Sync4j Developer's Guide

July 28, 2003

Change History

Date	Author	Description	Rev #
July 28, 2003	Stefano Fornari	Initial revision	1.0
October 30, 2003	Stefano Fornari	Added section on message processing pipeline	1.3

Table of Contents

1. Introduction

This document is intended for developers who aim to develop synchronization services based on Sync4j 1.0.x.

1.1. Comments and Feedbacks

The Sync4j team wants to hear from you! Please submit your questions, comments, feedbacks or testimonials to sync4j-users@lists.sourceforge.net.

2. Data Synchronization

All mobile devices – handheld computers, mobile phones, pagers, laptops – need to synchronize their data with the server where the information is stored. This ability to access and update information on the fly is key to the pervasive nature of mobile computing. Yet, today, almost every device uses a different technology for performing data synchronization.

Data synchronization is helpful in respect to many areas:

- Propagating updates between a growing number of applications
- Overcome the limitations of mobile devices and wireless connections
- Maximizing user experience while minimizing data access latency
- Keeping scalability of the IT infrastructure in an environment where the number of devices (clients) and connections tends to increase considerably
- Understanding the requirements of mobile applications, providing the user experience that helps and it is not an obstacle for mobile tasks.

Data synchronization is the process of making two sets of data look identical (Figure 1). This involves many concepts, the most important are:

- ID handling
- Change detection
- Modification exchange
- Conflict detection
- Conflict resolution
- Slow and fast synchronization

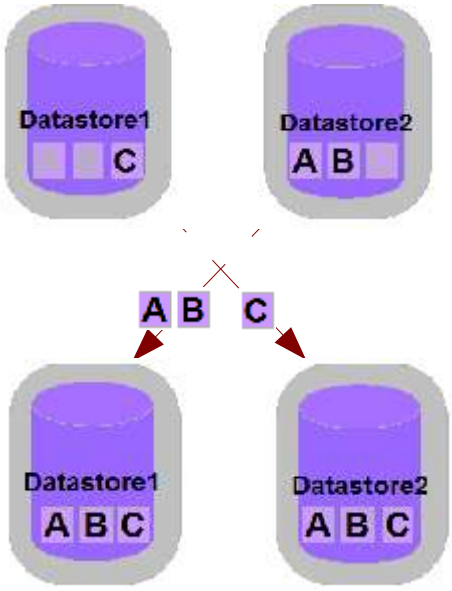

Figure 1 - Data synchronization process

2.1. Id Handling

At a first look, id handling seems a pretty straightforward process and of no interest. Instead, id handling is an important aspect of the synchronization process and it is not trivial. Each piece of data is usually uniquely identifiable by a subset of its content fields; for example, in the case of a contact entry, the concatenation of first name and last name uniquely selects an entry in your directory. In other cases, the id is represented by a particular field specifically introduced for that purpose. This may be the case, for example, of a Sales Force Automation mobile application, where an order is identified by an order number or reference. The way an item id is generated is not determinable a priori and it is application and device specific.
In an enterprise system, however, data is stored in a centralized database, shared by all users; each single item is known by the system with a unique global id. In same cases, two sets of data (i.e. the order on the client and the order on the server) represent the same information (*the* "order" made by

the customer) but they differ. What could be done to reconcile client and server ids in order to make the information consistent? Many approaches can be chosen:

- Clients and server agree on a *id scheme* (a convention on how to generate ids must be defined and used);
- Each client generates globally unique ids (GUIDs) and the server accepts client-generated ids;
- The server generates globally unique ids (GUIDs) and each client accepts those ids;
- Client and server generate their own ids and a mapping is kept between the two. Client side ids are called Local Unique IDentifiers (LUID) and server side ids are called Globale Unique IDentifiers (GUID). The mapping between local and global identifiers is referred as LUID-GUID mapping.

2.2. Change Detection

Change detection is the procedure of identifying which data is changed since a particular point in time (i.e. the last synchronization). This is usually achieved making use of additional information such as timestamps and state information. For example, a possible database enabled for an efficient change detection is the one depicted in Table 1.

ID	*first name*	*last name*	*telephone*	*state*	*last_update*
12	John	Doe	+1 650 5050403	N	2003-04-22 13:22
13	Mike	Smith	+1 469 4322045	D	2003-05-21 17:32
14	Vincent	Brown	+1 329 2662203	U	2003-05-21 17:29

Table 1 - A database enabled for efficient change detection

However, sometimes legacy databases do not provide the information needed to accomplish an efficient change detection. Therefore, the matter becomes more complicate and alternative methods must be adopted (for instance, based on content comparison).

2.3. Modification Exchange

A key component of a data synchronization infrastructure is the way modifications are exchanged between client and server. This involves the definition of a synchronization protocol that client and server have to use to initiate and carry on a synchronization session. In addition to the exchange modification method, a synchronization protocol must also define a set of supported modification commands. The minimal set of modification commands is represented by the following:

- Add
- Replace
- Delete

2.4. Conflict Detection

Let us suppose two users synchronize their local contact database with a central server in the morning, before going to the office. After syncing, they have exactly the same contacts on their PDAs. Let us now suppose that they change the telephone number of the same “John Doe” entry, but for some reason with a different number (maybe, one of the two made a mistake). What will happen when the next morning they will synchronize again? Which one of the two new version of the John Doe record should be taken and stored to the server? This condition is called a *conflict* and the server has the duty of identifying and resolving it.
The simplest way to do detect a conflict is by the means of a “synchronization matrix” (Table 2).

Database A → ↓ Database B	*New*	*Deleted*	*Updated*	*Synchronized/ Unchanged*	*Not Existing*
New	C	C	C	C	B
Deleted	C	X	C	D	X

Database A → ↓ Database B	New	Deleted	Updated	Synchronized/ Unchanged	Not Existing
Updated	C	C	C	B	B
Synchronized/Unchanged	C	D	A	=	B
Not Existing	A	X	A	A	X

Table 2 - The synchronization matrix

Because both users synchronize with the central database, we can consider what happens between the server database and one of the client databases at a time: let's call *Database A* the client database and *Database B* is the server database. The symbols in the synchronization matrix have the following meaning:

X : nothing to do
A : item A replaces item B
B : item B replaces item A
C : conflict
D : delete the item from the source(s) containing it

2.5. Conflict Resolution

Once a conflict arises and it is detected, a proper action must be taken. Different policies can be applied:

- User decides: the user is notified of the conflict condition and decides what to do; this strategy, like the following “Client wins” is a bit problematic in a server centric synchronization solution: each user may have the same right to modify an item and one users could not be able to decide whether his/her modification should win over the other ones.
- Client wins: the server silently replaces conflicting items with the ones sent by the client.
- Server wins: the client has to replace conflicting items with the ones from the server.
- Timestamp based: the last modified (in time) item wins
- Last/first in wins: the last/first arrived item wins
- Do not resolve

2.6. Slow and Fast Synchronization

There are many modes to carry on the synchronization process. The main distinction is between fast and slow synchronization. A *fast synchronization* involves only the items changed since the last synchronization between two devices. Of course, this is an optimized process that relies on the fact that, some time in the past, the devices where fully synchronized; this way, the state at the beginning of the sync operation is well known and sound. When this requisite is not true (because, for instance, the mobile device has been reset and has lost the timestamp of the last synchronization), a *slow synchronization* has to be performed. In this case, the client sends its entire database to the server, which compares it with its local database and returns to the client the modifications needed for it to be up to date again.
Either fast and slow synchronization modes can be performed in one of the following manners:

- Client to server: the server updates its database with client modifications, but sends no server-side modifications.
- Server to client: the client updates its database with server modifications, but sends no client-side modifications.
- Two-way: client and server exchange their modifications and both databases are updated accordingly.

3. The SyncML Initiative

With the many devices available today and the different applications data synchronization applies to, the need of a standard is evident. IT managers see the adoption of an industry standard as a way to protect their investments in IT infrastructure and devices. Even if applications or mobile devices will change in the future, if they speak the same *language*, servers and legacy systems will be only slightly impacted.
The de-facto standard for data synchronization is called SyncML (Synchronization Markup Language) which is now under the umbrella of the Open Mobile Alliance.

SyncML is defined as follows:

- SyncML is a new industry initiative to develop and promote a single, common data synchronization protocol that can be used industry-wide.
- SyncML is a specification for a common data synchronization framework and XML-based format for synchronizing data on networked devices.
- SyncML is a protocol for conveying data synchronization operations.

SyncML is targeted to personal and enterprise needs and it is application-agnostic: it defines how to establish, carry on and complete a data synchronization session and how to exchange data modifications and the commands to use. It does not specify, however, how to detect changes and conflicts or how conflicts should be resolved. This is one of the areas where SyncML client and server providers differentiate their offers.
SyncML has been designed to synchronize any type of data on different transport protocol (such as HTTP, WSP, OBEX, etc.); types of data may include:

- Common personal data formats, such as vCard for contact information, vCalendar and iCalendar for calendar, todo, and journal information
- Collaborative objects such as e-mail and network news
- Relational data
- XML (the Extensible Markup Language) and HTML documents
- Binary data, binary large objects, or “blobs”

To facilitate the adoption of the standard, SyncML initiative delivers:

- An architectural specification
- Two protocol specifications (SyncML representation protocol and SyncML synchronization protocol)
- Bindings to common transport protocols
- Interfaces for a common programming language
- An openly available prototype implementation of the protocol

4. Sync4j High-level Architecture

Sync4j is designed with modularity and flexibility in mind, being targeted to enterprise applications. The main modules that build up Sync4j are:

- The *Sync4j Engine*, which makes use of additional pluggable modules
- The *Transport Layer* module implements the transport specific binding of SyncML. In the case of the HTTP protocol, it is represented by a J2EE web module. Other transports can have specific implementation.
- The *SyncML* module is responsible for the encoding/decoding of SyncML messages, as specified by the representation specifications.
- The *Protocol* implements the SyncML synchronization protocol, which describes how SyncML messages are combined to represent a correct synchronization session.
- The *Services* module furnishes many horizontal services such as authentication, security, configuration, logging and so on.
- The *SyncSources* are the means Sync4j can integrate with external and legacy systems.

Sync4j is based on a rich programming framework that implements the most important functionalities and features that the different modules provide. Not all developers will have to deal with every module; however, in the following sections the framework is described in more detail with the purpose of helping the understanding of the inside aspects of Sync4j and driving the development of Sync4j extensions.

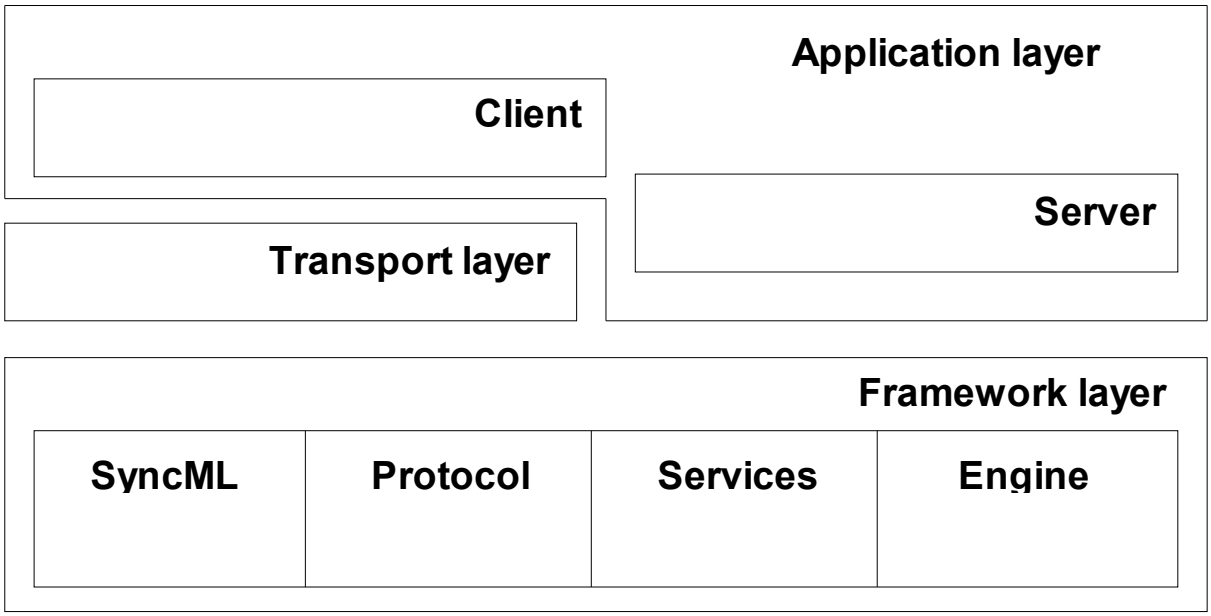

Figure 2 - Sync4j Framework architecture

4.1. Sync4j Framework

The Sync4j Framework architecture is conceptually divided in layers (Figure 2).

The bottom layer is a framework that implements and provides protocol implementation, horizontal services and the the synchronization engine interface on top of which the transport and application layers are developed. The application layer can be further divided in client and server, where *server* indicates the software that accepts and processes SyncML messages.

The server relays on the transport layer in order to receive messages delivered with different protocols such as HTTP, SMTP, OBEX, etc. In the current implementation of Sync4j the server is implemented as an EJB service deployable in a J2EE compliant application server.
Client applications take advantage of the services provided by the framework in order to code and decode SyncML messages and to send and receive SyncML messages on one of the supported transport protocol.

The framework includes many packages, the most important ones are:

- sync4j.framework.core;
- sync4j.framework.config;
- sync4j.framework.engine;
- sync4j.framework.logging;
- sync4j.framework.protocol;
- sync4j.framework.security;
- sync4j.framework.server.

sync4j.framework.core implements the block that in Figure 2 is called *SyncML* and groups the foundation classes used to represent a SyncML message. This module allows an easy translation of a XML stream into an objects tree, which is more manageable from a programming point of view. Vice versa, an object representing a message can be easily converted in the corresponding XML representation. The classes of the framework are responsible for checking that a given message is a valid SyncML message. Note that this validity check guarantees only that the XML structure can really represent a message, regardless of the context in which the message is processed. The scope of this check is to verify that the *representation* rules are all respected.
A SyncML communication is a sequence of correlated messages that must follow additional rules, dictated as well by the specification of the protocol. For instance, consider the following message:

```
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1028886155551</SessionID>
<MsgID>2</MsgID>
<Target>
<LocURI>URI:2002</LocURI>
</Target>
<Source>
<LocURI>http://www.sync4j.org/sync4j</LocURI>
</Source>
</SyncHdr>
</SyncML>
```

It is not a valid SyncML message in any context because it does not contain a <SyncBody> tag.

Consider the following instead:

```
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1028886155551</SessionID>
<MsgID>2</MsgID>
<Target>
<LocURI>URI:2002</LocURI>
</Target>
<Source>
<LocURI>http://www.sync4j.org/sync4j</LocURI>
</Source>
```

```
</SyncHdr>
<SyncBody>
<Status>
<CmdID>5</CmdID>
<MsgRef>1</MsgRef>
<CmdRef>3</CmdRef>
<Cmd>Sync</Cmd>
<TargetRef>db1</TargetRef>
<SourceRef>db1</SourceRef>
<Data>405</Data>
</Status>

<Add><CmdID>3</CmdID>
<NoResp/>
<Meta><Type xmlns='syncml:metinf'>...</Type></Meta>
<Item>
<Target>
<LocURI>item1</LocURI>
</Target>
<Source>
<LocURI>item1</LocURI>
</Source>
<Data>some data </Data>
</Item>
</Add>
</SyncBody>
</SyncML>
```

Even if it follows the representation rules, it is valid only in the case a previous initialization was made and the client requested the synchronization of the database *db1*. The package in charge of those aspects is *sync4j.framework.protocol.*

sync4j.framework.config is a utility module used to deal with the server and additional modules configuration. The Sync4j configuration architecture will be described later in this document.

The two packages *sync4j.framework.security* and *sync4j.framework.logging* represent the module that in Figure 2 is called *Services*. They implement logging and security services. Note that, for the security aspects, Sync4j adheres to the Java Authentication and Authorization Service (JAAS) delivered with the JDK 1.4. It is therefore possible to develop a proprietary authentication and authorization policy, configuring the system to use it instead of the standard module.

A package that plays an important role in the Sync4j architecture is *sync4j.framework.engine*. It provides a basic interface for a *synchronization engine*, allowing a pluggable architecture for customized engines. Generally speaking, the process of receiving and interpreting a synchronization message and the process of updating the data sources and producing the modifications for the client are distinct processes. They can also be applied independently one from the other. For example, from the synchronization point of view it does not really matter if a synchronization request came from a SyncML message or a simple HTTP request. In the same way, from the protocol point of view, it does not really matter which conflict resolution the synchronization engine will adopt. With this pluggable architecture, the business logic of the protocol and of the synchronization can be developed and extended separately (without modifying the server or the other modules) to meet at best the requirements.

The last package, *sync4j.framework.server* includes common classes for the development of server application and can be used to extend the standard Sync4j implementation.

As a developer, you might be interested in modifying one or more of the above components, but you are not forced to do it. Sync4j is a full featured SyncML synchronization server and provides a concrete implementation of the framework. However, flexibility and openness is the key in enterprise deployment: Sync4j allows you to customize and extend most of its features, if you need it.

In the following sections, we are going to tell more about each single framework layer.

4.1.1. Transport Layer

This layer implements the support for the various transport protocols SyncML can be bound to. Currently, Sync4j supports only the HTTP protocol, which is the most widely transport protocol used by the SyncML clients on the market. Other protocols might be added in future releases.

4.1.2. Application Layer

sync4j.server provides a basic structure for implementing a SyncML server.
In Sync4j, the server module is implemented as an Enterprise Java Bean that can be deployed into any J2EE 1.3 compliant application server. Behind this choice lay the following factors:

- decoupling between the transport protocol and the synchronization logic is pivotal in enterprise deployments;
- application servers provide many out-of-the-box facilities and services that should otherwise be redeveloped (i.e. connection management, thread management, security, scalability, availability, reliability), simply reinventing the wheel;
- J2EE it is a widely accepted standard in enterprises IT infrastructure;
- reusing the existing application server infrastructure simplifies management and deployment.

4.1.3. The Synchronization Engine

A synchronization server is not helpful without synchronization logic, such as the set of rules followed to:

- identify the sources and the destinations of data to be synchronized;
- identify what data needs to be updated/added/deleted
- determine how updates must be applied;
- detect conflicts;
- resolve conflicts.

In other words, the synchronization engine is the core of a data synchronization server.

Sync4j allows developers to plug in their own implementation of the synchronization engine. Therefore, developers can extend the basic behavior in order to meet their own requirements. Developers can even completely substitute the default implementation with a custom engine developed from scratch.
This brings a flexible and modular architecture, easier to reuse, extend and maintain.
The basic framework interfaces and classes are grouped in the package *sync4j.framework.engine*.

Since the synchronization process is the core of the synchronization engine, it is described in more detail in the following dedicated section.

5. The Synchronization Process

The synchronization process is driven by the *synchronization engine*, which in turn is a concrete implementation of the interface *sync4j.framework.engine.SyncEngine*.

The synchronization process is accomplished in three steps:

1. Preparation
2. Synchronization
3. Finalization

The Sync4j engine goes through these steps and coordinates the execution, but delegates most of the synchronization logic to an auxiliary class, implementation of the *SyncStrategy* interface.

As described before, two types of synchronization process are possible: *slow* and *fast*.
In a slow synchronization, the sources to be synchronized must be fully compared in order to reconstruct the right image of the data on both connection endpoints. The way the sets of items are compared is implementation specific and can vary from comparing just the item keys or the entire content of a *SyncItem*. In fact, in order to decide if two sync items are exactly the same or some fields have changed, all fields might require a comparison.
A slow sync is prepared by calling *prepareSlowSync(...)* of the *SyncStrategy* object.

In a fast synchronization, the sources are queried only for new, deleted or updated items since a given point in time. In this case, the status of the items can be checked in order to decide when a deeper comparison is necessary.
A fast sync is prepared by calling *prepareFastSync(...)* of the *SyncStrategy* object.

prepareSlowSync(...) and prepareFastSync() require an additional *java.security.Principal* parameter in input. The meaning of this parameter is implementation specific, but as a general rule, it is used to operate on the data belonging to a given entity such as a user, an application, a device, etc.

The following sections describe in more detail each phase of the synchronization process and other key aspects of the synchronization engine architecture. The section 5.5 puts all the pieces together, showing and describing the sequence diagram of the synchronization process.

5.1. Preparation

The preparation phase is the process of analyzing the differences between two or more sources of data (called *SyncSource*s) with the goal of obtaining a list of sync operations that, applied to the sources involved in the synchronization, will make the databases look identical (Figure 3).

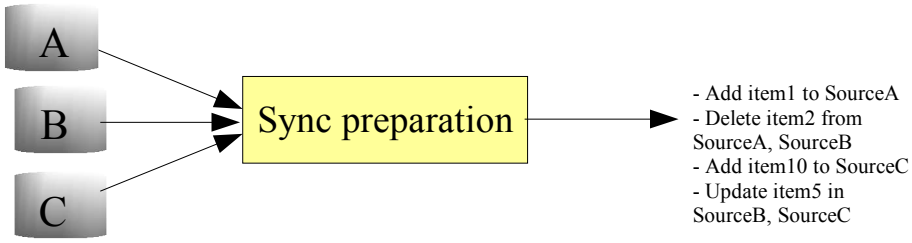

Figure 3- Preparation phase

5.2. Modifications Detection

Modifications detection is based on the sets of items represented in Figure 4, applying the modifications matrix of Table 3.

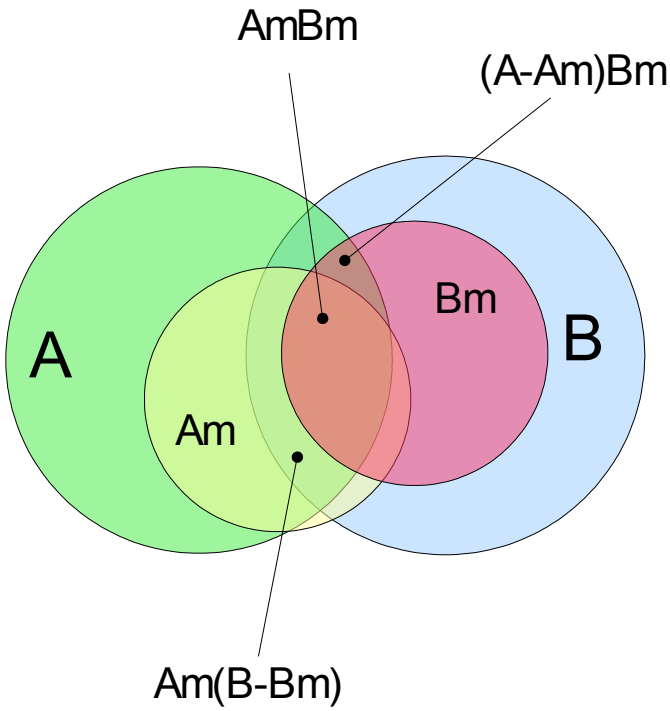

Figure 4 - Synchronization items sets

A – Items belonging to source A (as known via LUID-GUID mapping)
B – Items belonging to source B
Am – Modified items belonging to source A
Bm – Modified items belonging to source B
AmBm – Items modified either in source A and B (intersection between Am and Bm)
(A-Am)Bm – Items unmodified in A, but modified in B
Am(B-Bm) – items unmodified in B, but modified in A

Note that A is the server view of the A source: it contains the items mapped in the server as they are defined in the LUID-GUID mapping. If, for example, the client sends a new item that has never been mapped, this item will be in Am, but not in A. In order to be sure that the new item is not equal to some existing item in B, it must be looked up in B. If an item in B represents the same item as in Am, A is virtually augment of such item, so that at the end, Am will be a sub-set of A.

Another important aspect to point out is that the entire data sets A and B can be considerably big. Therefore, when possible, it is important to deal with the smallest possible sets of items instead of doing a full item-per-item comparison.

The preparation phase is slightly different depending on the type of the synchronization. In the case of a slow synchronization, all items in the sources must be compared looking for differences that will be translated into synchronization operations. This type of process does not depend on previous synchronizations and, in fact, it is used to fully recreate a database as if no synchronizations have ever taken place. This is achieved resetting the LUID-GUID mapping before starting the modification detection process.
On the contrary, when a fast synchronization is performed, it is assumed that the involved sources rely on a previous data synchronization, so that only the changes since the time of the last synchronization need to be considered.

The algorithm used in the preparation phase is as follows:

Given a set of sources A, B, C, D, etc, the synchronization process takes place between two sources at a time: A is first synchronized with B, then AB with C, then ABC with D and so on.
Given the sources to be compared, suppose A and B, the goal of the algorithm is to produce an array of *SyncOperation* objects, in which each element represents a particular synchronization action, i.e. create the item X in the source A, delete the item Y from the source B, etc. Sometimes, it is not possible to decide the action to perform, thus a *SyncConflict* operation is used. A conflict might be solved by something external the synchronization process, for instance by a user action. In order to create the *SyncOperation[]* array, each item in the source A is compared with each item in the source B (to be intended as the selected items depending on the synchronization type).

To determine which operation should be performed the *Synchronization matrix* defined above is used. We report the table here again for the sake of simplicity.

Database A → ↓ Database B	***New***	***Deleted***	***Updated***	***Synchronized/ Unchanged***	***Not Existing***
New	C	C	C	C	B
Deleted	C	X	C	D	X
Updated	C	C	C	B	B
Synchronized/Unchanged	C	D	A	=	B
Not Existing	A	X	A	A	X

Table 3 - Synchronization matrix

Where:
A : item A replaces item B
B : item B replaces item A
C : conflict
D : delete the target item
X : do nothing

Initially, items are compared based on a subset of the information they contain called *key (*in the synchronization engine it is called *SyncItemKey*). It is responsibility of the *SyncSource* to create proper and unique keys for each item. The *SyncItemKey* is stored in the *SyncItem* and can be obtained calling *getKey()*. The comparison is accomplished by the method *equals()* of the *SyncItemKey* object.

When the SyncStrategy performs a sync preparation, it returns the operations that have to be applied to the sources involved, in order to make them look equal. From a coding point of view, those

operations are represented by *SyncOperation* objects, which incapsulate the interested items and the operation itself.

5.3. Synchronization

The synchronization step is the phase where the sync operations prepared in the previous step are executed. Executing a *SyncOperation* means applying the required modification to the sync source involved.
For example, the *SyncOperation* represented by:

```
operation: new
item A: ITM0040102001  ← (the item key)
item B: null
```

results in the addition of *item B* to source *B*. Instead, if the operation is:

```
operation: new
item A: null
item B: ITM0376488440
```

The *item B* will be added to *source A*. The following combination will result in a conflict:

```
operation: new
item A:  ITM0040102001
item B:  ITM0040102001
```

The synchronization phase is implemented in the *sync(SyncOperation[])* method of *SyncStrategy*.

5.4. Finalization

The third and last step is intended for cleaning up purposes.

5.5. Synchronization Sequence Diagram

The sequence of operations that takes place during a fast synchronization is depicted in Figure 5, which serves as a guide for the following description.

The *SyncEngine* object drives the execution of all steps in its *sync()* method, where the requested sources are scanned for modified items. *SyncSourceA* and *SyncSourceB* represent the two sources involved in the synchronization process; generally, one source is the client view of the database, whilst the other source is the server view of the same data source.

First of all, *SyncEngine* calls *SyncStrategy.prepareSync(SyncSource[])* which returns an array of *SyncOperation*. Here, the synchronization engine has the opportunity to further processing the operations returned. For example, at this level the engine can decide how to solve conflicts.

After preparation and additional operation processing, the engine is ready to fire the execution of the real synchronization. Again, it performs the operation delegating the task to the *sync()* method of *SyncStrategy*.

Finally, *SyncStartegy.endSync()* is used to terminate the process.

The figure shows only the main tasks that *SyncStrategy* performs. First of all, it queries source A and B about which items have changed since the last synchronization and collects all of them in two lists, one for source A's items and one for source B's items. At this point, *SyncStrategy* is ready to compare those two sets of items and create the *SyncOperation[]* array. This is achieved by calling *checkSyncOperation(SyncItem[], SyncItem[])* where the rules described in the sections above and in the synchronization matrix are applied.

Note that the *SyncEngine* implemented in Sync4j makes use of the synchronization strategy object in the generic form represented by the interface *SyncStrategy*. The concrete implementation is configurable in the *Sync4j.properties* configuration file. Therefore, if you want or need to implement your own synchronization strategy, you can easily plug it into Sync4j just modifying that file.

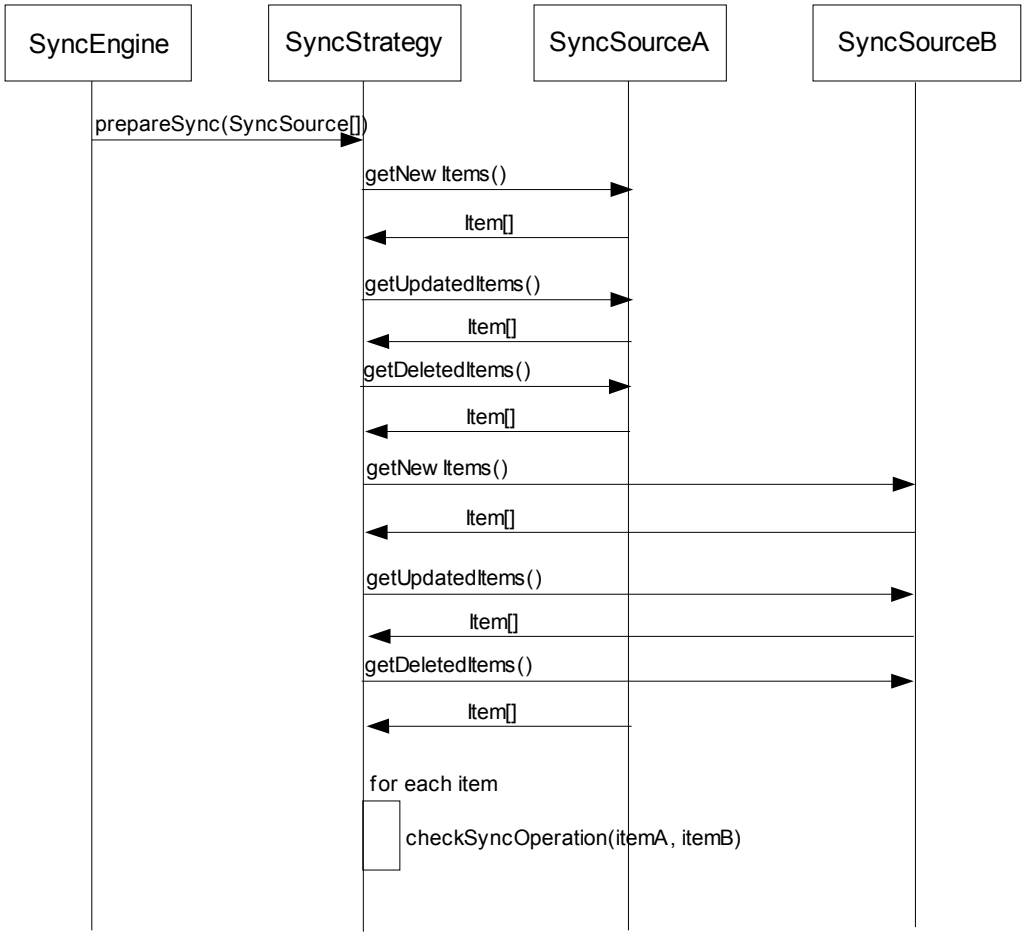

Figure 5 - Synchronization sequence diagram

6. Developing a SyncSource

A SyncSource is the means a set of data made available to Sync4j for synchronization. Therefore, in order to synchronize any type of data (files, database tables, calendar events and so on), there must be a proper SyncSource able to extract and store the data from and to the real data store.
Goal of Sync4j is to provide a collection of SyncSources for the most common uses (i.e. files), but new SyncSources can be independently developed and plugged in the synchronization engine so that Sync4j will be able to process synchronization requests targeted to virtually any data source.

6.1. The SyncSource Interface and Related Classes

The core of the SyncSource architecture is the interface *sync4j.framework.engine.source.SyncSource*. This interface does not make any assumption on the type of data being synchronized, so that its concrete implementations are completely free to access the storage they prefer.

A SyncSource is identified by a *name* and a *sourceURI;* the former represents a domain specific name, the latter is the URI that a SyncML client must specify in order to synchronize this particular SyncSource. Note that they must be both unique.
A SyncSource is also associated to a type, in the form of a mime type that represents the kind of data handled by the source.

The most important methods defined by the SyncSource interface are:

method	***description***
getUpdatedSyncItems	Called to retrieve the updated SyncItems for the given principal since the given point in time.
getUpdatedSyncItemKeys	Called to retrieve the SyncItemKey of the updated items for the given principal since the given point in time.
getNewSyncItems	Called to retrieve the new SyncItems for the given principal since the given point in time.
getNewSyncItemKeys	Called to retrieve the SyncItemKey of the new items for the given principal since the given point in time.
getDeletedSyncItems	Called to retrieve the deleted SyncItems for the given principal since the given point in time.
getDeletedSyncItemKeys	Called to retrieve the SyncItemKey of the deleted items for the given principal since the given point in time.
getAllSyncItems	Called to retrieve all the SyncItems for the given principal since the given point in time.
setSyncItem/s	Called to insert or update the given item(s).
removeSyncItem/s	Called to remove the given item(s).

Table 4 - SyncSource methods

When a synchronization requests reaches the engine, Sync4j looks for a source whose sourceURI matches the requested URI and computes the synchronization analysis calling the methods defined above.

6.1.1. Principal and Since Timestamp

SyncSource methods usually require two input parameters in order to retrieve the items:

- *principal* (of type *java.security.Principal*) and
- *since* (of type java.sql.Timestamp).

A principal represents any entity the data can be associated to. A principal is usually represented by a user id, but it may be something different (like a device or a client agent). The principal is used to limit the manipulation to the data related to the given entity, such as the contacts of a given user. If this parameter is null, all items in the datastore are considered for synchronization, regardless of the principal they belong to.
In Sync4j, a principal is composed of a userid and a deviceid, because the same user may make use of different devices.

The *since* timestamp represents the point in time of the last synchronization. It is used in fast synchronization to get the changed items since the last synchronization request.

In case of slow sync, *getAllItems()* is called instead of *get(Updated/New/Deleted)Items()*.

6.1.2. SyncItem

Items returned by a SyncSource are encapsulated in *sync4j.framework.engine.SyncItem* objects. SyncItem is a Java interface that the developer can implement in order to meet specif requirements. Sync4j provides a standard implementation of a SyncItem by the class *sync4j.framework.engine.SyncItemImpl*. SyncItem defines the following methods:

method	***description***
getKey	Returns the item key.
getState	Returns the item state.
setState	Sets the item state.
getProperties	Returns all item properties.
setProperties	Sets all item properties.
getProperty	Returns a specific item property.
setProperty	Sets a specific item property.
getPropertyValue	Returns a specific item property value.
setPropertyValue	Sets a specific item property value.
getSyncSource	Returns the SyncSource the item belongs to.

The content of an item is stored in *sync4j.framework.engine.SyncProperty* objects which represent a name-value pair. This suits almost any data representation requirements in a data synchronization context.

Two standard properties are defined and used by Sync4j: *BINARY_CONTENT* and *TIMESTAMP*.
BINARY_CONTENT is intended to store an item in a raw binary form. This is used, for instance, when the item is treated as a monolithic object identified only by the item key. No content parsing is implemented in order to identify fields and data.
TIMESTAMP contains the timestamp of the last change of the item state and it is used in the synchronization process, in order to determine the operation to be performed on the sources.

IMPORTANT: when a sync source creates SyncItems, it must always provide a value for at least the two properties BINARY_CONTENT and TIMESTAMP.

6.2. Sync4j Engine Configuration

To make a SyncSource available to Sync4j, it must be registered inserting a row in the *sync4j_sync_source* database table. This table binds the source URI to the bean implementing the SyncSource. For example, the test sync source might be configured as follows:

URI	***config***
test	com/funambol/Sync4j/engine/source/TestFileSystemSource.xml

The sources registered in sync4j_sync_source are loaded and initialized at engine startup.

The specified server bean configuration is a configuration file that must be available as prescribed by the configuration architecture (see later).

7. Configuring Sync4j and Sync4j Components

One of the Sync4j design goal is to provide a framework that can be used to implement any kind of synchronization service, extending existing modules or plugging in new modules. All this configuration info is easily accessible and editable, with the aim of avoiding complex and huge configuration files.

Sync4j uses three configuration techniques:

- Sync4j.properties
- J2EE deployment environment entries
- *Server JavaBeans*

In the following sections these three types of configuration are described in details.

7.1. Sync4j.properties

This is the main Sync4j configuration file, because it is used to initialize the engine. It is a standard properties file, read at engine initialization time so that the engine class (*sync4j.server.engine.Sync4jEngine*) can be instantiated with the properties needed to bootstrap. See the Sync4j administration guide for a list of all possible properties and their meanings.

7.2. J2EE deployment environment entries

A standard way to configure J2EE components is using EJB and WAR environment entries. In Sync4j, the session EJB SyncBean is the first component activated when a request comes in to the server. Therefore, it must be configured with the minimal information required to start the Sync4j engine. At this level the following parameters are specified:

Entry	***Description***	***Default***
syncengine/factory/bean	The name of the sync engine factory bean to use. Changing this value, you can make SyncBean processing requests with your own SyncEngine.	sync4j.server.engine.Sync4jEngineFactory
server/config_uri	URI that points to Sync4j.properties.	file://{sync4j-path}/config/Sync4j.properties
server/config_path	The config path for server JavaBeans.	file:///{Sync4j-path}/config/

The way SyncBean and Sync4jEngine interact is depicted in Figure 6.

SyncBean uses its minimal configuration to create the *SyncEngineFactory* and set the engine configuration read from the given *server/config_uri* environment entry. Later, when a new *SyncEngine* is needed, Sync4jEngineFactory creates the *Sync4jEngine* object with the configuration previously set.

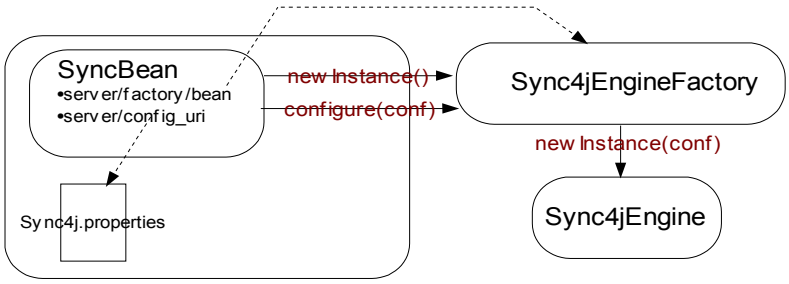

Figure 6 - SyncBean and SyncEngine Configuration

7.3. Server JavaBeans

As seen in the previous sections, some components are configured as *server JavaBeans*. Server JavaBeans are JavaBeans used server-side. The idea is to store a bean configuration as the serialized form of the bean itself. This way, a bean can be instantiated, configured and serialized to persist its configuration. Later, the bean can be deserialized in a properly configured instance.

However, it would not make sense to force the bean to be instantiated, configured and serialized any time its configuration changes. To solve this problem, Sync4j makes use of the standard java facility to serialize objects into XML (and to deserialize them from XML). This is achieved by the means of the classes *java.beans.XMLEncoder* and *java.beans.XMLDecoder*. Since configuration files created with such encoder/decoder are easy to use, read and write, they can be created and modified manually with a simple text editor, without the need of a dedicated GUI. An additional advantage of this approach is that server JavaBeans are not requested to implement *java.io.Serializable* because *XMLEncoder* does not require it.
This is an example of a server JavaBean:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.server.store.PersistentStoreManager">
  <void property="jndiDataSourceName">
    <string>java:/jdbc/sync4j</string>
  </void>
  <void property="stores">
   <array class="java.lang.String" length="2">
     <void index="0">
      <string>sync4j.server.store.SyncPersistentStore</string>
     </void>
     <void index="1">
      <string>sync4j.server.store.EnginePersistentStore</string>
     </void>
   </array>
  </void>
 </object>
</java>
```

In order to help server JavaBeans handling, Sync4j uses the factory class *sync4j.framework.tools.beans.BeanFactory,* which in turn makes use of a customized class loader; the class loader handles configuration files in a so called *config path*, in the same way a common class loader handles classes in a classpath.

7.3.1. The configuration path

Server JavaBeans are looked for in the configuration path, which is analogous to the class path for classes lookup. This is implemented reading the serialization files from a custom class loader, *sync4j.framework.config.ConfigClassLoader*. This may or may not make use of a parent classloader and can be configured with one or more URIs. In Sync4j 1.0.x, only one directory is used as config path as specified by the *server/config_path* SyncBean's environment entry.

7.3.2. Lazy Initialization

When a bean is deserialized from its XML form, the classloader that loads the serialization file calls first the empty constructor and then sets the bean properties values using the setXXX() methods provided by the class. However, some classes need additional operations to properly initialize (after setXXX() methods are called). To support this *lazy initialization* approach, these classes can implement *sync4j.framework.tools.beans.LayInitBean*, which defines a separate *init()* method. When Sync4j loads a *LazyInitBean*, after the bean instantiation (or deserialization), it calls its *init()* method, giving the bean the opportunity to complete its initialization.

8. Message Processing Pipeline

Goal of the message processing pipeline is to have a hook for adding additional processing and manipulation of the messages exchanged between the server and the client.
The kind of processing that is performed in the pipeline is a message level processing, such as the manipulation of the message elements. Possible applications are:

- Encoding/decoding of item content
- Item filtering
- Item ordering
- Message decoration (adding/removing elements on a custom basis)

This section describe the architecture and the design of the message processing pipeline implemented in Sync4j.

8.1. Architecture

The idea behind the message processing pipeline is to be able to modify both incoming and outgoing messages. In the former case, we want to be able to manipulate the message before it goes into the sync engine; instead, in the latter, we want to be able to change the message returned by the sync engine before send it to the client.

This is achieved with two different pipelines as outlined in Figure 7.

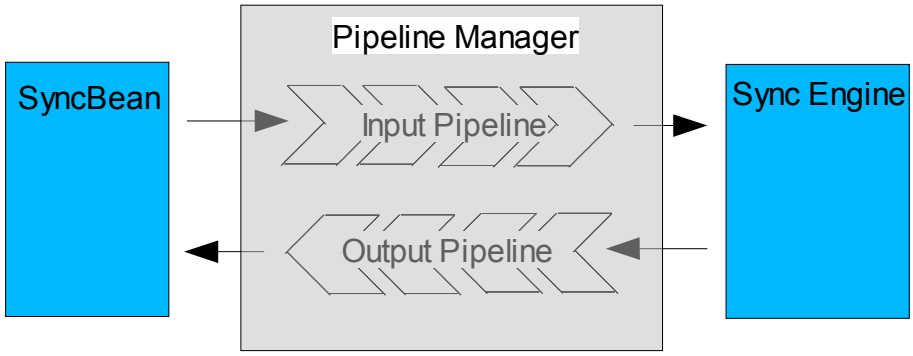

Figure 7 - Pipeline architecture

The input and output pipelines are constructed and managed by a Pipeline Manager component, which is configured with the list of components that build up the input pipeline and the list of components that build up the output pipeline.
The duties of the Pipeline Manager are:

- Creating the input and output pipelines at initialization time
- Provide a way to start the input or output message processing

- Coordinating the execution of the components in the pipelines
- Keep the “message processing context”, which is the state of one pipeline execution

8.2. Design

8.2.1. Overview

As said, the processing of a message starts just before an incoming message is submitted to the sync engine or just before an outgoing message is returned to the client.
As described in the sequence diagram of Figure 8, the pipeline manager creates a new “instance” of the input and output pipelines when a new synchronization session begins (so that in the *ejbCreate()* method of *SyncBean*). When a message comes from the client, instead of being processed immediately by the sync engine, it is passed to the input pipeline for preprocessing. Each component of the pipeline can then pre-process the message and apply its changes. The decorated message is then processed by the sync engine.

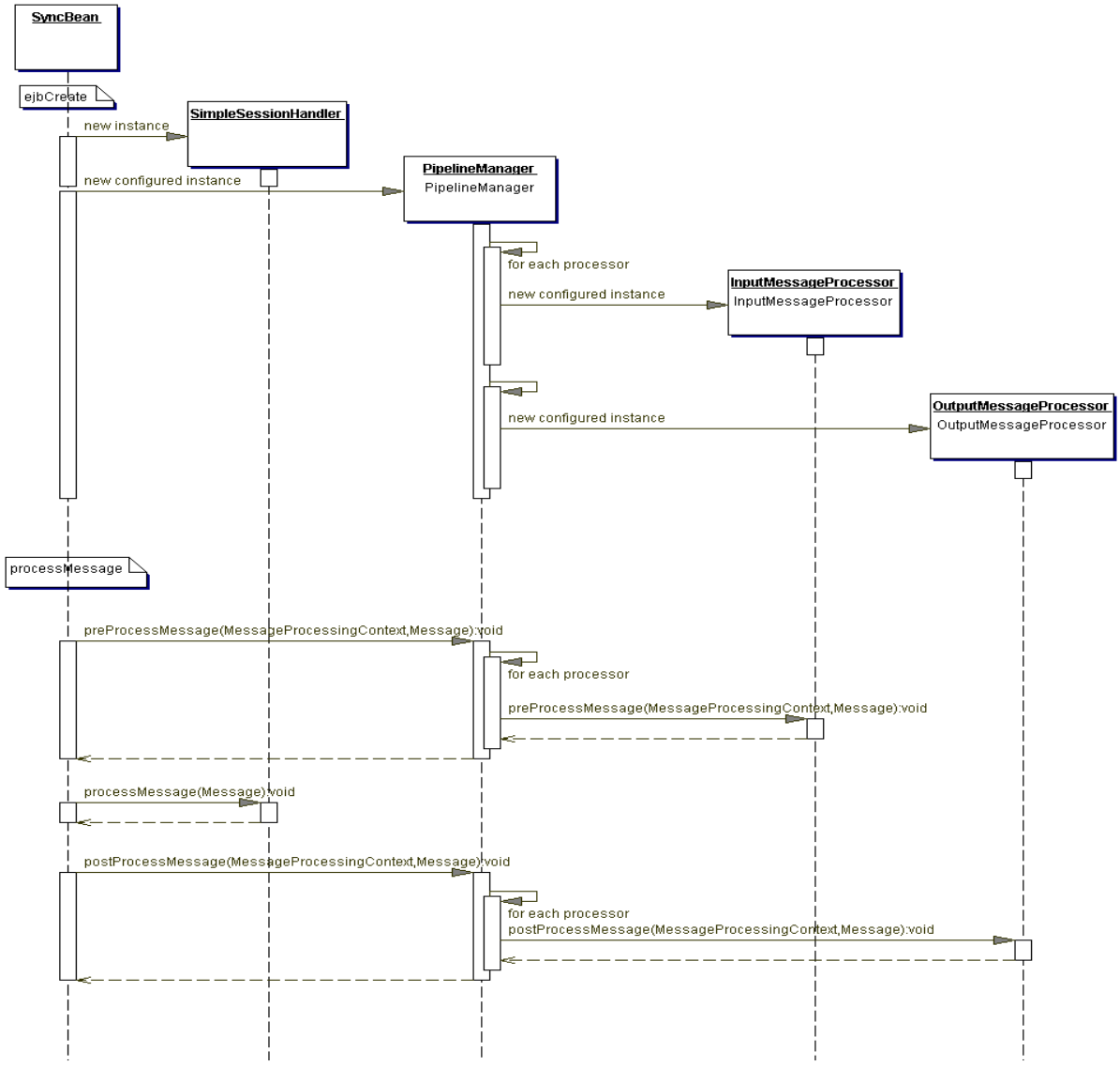

Figure 8 - Message processing pipeline sequence diagram

8.2.2. Class Diagram

The classes involved in the message processing pipeline architecture are depicted in Figure 9 and grouped under the package *sync4j.framework.server.engine.pipeline*.

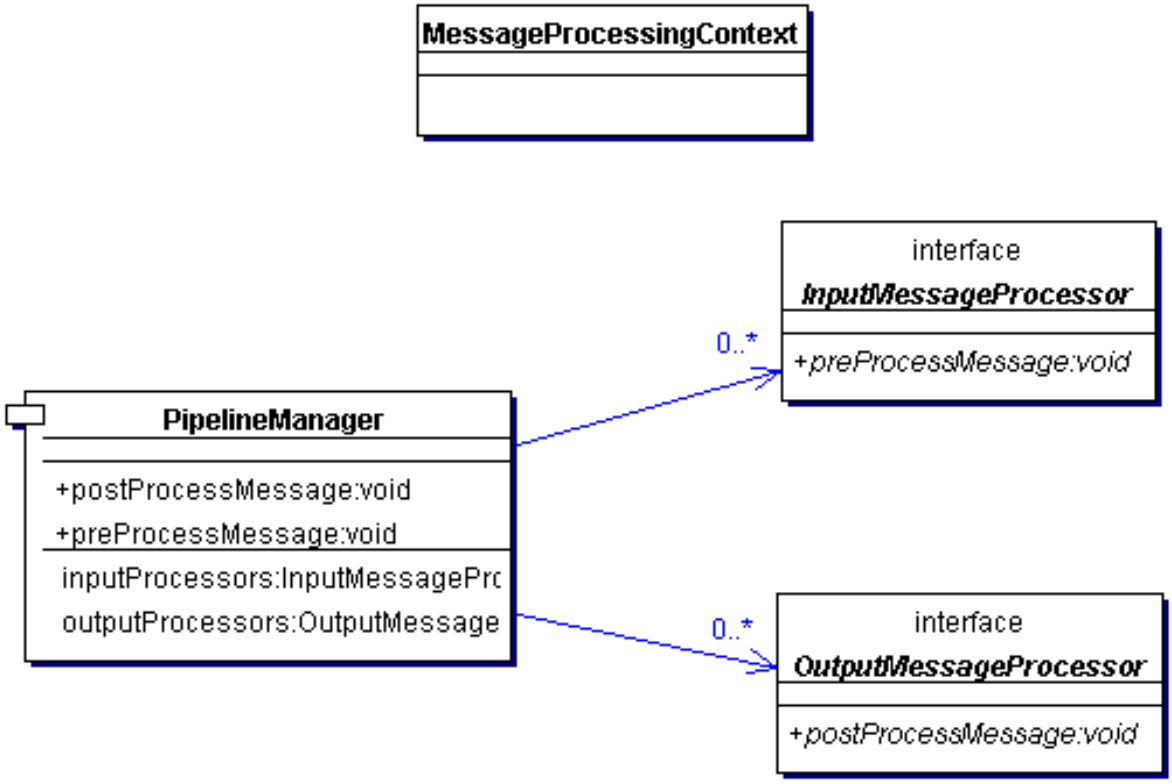

Figure 9 - Message processing pieline class diagram

Each component of the input pipeline must implement the *InputMessageProcessor* interface; on the other hand, each component of the output pipeline must implement the *OutputMessageProcessor* interface. The *PipelineManager* class implements the pipeline manager, which is the component that the transport layer deals with. The Pipeline Manager component is configured setting its *inputProcessors* and *outputProcessors* arrays directly in the server java bean configuration file. *postProcessMessage()* and *preProcessMessage()* are called to start the processing of messages. The message being processed is passed to the processing methods as a *sync4j.framework.core.Message* object, so that a processor can easily modify it. When one of those methods is called, *PipelineManager* creates a new *MessageProcessingContext* and loops through the *(input|output) Processors* arrays calling each element's *(pre|post)ProcessMessage()* passing the context and the message as parameters.

8.2.3. PipelineManager Configuration

The *PipelineManager* is configured as a common Sync4j server bean where the two properties *inputProcessors* and *outputProcessors* must be set as array properties accordingly to the java.beans.XMLDecoder specifications.

Here is an example of a PipelineManager configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.engine.pipeline.PipelineManager">
  <void property="inputProcessors">
    <array class="sync4j.framework.engine.pipeline.InputMessageProcessor" length="2">
     <void index="0">
       <object class="sync4j.framework.engine.pipeline.InputLogProcessor"></object>
     </void>
```

```
    <void index="1">
      <object class="com.funambol.syncserver.pipeline.ItemDecoder">
        <void property="type">
          <string>text/vcard</string>
        </void>
        <void property="version">
          <string>3.0</string>
        </void>
      </object>
    </void>
   </array>
  </void>
  <void property="outputProcessors">
    <array class="sync4j.framework.engine.pipeline.OutputMessageProcessor" length="1">
      <void index="0">
        <object class="com.funambol.syncserver.pipeline.ItemEncoder">
          <void property="type">
            <string>text/vcard</string>
          </void>
          <void property="version">
            <string>3.0</string>
          </void>
        </object>
      </void>
    </array>
  </void>
 </object>
</java>
```

8.2.4. Error Handling

Each processors can throw a *sync4j.framework.core.Sync4jException* in case of errors during the processing. The *PipelineManager* logs the error condition at INFO level to the logger *sync4j.engine.pipeline* and then carries on with the next pipeline component.

9. Error and Exception Handling

This chapter describes the use of exceptions to handle error conditions at the different levels in the layered architecture of Sync4j. Simplifying the structure depicted in Figure 2 and focusing on the execution flow of a SyncML request, we have the flow of Figure 10.

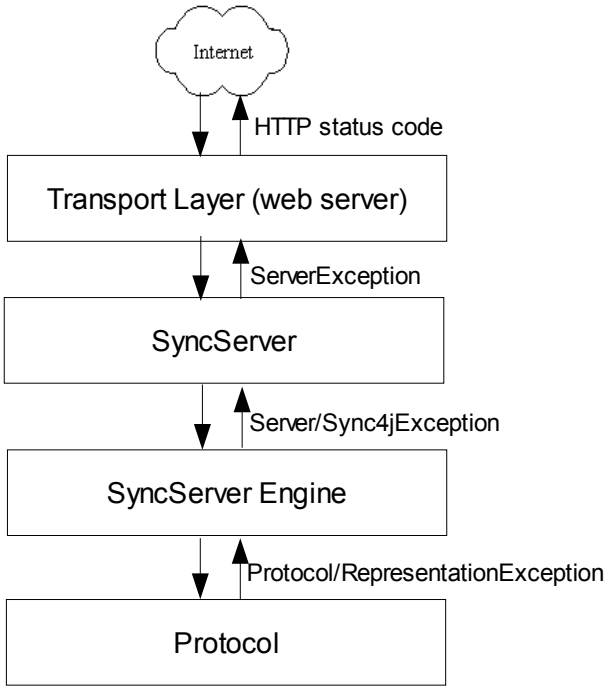

Figure 10 - Sync4j exception handling

The rule of thumb in handling error conditions is that when possible a SyncML message with an error status code should be returned instead of other kind of errors (such as transport level errors).
Like the picture suggests, only few types of exception should be thrown by the methods that cross the layers boundaries. Those exceptions should be strongly related with the responsibility of the throwing layer. However, inside the layer, other exception types can be defined and used as needed.
The following sections describe the error handling at each specific layer and the meaning of the different exceptions used in Sync4j and shown in Figure 11.

9.1. Sync4j Exception

Sync4jException is the base of most of the exceptions defined in Sync4j. It makes use of the new `Exception` object provided by the JDK 1.4, which allows chaining an exception with the causing exception. This functionality is very useful, because it allows to convert a low-level exception to an exception that crosses the layer boundary, while keeping the root cause of the error.

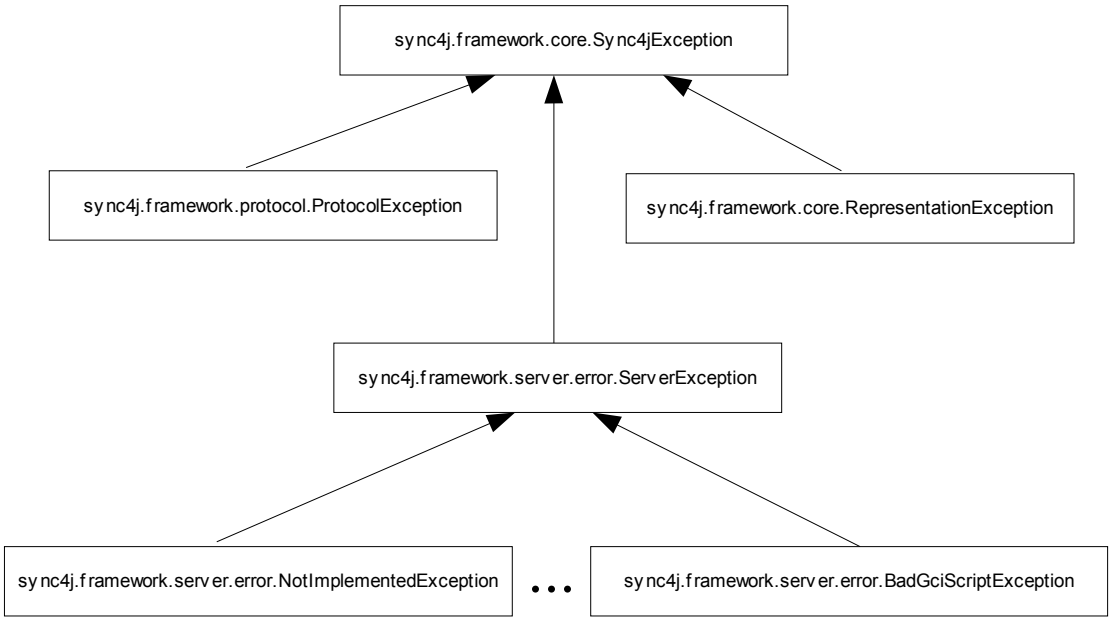

Figure 11 - Exception hierarchy

9.2. Server Exception

Sync4j and its extensions should only throw `ServerException` exceptions. This will include the causing exception if the error condition is due to one of the underlying layers. In addition to simply representing an error condition, `ServerException` stores also a status code associated to the error. This is particularly helpful when dealing with exceptions that encapsulate SyncML-level errors such as the SyncML error codes. `ServerException`s should be let reach the transport layer only when the error is in some way fatal, so that it is not possible to recover from it and produce a proper SyncML response message. When possible, it is recommended to create a SyncML message with a `<Status>` element representing the error condition with a proper status code in its `<Data>` subelement; status codes are defined by the SyncML specifications in [1].

9.2.1. SyncML Exceptions

Each SyncML error status code is encapsulated in a corresponding exception, for example `NotImplementedException` or `BadCGIScriptException`. All these exceptions are direct subclasses of `Sync4jException`.

9.3. Protocol Exception

A protocol error can be of two types:

1. SyncML representation error
2. SyncML protocol error

SyncML representation errors groups the errors occurring in the representation of a SyncML message: for example, when the XML document is not well formed, when there are parse or syntax errors, or when, even if the message is a well formed XML content, it does not represent a SyncML message (for example the `<SyncHeader>` element is missing). These are low level exceptions that may make impossible returning a proper SyncML response containing the error status code. When creating the response SyncML error message is impossible, the exception will bubble up to the transport level and a transport specific error response will be returned to the client.

SyncML protocol errors groups errors related to the violation of the SyncML protocol in terms of sequence of messages. Examples of these errors are the violations of the requirements mandated by the SyncML specifications for the initialization or modifications message (see [2]).
Following the classification above, two different exception classes are defined in Sync4j:

1. `core.framework.protocol.ProtocolExcepion`
2. `core.framework.core.RepresentationException`

10. Sync4j Modules

Sync4j modules represent the means third party developers can extend the way Sync4j works. A module is a packaged set of files containing classes, configuration files, initialization SQL scripts and so on, used by the installation procedure to embed the extensions into the Sync4j Enterprise Archives (the J2EE ear).

For more information on how to install Sync4j modules see [3].

For beginners information on how to build a Sync4j module see [4].

10.1. Building a Sync4j Module

A Sync4j module is a jar package named following the convention:

```
<module-name>-<major-version>.<minor-version>.s4j
```

Where <module-name> is the name of the module without spaces and with small caps only and <major/minor-version> are the major and minor version numbers. Changes in the minor version number must be backward-compatible, whilst changes in the major version number may require migration efforts.

The package must have the structure presented in Figure 12.

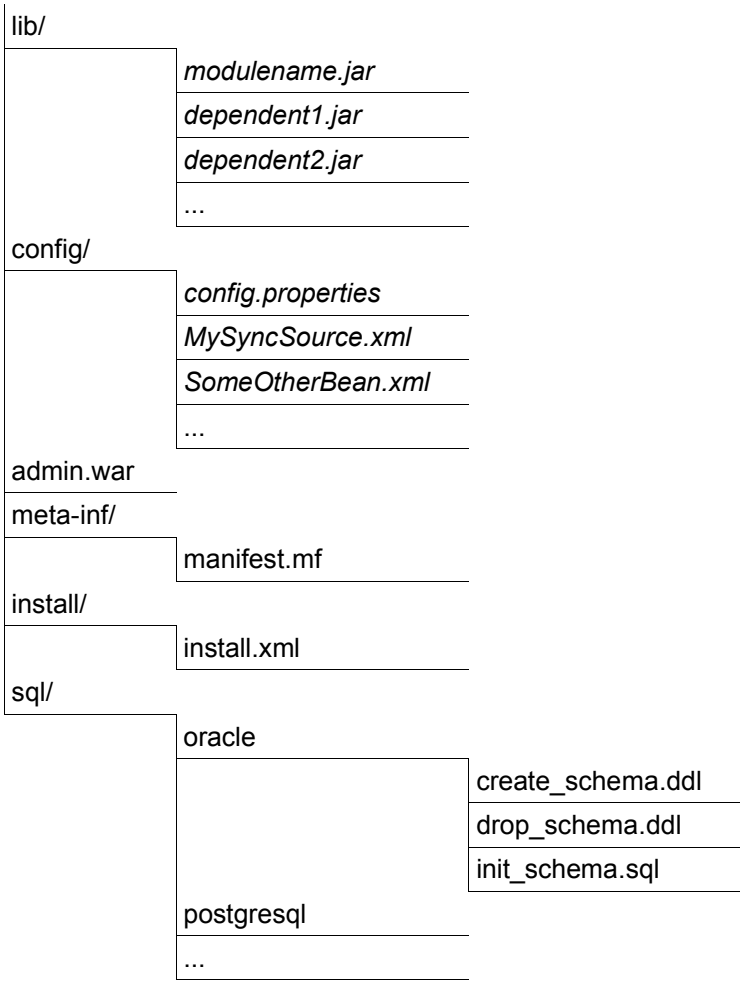

Figure 12 - Module package structure

In the figure, entries ending with a '/' represent directories and filenames in italic are given just as examples (in a real package they will be replaced with real filenames).
The module classes are packaged in a main jar file called <*modulename*>*.jar*. If this package requires additional libraries, it must use the java extension mechanism to make them available (in particular, depended libraries must be included in the *Class-path* manifest entry).
Configuration properties files and bean configuration files are stored under the package directory *config*, creating subdirectories as needed.
The directory *install* contains *install.xml,* which is an Ant script, called when the module is being installed; this is the hook where a module developer can insert module specific installation tasks. Installation specific files can be organized in subdirectories under *install*. If the module requires a custom database schema, the scripts to create, drop and initialize the database are stored under the *sql/*<*database*> directory, where <database> is the name of the DBMS as listed in the *install.properties* file.

11. References and Resources

11.1. References

[1] *SyncML Representation Protocol, version 1.1*,
http://www.syncml.org/docs/syncml_represent_v11_20020215.pdf
[2] *SyncML Sync Protocol, version 1.1*,
http://www.syncml.org/docs/syncml_sync_protocol_v11_20020215.pdf
[3] *Sync4j 3.0 Administration Guide*, Funambol, 2003
[4] *Sync4j 3.0 Module Development Tutorial*, Funambol, 2003

11.2. Resources

[1] www.syncml.org
[2] Java Authentication and Authorization Service, Reference Guide, JDK 1.4.x documentation