Sync4j SyncServer Developer's Guide

October 2004

Table of Contents

1. Introduction

This document is intended for developers who aim to develop synchronization services based on Sync4j 2.2 SyncServer.

1.1. Audience

1.2. 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 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

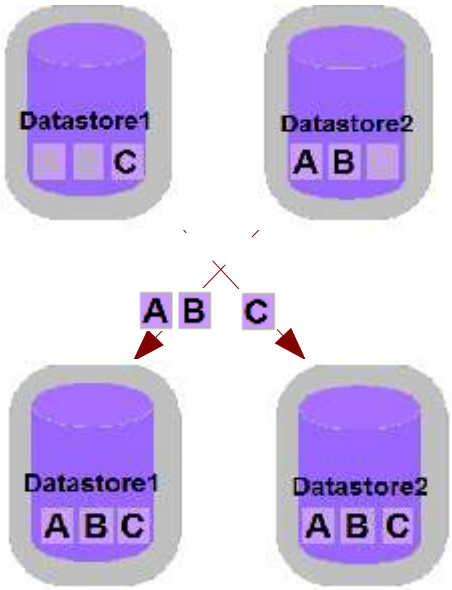

Figure 1 - Data synchronization process

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 process 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 shown 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 complicated and alternative methods must be adopted (based on content comparison, for instance).

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's suppose two users synchronize their local contacts 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's 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 versions of the John Doe record should be taken and stored into the server? This condition is called *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
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* 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 user 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 that it has to apply 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 SyncServer High-level Architecture

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

- The *Sync4j Engine*, which is extensible with 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 SyncServer can integrate with external and legacy systems.

Sync4j SyncServer 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 SyncServer and driving the development of Sync4j SyncServer extensions.

4.1. System Architecture

The system architecture of the SyncServer is shown in Figure 2: the transport and the business logic (protocol handling) are separated in two distinct blocks so that they can be handled respectively (but not necessarily) by a web application running in a J2EE web container and by an Enterprise Java Bean running in a J2EE EJB container.

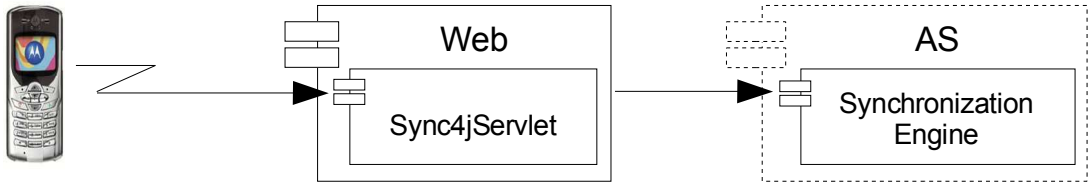

Figure 2- System Architecture

The web module implements the transport protocol (being OMA DM messages transported over HTTP). The EJB/Synchronization Engine layer contains the real synchronization logic implementation,

which is built of many components. Both the web layer and engine components are described in further details in the following sections.

4.2. SyncServer Architecture Overview

The SyncServer architecture is layered and modular (Figure 3).

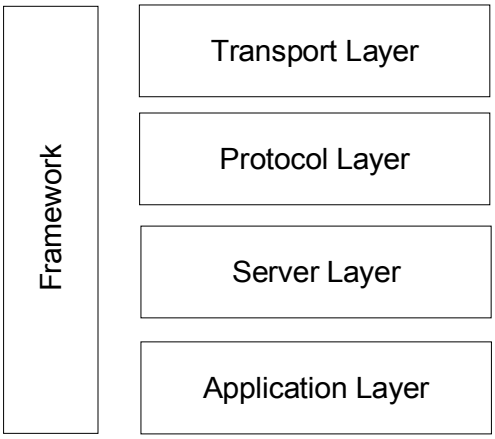

Figure 3 – DM Server layered architecture

Layers represent groups of functionality with well defined boundaries and communication interfaces. They are:

- Transport layer (i.e. HTTP)
- Protocol layer (i.e. SyncML)
- Server layer (i.e. synchronization server)
- Application layer (i.e. customer care front end)

The *transport layer* is the door through which client messages reach the system. The current implementation of the sync server implements the HTTP transport protocol and binding as defined by the HTTP binding OMA DS specification. The system is designed so that other transport protocols may be added in the future.

The *protocol layer* is responsible for the interpretation and handling of the SyncML protocol. It works at both representation and protocol levels. This layer is designed so that other synchronization protocols may be added in the future.

The *server layer* is the synchronization server implementation. The engine is entirely implemented as java module that can easily wrapped into a J2EE based application by the means of the *SyncBean* EJB. In this way, the server can be easily deployed on any J2EE compliant application server.

The *application layer* implements the way the synchronization server interacts with the external world. In addition to be a full implementation of simple and commonly used components, this layer implements also a framework used to extend the server in order to meet any application specific needs.

The *framework* implements and provides services and abstractions used by the different layers to implement the component they are built of. The most important services provided by the framework are:

- Core SyncML representation and protocol
- Configuration framework
- Logging framework
- SyncML DS engine framework

- Security framework
- Commonly used utilities

Those services are implemented in many packages, of which, the most important 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 and *sync4j.framework.protocol* implement the block that in Figure 3 is called *Protocol* and groups the foundation classes used to represent a SyncML message. This modules allow 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*.

sync4j.framework.config is 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* implement logging and security services. Note that, for the security aspects, Sync4j SyncServer 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 SyncServer architecture is *sync4j.framework.engine*. It provides a basic interface of 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 cames 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 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 classes for the development of server applications and can be used to extend the standard Sync4j SyncServer 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 SyncServer 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 SyncServer allows you to customize and extend most of its features, if you need to do it.

4.2.1. The Synchronization Engine

A synchronization server is not helpful without synchronization logic, so that a set of rules followed to:

- identify the sources and the destinations of the data sets 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 SyncServer 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 a dedicated section later in this document.

4.3. The Execution Flow

The execution flow of an OMA DS request is shown in Figure 4.

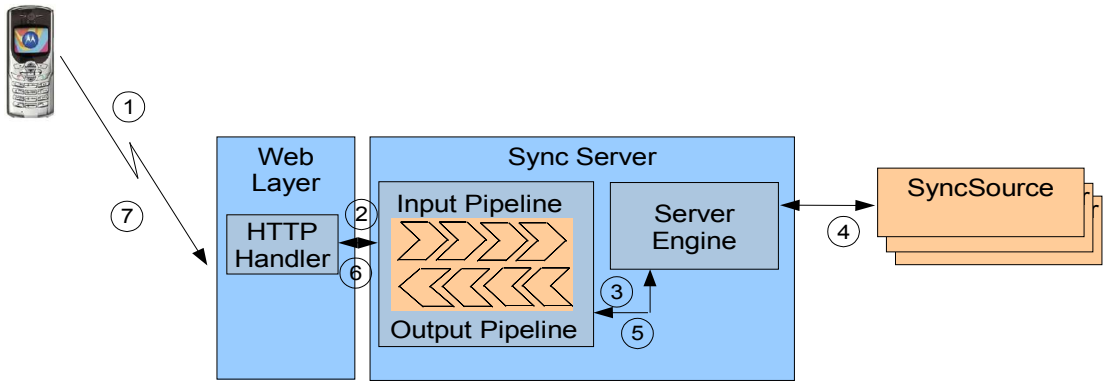

Figure 4 - Execution flow of an OMA DM request

In Figure 4, blue and light blue blocks are part of the server implementation; light red blocks instead a re customizable components that developer can build/extend/embed.
Finally, the orange blocks are components added and customized by developers to meet the end-user management application needs.

A synchronization session starts with the client device sending a first SyncML message to the server. The request then follows the flow described below.

1. When a new request comes from the client, the HTTP handler takes care of it. After some processing, for example the translation of the binary message into a more manageable form or the association of the incoming message to an existing synchronization session, the HTTP handler passes the request to the synchronization server.
2. The message first goes through the *input message processing pipeline* (see later) where it can be preprocessed accordingly to the application needs.
3. The manipulated message comes out from the input pipeline and goes into the server engine for the synchronization processing.
4. When needed, the server engine calls the services of the external (and custom) sync sources in order to access the real data stores.
5. After processing the incoming message, the server engine builds the response message, which goes through the *output message processing pipeline* for postprocessing.
6. The so postprocessed message is then returned to the HTTP handler, which packs the SyncML message into a HTTP response and sends it back to the device.

5. The Synchronization Process

The synchronization process is accomplished in three steps:

1. Preparation
2. Synchronization
3. Finalization

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

There are many types of synchronization; the ones specified by the SyncML protocol are:

Sync Type	***Description***
Two-way sync (fast)	A normal sync type in which the client and the server exchange information about modified data in these devices. The client sends the modifications first.
Slow sync	A form of two-way sync in which all items are compared with each other on a field-by-field basis. In practice, this means that the client sends all its data from a database to the server and the server does the sync analysis (field-by-field) for this data and the data in the server.
One-way sync from client only	A sync type in which the client sends its modifications to the server but the server does not send its modifications back to the client.
Refresh sync from client only	A sync type in which the client sends all its data from a database to the server (i.e., exports). The server is expected to replace all data in the target database with the data sent by the client.
One-way sync from server only	A sync type in which the client gets all modifications from the server but the client does not send its modifications to the server.
Refresh sync from server only	A sync type in which the server sends all its data from a database to the client. The client is expected to replace all data in the target database with the data sent by the server.
Server Alerted Sync[1]	A sync type in which the server to alerts the client to perform sync. That is, the server informs the client to starts a specific type of sync with the server.

Table 3 - Sync modes defined by SyncML

The first two are the most important, since the others are derivation of slow and fast sync modes.
In a slow synchronization, the client sends all its items to server, which compare them with the server database and then it sends back the modification that the client has to apply in order to be in sync again. In the case of slow sync, the sources to be synchronized must be fully compared in order to reconstruct the right image of the data on both synchronization endpoints. The way the sets of items

1 The SyncML specification does not tell anything about how server alerted sync should be achieved, therefore each product can implement it in a different and not interoperable way. As per nowadays, only few devices are known to support this feature. Sync4j does not currently implement server alerted sync.

are compared is implementation specific and can vary from comparing just the item keys or the entire content of an item.

A slow sync is prepared by calling *prepareSlowSync(...)* of the *SyncStrategy* object.

In a two-way fast synchronization, the sources are queried only for new, deleted or updated items since a given point in time (and for a given user). In this case, the status (deleted/updated/new) and the modification timestamp 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 5).

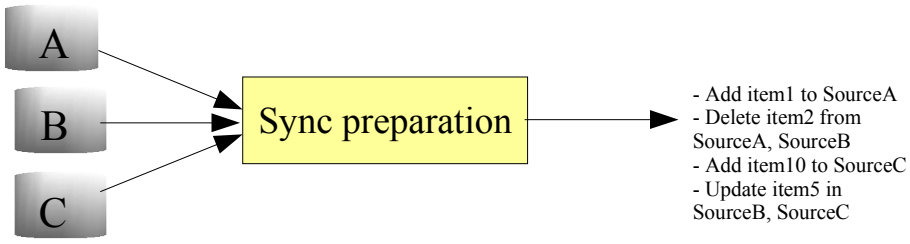

Figure 5- Preparation phase

5.2. Modifications Detection

Modifications detection is based on the sets of items represented in Figure 6, applying the modifications matrix of Table 4.

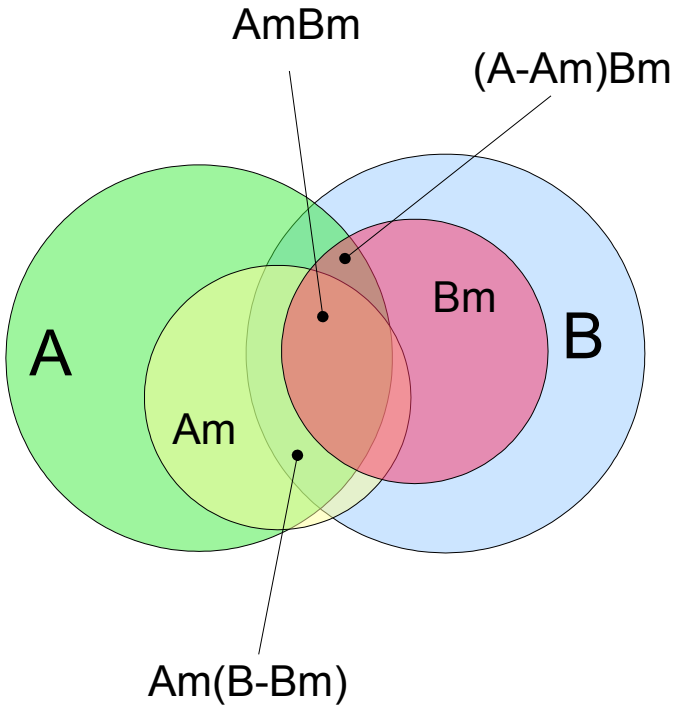

Figure 6 - Synchronization items sets

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.

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 4 - 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 encapsulate the interested items and the operation itself.

When a client item should be updated or inserted on the server, but the server does not have a mapping for it, a deeper comparison must be performed. In fact, the new/updated item could have the same content of an existing item on the server; this could even turn into a conflict. For example, suppose that a client tries to insert a new appointment with id xyz at 20041029T1400Z in the meeting room OceanSide. Even if there is no matching item on the server, if OceanSide is already busy at the same time, this could be considered a conflict.
In order to detect such situations, the synchronization engine will ask for items *similar* to the one that is trying to add/update. Those similar items are called *Twins*. Note that we used by choice *similar* and not *equal*. This is because how much an item should look like an existing item in order to be considered a twin is not fixed. Each source should be able to find twin items accordingly to its own logic.

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 7, 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 SyncServer 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 SyncServer just modifying that file.

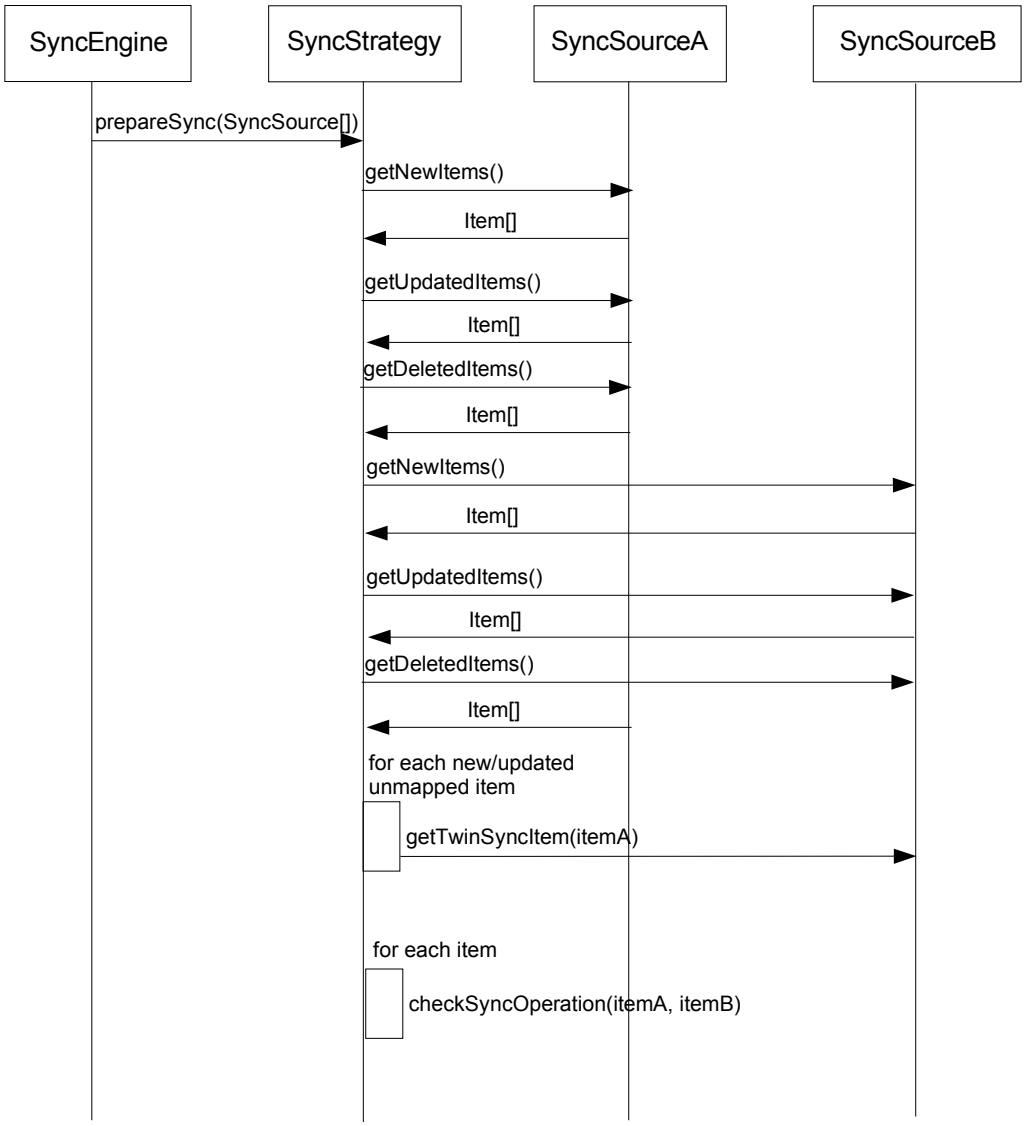

Figure 7 - Synchronization sequence diagram

6. Extending the SyncServer with Sync4j Modules

Sync4j modules represent the means by which third party developers can extend the way Sync4j works. A module is a packaged set of files containing classes, installation scripts, 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].

6.1. 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 8.

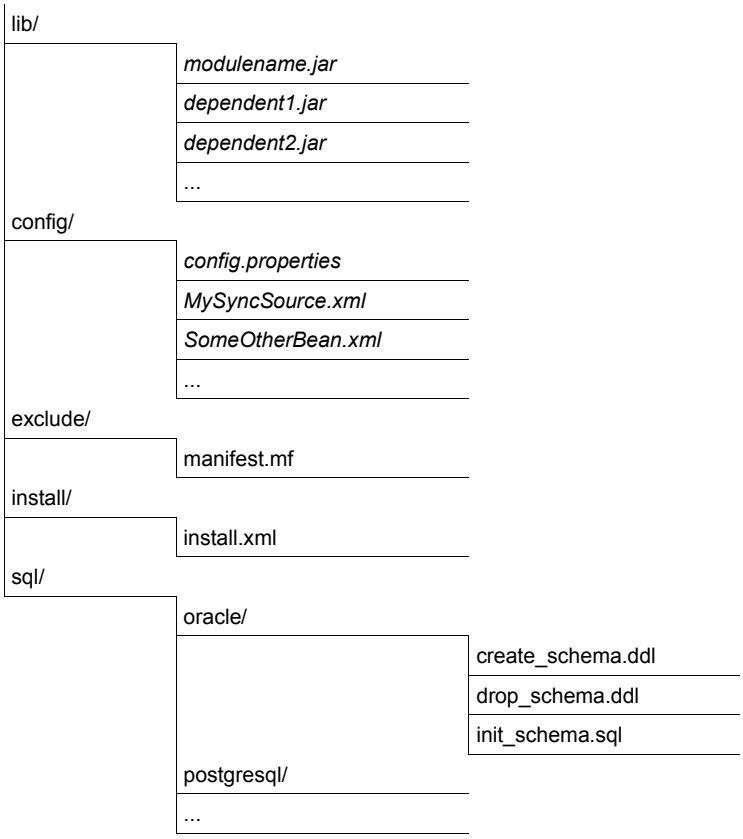

Figure 8 - Module package structure

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. Finally, the *exclude* directory is used to store files that will be used by the installation procedure, but that will not be included in the final server ear.

6.2. Modules, SyncConnectors and SyncSource Types

As already stated, the module is a container for anything related to one or more server extensions. Those extensions may include one or more sync connectors. A *SyncConnector* is an extension to the server intended to support the synchronization of a particular set of data sources. The Funambol's SyncConnector DB, for example, provides a GUI and runtime classes for the synchronization of generic data stored into a RDMS. The Sync4j Foundation module provides a *SyncConnector FileSystem* that allows to synchronize data stored in a directory of the file system. A key piece of software grouped under the umbrella of the SyncConnector is the sync source type. A *SyncSource Type* represents the template from which a real *SyncSource* can be created. For example, the *FileSystemSyncSource* type is the means the SyncServer can synchronize data stored in the file system. However, it does not represent a particular directory to synchronize. To synchronize a specific directory (for instance */data/contacts*) a real SyncSource *instance* must be created and configured

with the wanted directory. Since this is a guide for developers, you can think of a SyncSource Type as a class and of a SyncSource as an instance.

6.2.1. Registering Modules, SyncConnectors and SyncSource Types

Modules, SyncConnectors and SyncSource types are registered filling the following database tables:

- sync4j_module for Module information
- sync4j_connector for SyncConnector information
- sync4j_sync_source_type for SyncSource Type information
- sync4j_connector_source_type for SyncConnector-SyncSource type associations
- sync4j_module_connector for Module-SyncConnector association

Note that the last two tables are used to create the hierarchy Module-SyncConnector-SyncSource Type that you can see in the SyncAdmin.

As an example, we are going to have a look at the foundation module registration. When Sync4j SyncServer is installed, the foundation module is installed too. It brings a SyncConnector called SyncConnector File System, which, in turn, contains the SyncSource type FileSystemSyncSource (Figure 9).

Figure 9 - Foundation Sync4j Module in the SyncAdmin tool

This hierarchy is obtained with the following SQL commands:

1. Module registration:
```
insert into sync4j_module (id, name, description)
values('foundation','foundation-1.0','Foundation ver.1.0');
```
2. SyncConnector registration:
```
insert into sync4j_connector(id, name, description, admin_class)
values('foundation','SyncConnectorFoundation','SyncConnector
Foundation','');
```
3. The SyncConnector Foundation belongs to the foundation-1.0 module:
```
insert into sync4j_module_connector(module, connector)
values('foundation','foundation');
```
4. The FileSystem SyncSource type (fs-foundation) belongs to the SyncConnector Foundation:
```
insert into sync4j_connector_source_type(connector, sourcetype)
values('foundation','fs-foundation');
```
5. Finally, the SyncSource Type registration:
```
insert into sync4j_sync_source_type(id, description, class, admin_class)
values(
 'fs-foundation','FileSystem SyncSource',
 'sync4j.foundation.engine.source.FileSystemSyncSource',
 'sync4j.foundation.admin.FileSystemSyncSourceConfigPanel'
)
```

Note that in the SyncSource Type registration two classes are specified: *FileSystemSyncSource*, which actually implements the SyncSource interface and *FileSystemSyncSourceConfigPanel*, which instead is used to create a new SyncSource instance and to configure it in the SyncAdmin.

We will see how those implementing classes are developed in the following section.

Note also that in this guide very often SyncSource and SyncSource Type are considered synonyms, even if they are in the template-instance relationship seen before.

7. Developing a SyncSource

A SyncSource is the means a set of data is made available to SyncServer for the purpose of the 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 the Sync4j platform 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 the server will be able to process synchronization requests targeted to virtually any data source.

7.1. 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 their own underlying storage.

A SyncSource is identified by a *sourceURI* and usually a *name;* the former is the URI that a SyncML client must specify as target in order to synchronize this particular SyncSource; the latter is a reader-friendly name used for display purposes only. 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 type of data handled by the source.

The most important methods defined by the SyncSource interface are:

method	***description***
beginSync()	This is the first SyncSource method the sync engine calls and it is used to specify who is going to synchronize and which type of synchronization is requested.
endSync()	This is the latest SyncSource method the sync engine calls and it may be used to perform finalization tasks.
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.

method	*description*
setSyncItem/s	Called to insert or update the given item(s).
removeSyncItem/s	Called to remove the given item(s).
getSyncItemFromTwin	Called to find items that represent the same information as the given item. It is used in conflict detection and during slow sync to associate a client item with a server item which is *similar enough*.

Table 5 - SyncSource methods

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

For an example on how to develop a SyncSource see [4] or the Sync4j source code.

7.1.1. Principal and Since Timestamp

SyncSource methods usually require two input parameters in order to retrieve the requested 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 data manipulation to the data set related to the given entity (for example, the contacts of a specific user). If this parameter is null, all items in the datastore are considered for synchronization, regardless the principal they belong to.

In Sync4j SyncServer, a principal is composed of a couple (user,device) 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 during 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()*.

7.1.2. 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 SyncServer provides a standard implementation of a SyncItem represented by the class *sync4j.framework.engine.SyncItemImpl*, which is ready to use and should be enough for the majority of the SyncSources needs. SyncItem defines the following methods:

method	*description*
getKey	Returns the item key.
getMappedKey	Returns the mapped key corresponding to this item's 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.

Table 6 - SyncItem methods

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 SyncServer: *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 its 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 a SyncItem, it must always provide a value for at least the two properties BINARY_CONTENT and TIMESTAMP.

7.1.3. Twin Items

The concept of *twin item* is specific to Sync4j. An item X is a twin of an item Y when from the SyncSource point of view, X and Y represent the same information.
For example, two event items, both at the same time in the same place may be considered the same appointment, even if the associated note is different. Or two vcard objects may be considered the same contact if they have the same first, middle and last name, but different phone number.
Because the SyncSource is the only Sync4j object with knowledge about how data are stored in the data source, the SyncSource is the only component that can select the twins of a given item. This is done calling *getSyncItemsFromTwin()*.

Twin items are necessary in two circumstances:

1. During slow sync
2. During conflict detection

During slow sync, the client sends all its items and the server has to discover which operation the client should apply in order to make its data set look identical to the one stored on the server. Because the two devices are supposed to be out of sync, the server cannot relay on LUID-GUID mappings, and then on a simple keys comparison. In this case, for each item given by the client, the server must search for twin items. If a twin is not found, a delete command will be issued. If at least a twin is found, the server will consider to have such client item and won't send back a command for it.

The same process is valid during conflict detection. When during fast sync a client sends a new item into an add or replace command, the server should first check if this item is conflicting with something

else. Therefore, the server calls the SyncSource's getSyncItemsFromTwin(). If this method returns something, a conflict is decteced and handled accordingly. If no twin is found the item can be added.

The SyncSource developer is free to apply the more appropriated comparison logic accordingly to the type of data the SyncSource deals with. This content based conflict detection can be disabled returning always no twin items.

One final note about twin computation is that it should be kept as simplest as possible, since it may have a big impact on performance during slow sync and conflict detection.

7.1.4. The SyncAdmin Configuration Panel

One of the most interesting feature introduced with SyncServer 4.0 was the SyncAdmin management console. By the mean of the SyncAdmin tool it is possible to perform many administration tasks, including the configuration of the standard Sync4j SyncServer sync sources.
After being logged in in the SyncAdmin and connected to the server, you will see something like what shown in Figure 10. Selecting an existing SyncSource, a configuration panel is displayed in the middle of the window.

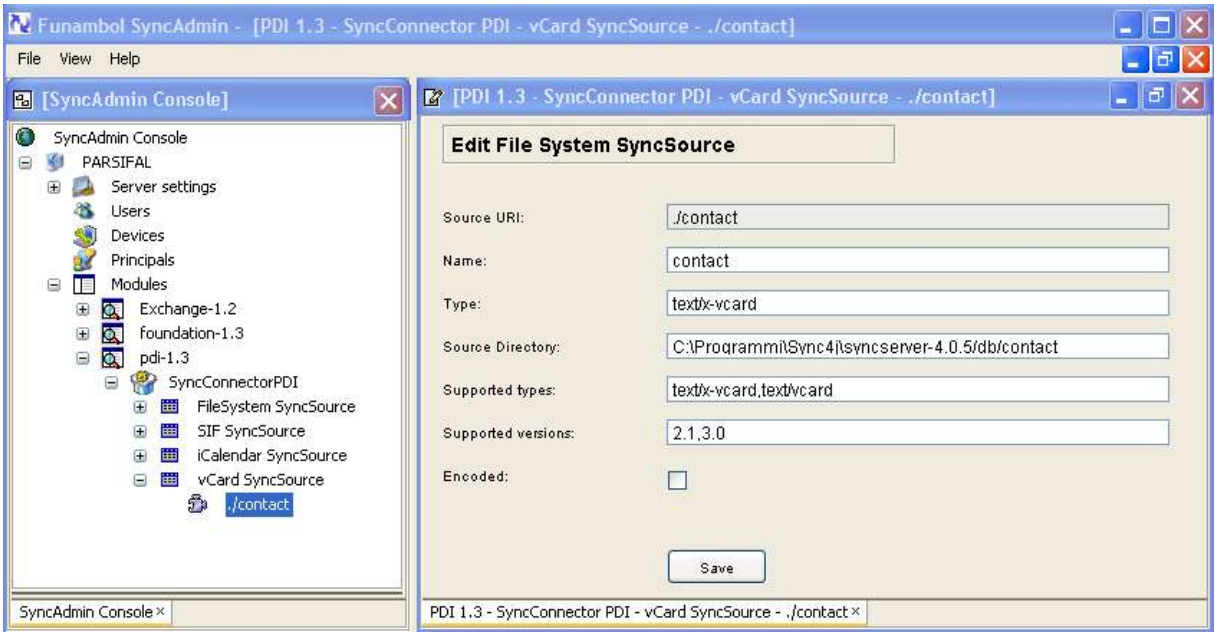

Figure 10 - SyncAdmin showing connectors, modules and SyncSources

The SyncAdmin tools is designed to be extended by developers so that a new custom sync source type can be configured into the SyncAdmin through a custom management panel.
A management panel for a custom sync source is represented by an extension of the abstract class *sync4j.syncadmin.ui.ManagementPanel*. This has the interface described in Table 7.

Public methods	
method	***description***
setState(int state)	Defines if the panel is creating a new sync source instance or editing an existing one.
int getState()	Returns the creation sync source instance state (creation or editing).
setModuleId(id)	Sets the id of the module the SyncSource instance belongs to.
String getModuleId()	Returns the id of the module the SyncSource instance belongs to.
setConnectorId(id)	Sets the id of the connector the SyncSource instance belongs to.
String getConnectorId()	Returns the id of the connector the SyncSource instance belongs to.
setSetSourceTypeId(id)	Sets the id of the SyncSource type the SyncSource instance belongs to.

Public methods	
String getSourceTypeId()	Returns the id of the SyncSource type the SyncSource instance belongs to.
notifyError()	Used to notify the user that an error occurred.
loadSyncSource(source)	Used to display the values of a SyncSource instance.
Private methods	
setSyncSource(source)	Used to notify the SyncAdmin that a SyncSource instance has to be saved.
DeleteSyncSource(sourceURI)	Used to notify the SyncAdmin that a SyncSource instance has to be deleted.

Table 7 - ManagementPanel interface

A SyncSource configuration panel can be in one of the two states *INSERT* or *UPDATE*. The former is for when a new SyncSource is created, the latter for when it is edited.

One important thing to point out here is that the classes that implement both the SyncSource and its configuration panel, are installed on the server at module installation time. The SyncAdmin retrives them on demand via a customized class loader. The SyncAdmin knows about which classes to use, thanks to the SyncSource Type registration seen before; for example, the statement

```
insert into sync4j_sync_source_type(id, description, class, admin_class)
values(
 'fs-foundation','FileSystem SyncSource',
 'sync4j.foundation.engine.source.FileSystemSyncSource',
 'sync4j.foundation.admin.FileSystemSyncSourceConfigPanel'
)
```

tells the server that the fs-foundation SyncSource Type is implemented by the *sync4j.foundation.engine.source.FileSystemSyncSource* class and that its configuration panel is implemented by the class *sync4j.foundation.admin.FileSystemSyncSourceConfigPanel*.

The way the SyncAdmin and the SyncSource management panel interact is pretty simple; the case of creating a new SyncSource is shown in Figure 11.

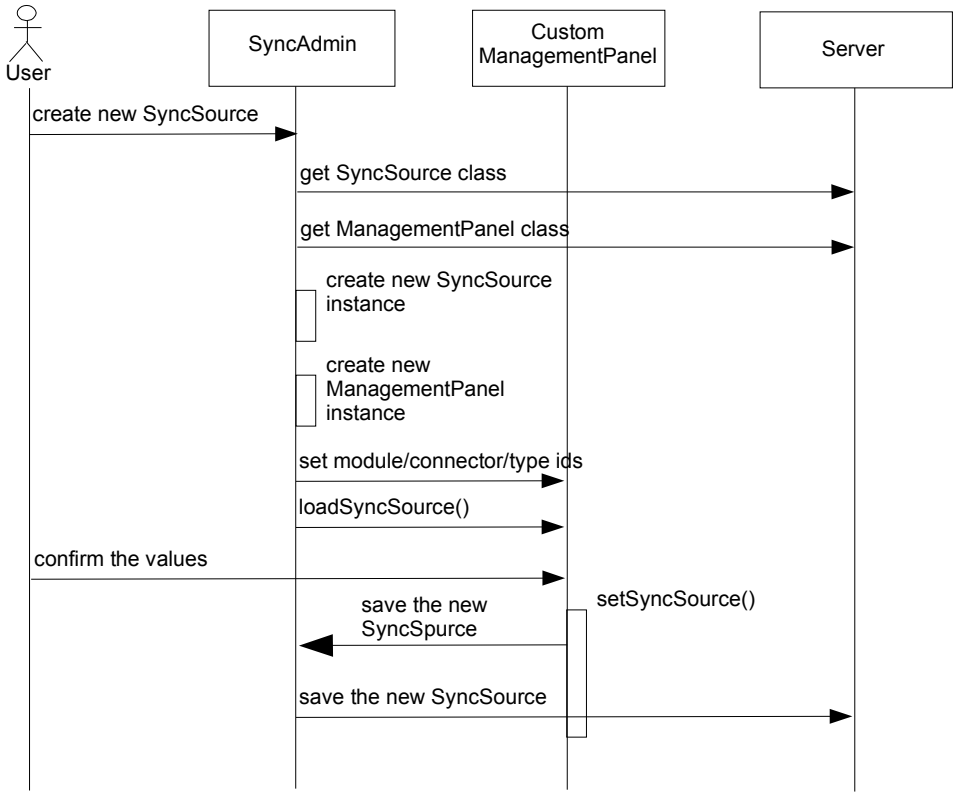

Figure 11 - SyncAdmin-ManagementPanel interaction diagram
First of all, when the user right clicks on a SyncSource type and selects Add, few things happen:

- The SyncAdmin knows the name of the SyncSource and management panel classes, therefore it instantiates a new SyncSource and ManagementPanel objects. If the classes are not found locally, they are downloaded from the server.
- The SyncAdmin displays the configuration panel and calls its setModuleId(), setConnectorId(), setSourceTypeId() passing the appropriate ids.
- The SyncAdmin calls the panel's loadSyncSource() passing the newly created SyncSource instance.

When the user commits the changes (for example pressing a button provided by the UI of the custom management panel), the new values should be sent back to the server. This is done by the *setSyncSource()* method implemented in the ManagementPanel class. The developer does not need to know the details on how this is done, since it is a SyncAdmin specific functionality.
For example, a developer could add the following code to create a button that saves the newly created SyncSource to the server.

```
Jbutton add = new Jbutton("Add");

add.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event ) {
        try {
            validateValues(); // some validation code
            getValues();      // store the values in the UI elements into
                              // the SyncSource instance

            setSyncSource(syncSource);
        } catch (Exception e) {
            error(e.getMessage());
        }
```

```
    }
});
```

The update process is very similar, the only difference is that the SyncSource instance will not be created from scratch; instead, the existing SyncSource instance will be used. Changes to the SyncSource will be reflected on the server with the same mechanism described above.

For a complete example of how to develop a custom management panel see [4].

8. Configuring Sync4j and Sync4j Components

One of the Sync4j SyncServer 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 require a lot of configuration information and possibly an easy way to add module configuration. Configuration files should be easily understandable, accessible and editable.

Sync4j uses mainly three configuration techniques:

- System properties
- Sync4j.properties
- *Server JavaBeans*

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

8.1. System Properties

The only system property used by SyncServer 4.0.5 is sync4j.home which must point to the directory where the Sync4j package is installed (commonly referenced as $SYNC4J_HOME).
This property is specified at JVM invocation time using the -D option. On many systems, it is sufficient to set the JAVA_OPTS environment variable in order to get it included into the JVM launching command.

8.2. Sync4j.properties

This is the main Sync4j SyncServer 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 classes can be instantiated with the properties needed to bootstrap. See the Sync4j SyncServer administration guide[3] for a list of all possible properties and their meanings.

8.3. Server JavaBeans

Many Sync4j SyncServer 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 memory as a properly configured instance.

Sync4j SyncServer makes use of the standard java facility to serialize objects into XML (and to deserialize them from XML). This is achieved by using 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 SyncServer 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 the classpath.

The XML syntax uses the following conventions:

- Each element represents a method call.
- The "object" tag denotes an expression whose value is to be used as the argument to the enclosing element.
- The "void" tag denotes a statement which will be executed, but whose result will not be used as an argument to the enclosing method.
- Elements which contain elements use those elements as arguments, unless they have the tag: "void".
- The name of the method is denoted by the "method" attribute.
- XML's standard "id" and "idref" attributes are used to make references to previous expressions - so as to deal with circularities in the object graph.
- The "class" attribute is used to specify the target of a static method or constructor explicitly; its value being the fully qualified name of the class.
- Elements with the "void" tag are executed using the outer context as the target if no target is defined by a "class" attribute.
- Java's String class is treated specially and is written <string>Hello, world</string> where the characters of the string are converted to bytes using the UTF-8 character encoding.

Although all object graphs may be written using just these three tags, the following definitions are included so that common data structures can be expressed more concisely:

- The default method name is "new".
- A reference to a java class is written in the form <class>javax.swing.JButton</class>.
- Instances of the wrapper classes for Java's primitive types are written using the name of the primitive type as the tag. For example, an instance of the Integer class could be written: <int>123</int>. Java's reflection is internally used for the conversion between Java's primitive types and their associated "wrapper classes".
- In an element representing a nullary method whose name starts with "get", the "method" attribute is replaced with a "property" attribute whose value is given by removing the "get" prefix and decapitalizing the result.

- In an element representing a monadic method whose name starts with "set", the "method" attribute is replaced with a "property" attribute whose value is given by removing the "set" prefix and decapitalizing the result.
- In an element representing a method named "get" taking one integer argument, the "method" attribute is replaced with an "index" attribute whose value the value of the first argument.
- In an element representing a method named "set" taking two arguments, the first of which is an integer, the "method" attribute is replaced with an "index" attribute whose value the value of the first argument.
- A reference to an array is written using the "array" tag. The "class" and "length" attributes specify the sub-type of the array and its length respectively.

8.3.1. 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 class loader (which is instead our server beans loader), is configured to read objects from the configuration path. The config path is built appending "/config" to the sync4j.home system property value. For example, if the sync4j.home is set to "/opt/Sync4j-2.2/syncserver-4.0.5", the config path would be "/opt/Sync4j-2.2/syncserver-4.0.5/config".

8.3.2. Lazy Initialization

When a bean is deserialized from its XML form, the classloader that loads the serialization file calls the empty constructor first and then it sets the bean property values using the setXXX() methods provided by the class. However, some classes need additional operations to be performed in order 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 the SyncServer loads a *LazyInitBean*, after bean instantiation (or deserialization) and configuration (calling the setter methods), it calls the bean's *init()* method, giving the bean the opportunity to complete its initialization.

8.4. How to Configure a Standard Component

Making a change to a configuration bean is as easy as editing a text file. Let's take as example the configuration file for the DBOfficer component. The configuration bean full path is sync4j/server/security/DBOfficer.xml (remember: this path is relative to the configpath) and its content is below:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="sync4j.server.security.DBOfficer">
  <void property="clientAuth">
   <string>syncml:auth-basic</string>
  </void>
  <void property="serverAuth">
   <string>none</string>
  </void>
 </object>
</java>
```

The object element specifies which Java class will be instantiated and the property element sets the corresponding instance property. Therefore, to change the preferred client authentication type, it is sufficient to edit the file, change the clientAuth property and save. The next time this bean will be used, the new configuration value will be picked up.

8.5. How to Create a Custom Configurable Object

With this technique, any Java object can be configured, from a simple Java class to a very complex Java object tree. For example, this configures a String object:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <string>This is a String!</string>
</java>
```

A more interesting example is given, for instance, by the class *sync4j.framework.config.LoggingConfiguration*. The class looks like the following:

```
public class LoggingConfiguration {
    // ------------------------------------------------------------ Private data
    private ArrayList loggers;
    // ------------------------------------------------------------ Constructors

    /** Creates a new instance of LoggingConfiguration */
    public LoggingConfiguration() {
    }

    /**
     * Getter for property loggers.
     *
     * @return Value of property loggers.
     */
    public ArrayList getLoggers() {
        return loggers;
    }

    /**
     * Setter for property loggers.
     *
     * @param loggers New value of property loggers.
     */
    public void setLoggers(ArrayList loggers) {
        this.loggers = loggers;
    }
}
```

A possible configuration file for such a class could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_04" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.config.LoggingConfiguration">
  <void property="loggers">
   <object class="java.util.ArrayList">
    <!--
         sync4j
    -->
    <void method="add">
     <object class="sync4j.framework.config.LoggerConfiguration">
      <void property="append">
       <boolean>true</boolean>
      </void>
      <void property="count">
       <int>1</int>
      </void>
      <void property="fileOutput">
       <boolean>true</boolean>
      </void>
      <void property="level">
       <string>INFO</string>
      </void>
      <void property="limit">
```

```
       <int>100</int>
      </void>
      <void property="name">
       <string>sync4j</string>
      </void>
      <void property="pattern">
       <string>logs/syncserver.log</string>
      </void>
     </object>
    </void>

    <!--
         sync4j.engine
    -->
    <void method="add">
     <object class="sync4j.framework.config.LoggerConfiguration">
      <void property="append">
       <boolean>true</boolean>
      </void>
      <void property="count">
       <int>1</int>
      </void>
      <void property="inherit">
       <boolean>true</boolean>
      </void>
      <void property="level">
       <string>INFO</string>
      </void>
      <void property="limit">
       <int>100</int>
      </void>
      <void property="name">
       <string>sync4j.engine</string>
      </void>
      <void property="pattern">
       <string>logs/syncserver.engine.log</string>
      </void>
     </object>
    </void>

   </object>
  </void>
 </object>
</java>
```

NOTE: see later how to create such a file authomaically.

8.6. How to Get a Configured Instance

Configuration beans are accessed through the singleton *sync4j.framework.config.Configuration* object. For example, to instantiate a configured DeviceInventory instance, use the code below.

```
Configuration c = Configuration.getConfiguration();

LoggingConfiguration logging = c.getBeanInstanceByName("sync4j/server/logging/logging.xml");
```

8.6.1. Tips and Tricks

It is not necessary to write a configuration file by hands from scratch. To write a bean instance for the first time use the *sync4j.framework.tools.beans.BeanFactory*'s *saveBeanInstance()* method to save a configured instance into a file. For example:

```
Jbutton b = new Jbutton(“press me”);

BeanFactory.saveBeanInstance(b, new File("button.xml");
```

The result is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_04" class="java.beans.XMLDecoder">
 <object class="javax.swing.JButton">
  <string>press me</string>
 </object>
</java>
```

9. Customizing Message Processing

This section explains how to extend the SyncServer customizing the processing of incoming and outgoing messages.

9.1. Overview

The OMA DS protocol is an XML-based protocol. This means that each OMA DS message is an XML document.
When a OMA DS message reaches the DM Server, it passes through some transformations. These are divided into *XML level transformations* and *message transformations*. The former works on the message in its XML representation, the latter on a Java representation of the message.

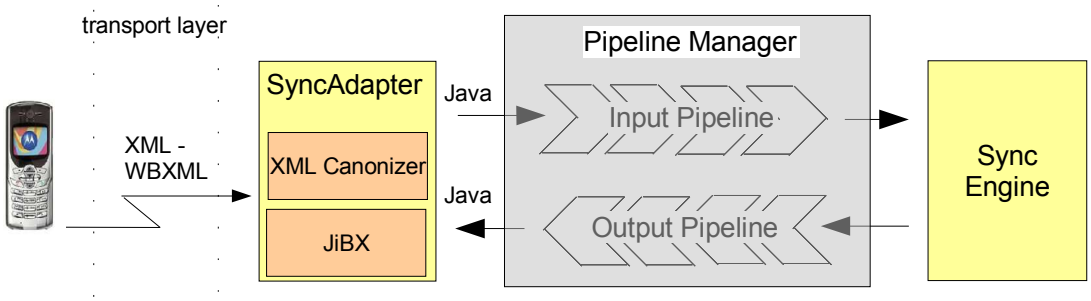

Figure 12 - Message processing architecture

In order to save bandwith and processing power, OMA DS messages can be also WBXML encoded. Regardless how the message is encoded, its content is first delivered to a SyncAdapter component by the transport layer (Figure 6). The SyncAdapter first translates the message in XML if it was WBXML encoded and then the XML message is reduced to a “canonical” form in order to get rid of device specific singularities. XML canonization is the XML level transformation.

Even when in the canonical XML form, the message is still hard to manipulate, since XML needs to be parsed. Plus, each component that needs to access any of the OMA DM message elements would have to parse the XML again, with a big impact on performance. For these reasons, the canonic XML message is translated into an object tree that represents exactly the message.

After an incoming message has been translated into an object tree, it passes through the *input message processing pipeline* before it gets to the SyncEngine. This gives the opportunity of further processing the message when it is in a more manageable representation. In a similar way, a response message going out from the SyncEngine, passes through the *output message processing pipeline* before getting translated to its XML (and then WBXML) representation.
The input and the output pipelines are completely customizable, so that custom message pre and post processing can be easily added to the system.

Input and output message processing components are also called "*synclets*".

9.2. Preprocessing an Incoming Message

To preprocess an incoming message we have to create an input processor component and to configure the pipeline manger accordingly. This is described below.

9.2.1. Creating an Input Synclet

An input synclet is a class that implements the *sync4j.framework.engine.pipeline.InputMessageProcessor* interface. This interface defines just one method: *void preProcessMessage(MessageProcessingContext context, SyncML msg). context* is a parameter that is shared amongst all the synclets (both input and output) involved in the message processing. *msg* is the object tree representing the SyncML message. The object tree is composed of instances of classes in the *sync4j.framework.core* packages and represents a hierarchical view of the message.

For example, the synchronization message below will be translated in the object hierarchy of Figure 12.

```
<SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>12345678</SessionID>
<MsgID>2</MsgID>
<Target><LocURI>http://localhost</LocURI></Target>
<Source><LocURI>syncml-phone</LocURI></Source>
<Cred>
 <Meta><Type>syncml:auth-basic</Type></Meta>
 <Data>Z3Vlc3Q6Z3Vlc3Q=</Data>
</Cred>
</SyncHdr>
<SyncBody>
<Alert>
<CmdID>1</CmdID>
<Data>200</Data>
<Item>
<Target><LocURI>test</LocURI></Target>
<Source><LocURI>test</LocURI></Source>
<Meta>
<Anchor>
<Last>234</Last>
<Next>276</Next>
</Anchor>
</Meta>
</Item>
</Alert>
<Final/>
</SyncBody>
</SyncML>
```

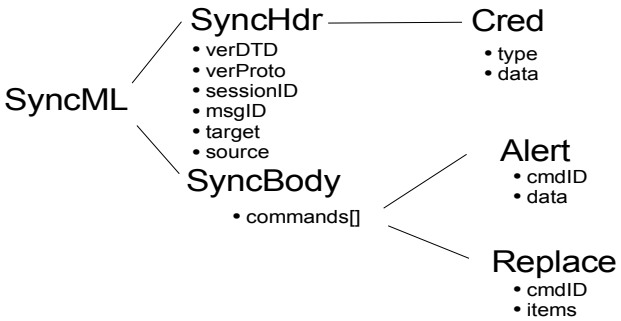

Figure 13 - sync4j.framework.core object tree example

```
public class LoggingSynclet
implements InputMessageProcessor {
    // ------------------------------------------------------------ Private data
    private static final Logger log = Sync4jLogger.getLogger("engine");

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

    /**
     * Logs the input message and context
     *
     * @param processingContext the message processing context
     * @param message the message to be processed
     *
     * @throws Sync4jException
     */
    public void preProcessMessage(MessageProcessingContext processingContext,
                                  SyncML                    message)
    throws Sync4jException {
       if (log.isLoggable(Level.INFO)) {
            log.info("------------------------------------------------------------");
            log.info("Input message processing context");
            log.info("");
            log.info(processingContext.toString());
            log.info("------------------------------------------------------------");

            log.info("Input message");
            log.info("");
            log.info(Util.toXML(message));
            log.info("------------------------------------------------------------");

            //
            // Sets the device id to foo
            //
            message.getSyncHdr().getSource().setLocURI("foo");


       }
    }
}
```

Scope of this synclet is pretty evident: it just logs to the sync4j.engine logger the input message. Plus, it modifies the message setting the device id to “foo”.

9.2.2. Configuring an Input Synclet

The input synclet so created, is configured telling the Pipeline Manager to insert the new synclet in the input pipeline. This is done like in the following server side JavaBeans.

```
<?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="1">
      <void index="0">
        <object class="sync4j.foundation.synclet.InputMessageProcessor"/>
      </void>
    </array>
  </void>
  <void property="outputProcessors">
    <array class="sync4j.framework.engine.pipeline.OutputMessageProcessor" length="0"/>
  </void>
 </object>
</java>
```

9.3. Postprocessing an Outgoing Message

To postprocess an outgoing message we have to create an output processor component and to configure the pipeline manger accordingly. This is described below.

9.3.1. Creating an Output Synclet

An output synclet is a class that implements the *sync4j.framework.engine.pipeline.OutputMessageProcessor* interface. This interface defines just one method: *void postProcessMessage(MessageProcessingContext context, SyncML msg).*
The concepts behind the output message processing are the same as per input processing.

An example of an output synclet is the class shown below. The scope of this synclet is to inject into the outgoing message a Get command to request client capabilities:

```
public class AddGetSynclet
implements OutputMessageProcessor {
    // ------------------------------------------------------------- Constants

    public static final String PARAM_SESSION_ID = "sid";


    // ------------------------------------------------- OutputMessageProcessor

    public void postProcessMessage(MessageProcessingContext processingContext,
                                   SyncML                   message          )
    throws Sync4jException {
        AbstractCommand[] commands = message.getBody().getCommands();

        AbstractCommand[] newCommands = new AbstractCommand[commands.length+1];

        Meta meta = new Meta();
        meta.setType("application/vnd.syncml-devinf+xml");

        Item item = new Item(
            new Target("/devinf11"),
            null,
            null,
            null,
            false
        );

        Get get = new Get(
            new CmdID(newCommands.length),
            false,
            null,
            null,
            meta,
            new Item[] { item }
        );

        System.arraycopy(commands, 0, newCommands, 0, commands.length);
        newCommands[commands.length] = get;
    }
}
```

9.3.2. Configuring an Output Synclet

The output synclet so created, is configured telling the Pipeline Manager to insert the new synclet in the outut pipeline. This is done like in the following server side JavaBeans (keeping the same configuration of the input pipeline as the previous example).

```
<?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="1">
      <void index="0">
        <object class="sync4j.foundation.synclet.InputMessageProcessor"/>
      </void>
    </array>
  </void>
  <void property="outputProcessors">
    <array class="sync4j.framework.engine.pipeline.OutputMessageProcessor" length="1">
      <void index="0">
        <object class="com.foo.synclet.AddGetSynclet"/>
      </void>
    </array>
  </void>
 </object>
</java>
```

10. References and Resources

10.1. 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 SyncServer 4.0 Administration Guide*
[4] *Sync4j SyncServer 4.0 Module Development Tutorial*

10.2. Resources

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