FUNAMBOL mobile open source

JavaME Common API Design Document

December 2007

Changes History

Date	*Author*	*Changes*
2006.10	Ivano Brogonzoli	Initial Draft
2006.11.13	Stefano Fornari	Minor changes due to a first review
2006.11.14	Ivano Brogonzoli	Added method description for com.funambol.uti and com. funambol.storage classes Added new classes documentation and method specifications
2007.05.30	Ivano Brogonzoli	Added new classes documentation and diagrams. New Class Documentation organization into the three main packages.
2007.06.20	Andrea Gazzaniga	Reviewed version
2007.08.21	Marco Garatti	Added SyncListener description and updated the ThreadPool description.
2007.12.07	Edoardo Schepis	Added SocketAppender details Added paragraph for Server Alerted Sync and OTA Config Provisioning
2007.12.08	Edoardo Schepis	Review of: SAN and OTAConfig paragraphs Added CTPService paragraph
2007.12.14	Marco Garatti	Added some details on the CTPService implementation
2008.09.22	Marco Garatti	Added UpdaterManager description
2008.10.13	Ivano Brogonzoli	Updated Persistent storage management description
2008.10.15	Ivano Brogonzoli	Updated CTP service with classes description
2008.11.12	Ivano Brogonzoli	Connection Framework description

Table of Contents

1. Overview

1.1. Scope

This document describes the Funambol JavaME Common API library, which purpose is giving support and providing basic functionalities to all the other components developed for the J2ME platform. These functionalities are: a persistent data storage framework, a logging framework, a framework to read different streams of byte, a thread monitoring system and a set of classes useful for data encoding and string manipulation. This library may grow in future, as new functions will be seen as common to different J2ME components.

1.2. Document Conventions

All diagrams in this document follow the conventions described in the paragraphs 1.2.1 and 1.2.2.

1.2.1. 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 methods or functions 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.

1.2.2. Class Diagrams

In general for each class only the main methods are described, but all diagrams follow these rules:

- Each class is represented as a box;
- Data members and methods are separated by an horizontal line;
- Plain titles represent classes, italicized titles represent interface (abstract classes);
- **+** next to a method or data member name means “public”;
- **-** next to a method or data member name means “private”;
- ***** next to a method or data member name means “protected”;
- **>** next to a data member name means it is a property with get/set accessors;
- Inheritance is represented by an arrow pointing to the base class;
- Italicized methods names represent abstract method.
- Connecting lines without arrow termination (connections) represent relation between classes or interfaces and the optional number written over a connection – near a box - represents the “How many” relation between the two entities.

1.3. The Funambol J2ME Common API

The J2ME Common APIs are structured into the five packages:

- Storage
- Util
- Tools
- Updater
- Push

Each package is briefly described in the following paragraphs.

1.3.1. com.funambol.storage package

The J2ME Storage package provide some classes to make the usage of the JavaME RMS easier.

The RMS has a very simple interface: you can only store a byte array in a record indexed by number. The classes in this package allows to store an instance of an object that implements the Serializable interface, thus providing a common way to store complex object in a record, and to access a record by name instead of the positional index, without high performace overhead.

The key classes to make this are:

- Serializable interface: gives a common interface for objects that are able to write their content to a stream, and to read it back
- ObjectStore: store and retrieve Serializable objects to/from the RMS, accessing them with the record index.
- NamedObjectStore: uses an index to access Objects in the RMS by name instead of by record number.

1.3.2. com.funambol.util package

The com.funambol.util package provides a bunch of utility classes for different purposes. This is the outline of them:

- Logging framework: represents a useful instrument to take trace of the operations executed by a program running in memory and to store them into device's persistent storage.
- StreamReader framework: a set of classes useful to read data from different stream of bytes;
- Thread Monitoring System: allows to track the number of threads started by an application, and to queue them if the limit of parallel threads is reached. On JavaME the number of threads guaranteed by the specification is only 5.
- Observable/Observer/Stoppable: a set of interfaces to implement a common java pattern in a Model/View/Controller architecture.
- String Utility set: a collection of utility classes useful to accomplish common operations of logging, manipulating and encoding strings (StringTools, Base64, QuotedPrintable). A class ChunkedString, that allows to work on substrings without really allocating new memory, is also provided.
- Connection framework: a Singleton pattern implementation to manage all of the connections provided by the J2ME CLDC Connector.

1.3.3. com.funambol.tools package

This package only refers to one class that is a utility used according to the Logging framework in order to display log entries to the user.

1.3.4. com.funambol.push package

This package contains the client push framework. Push is aimed at notifying clients of new events from a server. This package provides push mechanisms that can be coupled with Funambol push server. A client can instantiate a push manager and be notified for incoming events.

1.3.5. com.funambol.updater package

This package contains a component that can be used to check for new version of the application. The component can be coupled to a Funambol update server.

2. Functional description

2.1. Funambol J2ME Storage

The package com.funambol.common.storage is responsible of storing and retrieving user's data. The most important key concept is Serialization that make possible to store entire object data types on the device using the interface com.funambol.storage.Serializable. In particular the com.funambol.storage package is intended to be a wrapper framework between the concept of device's RecordStore in J2ME CLDC and the classes of ObjectStore: in this way particular attention has to be given to the interfaces shown in picture 2.1; ObjectStoreListener, ObjectComparator, ObjectFilter and ObjectEnumeration are entities that make possible to treat an object into an ObjectStore of the Funambol common API as a record in the RecordStore of the CLDC specification or a Persistable object (described into the RIM API for Blackberry). For more details about the behavior of the following classes refer directly to the Javadoc of both the mentioned specification.

An important point is that in order to extend the support of this package to Blackberry devices there is a big storage limitation using the CLDC RecordStore class: Blackberry devices cannot have RecordStore bigger than 64 Kb. All this said and due to the fact that RIM show a particular interface (net.rim.java.util.Persistable) to be used in order to persist data (bigger than 64 Kb) into the device memory, it has been necessary to generalize the concept of RecordStore into the com.funambol.storage.AbstractRecordStore class and then implement the devices' specific data managers that are com.funambol.storage.RmsRecordStoreWrapper and com.funambol.storage.BlackberryRecordStore. They defines the basic methods to store and retrieve data, both on a J2ME CLDC native or Blackberry powered devices.

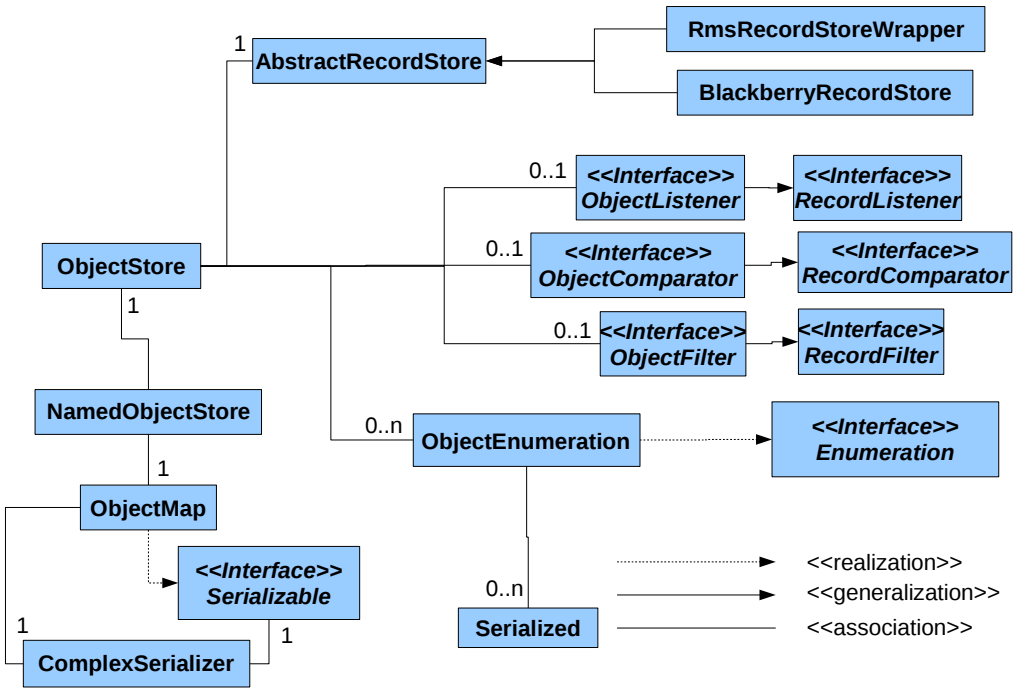


Figure 2.1: Funambol J2ME Storage class diagram

2.1.1. com.funambol.storage.Serializable

Serialization is the process to write an object into a byte stream; the object's information can be read back by a symmetric process called deserialization and the result is an object with the same content. In other words Serialization make possible to take a snapshot of an object into a certain state and read it back in form of stream of byte. Objects serialization is not implemented by the default Java CLDC libraries and this functionality is needed each time an application needs to store data on a persistent storage, or to send them over a byte stream. Funambol J2ME Common API's Serialization is based on a simple approach: an interface called Serializable must be implemented by all classes that wants to be serialized. This interface exposes two methods:

- serialize(DataOutputStream out): convert object into a byte stream, suitable to be stored in the device storage (i.e. RMS storage and flash memory cards), to be transferred over a socket connection (i.e. HTTP and Bluetooth), and so on;
- deserialize(DataInputStream in): read back the data from the input stream and build a copy of the original object (in the original state).

2.1.2. com.funambol.storage.ComplexSerializer

As J2ME CLDC specification doesn't include an object serializer framework, this class is useful to convert various type of objects into byte streams: in this way they will be stored and retrieved to/from the recordstore. Particular supported types are Vector and Hashtable and arrays of object. Finally this class provides methods to convert serialized objects into the original ones.

2.1.3. com.funambol.storage.ObjectStore

We said once an object implementing Serializable interface is serialized, its related stream of byte can be stored into device storage: ObjectStore is the class that manages this process. Each instance of ObjectStore can be bound to one RMS RecordStore using the open() and create() methods: the first can be used only on existent RecordStores, while the second creates the RecordStore if it doesn't exist or just open it if it has been previously created. Subsequent calls to open() has no overhead if done on the same RecordStore, while a call with a different RecordStore

name closes the current one and open the other. Finally, when an access to a not existent recordstore is made, a RecordStoreNotFound exception is thrown. Since J2ME does not support object finalization, the ObjectStore itself cannot close the RecordStore in the finalize() method, and the user has to call the close() method explicitly. The two methods store() and retrieve() are essentially responsible to call the serialize() and deserialize() method of the given object to set/get the related byte array, suitable to be stored. It may be noticed that there are 2 store methods:

- int store(Serializable object); it creates a new record on the opened RecordStore which index is automatically assigned by the device's RMS.
- int store(int index, Serializable object); this is useful to access a record of which index is known; i.e. to replace an updated ObjectMap – see 2.1.3.1 – the parameter index will be 1 and the object to be serialized will be the recordstore ObjectMap (which is located at the first record of each recordstore).

2.1.4. com.funambol.storage.ObjectEnumeration

This class implements Enumeration interface (from CLDC native specification) and represents a numbered list of a subset of the object contained into the ObjectStore; it's behavior is similar to the Enumeration, but it has a private fields that contains the number of objects (like the size for vectors) to be enumerated.

2.1.5. com.funambol.storage.Serialized

This entity is used when an ObjectEnumeration is required to be created from an ObjectStore. All the serializable object of that enumeration are associated to their related Record index on the RecordStore: that index becomes the object index of the serialized entity; this trick makes faster the process of search of an object into an ObjectStore.

2.1.6. com.funambol.storage.ObjectFilter

This interface extends CLDC RecordFilter interface and is useful to filter subset of objects contained into the referenced ObjectStore. In other words this is a wrapper to filter objects contained into an ObjectStore.

2.1.7. com.funambol.storage.ObjectComparator

This interface extends CLDC RecordComparator interface and it is ideal to create sorted ObjectEnumeration objects contained into the referenced ObjectStore. The sorting operation is made by comparing Object states and giving them a certain sort order determined by the sorting criteria.

2.1.8. com.funambol.storage.ObjectStoreListener

Extends the RecordListener interface: this interface take care about changes related to a particular ObjectStore() like add, remove and update operation on its objects. This object's behavior is similar to the one of “trigger” object in old DBMS.

2.1.9. com.funambol.storage.NamedObjectStore

This class is responsible to store and retrieve objects in the persistent storage accessing them by object name. The idea is to use give the developer the capability to access the record in the RMS using a symbolic name instead of the record index.

Since the search of a string in the records would have performance issues as the number of records grows, the implementation of this mechanism is achieved using the private class ObjectMap, which is basically an hastable of names stored in the first record of the RecordStore.

This mechanism speedup the access time, making the update slower because two writes are needed: one for the record and one for the index. Anyway, this overhead doesn't grow significantly when the number of records increases.

2.1.9.1. com.funambol.storage.NamedObjectStore.ObjectMap

This is a private class used by NamedObjectStore to implement the name index. The class uses an hashtable to maintain the link between a String name and an index in the RecordStore.

2.1.10. com.funambol.storage.DataAccessException

This class represents an Exception that can be thrown if something fails during reading/writing access to the recordstore. It can be useful to be thrown in case of an access to a non existent ObjectStore.

2.1.11. com.funambol.storage.AbstractRecordStore

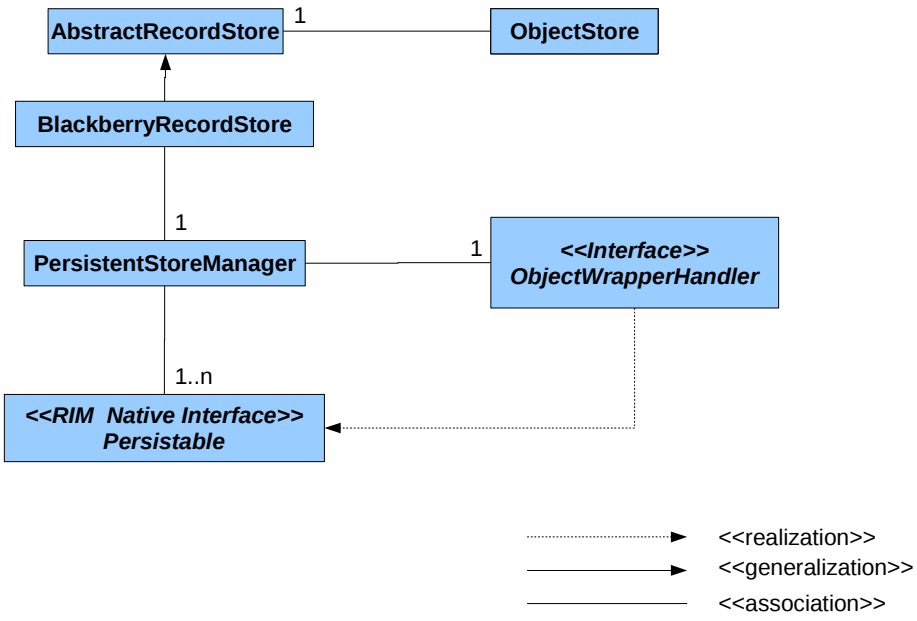
This is the storage wrapper class designed in order to manage different storage systems on different devices. Now it just wrap the concept of pure J2ME CLDC device and Blackberry devices. Now the implementation has been realized using the Sun Java WTK preprocessor in order to distinguish the building platform and use the correct storage manager, in fact, as per now, it has only the 2 implementations that are com.funambol.storage.RmsRecordstoreWrapper and com.funambol.storage.BlackBerryRecordStore.

2.1.12. com.funambol.storage.RmsRecordStoreWrapper

The implementation of AbstractRecordStore class that make possible the persistent data management on CLDC native devices.

2.1.13. com.funambol.storage.BlackberryRecordStore

The implementation of AbstractRecordStore class that make possible the persistent data management on Blackberry devices. This class uses the inner PersistentStoreManager and the ObjectWrapperHandler interface to manage Persistable objects storing and retrieving them on the device memory.



2.1.14. com.funambol.storage.BlackberryRecordEnumeration

Used by BlackberryRecordStore class to wrap the CLDC RecordEnumeration native class around Blackberry devices.

2.1.15. com.funambol.storage.ObjectWrapperHandler

Interface that must be implemented in order to give the information on how to manage Persistable object on the blackberry environment. Older implementation of the Persistable object Management made use of class called “ObjectWrapper” that was inner to the BlackberryRecordStore class: unfortunately it generated conflicts (Multiply class defined error) when more than one application using the same funambol storage API was installed on the same device. The actual implementation avoid the conflict problem for Blackberry devices but requires an high level implementation. Here's a simple example of how the class must be implemented for a correct storage usage on a Blackberry application:

```
public class DummyObjectWrapper implements Persistable {

    private Object object;
    public DummyObjectWrapper(Object o) {
        this.object = o;
    }

    public Object getObject() {
        return object;
    }

    public void setObject(Object object) {
        this.object = object;
    }
}

class DummyObjectWrapperHanlder implements ObjectWrapperHanlder {
    public Persistable createObjectWrapper() {
        return new DummyObjectWrapper();
    }

    public Object getObject(Persistable p) {
        DummyObjectWrapper ow = (DummyObjectWrapper) p;
        return ow.getObject();
    }
}
```

A call to the BlackberryRecordStore.init() method must be finally made in the high level client in order to close the circle:

```
public static void main(String[] args) {
    ...
    BlackberryRecordstore.init(new DummyObjectWrapperHandler());
    ...
}
```

2.2. Funambol J2ME Util

This is an utility package that provides the following basic but useful functionalities:

1. a light but flexible logging system;
2. an efficient I/O Stream reader framework under the Factory Design Pattern;
3. a set of classes to monitor threads running on the JVM when a MIDlet is running;
4. A couple of classes that realizes the Observer-Observable paradigm.
5. a set of tools for strings manipulation and encoding/decoding algorithm that can be useful for basic encryption.

For each group of classes class diagrams will be provided in the following paragraphs. For more details about the behavior of the following classes refer directly to the Javadoc.

2.2.1. The logging framework

This framework provides to the creation managing and deletion of Logging mechanism in every application. It is useful to create and manipulate logs by the related appenders, object instantiated in order to implement various logging strategies. This framework related class diagram is shown in figure 2.2.

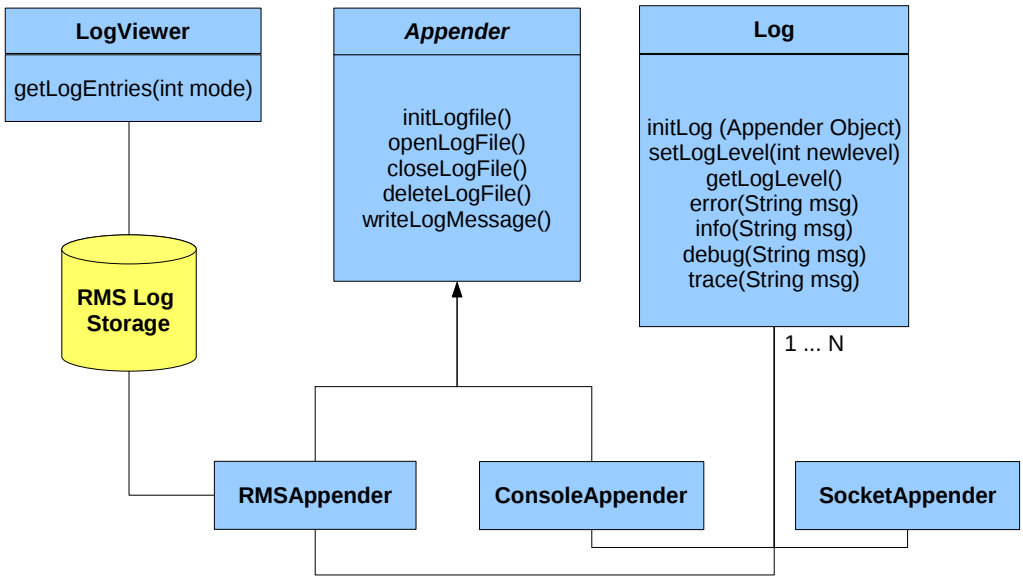


Figure 2.2: Logging class diagram

2.2.1.1. com.funambol.util.Log

This class is responsible for creating logs and deciding the logging levels.

Methods error(), info(), debug() and trace() can be used directly, just like, println(); three appenders (ConsoleAppender, RMSAppender, FileAppender) are created in order to write log entries on different logging media (the standard output, a RMS and a file respectively).

The default Appender is ConsoleAppender: Log.error() do exactly a System.out.println(). The default log level is INFO, so if you want to see all the messages you just have to change this value in an application that uses the static methods of 'Log' calling the method setLevel(), e.g. setLevel(Log.DEBUG). The different log levels should be used as follows:

- error(): an error occurred in the process and we want to always write this message
- info(): an average user should understand the progress of the application (in particular no development skills should be required); i.e. for a sync, he should see something like:

  Start sync

  Syncing contacts

  Sending contact: ID

  Receiving contact: ID

  .....

  End sync
- debug(): any information useful for debug; this information is addressed to developers;
- trace(): info to trace the flow of the program (Entering function()/Exiting function()).

2.2.1.2. com.funambol.util.Appender

A simple interface with logging constants and methods: it must be implemented by all Appender-type classes (2.2.3, 2.2.4, 2.2.5).

- void initLogFile(); initialize a new log file for the appropriate appender;
- void openLogFile(): open an existing log file;
- void closeLogFile(): close an existing log file;
- void deleteLogFile(): delete an existing log file;
- void writeLogMessage(String level, String msg): writes a message to on open log file with the specified level (0: error, 1:info, 2:debug, 3:trace);

2.2.1.3. com.funambol.util.ConsoleAppender

This Appender prints log messages to the standard output. It is the default Log mechanism and it is initialized by default if no other appender is specified. It's suitable to use this appender instead of System.out.println(msg) command. This class only implements writeLogMessage method.

2.2.1.4. com.funambol.util.RMSAppender

Write log file into using device's RMS. The log file generated is a recordstore that has one record per log message. In case of RecordStoreFullException the log file is auto-resized invoking the private method resizeLogStore(RecordStore dbStore, msgSize): dbStore is the RecordStore associated to the current log file, while msgSize is the size (in bytes) of the log message to be written before exception occurred. In case of RecordStoreFullException the RMS frees 10*msgSize bytes on the dbStore.

2.2.1.5. com.funambol.util.SocketAppender

This appender creates a socket connection to a server-side LogServer and writes the log using this socket. The SocketAppender is created passing the server to contact for logging and it can be passed to the initLog method of Log class to configure the logging via socket.

2.2.1.6. com.funambol.util.LogViewer

This will be useful to create a view of the the last Log from the RMS.

- public String[] getLogEntries(int mode): receives the parameter mode that iondicates wich type of log must be retrieved; actually the only mode is RMS_LOG defined with the final 0 value. This class already include JSR75 management capability.

2.2.2. The implementation of streaming readers

2.2.2.1. com.funambol.util.StreamReader

This interface simplifies the exchange of data stream between two objects. It represents the solution to the problem of reading different data stream from different data sources returning in the same time the same object type: all object that need to read a data stream must implement this interface. The only method shown by this interface is the following:

- byte[] readStream(InputStream is, int buffersize) throws IOException, StreamReaderException: it receives the InputStream object to be read and an integer that represents the buffer size to be read; return the byte array corresponding to the result of the reading operation. A valid example of the StreamReader usage is given into com.funambol.syncml.HttpTransportAgent when the data stream coming from server must be read.

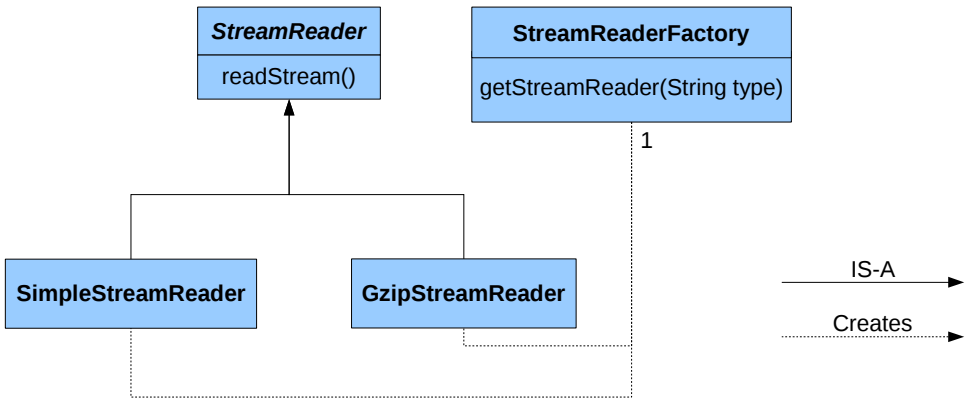


Figure 2.3: Funambol J2ME StreamReader Framework Class diagram

2.2.2.2. com.funambol.util.StreamReaderFactory

This class shows just a static method:

- public static StreamReader getStreamReader(String type): receives the string parameter that is related to the type of stream to be read and returns the correct reader object for that data stream.

2.2.2.3. com.funambol.util.SimpleStreamReader

This class implements StreamReader interface in order to read input stream based on UTF text data encoding. It receives a text input stream and translates it into the corresponding byte array.

2.2.2.4. com.funambol.util.GzipStreamReader

This class implements StreamReader interface in order to read stream based on Gzip encoding standards.

2.2.3. The Observer Pattern

The observer pattern (sometimes known as publish/subscribe) is useful to observe the state of an object in a given period of time. One or more object are registered to observe (even to listen to) the behavior of another object in the same time. Funambol J2ME Common API offer two classes to implement this pattern, Observable and Observer. The interaction of the two interfaces is shown in figure 2.3.

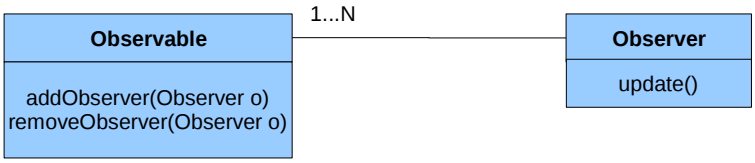


Figure 2.4: Funambol J2ME Common API Observer Design Pattern Class diagram

2.2.3.1. com.funambol.util.Observable

Every instance implementing this interface become observable objects: this means that somewhere in the code another class – maybe an Observer - can call them in order to receive information about the status of their related process.

- public boolean addObserver(Observer o): add an observer to this instance. This means that an observer can communicate with an observable through this interface and more observers can be registered to an observable object;
- public boolean removeObserver(Observer o): remove an observer from the list of observer related to this object.

2.2.3.2. com.funambol.util.Observer

As the name suggests this entity in complementary to Observable interface: it just has an update() method that refers to an Observable instance in order to understand its related status.

2.2.4. The thread monitoring and management system

This set of classes implements the thread life-cycle monitoring using

2.2.4.1. com.funambol.util.Queue

This entity is based on the java.util.Stack class and is a container for threads that work like a queue. It accepts Runnable object to be pushed at the end of the queue. All methods are synchronized.

- public synchronized void add(Object runnable): allow to put threads in the queue;
- public synchronized void remove(): allow to remove threads from the queue; lock if the task is not terminated;
- public synchronized Object removeNoWait(): removes the item immediately from the queue without locking if its task is not terminated.

As of today this class is not used by the API and is left mainly for backward compatibility.

2.2.4.2. com.funambol.util.ThreadPoolMonitor

This class allows a program to implement a mechanism to catch runtime exceptions that are not caught and rise up to the run method of a thread. If the thread is created by a ThreadPool (see below) and a runtime exception is thrown and not captured, then a ThreadPoolMonitor is used to handle such an exception. A ThreadPoolMonitor can be provided to the constructor of ThreadPool for the purpose of handling runtime exceptions. There is a default implementation that simply logs the exception, but clients can derive from the base class and implement more sophisticated exceptions handling mechanisms.

Main method:

- public void handleThrowable(Class clazz, Runnable runnable, Throwable throwable):

  when invoked receives the calling class, the related thread and the throwable as parameters: prints Log entries and the throwable related stack trace according to them;

2.2.4.3. com.funambol.util.ThreadPool

This class handles a pool of threads that can be monitored for runtime exceptions. The pool has also the concept of maximum size (in terms of concurrent alive threads) but this restriction is not enforced by ThreadPool.

ThreadPool allows clients to start a new thread in this pool. The client shall provide a Runnable object which is encapsulated into a thread created by the pool. This thread execution is guarded for possible runtime exceptions. If a runtime exception is thrown and not handled by the client runnable object, then the pool will handle it by the ThreadPoolMonitor which can be specified at ThreadPool construction time. If such a monitor is not provided, then a default ThreadPoolMonitor is used. The default implementation logs the exceptions.

A ThreadPool is always created with a maximum number of concurrent threads. At the moment the class does not really enforce any restriction on the number of alive threads. If the client fires more threads than the maximum number then an error is logged, but the pool will attempt to start a new thread anyway. If the JVM cannot cope with this request, then an exception will be thrown. From an implementation point of view it would be rather simple to wait for a thread to finish before starting the new one (thus enforcing the maximum number of alive threads in the pool). But from a semantic point of view it is unclear if this would be a good choice. The client could end up in a deadlock or the program may not function properly in these cases. In our J2ME mail client we set a maximum number of thread rather low (5) and we check that we never exceed this value. If we do, then we rather change the client code so that we do not reach the limit.

Main methods are:

- public Thread startThread(): starts a thread in the queue;
- public int getRunnableCount(): returns the number of running thread in the queue;

2.2.5. String utilities

This is not a framework but just a set of classes to encode/decode and manipulate Strings.

2.2.5.1. com.funambol.util.StringUtil

Many methods useful to manipulate strings. The use of this class is better explained in Javadoc.

2.2.5.2. com.funambol.util.Base64

It provides methods to encode/decode strings by Base 64 encoding.

2.2.5.3. com.funambol.util.ChunkedString

Given a String object this class receives its reference or a reference to a part of it. This class is useful when parsing big strings that include substrings to be parsed: in this case the substring() final method of String class would double the amount of memory to be occupied making a couple of the given string. This class allows to reference a given string and not to copy it, avoiding duplicate creation and waste of memory. Many methods have been implemented symmetrically to StringUtil. Pay great attention to the constructors.

2.2.5.4. com.funambol.util.DateUtil

A collection of methods useful to manipulate and convert Dates in different encoding types.

2.2.5.5. com.funambol.util.MailDateFormatter

A collection of methods to to convert date information contained in <code>Date</code> objects into RFC2822 and UTC ('Zulu') strings, and to build Date objects starting from string representations of dates in RFC2822 and UTC format.

2.2.5.6. com.funambol.util.QuotedPrintable

A class containing static methods to perform decoding from “quoted printable” content transfer encoding and to encode into.

2.2.5.7. com.funambol.util.XmlUtil

A set of methods useful to parse XML String types.

2.2.5.8. com.funambol.util.XmlException

This class extends the java.lang.Exception class. It is thrown when an XML exception occurs. Two constructors are provided. The first generates a generic XML Exception, while the second allow to insert a specific exception message

2.2.5.9. com.funambol.util.Entities

This Class provides some methods to escape/unescape characters according to XML specifications. All new Entities are mapped on two Hashtable: one is used to show store the entity name and value in memory, while the second is useful to retrieve the Entity name, given the Entity value.

- String escape(String str): Escape special characters in a given string;
- String unescape(String str): Unescape special characters in a given string.

2.2.6. com.funambol.util.CodedException

In JavaME, a lot of Exceptions classes causes an overhead often not acceptable on more contrained devices. To overcome this, but also allowing a good exception handling design, the CodedException class has been introduced.

The idea is to have one Exception class with a numeric code along with the String message. The catcher can then process the code to take the proper action.

The CodedException defines some basic codes, but developers can derive from it and add other application specific codes, paying attention to not overlap the definitions.

This class extends RuntimeException exception, to allow it to be catched from another thread (see also ThreadPoolMonitor).

- public CodedException(int code, String msg): the constructor receives the code of the exception and the message;
- public int getCode(): returns the code of this exception;

2.2.7. The Connection framework

This framework is a facilitator for all of the possible connections requests. Opening a connection means not only to open a socket or an http channel to the network, but the concept is wider: it means open one of the connection provided by J2ME CLDC javax.microedition.io.Connector class. As per now this framework is only able to manage the following kind of connections:

- javax.microedition.io.MessageConnection;
- javax.microedition.io.SocketConnection;
- javax.microedition.io.HttpConnection;
- javax.microedition.io.HttpsConnection;

The implementation is centered on the ConnectionManager class that a Singleton pattern realization. When a service or a Transport agent requires one of the connection listed above this class can manage the open action with the call to Connector.open(String url) method. Also a listener interface (com.funambol.util.ConnectionListener) for the connection manager is provided in order to understand the connection status and the class com.funambol.util.BasicConnectionListener represents its simplest implementation. Here follows the class diagram for the standard Connection framework implementation.

(Picture)

This kind of manager is very useful on devices that require a little step of configuration in order the connection to be opened. Devices like that are the Blackberry family. In such these devices the connection requires some optional parameters on the url in order to make the device able to request for the returned connection. In the Blackberry implementation the following classes were added in order to provide the connection configuration:

- com.funambol.util.BlackberryConfiguration;
- com.funambol.util.ConnectionConfig;
- com.funambol.util.WapGateway;
- com funambol.util.BlackberryUtils;

2.2.7.1. com.funambol.util.ConnectionManager

The core class implemented into the device dependent directory. We have one for J2ME standard clients and it results in just a wrapper for the Connector.open(...) method call. For the blackberry device family it uses an array of suitable and valid BlackberryConfigurations in order to return a duitable url configuration (only for http, https and socket requests, while SMS Message connection requests are managed without additional parameters). The configuration priority is the following:

- Wifi if available (it is mandatory to have a blackberry OS >= 4.2.1 for this requirement);
- User TCP defined APN (The apn that the user can configure manually on his own device);
- WapGateway table (the client's predicted configuration to use the GPRS bearer)
- ServiceBook WAP configuration (The content of the Blackberry WAP/WAP2 Transport ServiceRecord entry).

This class has a ConnectionListener object associated. If it is not set externally defined it is assigned to an instance of the BasicConnectionListener.

2.2.7.2. com.funambol.util.ConnectionListener

This interface is defined to be the connection notifier object. Its method are clearly useful to know the connection status as they are:

- ConnectionOpened
- ConnectionClosed
- ConnectionConfigurationChanged
- RequestWritten
- ResponseReceived
- isConnectionConfigurationAllowed

See the javadoc to understand the above method usage.

2.2.7.3. com.funambol.util.BasicConnectionListener

It is an implementation of the com.funambol.util.ConnectionListener interface and it is used as default by the ConnectionManager class when no other Listener is set. The more interesting method is the one used to confirm that a connection configuration is allowed to be used: this means that another implementor could manage the case to request the user if she really wants to use the configuration suggested by the system. The basic implementation returns always true and allows the configuration usage.

2.2.7.4. com.funambol.util.BlackberryConfiguration

This class is owned by the Blackberry implementation only and it aims to return suitable configurations for http, https and socket connections type. It is a container for a configuration that is a set of:

- Url parameters;
- Permission;
- Description;

2.2.7.5. Com.funambol.util.BlackberryUtils

This is a utility class that wraps out some of the net.rim.api from RIM Blackberry. It mainly provide useful methods to request run-time system property such as The GPRS coverage and wifi network availability. It also provides useful methods to access the ServiceBook and retrieve ServiceRecords poperty in order to interact with the com.funambol.util.ConnectiongConfig and give suitable configurations.

2.2.7.6.

2.2.7.7. com.funambol.util.WapGateway

A container class that wrap billed APN from free ones.This is useful, because depending by the flat plan registered on the SIM, it could possible to have more than one suitable configuration for the APN where to send the request. This configuration actually knows that in some countries same APN provided by the service book are billed by the carrier, others are not, so it just wrap the service book entry not to use the billed APN. I.e.: in Italy Tim offers both the WAP and GPRS traffic flat plan, but the service book report only the wap entry in some cases, as “wap.tim.it”. That entry is changed into “ibox.tim.it” that is the predefined in order to use GPRS traffic on the most of devices.

2.2.7.8. com.funambol.util.ConnectionConfig

This is the class that provides the BlackberryConfiguration objects array to the ConnectionManager class. As per now 4 configurations are provided and they are:

- Wifi configuration: as per the blackberry implementation optional parameter have to be added to the url in order to use the wifi interface (the string: “;deviceside=true;interface=wifi”);
- TCP user defined configuration: the device will use the parameters provided by the user in the TCP settings field of the device.
- Custom client APN Gateway: As per now it refers just to the class WapGateway and it is defined for Italian and United states carriers.
- Service book WAP/WAP2 Transport APN entry: This configuration is exactly the one contained into the device's service book and it is defined as the WAP/WAP2 transport APN.

2.3. Tools package

This package has specifically been designed to accommodate MIDlet tools: actually only one class belongs to this package and it's a utility to view the log entries of the device.

2.3.1. com.funambol.tools.LogViewerMIDlet

This class must be included as a standalone MIDlet into the jad file specification of a Midlet Suite. If a the log mechanism described in paragraph has produced some entries into the device's storage, run this MIDlet after the execution of the principal MIDlet to see the Log entries displayed on the screen. This class is coupled with com.funambol.util.LogViewer class described in par. 2.2.5.

2.4. Push package

The push package is described in chapter 3.

2.5. Updater package

The Funambol update protocol is described in [FUN-UPD]. This package implements the client side part. The following diagram shows how classes are organized.

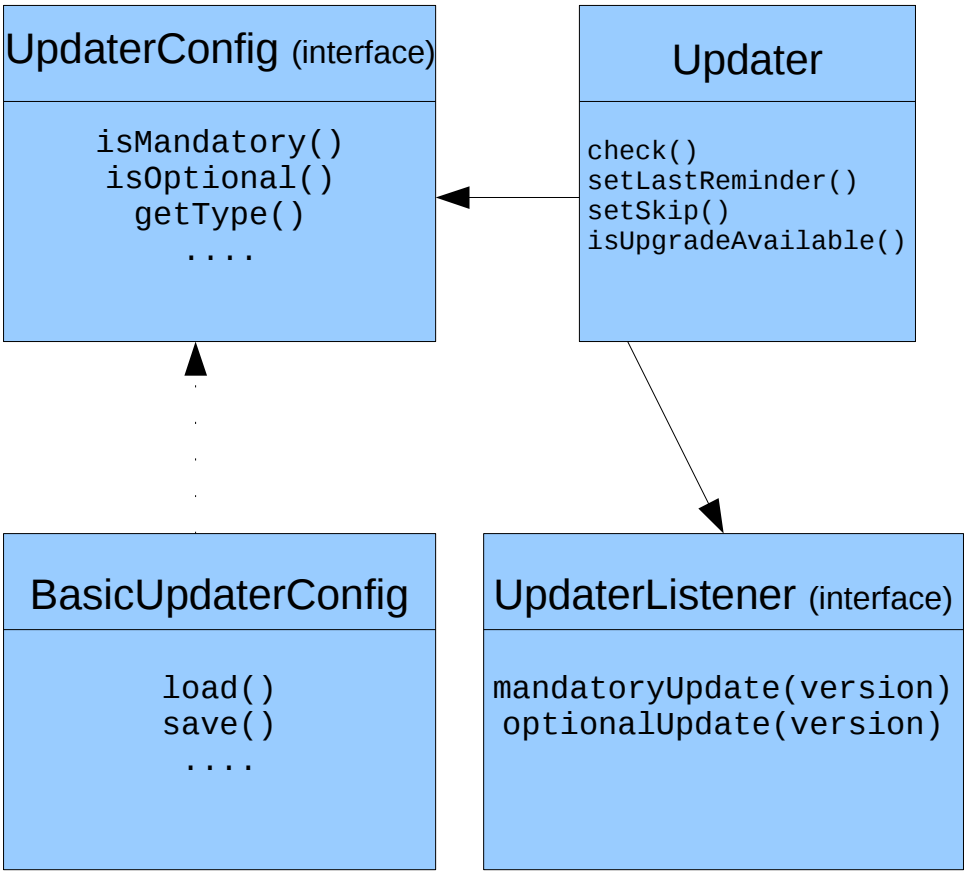
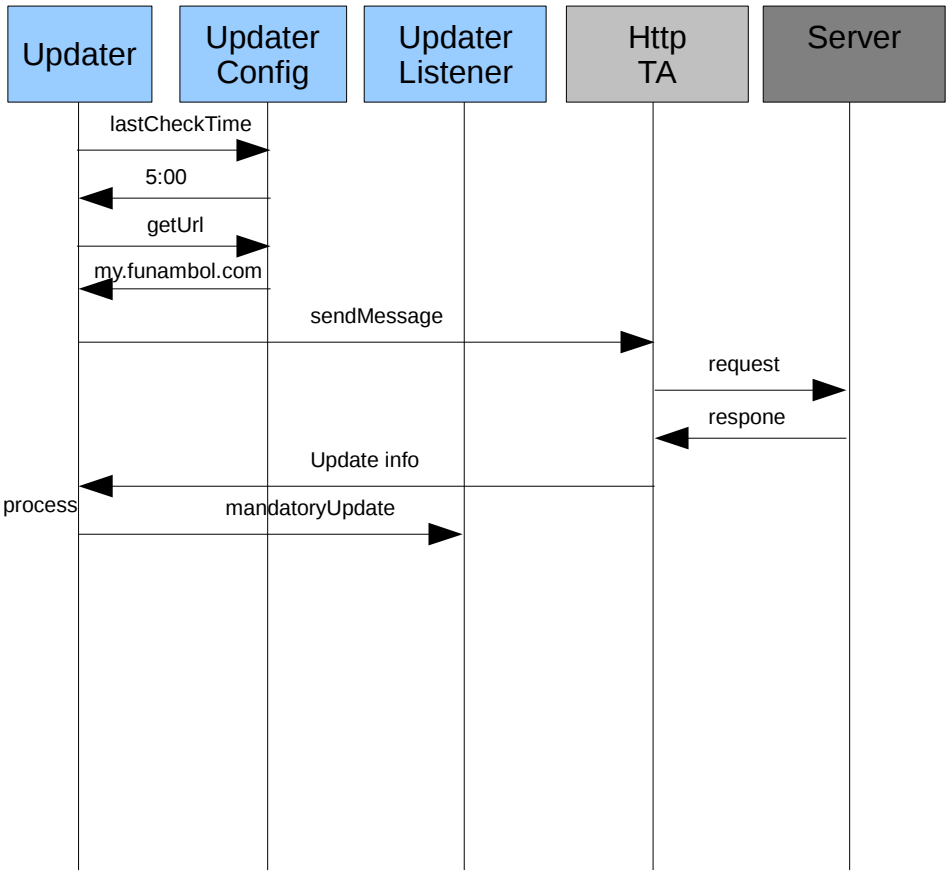


Illustration 1: Updater component organization

The Updater is the class in charge of interacting with the update server. This class has knowledge is instantiated with the name of the component and the current version. It then uses an UpdaterConfig to store its configuration and persist it across different sessions. Since the Funambol update service is http based, the Updater uses HttpTransportAgent to connect to the server and obtain the necessary information. When a new version is discovered, the Updater invokes client code via the UpdaterListener interface. A client must implement and register this interface to get notifications.

Figure ?? is a typical use-case. In this case the Updater is asked to check for new versions. The first operation performed is checking whether enough time has passed since the last check (there is an interval checking property in the configuration). In such a case the Updater retrieves extra information from the configuration, in particular the update server url. It then uses the HttpTransportAgent to query for updates. The response is processed and parsed and if the version available on the server is newer than the current one, the client is notified via the listener. It is up to the client to decide when the user shall be notified. It is important to note that the client must communicate back to the Updater the time at which the user got notified. This is necessary because there is a minimum interval between successive user notification. This property can be set in the UpdaterConfiguration.


Updater
Updater Config
Updater Listener
Http TA
Server
lastCheckTime
5:00
getUrl
my.funambol.com
sendMessage
request
respone
Update info
process
mandatoryUpdate

3. Server Alerted Sync – com.funambol.push package

Most devices can receive unsolicited requests to start a new synchronization from the server, sometimes referred to as "notifications". This can be done in many different ways, but on mobile phones the most common is using SMS binary messages. Another possibility available under some circumstances is by the means of UDP or TCP packets.

The notification message (also called *Package 0* or *PKG#0*) provides the means for a server to notify a client to start a SyncML session with the server. This mechanism is called *Server Alerted Sync.*

3.1. OTAService: SMS based push

In order to be able to be alerted by the server the client application must be able to listen to incoming notifications not only when it is active and running, but also when it isn't active. In J2ME, this is allowed by the *Push Registry,* offered by the device's Application Manager implemented as per Sun's MIDP 2.0 profile.

In order to achieve this goal, the PKG#0 must not be intercepted by the normal phone's messaging application, but intercepted and parsed by the J2ME application itself. This is achieved sending the notification to a particular port number for which the J2ME application is registered to listen to. When a client application is implemented by a MIDlet, this information needs in both cases to be set in the Java Application Descriptor file (JAD) like in the following example where the MIDlet wait for an SMS on port 50001.

```
MIDlet-1: WAPPushTester,,com.funambol.mailclient.push.WAPPushTester
MIDlet-Jar-Size: 2892
MIDlet-Jar-URL: WAPPushTester.jar
MIDlet-Name: WAPPushTester
Midlet Suite MIDlet-Permissions: javax.microedition.io.PushRegistry,
javax.wireless.messaging.sms.receive
MIDlet-Push-1: sms://:50001,com.funambol.mailclient.push.WAPPushTester,*
MIDlet-Vendor: Funambol, Inc
MIDlet-Version: 1.0.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
SMS-Port: 50001
```

The registration has to be static through adequate entries in the JAD file, because the application needs to be started even if it is not running by an incoming notification from the server.

As stated above, a client application can listen to incoming SMS messages. This is achieved by implementing the interface **javax.wireless.messaging.MessageListener** of the Wireless Messaging API (JSR-120).

The OTAService class implement the SMS push management system and is in charge of handling incoming SMS. It implements the MessageListener interface and the related *notifyIncomingMessage* method.

The methods provided by the OTAService class are:

- **notifyIncomingMessage**:called by the platform when an incoming message arrives over the connection (MessageConnection, also defined in the JSR-120) where the application

has registered a listener object (the application registers a *PushNotificationListener*, by invoking setMessageListener on the connection object).

- **startService** and **stopService**: called by the client to start/stop the service, therefore to open and close the connections to the SMS incoming messages.

Notice that the port number to which the SMS has to be sent by the server has to be used when getting the connection object (it is part of the string parameter passed to the open() method of the class Connector).

For example:

```
//initializing the network connection
smsPort = getAppProperty("SMS-Port");// in the JAD file
String smsConnection = "sms://:" + smsPort;
if (smsconn == null) {
    try {
        smsconn = (MessageConnection)Connector.open(smsConnection);
        smsconn.setMessageListener(this);
    } catch (IOException ioe) {
        //manage the exception
    }
}
```

The operations that are to be executed when the application receives an SMS notification are then set in the method **notifyIncomingMessage**.

For example (notice that all operations to be done, grouped in the **run()** method of the application, are managed as a separate thread):

```
public void notifyIncomingMessage(MessageConnection conn) {
    if (thread == null) {
        thread = new Thread(this);
        thread.start();
    } else {
        thread.run();
    }
}
```

3.2. CTPService: TCP/IP based push

The alternative for pushing notifications to JavaME handsets is to use the TCP/IP protocol.

A persistently opened TCP channel connects the client to the server, allowing the server to push notifications every time an update is available.

The Client TCP Push (CTP) protocol requires that the connection is initiated by the client opening a socket to the CTPServer. The client is also in charge of maintaining open and “alive” the connection using periodical heartbeats. The CTPServer registers the client for any further notification and keeps the connection open until the client closes it. Usually clients close the channel only when the application is closed by the user or in cases of network connection loss. In these cases the server switches to SMS push method for future notifications.

The CTPService implemented in the JavaME common API, allows the server to notify the client only when the client is up and running. In fact when the client is closed the CTP channel is closed and the client is not reachable.

For the details regarding CTP protocol and working mechanism please refer to [DSPD]

CTPService implements the following features:

- authentication to the CTPserver: it is implemented using MD5 authentication mode and **MD5** class
- management of lifecycle of the CTP connection and the related heartbeats through the **HeartBeatGenerator** inner class
- management of connection errors: using the **ConnectionTimer** inner class, the CTPService is able to monitor read/write network operations. In some cases network operations do not generate any exception even if the connection is broken. CTPService uses ConnectionTimer to monitor the time taken by network operations to complete. If these operations take longer than a programmable timeout, then the operations are terminated by closing the corresponding channel.
- management of the CTP messages: the **CTPMessage** inner class wraps the CTP messages exchanged with the server on CTP channel. When the server sends a message to notify the presence of new emails, the CTP service invokes the method *handleMessage* of the registered *PushListener*. This method is in charge of starting a new synchronization to pull in new messages.

The following diagram describes the lifecycle of CTP protocol implemented in the CTPService.

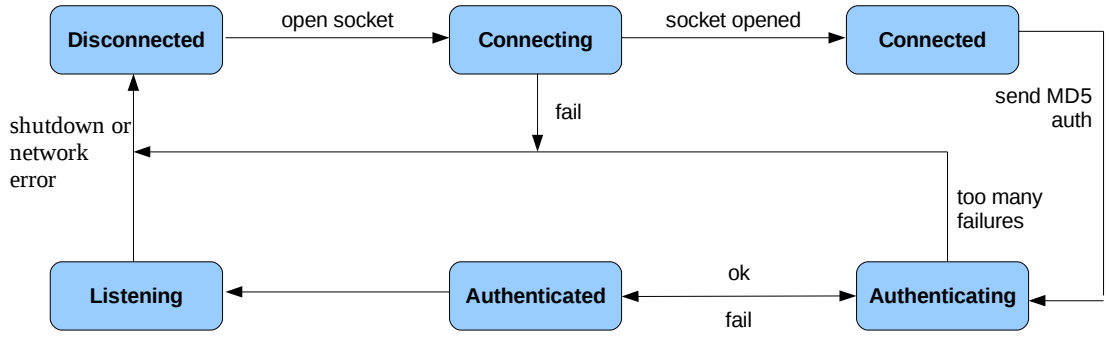


Figure 1: Lifecycle of CTP protocol implemented in CTPService class

3.2.1. com.funambol.push.CTPListener

An interface that provides methods to notify the changes in the CTP status. It is useful to keep trace of the CTP state changes.

3.2.2. com.funambol.push.CTPService

A class that shows all of the methods to start, stop and manage the CTPService behavior. It uses the CTPListener in order to notify its state's changes.

3.3. SAN Message Parsing

Any service registered for incoming SAN messages delegates the SAN message parsing to the **SANMessageParser** class, but different preprocessing is applied to the payload before it is actually parsed.

- **OTAService** gets the notification's binary payload in the form of an array of bytes contained in the SMS message. The OTA Service is also able to get SAN messages through a SMS text message and therefore it needs a specific logic to get the SMS type (text or binary) and accordingly parse the SAN Message.
  Notice that the first 3 bytes + the number of bytes indicated in the 3rd byte of the binary SMS message (actually always 3, also 3+3 = 6 bytes altogether) from the server are to be skipped before they are parsed, because they are part of the WAP Push header used to deliver the SMS.
  The sixth byte of this part of header carries the type of the message received, if SAN or OTA (Over The Air Configuration Provisioning, see below).
- **CTPService**, when receives a SAN Message, doesn't need to skip the first bytes because there's no WAP Push overhead (no SMS) and therefore it can directly pass the payload to the SANMessageParser class. In this case the payload is implemented with a CTPMessage object.

The format of the SAN Message content is defined by the Open Mobile Alliance (see [SAN]). Because this information is split over well determined sequences of bits that don't match the one-byte boundaries of the returned array's elements, the parser has to retrieve this information in a per-bit basis.

The class **SANMessageParser** provides a binary parser with the method **parseMessage**. At the end of the process, a **SANMessage** object is populated with the values needed by the client to start the synchronization, accessible through these keys: Digest, Version, UiMode, Initiator, SessionId, ServerId, NumberOfSync, syncInfo. For a detailed description of the syntax of the Notification Package see [SAN], §7. A helper class, **SyncInfo**, is also currently used.

4. Over The Air (OTA) Configuration Provisioning

The configuration of a client by the user on a mobile phone can be difficult, because of the limited keyboard and screen size. For this reason, many services for mobile devices provide a way to send the configuration data to the device, called **Over The Air Configuration**. This is done using one or more SMS sent by the server to the device, with data stored in a WBXML message.

To maximize the efficiency of transport of the config information needed by a client, the Funambol server is able to send these data also in a custom format described below.

The Funambol Common API provides a method to parse this format and to populate the objects SyncConfig and SourceConfig.

The simple protocol used is based on different sections, indicated by an initial byte, each of those are composed by a fixed number of strings, in a defined order.

Each string is composed by a lead byte containing the string length, and a sequence of bytes containing the characters. This means that each string cannot be greater than 127 characters, but, since the OTA config messages are tipically transferred ove SMS, this limit should not be a problem, but it must be taken into account by the server side UI to prevent the user to set parameters greater than this limit.

The section and fields are described by the following table:

Section ID (Type)	***Section Description***	***SECTION PARAMETERS***
1	SyncML	URL + User + Password
2	Mail	Remote URI + Visible Name + Mail Address
3	Contact	Remote URI + Format*
4	Calendar	Remote URI + Format*
5	Task	Remote URI + Format*
6	Note	Remote URI + (Local URI)
7	Briefcase	Remote URI + (Local URI)
*Format = S or V for SIF or Vcard format		

The message will have the following structure:

Section 1					**Section i**	**Section N**		
Type	***Length***	***Value***	***Length***	***Value***	***...***	***Type N***	***Length***	***Value***
First Section type	First Field Length	First Field Value	Second Field Length	Second Field Value	...	Nth Section Type	Nth Field Length	Nth Field Value

Finally, here's an example of configuration message:

1	28	http://www.funambol.net/sync	5	guest	5	guest	2	5	imail	7	John Doe	13	jdoe@mail.com

Interpretation in the client:

1: SyncML Section (Type);

28: URL field length;

http://www,funambol.net/sync: URL value;

5:User field Length;

guest: User value

5:Password field length

guest: Password value

2: Mail

5: Remote URI length

imail: Remote URI value

7: Visible name length

John Doe: Visible name value

13: Mail address length

jdoe@mail.com: Mail Address value

The OTA Configuration management has been implemented with the classes **OTAConfigMessage** and **OTAConfigMessageParser**.

In order to handle both SAN and OTAConfig messages with one service (OTAService), a client can registers a listener for incoming SMS on a specific port and uses a byte from the SMS header to get the information of which message has arrived.

5. Appendices

5.1. Appendix A – References

[JavaME-JSR] List of Standard JavaME JSRs: http://java.sun.com/javame/reference/apis.jsp#api;

[OMA-DS] SyncML Data Synchronization Protocol, version 1.2, Open Mobile Alliance

[API-J2SE] Funambol Java API J2SE Developer Guide, version 1.0, Funambol Inc.

[DSPD] Funambol DS Server Push Design, version, Funambol Inc.

[SAN] Open Mobile Alliance, SyncML Server Alerted Notification (http://www.openmobilealliance.org/release_program/docs/CopyrightClick.asp?pck=Common&file=v1_2-20050509-C/OMA-TS-SyncML_SAN-V1_2-20050509-C.pdf)

[FUN-UPD]