Java ME SyncML API
Design Document

Last modified: September 11, 2009

Changes History

Date	*Author*	*Changes*
2006.11.06	Andrea Gazzaniga	Initial Draft
2006.11.06	Giuseppe Monticelli	Server Alerted Notification
2006.11.09	Ivano Brogonzoli	OTA Configuration
2006.11.10	Andrea Gazzaniga	Version 1.0 (first review completed)
2006.11.15	Giuseppe Monticelli	SAN Java examples, UML class diagrams
2006.11.16	Andrea Gazzaniga	Editorial changes
2006.11.17	Stefano Fornari	Editorial and content review
2006.12.04	Andrea Gazzaniga	Version 1.1 (after public review)
2007.12.07	Edoardo Schepis	SAN and OTAConfig paragraphs moved to common package documentation Added Large Object documentation
2008.09.16	Ivano Brogonzoli	Added The MappingManager part. It explains how lost mappings informaions are recovered by the API and how it solves the problem of duplicated items.
2008.12.12	Marco Garatti	Reviewed large object section
2008.12.12	Marco Garatti	Added TrackableSyncSource and ChangesTracker sections
07/21/09	Giulia Zanchi	Fixed copyright info and front page

Table of Contents

1. Overview

The Funambol SyncML API allows application developers to embed SyncML capabilities in a J2ME application, thus giving it the access to a powerful data synchronization framework to keep the application data in sync with other devices.

This document explains, from a developer point of view, the architecture of the Funambol SyncML API for Java 2 Micro Edition.

1.1. Document Conventions

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

1.1.1. Sequence Diagrams

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

1.1.2. Class Diagrams

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

1.2. Functional overview

The Funambol SyncML API for J2ME is currently focused on data synchronization only. The device management protocol (OMA-DM), available on other platforms, is not implemented in this version of the API.

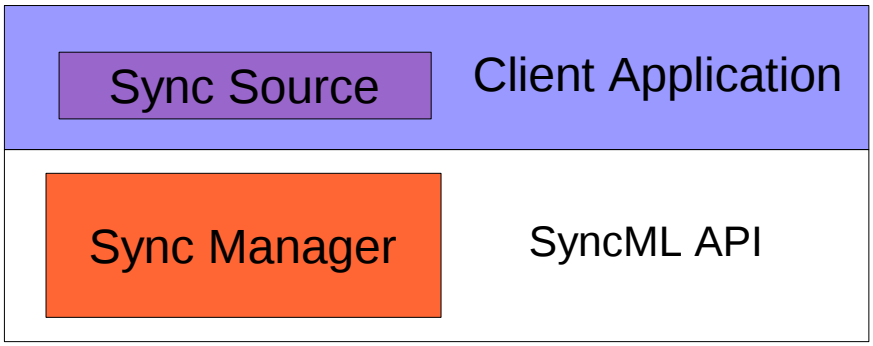


Figure 1.1 - SyncClient API architecture

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

2. Data Synchronization Layer

This chapter describes the design of the Funambol J2ME SyncML API Data Synchronization Layer.

The Data Synchronization process implemented in the Funambol API follows the OMA DS 1.2 specification (see [OMA-DS] and related documents).

2.1. The Synchronization Process

In this section, a brief introduction to the synchronization process is provided to better understand the use of the SyncML API. For a more detailed description see the OMA DS specification ([OMA-DS]).

The synchronization process is logically a sequence of three phases:

1. Initialization
2. Modifications exchange
3. IDs Mapping

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

After the initialization phase, first the client sends all client-side modifications and receives the status of the execution of the commands on the server; then, the server sends server-side modifications. The client applies the changes and sends the proper status back to the server.

In the case the server issued a new item to the client, the latter creates a new local id for it and therefore needs to communicate such new key to the server (see LUID-GUID mapping in [OMA-DS]). This is usually done in the mapping phase (or at least, the Funambol client API will send the mapping just after the modifications exchange phase).

The synchronization flow is illustrated by the sequence diagram of Figure 2.1, where the entities that participates to the sync have the following meaning:

- Client Application: a final client application using the Funambol SyncML API;
- SyncManager: the SyncML API synchronization engine;
- SyncSource: the interface for a client data source;
- SyncListener: the interface for monitoring a synchronization process;
- Sync Server: the SyncML server

As shown in the figure, the client application just kicks off the synchronization process giving a sync source to synchronize. The SyncManager takes control of the synchronization process. The macros initialization, modifications exchange and client mapping involve the

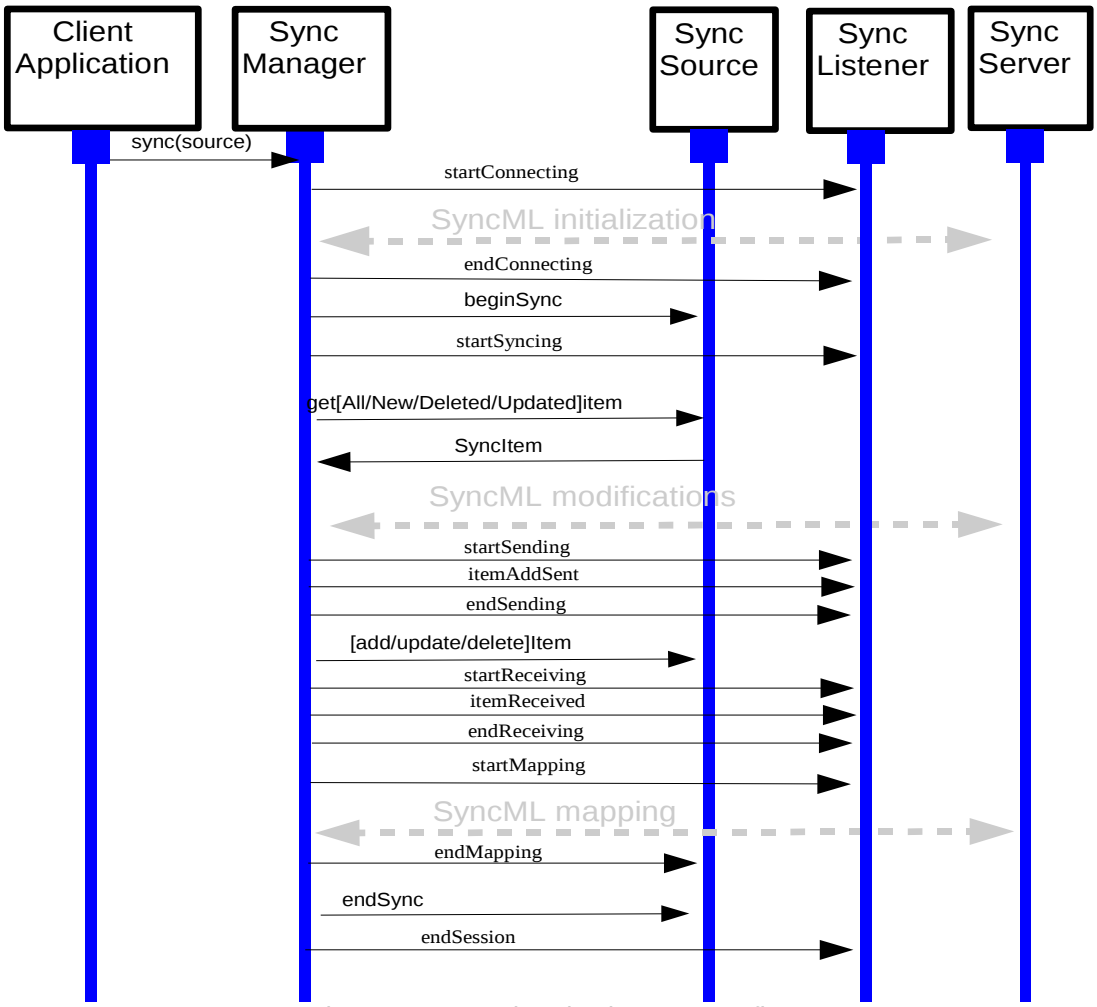


Figure 2.1 - Synchronization process flow

interaction with the server and include the required SyncML messages exchange.

To simplify the figure, some back arrows are not shown as they do not carry information (for example the ones from the listener back to the SyncManager).

2.1.1. Initialization

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

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

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

The specification defines three important tasks performed in the initialization phase:

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

The server authentication is not implemented in the current version of the J2ME SyncML API, so it will not be further described.

Client Authentication

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

The SyncML specifications mandate that client implementations must support at least *basic* and *MD5 Digest* authentication.

Client authentication is delivered in the Cred element of the SyncML message header. Plus, even if the client sends in the first message its credentials in one of the supported types, the server can refuse it and challenge the client for a different kind of credentials. For example, if the client starts sending Basic credentials and the server requires MD5 authentication, the server responds with a 401 status to the client credentials and provides a Chal element with the requested authentication.

The Funambol J2ME SyncML API implements only the basic one at the moment. If the server requires an MD5 authentication, the client recognize the request and report an an invalid authentication method exception.

For additional information on Basic and MD5 authentication see section 2.5 of [OMA-DS].

Basic authentication is identified by the URI syncml:auth-basic; it is pretty simple and it is very similar to what happens in Web applications.

In this case, credential data are encoded as follows:

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

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

Database Alerting

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

Sync Type	***Description***	***Alert Code***
Two-way	A normal sync type in which the client and the server exchange	200

Sync Type	*Description*	*Alert Code*
	information about modified data in these devices. The client sends the modifications first.	
Slow	A form of two-way sync in which all items are compared with each other on a field-by-field basis. In practice, this means that the client sends all its data from a database to the server and the server does the sync analysis (field-by-field) for this data and the data in the server.	201
One-way from client only	A sync type in which the client sends its modifications to the server but the server does not send its modifications back to the client.	202
Refresh from client only	A sync type in which the client sends all its data from a database to the server (i.e., exports). The server is expected to replace all data in the target database with the data sent by the client.	203
One-way from server only	A sync type in which the client gets all modifications from the server but the client does not send its modifications to the server.	204
Refresh from server only	A sync type in which the server sends all its data from a database to the client. The client is expected to replace all data in the target database with the data sent by the server.	205

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

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

2.1.2. Modifications Exchange

The modification phase is when client modifications are detected on the client and sent to the server who replies with the server side updates. These updates are applied to the local database. Accordingly to the SyncML specs, updates are exchanged with commands like Add, Replace, Delete, Copy, Move, etc.

However only Add, Replace and Delete are mandatory to SyncML implementations; the others are optional (see the specs for details).

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

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

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

The interaction between the SyncManager and the SyncSource is the core of the synchronization process. Since the number of items in the local data source may be relevant, and thinking of the limited resources of a J2ME device, the SyncSource works like a cursor: it provides getItem()/getNewItem()/getUpdatedItem()/getDeletedItem() to retrieve the first and next items in the data store. The SyncSource implementation must reset the cursor

after a call to the beginSync() method, and return the next item of the requested type at each call.

In this way the SyncManager has the opportunity to choose to add the item in the current message or in the next one, to implement the multi-message mechanism, described later.

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

On the opposite direction, when modifications are received from the server, the SyncManager decodes each command and call add/update/deleteItem() on the SyncSource implementation. These calls apply the change and return a status code (the same value that will be sent in the SyncML message).

Note that addItem() returns also the new local ID generated by the data store (LUID); this is stored in the SyncSource along with the GUID received by the server so that the mapping will be communicated back to the server in a Map command (see the next phase).

Multiple messages in one package

OMA-DS specification defines a way to split the modification package over multiple messages, which means the client can send its modified items in more messages and get the server items in the same way. This process is commonly called “multi-messaging” and it is used to meet the limited resources requirements of mobile devices. The Funambol J2ME SyncML API supports multi-messaging, as summarized below:

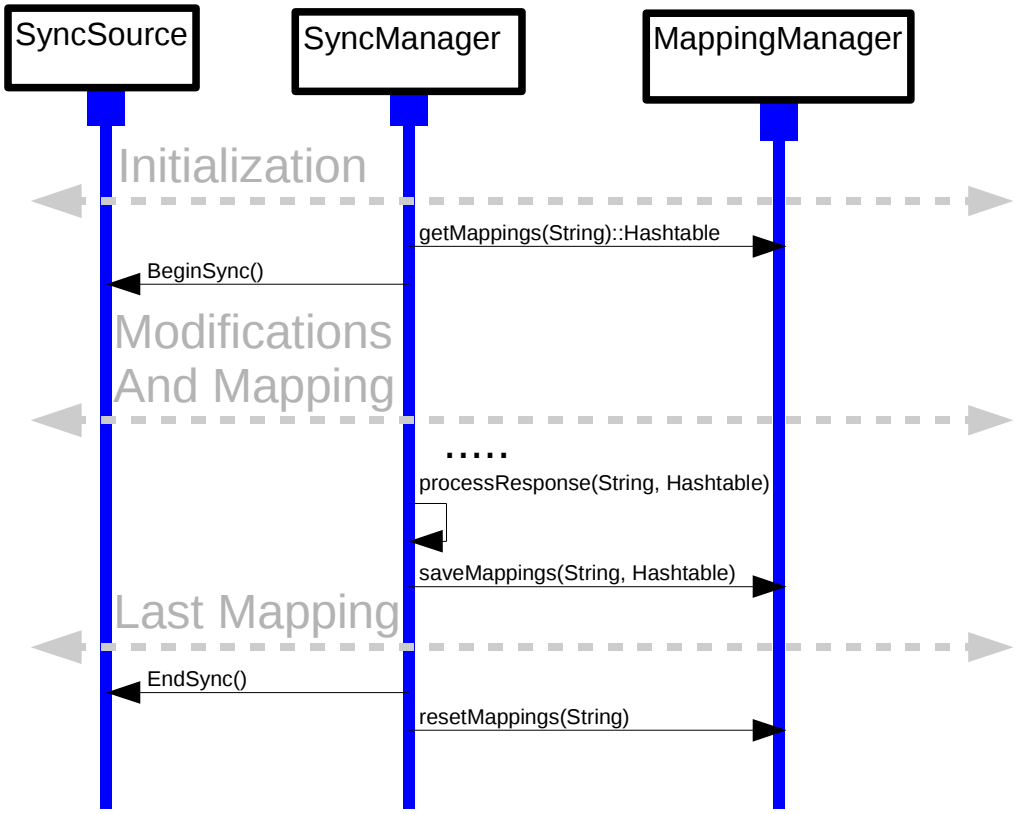
1. Client sends the first n items in the first message of PKG #3, without the <Final/> element;
2. Server applies the changes and replies with a message containing only status commands; the server is not allowed to send its own changes until client finished to send client side changes (which means unless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

2.1.3. LUID-GUID Mapping

After the server has done with its modifications, along with the latest status command, the client is requested to send also the LUID-GUID mapping, if there is any. As already mentioned, such mapping is stored temporarily in the SyncSource implementation's instance, therefore the SyncManager can easily extract them and embed them into the message. However, it may happen that the communication falls in the middle of sending the message or before receiving the response from the server. For this reason, mappings must

be permanently stored before sending them and can be removed only after having received a successful status for the sent Map command.

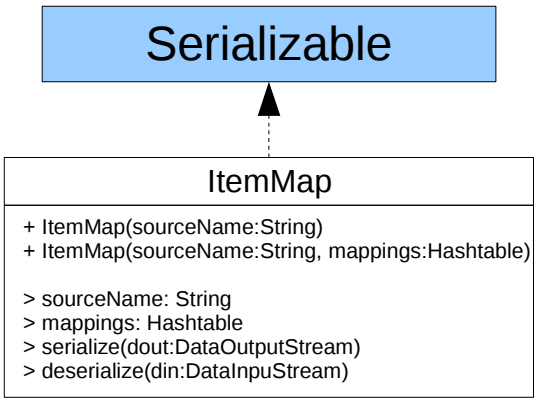
If, when a new synchronization starts, the persistent mapping is still present, it must be sent again. The Class that manages this task is called MappingManager and it is instantiated by the sync manager, that invokes it during the sync process as per the diagram below:



The MappingManager class refers to a particular Serializable object called the ItemMap, that contains the name of the source mappings to be persisted and the hashtable with the mappings informations.

Before the sync begins the SyncManager cheks if there are mappings to be sent for the given source. Due to the fact that the sync implements the multimessage tecnique every time a mapping information must be sent, the SyncManager saves the mappings before sending the message: when a network problem occurs the persisted mappings will be sent on the next sync process only if the next sync type is two way or one way from server (in case of slow syncs or refresh syncs no mapping message recovery is needed). The call to the reset method is done when the sync ended succesfully.

The ItemMap class is owned by the MappingManager and its diagram is shown below:



The ItemMap class implements the Serializable interface. The two methods serialize(..) and deserialize(...) are implicitly used by the store when the source mappings are persisted into the RMS. Due to performanse issues the mappings table is persisted every time a mapping list is filled after the response by the server is processed. This is a bit different from the C++ API implementation that saves every mappings anytime a new item is added on the client. Doing the same on the RMS wuold create loss of performance into the sync process, so it has been decided to implement the mapping caching once per message.

2.2. Data Synchronization Layer Architecture

This section defines the design of the data synchronization layer. The fundamental components are: SyncManager, SyncSource and SyncItem. Roles and responsibilities of these components are summarized in the table below.

Component	***Implementing class / Interface***	***Roles and responsibilities***
SyncManager	com.funambol.syncml.spds.SyncManager	This is responsible for the implementation of each single phase of the synchronization. It is also responsible of driving the process of building the proper SyncML messages to send to the server through the TransportAgent and to interpret the server response and take the consequent actions.
SyncSource	com.funambol.syncml.spds.SyncSource	An implementation of this interface represents a client database. It is used by the SyncManager when it needs to read or store data from and to the local database. SyncSource is purely an interface, which client developers must implement in order to access the most disparate data sources.This componentn
SyncItem	com.funambol.syncml.spds.SyncItem	This class is a container for the items exchanged between the SyncManager and the SyncSource. It is designed to let the SyncManager handle any data in the same way.
SyncListener	com.funambol.util.SyncListener	This class is a generic listener to monitor a synchronization session.

Figure 2.2 represents the class diagram for the data synchronization layer (only the main classes with the relevant details are shown, for a complete description of the classes please refer to the Javadoc).

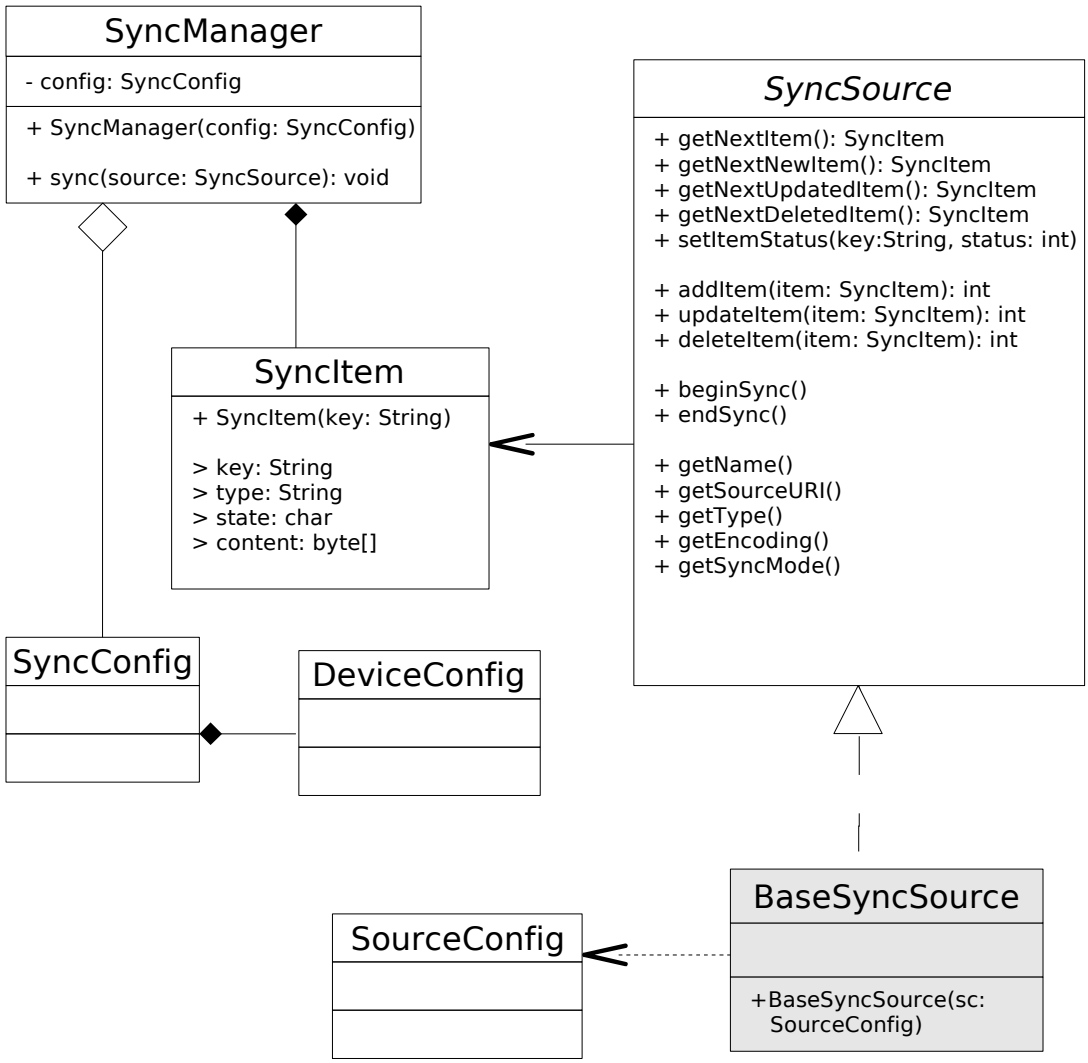


Figure 2.2: SyncManager class diagram

2.2.1. SyncManager

SyncManager represents the core of the data synchronization API, and is the contact point between the application and the synchronization engine.

The SyncManager needs some configuration parameters to work, and these parameters are passed during the creation of a new instance of the class using the SyncConfig class. It is client's responsibility to fill or retrieve the config before creating the SyncManager, but the SyncConfig already implements the Serializable interface (see [COMMON]) to make it easier for the client to store and retrieve it to and from the device persistent storage. See below for a description of the SyncConfig class.

A simplified example of SyncManager creation can be:

```
SyncConfig conf = new SyncConfig();
// set something in the conf...
conf.serverUrl = "http://www.somewhere.com/funambol.ds"
SyncManager sm = new SyncManager(conf);
```

Trigger synchronization

To actually start a synchronization, the client application must call the sync() method, passing the SyncSource to synchronize as a parameter (see the SyncSource section for more details on how to create a SyncSource):

```
sm.sync(source)
```

When `sync()` is invoked the SyncManager start the sync process as described in section 2.1.

This version of the API does not implement large object handling yet (which means the ability to split a big item amongst multiple messages). This feature will be implemented in a future version of the API.

Detecting, Sending and Committing Client Modifications

Note that the way changes are detected is left to the SyncSource implementor. It may be based on flag modification, change log handling, versioning... this is hidden to the SyncManager who just asks for modifications calling getNext***Item() on a SyncSource implementation's instance.

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

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

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

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

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

Applying Server Modifications

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

Note that the item parameter passed to addItem() on return may have a changed key. This represents the local id (LUID) that will be returned to the server in a Map command. See the table in the SyncSource Interface paragraph for details on the addItem() method.

Device capabilities handling

SyncManager also provides device capabilities. This information combines device specific information with data source related information, therefore properties from many configuration objects will be used to build the DevInf command to send to the server. DevInf properties are retrieved from the configuration (SyncConfig object) passed to the SyncManager constructor (see SyncManager and SyncSource configuration).

If it is the first time SyncManager connects to a server, client capabilities are sent in a Put command (see the client capabilities handling session).

In the same way, if during initialization the server requests client capabilities sending, SyncManager will send them in a Results command.

2.2.2. The SyncSource Interface

SyncSource is an interface that implementors must implement in order to access specific data sources. The following table gives a list of the main methods with a brief description. Please refer to the Javadoc for a complete description of the interface.

Method	***Description***
void beginSync(int syncmode)	Called by the SyncManager after the initialization phase, when the sync mode sent by the server is known. The syncmode is normally used by the SyncSource implementation to prepare the lists of items.
void endSync()	Called by the SyncManager before committing the source. The SyncSource implementation can stop the commit phase throwing an exception here.
SyncItem getNextItem()	Returns the next item regardless its modification state, or NULL if the source contains no items.
SyncItem getNextNewItem()	Returns the next new item or NULL if there are no more new items.
SyncItem getNextUpdatedItem()	Returns the next updated item or NULL if there are no more updated items.
SyncItem getNextDeletedItem()	Returns the next deleted item or NULL if there are no more deleted items.
void setItemStatus(String key, int status)	Called when a Status for a modification command previously sent is returned by the server. This gives the opportunity to the SyncSource to know if the modification has succeeded and to reset the related modification flag.
int addItem(SyncItem item)	Adds a new item to the underlying data source. The call returns the status code that SyncManager shall return to the server in the Status command. If the SyncSource implementation generates a new local ID for the added item, it must set the new key in the Item object, so that the SyncManager will be able to send it back to server in a Map command.
int updateItem(SyncItem item)	Updates an item into the underlying data source. The call returns the status code that SyncManager shall return to the server in the Status command.
int deleteItem(SyncItem item)	Deletes an item from the underlying data source. The call returns the status code that SyncManager shall return to the server in the Status command.
void dataReceived(String date, int size)	Called from the engine when new data arrives from the server. The date is in the format of the HTTP header, if available, or null otherwise. The size is the total size of the message reveiced, invluding the protocol overhead.
SyncFilter getFilter()	Returns an implementation of the SyncFilter interface, with the filter to be used for this SyncSource. See section 2.5. for more details.
void setFilter(SyncFilter filter)	Sets the filter that should be used in preparation for the synchronization of this SyncSource. See section 2.5 for more details.

2.2.3. BaseSyncSource

BaseSyncSource is an abstract class implementing the SyncSource interface. It contains a SourceConfig (see 2.2.6) instance with the source configuration data, and implements the most common methods. An implementor can extend this class implementing only the getXXXItem() and the add/update/deleteItem() methods.

The constructor of BaseSyncSource, and generally of all the SyncSource implementations, requires a SourceConfig parameter (a SyncSource implementation can also use a class derived from SourceConfig). Given a TestSyncSource that extends BaseSyncSource, and a NamedObjectStore instance used to retrieve the config (see [COMMON]), an example of use can be:

```
SourceConfig sc = new SourceConfig();
namedObjectStore.retrieve("source.test", sc);
TestSyncSource tss = new TestSyncSource(sc);
// Perform a sync: the process can change the SourceConfig (the
anchors,
// for instance)
sm.sync(tss);
// Save the modified config
namedObjectStore.store("source.test", sc);
```

2.2.4. TrackableSyncSource

A trackable sync source is an extension of BaseSyncSource that adds the concept of a tracker of changes made since the last synchronization. This concept is abstracted by an interface named ChangesTracker described in the next section.

When a TrackableSyncSource is created it needs a tracker used when the set of modified items is computed. This source at this point is capable of computing the list of changed items (for fast syncs). The only requirement for deriving classes is to implement a method to get the keys of all items in the store. This method is typically used in two scenarios:

- during initAllItems (invoked by the SyncManager for slow syncs)
- to track changes in a fast sync when the tracker is based on fingerprints (for example the CacheTracker, descibed in next section).

This sync source does not implement methods such as addItem, updateItem, deleteItem and getItemContent which are left to derived classes, but it simplifies the writing of new classes as it provides the methods to fetch items from the sync source.

2.2.5. ChangesTracker

This is an interface describing a tracker of changes in a source. The interface is very general and allows very different implementations. In particular there are two basic different ways of implementing a tracker:

- based on fingerprints or items properties
- based on modification listeners

The first type is built around the idea that a snapshot of the data store is taken at the last sync and it is compared with the current status. This allows the detection of new, updated and removed items. This approach is always possible, even though it is not very efficient as it requires a complete scan of the data store.

The second type of tracker is built around the idea that the underlying platform notifies an application when a change is made to the data store. Many applications offer this possibility that leads to high efficient data sources.

The two different types of tracker works in a very different way and finding a common interface to describe both implementations required to be very generic. The reason is mostly related to when changes are discovered. A fingerprint based implementation needs to compare the old and the current status to compute modifications. Once this done, any other change to the data store is not caputred until the next synchronization. A tracker based on listeners can behave differently as it can detect changes made after the sync started.

The interface has two sets of methods:

- methods to retrieve list of changes. Changes are returned as enumerations containing item's keys
- methods to inform the tracker about what is happening in the sync

It is interesting to observe the behavior of the tracker related to time. Figure 18 shows a temporal diagram. Let “b” be the time when the sync source informs the tracker that it is going to ask for changes soon. At this time the source must guarantee that all changed happened before b and after the previous synchronization will be detected with the next calls to the method that retrieve list of changes. Calling the “begin” has the function to set a point in time and this guarantees the enunciated property.

Now let “c1” be the time when a new item is added to the store and “g” the time when the sync source asks for the new items. The interface guarantees that all items added before b are detected in g, but the item added in “c1” can be part of this set or not. No behavior is enforced. A fingerprint tracker is likely to ignore it, while a listener based one will probably include it. Both behaviors are acceptable,

When getNewItems is invoked, the method returns the set of changed items and this set is frozen at time “g”. Changes made afterward (e.g. in c2) are not part of the changeset. This is true for any tracker and an expected behavior of the tracker.

The next interesting point in time is “s” when the source receives the status of an item sent to the server. If the item was exchanged correctly, then the tracker can remove it from its lists. There is just one exception to this rule, that must be taken into account. Suppose an item was detected as updated in the current changeset and then it is updated again. When a successful status is reported the item shall not be removed from the list of changes to be reported at the next sync. The interface allows this case to be handled for both fingerprint and listener trackers.

Time “e” is when the sync ends and “b” begins again. The changeset in b will contain the item added in c2 and possibly (depending on the implementation) the item added in c1.

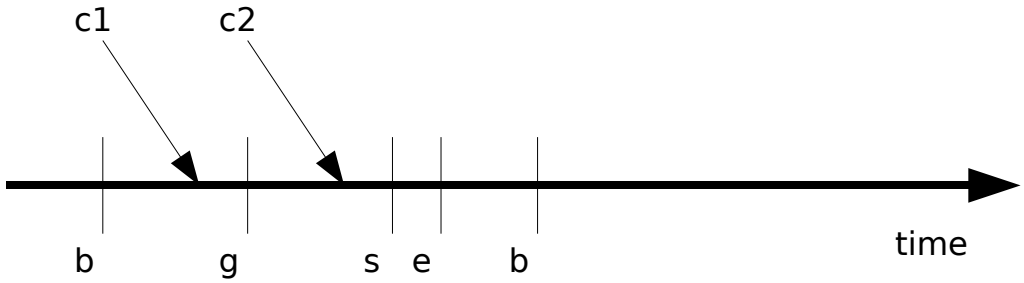


Figure 2.3: Changes tracker behavior in time

2.2.6. CacheTracker

This is a default implementation of ChangesTracker. This implementation is based on fingerprint computed with MD5. The CacheTracker stores a fingerprint for each item in the store. When the begin is invoked it computes the fingerprint for all the items currently in the store and detects all the changes. The methods that returns the list of changes will simply return what was computed by the begin method.

The behavior of the setItemStatus (invoked to notify the status of a change is interesting). If a command was successfully applied on the server, then the source status is updated. Suppose an “add” command was sent to the server and the item was correctly added to the server. In this case the setItemStatus will add this item into the cache so that it won't reported as a new item at the next sync.

The implementation uses MD5 to compute fingerprints. If a client prefers a different fingerprint computation it can derive the class and redefine the method that computes the fingerprint for an item.

The cache is stored in a StringsKeyValueStore which is an abstract class defined in the common APIs. This class allows a string map to be persisted and reloaded. JME does not have the concept of Serializable and therefore custom implementations are necessary.

2.2.7. SyncConfig

SyncConfig is a container class that groups the configuration information needed by the SyncManager.

The properties of SyncConfig are described below, for a complete description of the fields, please refer to the Javadoc.

Property	***Description***
syncURL	The syncURL value. If the URL does not start with http:// (or HTTP://) or https:// (or HTTPS://), http:// is prepended to the given string.
username	The username to use for client authentication
password	The password to use for client authentication
userAgent	The user agent string, will be sent in the HTTP header to identify the client on server side. It should be a short description with the

Property	*Description*
	client name plus its version.
deviceConfig	The device settings, used to build the DevInf tag.
devInfHash	This is a hash value generated from all properties that are used for the <DevInf> element, plus the syncURL property from SyncConfig. Initial value = “0”. TODO: check this on J2ME

Notes:

'userAgent' value should be specified by the client. If the property is left empty, the user agent will have the form: “Profile/<MID profile> Configuration/[CLDC 1.x]”.

'devInfHash' property is used to verify if any DevInf element (or syncURL) has changed since last sync; in that case the devInf is sent to the server (see next paragraph and Client Capabilities Handling chapter).

2.2.8. DeviceConfig

The DeviceConfig class is a container for properties related to the device and to the client application. Most of DeviceConfig properties are used to generate the <DevInf> element for client capabilities (see Client Capabilities Handling chapter). DeviceConfig properties are described below.

Property	*Description*
verDTD	Specifies the major and minor version identifier of the Device Information DTD used in the representation of the Device Information. The value MUST be “1.2”. This property is mandatory.
man	Specifies the name of the manufacturer of the device. This property is optional.
mod	Specifies the model name or model number of the device. This property is optional.
oem	Specifies the OEM (Original Equipment Manufacturer) of the device. This property is optional.
fwv	Specifies the firmware version of the device. This property is optional.
swv	Specifies the software version of the device. This property is optional.
hwv	Specifies the hardware version of the device. This property is optional.
devID	Specifies the identifier of the source synchronization device. The content information MUST specify a theoretically, globally unique identifier. This property is mandatory.
devType	Specifies the type of the source synchronization device. Type values for this element type can be e.g. “pager”, “handheld”, “pda”, “phone”, “smartphone”, “server”, “workstation”. Other values can also be specified. This property is mandatory.
dsV	Specifies the implemented DS version. This property is optional.
utc	Boolean. Specifies that the device supports UTC based time. If

Property	*Description*
	utc = TRUE, the server SHOULD send time in UTC format, else MUST send in local time. Client MAY send time in local or UTC format. Default value is TRUE.
loSupport	Boolean. Specifies that the device supports handling of large objects. Default value is FALSE.
nocSupport	Boolean. Specifies that the device supports number of changes. Default value is FALSE.
maxMsgSize	The maximum message size (byte) accepted for XML messages received from server. The same size is used for messages sent to the server.
maxObjSize	Specifies the maximum object size allowed by the device. Default value is 0 (no maxObjSize set).

Notes:
If there is no firmware/software/hardware version of the device available (fwv, swv, hwv), then their content information can also be a date, for example, 19980114 or 19990714T133000Z. Only hours, minutes and second MUST be specified in the time component.

2.2.9. SourceConfig

The class SourceConfig is a container for the configuration parameter of a SyncSource. It is used by BaseSyncSource. The most important SourceConfig properties are described in the table below.

Property	*Description*
version	A version number to handle software upgrades
encoding	Specifies how the content of an item should be encoded. The form of this parameter is a semi-colon separated list of formats that must be applied in sequence from the leftmost to the rightmost. For example, if format is “des;b64”, when the item will be output in the message, the content must be first transformed with the “des” encoder and than with the Base64 encoder.
syncMode	This is the current sync mode used by the client. The parameter MUST be one of the sync modes specified in property “syncModes”.
type	The mime type of the items sent by this source.
remoteUri	The remote URI of this source on the SyncML server.
lastAnchor	Long value that specifies the last timestamp for this source.
nextAnchor	Long value that specifies the last timestamp for this source.
ctCap	Specifies the content type capabilities for this SyncSource. This information describes which properties and which values the client can store. **Not implemented yet.**

2.3. Synchronization Events Notification

The synchronization process can be monitored by installing a proper listener that can be associated to a SyncSource. The model for monitoring the synchronization is different in J2ME (compared to J2SE [API-J2SE] and C++ API [API-CPP]).

A general SyncListener interface can be implemented or a utility BasicSyncListener can be extended to implement custom listener.

From a synchronization listener point of view, a synchronization is a sequence of events that happen in a certain order.

There are three different phases (as described in the SyncML synchronization process).

1. Connection set up
2. Modification exchange
3. Mapping

The phases are encapsulated into a sync session which starts with a startSession and is terminated by an endSession. Each phase is mandatory and the order cannot change.

The Modification exchange phase is the most interesting from the point of view of the SyncListener, because it can be further split into more sub events. The modifications are exchanged with a sequence of send/receive sesssions. Each time modifications are about to be sent to the server a startSending event is generated. Then we can have an arbitrary sequence of itemAddSent, itemReplaceSent and itemDeleteSent. At the end of the sending step a endSending event is generated. Similarly for the receiving phase.

The sequence of events can be described by the following grammar:

S -> startSession C U M endSession

C -> startConnecting endConnecting

U -> startSyncing (S R)+ endSyncing

M -> startMapping endMapping

S -> startSending (itemAddSent | itemUpdateSent | itemDeleteSent) endSending

R -> startReceiving (itemReceived | itemUpdated | itemDeleted | dataReceived)*
endReceiving

Unlike in other APIs, an event is not described by an EventInfo. This is just a pragmatic choice as today such an object would not contain useful information but it would require extra memory and CPU cycles. In the future we may need and introduce such an object to describe events.

The SyncManager notifies both the SyncSource and the SyncListener for some events (such as the server provides new items), but in some case it only notifies the SyncListener (for example on the session start and end). The two objects have a different vision of the synchronization process. The listener is notified about everything that happens (with adequate information that describe the events), while the SyncSource is only notified on events that require an action on its side.

Providing a SyncListener is a possibility, but not a requirement. If a simple client is not interested at observing the synchronization, it can avoid registering any listener.

The class diagram of Figure 2.4 shows the classes used for event notification.

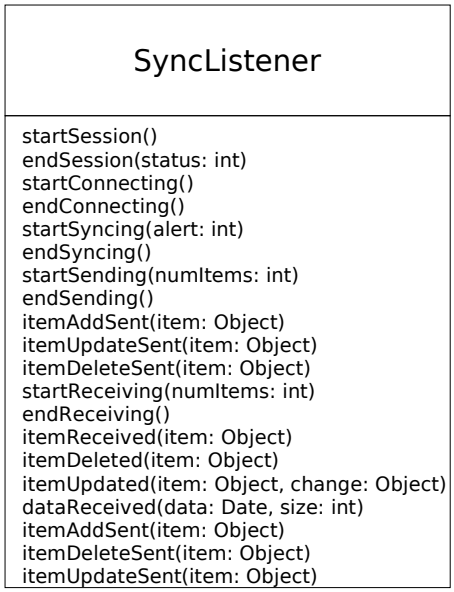


Figure 2.4: Listeners class diagram

2.4. Client Capabilities Handling

The SyncML specification ([OMA-DS], § 6.7) state:

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

Therefore, we have two scenarios:

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

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

For case 1, the SyncManager determines if it has to include client capabilities calculating the value of the devInfHash: if its value is changed it means that some DevInf parameter is changed, or the syncUrl property has been modified.

Note: the syncUrl parameter has been added to the generation of devInfHash, to ensure the client sends device informations when connecting to a different server.

However, a client application can decide to reset devInfHash when any other relevant configuration settings is changed.

In case 2, the SyncManager determines if it has to send the client capabilities inspecting the messages received by the server during initialization. If any contains a Get command, in the reply the SyncManager will send a Results command containing the DevInf build from the various configuration objects.

2.5. Filtering

Filtering is a new feature, introduced in the SyncML DS 1.2 specification, that allows a SyncML client to synchronize a database restricting the set of data returned by the server to the items that fall into a given filter. See [OMA-DS], paragraph 5.13, for details.

For the J2ME platform, a simplified interface to add filtering capabilities is defined, so that a client can send a filter to the server, if needed, without overhead for a client that does not need it.

The sync engine uses the SyncFilter interface, that provide a method to format an XML string representing the SyncML filter. The SyncSource interface provide a method getFilter() to obtain the filter defined for that SyncSource, if defined.

If a client need to filter data for a certain SyncSource, it has to implement the getFilter() method of the SyncSource, returning the needed filer. The class BaseSyncSource already implement the interface: a client extending the class has to set the desired filter calling setFilter() before starting the sync.

The SyncFilter interface

Method	***Description***
toSyncML()	Return a string containing the SyncML filter to apply for the session.

2.6. Large Objects Handling

The SyncML specification ([OMA-DS], § 6.7) states:

"*While synchronizing, object reception can be limited by two factors: the maximum message size the target device can receive (declared in <MaxMsgSize> tag), and the maximum object size the target device can receive (declared in <MaxObjSize> tag).*

This feature provides a means to synchronize an object whose size exceeds that which can be transmitted within one message (e.g. the maximum message size – declared in <MaxMsgSize> element – that the target device can receive). This is achieved by splitting the object into chunks that will each fit within one message and by sending them contiguously. The first chunk of data is sent with the overall size of the object and a <MoreData/> signaling that more chunks will be sent. Every subsequent chunk is sent with a <MoreData/> tag, except from the last one: the final chunk is sent with no <MoreData/> tag. The target device, having received the final chunk, has to re-construct the object and consequently acts as it had received it in one piece (e.g. apply the requested command). The appropriate status MUST then be sent to the originator. A command on a chunked object MUST implicitly be treated as atomic, i.e. the recipient can only commit the object once all chunks have been successfully received and reassembled."

In order to implement the support of Large Objects the SyncML API provides basic mechanisms that need to complemented/extended by client code in the sync source implementation. In other words the large object support is not transparent to clients that need to handle part of the work.

As stated above, large objects are split in chunks during a synchronization, both when sent and received. The SyncManager is responsible to get the chunks and to create a single SyncItem for each chunk. Such a SyncItem is marked as a 'chunk'. The SyncManager does

not keep in memory chunks but it rather discards them as soon as they have been processed by the proper SyncSource.

The SyncSource is responsible to manage the chunks, reassembling the data of the different sync items and is in charge also to maintain the entire large object in memory or serialize it depending on the needs of the running platform.

The SyncSource is also responsible for chopping a large object during sending. A large object is split into chunks that are provided to the SyncManager. Such items are marked as 'chunks' so that the SyncManager can handle them properly (adding the <*MoreData/*> tag in SyncML).

For java platforms with restrictions in terms of runtime memory, the SyncSource should store the large object in local storage as soon as the chunk is set in the item. In this way the whole large object won't be loaded in the heap space, but only the single chunk. This allows to implement the support of Large Objects also on java platforms with small heap space.

The idea behind this design is to avoid adding too much complexity to the SyncManager and to give maximum freedom to each implementation. One extreme is no large object support. In such a case there is almost no complexity added by the large object support in the APIs (and in the client). The drawback of this approach is that each sync source willing to handle large objects must do part of the work. This leads to some code duplication, but not much. And on top of this each source may decide to handle large objects in a different way.

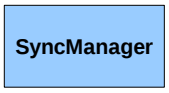
2.6.1. Receiving large objects

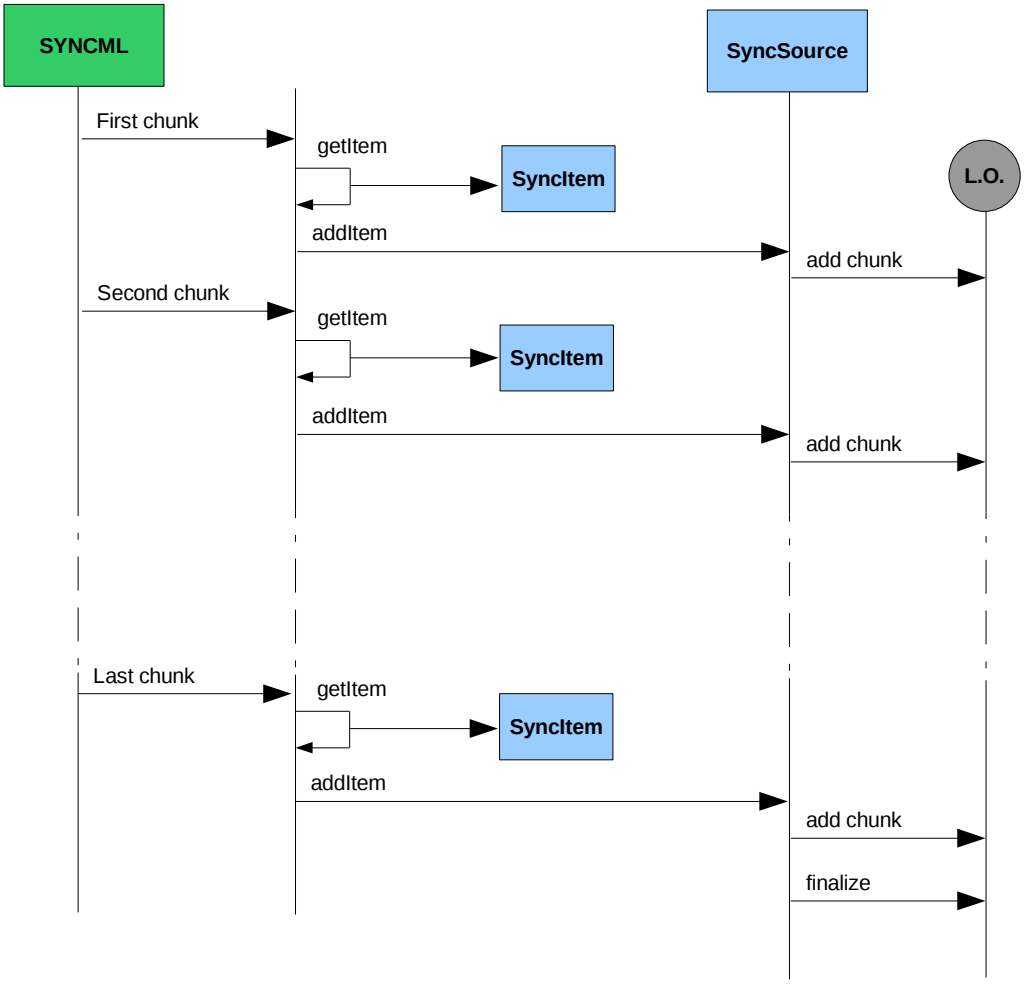
The SyncManager is able to check if a new item is a chunk by checking the <*MoreData/*> tag in SyncML and in this case it sets the *moreData* property of the SyncItem. The item is passed to the SyncSource and then discarded by the SyncManager. A SyncSource may decide to keep each chunk in memory or to stream them somewhere. No behavior is enforced by design.

On the last chunk of a large object the SyncManager does not set the SyncItem moreData property and this allows the SyncSource to recognize this is the last chunk and to finalize the large object.

Only after the end of the large object synchronization the SyncSource is able to create the large object client representation (i.e. File, Contact Photo, Message Attachment) converting the reassembled data chunks in the specific format.

The following picture shows the sequence diagram for the large objects client handling in case of new Item from Server.




SYNCML
SyncSource
First chunk
getItem
SyncItem
L.O.
addItem
add chunk
Second chunk
getItem
SyncItem
addItem
add chunk
Last chunk
getItem
SyncItem
addItem
add chunk
finalize

One interesting aspect is related to data encoding. The SyncManager is responsible for b64 encoding/decoding (as of today b64 is the only supported encoding). This topic is covered in section [2.6.3].

2.6.2. Sending large objects

Items being sent to the server must be split by the SyncSource which must guarantee the item size does not exceeds the *maxMsgSize* and the *maxObjSize*. If an item is too big, then it must be split in chunks. The SyncSource creates a SyncItem for each chunk and set the *hasMoreData* property (for all items but the last one).

The SyncManager generates the proper SyncML based on the given sync item. Since the protocol requires that the first chunk of a large object contains a metadata indicating the whole item size, SyncItem has a property to specify the item size. This property is only used for large objects and a sync source is only required to fill it for large object items.

A sync source has no knowledge of how many bytes are available for the item currently being processed. For example when the *getNextNewItem* is invoked, there is no information of how many bytes are available in the SyncML message under preparation. The SyncSource can make a simplifying assumption here, based on how the SyncManager works. If the current item fits (alone) in a SyncML message, then it does not need to be split. This is OK because if the SyncManager fetches an item that does not fit in the current message, then it terminates such message and then starts a new one. The only requirement is that the sync source takes into account the encoding. When checking the size of an item, the encoded size must be used.

The SyncManager is responsible for items encoding and there must be some sort of agreement between the SyncManager and the SyncSource to allow encoding of chunks. This aspect is described in section 2.6.3. But it is important that

2.6.3. Encoding and legacy support

The design described so far has a couple of shortcomings:

1. it does not address how encoding/decoding is performed for large objects
2. it does specify how existing sync sources will work in the new context

This section provides answers to the above issues.

As per existing implementation of the SyncManager, this component is responsible for performing items encoding/decoding. Since the SyncManager does not have visibility on entire large objects, it must be able to perform encoding/decoding on single chunks. This is

possible only for some encodings but the SyncManager is limited to base 64 and in this case we can guarantee this property. Therefore the design is done assuming that the SyncManager handles only encodings that can be performed on individual chunks.

The encoding issue is slightly different for sent and received items. For items being sent a SyncSource is required to generate chunks that can be encoded in chunks. For example in b64 the item's size must be amultiple of 16 bytes, so that no padding is required during encoding. This property guarantees that the SyncManager can simply encode each chunk and the result is equivalent to encoding the whole item.

For items being received we cannot guarantee the item has been chopped in chunks whose size is multiple of 16. Therefore each item may contain padding. One way of handling this issue would be to leave the problem to the *SyncSource*, but this does not work well, especially when legacy is taken into account (see below). Therefore a different approach must be taken. The idea is to have a class *SourceLOHandler* that works between the *SyncManager* and the *SyncSource* and provides some basic functionalities when it comes to items received from the server. One duty is to handle encoded items. When items are received the *SyncManager* decodes them if the *SyncSource* requires it. Then the item is passed to the *SourceLOHandler* which checks if the chunk has “padding”. In such a case the item is truncated at its maximum size multiple of 16. The remaining bytes are stored and kept to be put in front of the next chunk. This way a *SyncSource* can simply take all the chunks and concatenate them to form the large object. Isolating this functionality into a different object is crucial to keep the *SyncManager* as simple as possible.

The *SourceLOHandler* has another important role, related to backward compatibility. *SyncSource* is a pure interface and we don't want to change all possible implementations of this interface. Or at least we want to minimize the impact on existing implementations. Supporting large object is a device capability and not a sync source property. If a device specifies its capabilities specifying LO support, then it must be ready to receive large objects on any sync source. But this would require changing any existing implementation. To avoid this problem we rely on *SourceLOHandler*. New and updated items received from the server are routed to the *SourceLOHandler* (there exists one instance per sync source). The handler checks if the corresponding source supports large objects and if it does not then performs reassembling in memory. Once an item is complete it is passed to the sync source. If the corresponding sync source supports large objects, then the chunk is directly passed to the sync source.

3. References

[OMA-DS] SyncML Data Synchronization Protocol, version 1.2, Open Mobile Alliance
[COMMON] Funambol J2ME Common API Design Document, version 1.0, Funambol Inc.
[API-J2SE] Funambol Java API J2SE Developer Guide, version 1.0, Funambol Inc.
[API-CPP] Funambol 3.0 Client API C++ Design Document, version 1.0, Funambol Inc.