not set, the current directory is picked up.

3.2. Examples

3.2.1. Getting a DeviceManager Instance

```
import sync4j.syncclient.spdm.DeviceManager;
import sync4j.syncclient.spdm.SimpleDeviceManager;

...

System.setPropety("spdm.dir.base", "/config");
DeviceManager dm = SimpleDeviceManager.getDeviceManager();
```

NOTE: each DeviceManager implementation is also a factory for concrete instances.

3.2.2. Getting the Root Management Tree

```
import sync4j.syncclient.spdm.ManagementNode;
import sync4j.syncclient.spdm.DeviceManager;
import sync4j.syncclient.spdm.SimpleDeviceManager;

...
DeviceManager dm = SimpleDeviceManager.getDeviceManager();

ManagementNode rootNode = dm.getManagementNode();
```

3.2.3. Reading Management Node Configurable Properties

```
import sync4j.syncclient.spdm.ManagementNode;
import sync4j.syncclient.spdm.DeviceManager;
import sync4j.syncclient.spdm.SimpleDeviceManager;

...
DeviceManager dm = SimpleDeviceManager.getDeviceManager();

ManagementNode rootNode = dm.getManagementNode();

Hashtable params = rootNode.getNodeValues();

System.out.println("syncml-url: " + params.get("syncml-url"));
```

3.2.4. Reading Node Children

```
import sync4j.syncclient.spdm.ManagementNode;
import sync4j.syncclient.spdm.DeviceManager;
import sync4j.syncclient.spdm.SimpleDeviceManager;

...
DeviceManager dm = SimpleDeviceManager.getDeviceManager();

ManagementNode rootNode = dm.getManagementNode();

ManagementNode sourcesNode = rootNode.getChildNode(CONTEXT_SOURCES);

ManagementNode[] sources = sourcesNode.getChildren();

Hashtable sourceConfig = null;
for (int i=0; i<sources.length; ++i) {
  sourceConfig = sources[i].getValues();
  System.out.println(sourceConfig.get("sourceURI"));
}
```

3.2.5. Update Configurable Properties

```
import sync4j.syncclient.spdm.ManagementNode;
import sync4j.syncclient.spdm.DeviceManager;
import sync4j.syncclient.spdm.SimpleDeviceManager;

...
DeviceManager dm = SimpleDeviceManager.getDeviceManager();

ManagementNode rootNode = dm.getManagementNode();

ManagementNode testNode = rootNode.getChildNode(CONTEXT_SOURCES + "/test");

testNode.setValue("property", "value");
```

4. Developing a Test Application

In this section, we are going to develop a test application from scratch. Our test application is a J2SE application, composed of the following files:

- Test.java: the main test application.
- DummySyncSource: a sync source that just prints out messages when its callback methods are called.
- Configuration files for the Device Manager
- The Sync4j SyncClient API jar files (SPDM.jar and SPDS.jar)

You can find all those files in the examples directory of the SyncClient API installation directory. The following sections explain the main files and the steps to build and run the example.

4.1. Test.java

This is the java program that we are going to launch, since it contains the *main()* method. This is all you need to use the Sync4j SyncClient data synchronization functionality:

```
package sync4j.syncclient.test;

import sync4j.syncclient.spdm.SimpleDeviceManager;
import sync4j.syncclient.spds.*;
import sync4j.syncclient.spds.engine.*;

public class Test {
    public static void main(String[] args) throws Exception {
        System.setProperty(SimpleDeviceManager.PROP_DM_DIR_BASE, "config");

        //
        // Starts and initializes the Sync Manager with application URI
        // sync4j.org/test
        //
        SyncManager syncManager =
           SyncManager.getSyncManager("sync4j.org/test");

        //
        // Synchronize!
        //
        syncManager.sync();
    }
}
```

4.2. DummySyncSource.java

This is a test implementation of a SyncSource, with the only goal of showing when its methods are called and with which parameters.

```
package sync4j.syncclient.test;

import java.security.Principal;
import java.util.Date;
```

```
import sync4j.syncclient.spds.engine.*;
import sync4j.syncclient.spds.SyncException;

public class DummySyncSource implements SyncSource {

    private String name       = null;
    private String type       = null;
    private String sourceURI  = null;

    private SyncItem[] allItems     = null;
    private SyncItem[] newItems     = null;
    private SyncItem[] deletedItems = null;
    private SyncItem[] updatedItems = null;

    // ------------------------------------------------------------ Constructors

    /** Creates a new instance of AbstractSyncSource */
    public DummySyncSource() {
        newItems     = new SyncItem[] {
            createItem("10", "This is a new item", SyncItemState.NEW)
        };
        deletedItems = new SyncItem[] {
            createItem("20", "This is a deleted item", SyncItemState.DELETED)
        };
        updatedItems = new SyncItem[] {
            createItem("30", "This is an updated item", SyncItemState.UPDATED)
        };

        allItems = new SyncItem[newItems.length + updatedItems.length + 1];

        allItems[0] = createItem("40", "This is an unchanged item",
SyncItemState.SYNCHRONIZED);
        allItems[1] = newItems[0];
        allItems[2] = updatedItems[0];
    }

    // ---------------------------------------------------------- Public methods

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("setName(" + name + ")");
        this.name = name;
    }

    public String getType() {
        return this.type;
    }

    public void setType(String type) {
        System.out.println("setType(" + type + ")");
        this.type = type;
    }

    public String getSourceURI() {
        return sourceURI;
    }

    public void setSourceURI(String sourceURI) {
        System.out.println("setSourceURI(" + sourceURI + ")");
        this.sourceURI = sourceURI;
    }

    public void setParam1(String value) {
        System.out.println("setParam1(" + value + ")");
    }


    public SyncItem[] getAllSyncItems(Principal principal)
    throws SyncException    {
        System.out.println("getAllSyncItems(" + principal + ")");
        return allItems;
    }
```

```
    public SyncItem[] getDeletedSyncItems(Principal principal,
                                          Date since         )
    throws SyncException {
        System.out.println("getDeletedSyncItems(" + principal + " , " + since + ")");

        return deletedItems;
    }

    public SyncItem[] getNewSyncItems(Principal principal,
                                      Date since         )
    throws SyncException {
        System.out.println("getNewSyncItems(" + principal + " , " + since + ")");
        return newItems;
    }

    public SyncItem[] getUpdatedSyncItems(Principal principal,
                                          Date since         )
    throws SyncException {
        System.out.println("getUpadtedSyncItems(" + principal + " , " + since + ")");

        return updatedItems;
    }

    public void removeSyncItem(Principal principal, SyncItem syncItem)
    throws SyncException {
        System.out.println("removeSyncItem(" + principal + " , " + syncItem.getKey().
getKeyAsString() + ")");
    }

    public SyncItem setSyncItem(Principal principal, SyncItem syncItem)
    throws SyncException {
        System.out.println("setSyncItem(" + principal + " , " + syncItem.getKey().
getKeyAsString() + ")");
        return new SyncItemImpl(this, syncItem.getKey().getKeyAsString()+"-1");
    }

    // ------------------------------------------------------- Private methods

    private SyncItem createItem(String id, String content, char state) {
        SyncItem item = new SyncItemImpl(this, id, state);

        item.setProperty(
            new SyncItemProperty(
                SyncItem.PROPERTY_BINARY_CONTENT,
                content.getBytes()
            )
        );

        return item;
    }
}
```

4.3. Configuration Files

Because the Sync Manager uses the SimpleDeviceManager as Device Manager, the configuration properties can be stored in the file system configuration tree as properties files. This has the advantage of making easy changing the configuration by hands, without other tools than a simple text editor.

The example has the following configuration structure:

```
<installation dir>
  + config
      + sync4j.org
          + test
              + spds
                  - syncml.properties
                  + sources
                      - test.properties
```

Note that the management tree starts at the sync4j.org directory, since the spdm.dir.base system property points to <installation directory>/config and the application URI is sync4j.org/test.

4.3.1. syncml.properties

```
#
# Configuration file for the SyncML client agent
#

#
# The initial URL for the SyncML request
#
syncml-url=http://localhost:8080/sync4j/sync


#
# The target URI of the server being contacted
#
targetLocalUri=sync.sync4j.org

#
# Username and password for authentication to the sync server
#
username=test
password=test

#
# The identification tag of this SyncML agent
#
device-id=test
```

4.3.2. test.properties

```
name=test
type=clear/text
last=1056201555322
class=sync4j.syncclient.test.DummySyncSource
sourceURI=test
param2=value2
param1=value1
```

4.4. Building and Running

To compile the classes, use the following commands:

```
set CLASSPATH=..\output\SPDM.jar;..\output\SPDS.jar;config\xml
%JAVA_HOME%\bin\javac -d classes src\sync4j\syncclient\test\*.java
```

If they compiled without errors, run the test program:

```
set CLASSPATH=classes;..\output\SPDM.jar;..\output\SPDS.jar;..\lib\sync4j-
clientframework.jar
%JAVA_HOME%\bin\java sync4j.syncclient.test.Test
```