Developing and Managing Mobile Applications with SyncML and Funambol

September, 2007

Table of Contents

Preface

Why a book on mobile application development and management?

Because mobile is the next big thing. Over one billion mobile devices will be sold this year. Half of the population on earth will have a mobile device by the end of the year. The diffusion of mobile devices will surpass by every measurement that of personal computers. If the Internet enabled an explosion of web applications, the new era of wireless will create incredible demand for mobile applications and services.

How will developers build these applications? As they did for the web, they will use standards. SyncML is to mobile applications what HTML and HTTP are to web applications. SyncML allows you to build an application that works even when a device is disconnected – a key usability factor for mobile applications.

What will developers use to build mobile applications? They will use open source. Open source is freely available, it supports more devices than proprietary software and it is widely used, resulting in more expertise and resources from a global community.

Many mobile applications are being built using Funambol, the leading mobile open source project in the world. Funambol had been downloaded over one million times, with downloads exceeding 70,000 per month.

We hope you enjoy reading this document, learning about mobile application development, the Open Mobile Alliance (OMA) Data Synchronization (DS) and Device Management (DM) standards (aka SyncML) and the Funambol open source project.

1. Mobile Application Development and Management

One of the primary challenges mobile operators, enterprises, and end users encounter with mobility is synchronizing the data accessed or used by a wireless device.

There has recently been an impressive increase in the number of personal devices. Many people own more then one desktop PC, a VCR, a DVD player, a satellite set-top box, Tivo, etc. When traveling, they often carry a cell phone, laptop, PDA and MP3 player. These devices partially share the same data yet keeping their information in sync is a difficult task. For example, few users have the phone numbers in Outlook consistent with the ones in their mobile phone.

If we extend the concept of data synchronization to multiple mobile devices and corporate servers, we begin to see the magnitude of the problem. Synchronization is a fundamental component in every wireless project: -- data must be consistent when accessed by multiple users who are not always connected. Data changed in one device should be reflected in all others. When a customer phone number is added in someone's mobile device, it should be reflected in the corporate CRM system and in their colleagues' cell phones as well.

A key reality, sometimes obscured by marketing, is that wireless devices are not always connected. Coverage is not yet universal, connections often get dropped and roaming is expensive and difficult. New network technologies are expected to bring the always-connected dream to users. However, as it happens in the wired world, a user might not be willing to download all of their applications and data on the fly.

One of the reasons behind the failure of network computing is that users prefer to have their data and applications on the device they own versus leaving it on a machine owned by someone else. Further, they do not want to rely on the network, even if it is highly reliable. If the network computing idea did not work well in the wired world, it is even less likely to succeed on devices that have limited memory and processing power. In particular, if a mobile operator's revenue model is based on charging per byte or minute (i.e. the more applications or data downloaded, the more the user pays), users will naturally prefer downloading once and syncing over paying per each access.

Synchronization is crucial: the data I own is on my device and I update it when I want (manually, scheduled, application controlled, etc.). I only pay for downloading information I really need. Applications need to work offline and synchronize with the world when they are back online. Real-time access is left to specific transactions that require immediate action.

An even greater challenge for IT managers will be to remotely provision a device with the proper applications as well as ongoing configuration management. This must occur in an increasingly complex environment with a plethora of varying devices and operating systems. A smartphone has become a tiny personal computer with software that needs to be installed, updated and configured remotely and on the fly. If a mobile device gets lost, its critical data should be erasable with a remote command.

In sum, the requirements for a mobile application include:
- Data Synchronization
    - Offline use
    - Multi-directional synchronization on demand
- Device Management
    - Application Provisioning
    - First installation based on device characteristics
    - On-the-fly application upgrade (e.g. bug fixes deployed)
    - Remote personalization and customization of devices
    - Remote monitoring of devices (e.g. are they working properly?)
    - Remote device repair

The three aspects of the equation – data synchronization, application provisioning and device management – are often approached separately. They are, however, intrinsically part of a common goal: keeping a device synchronized with the data, applications and configuration stored on a remote server.

With this in mind, the open source Funambol platform includes a server coupled with small footprint clients specific to different devices as well as connectors to various data sources.

Funambol is based on SyncML (Synchronization Markup Language), the Open Mobile Alliance (OMA) standard for Data Synchronization (DS) and Device Management (DM). Many leading device manufacturers now include a SyncML client in their new devices. This is driving the need for a synchronization infrastructure based on the SyncML protocol.

Funambol is the leading mobile open source project in the world, with more than one million downloads by 10,000 contributors in 200 countries. Usage by so many people combined with the continuous community feedback inherent in open source projects guarantees high quality software and fast development cycles. In parallel with open source licensing, Funambol software is also available under a commercial license that removes open source restrictions (such as sharing code changes).

2. Advantages of Synchronization

Connecting to a network when working outside the office is not always feasible. It is often difficult to maintain a voice call while traveling along some of the most common highways; data coverage is even worse. Add to this poor coverage in buildings and it quickly becomes apparent that 100% network connectivity is still a long ways off.

As a result, the best solution for companies that need to deploy mobile applications is to keep them up-to-date and operational even when not online. Additionally, these applications can take advantage of a mobile device's processor and memory, delivering superior responsiveness versus a browser-based application. Sync sessions can occur in the background when connectivity is available, transferring only the information that has changed since the last sync.

Why Synchronization's Time Has Arrived

If a local application that synchronizes is more effective then a network only application, why are they not more widespread? It's basically a hardware issue. To gain the advantages of synchronization, you need to keep the bulk of data and the application itself on the device and only distribute the changes. For many applications, this requires that the mobile device be capable of storing and processing hundreds of megabytes of data.

Notebooks have had this capacity for years but only recently have smartphones and other mobile devices become available with this type of power. This pace of change is accelerating with smartphones with more storage and processing capacity.

In addition to more powerful hardware, synchronization is becoming more common due to the inclusion of SyncML in hundreds of millions of mobile devices. This means that sync software does not need to be downloaded to the device over the air or using cables, making it much easier and quicker to synchronize data.

SyncML-based Mobile Applications

Here are several examples of mobile applications that involve syncing data.

Telcos. Obvious consumer applications include mobile backup, PIM sync and mobile email. In addition to these basic services, several others can be provided via a SyncML server. For example, mobile operators can offer over the air (OTA) firmware upgrades and device management. Consider this example: an mobile virtual network operator (MVNO) distributes a generic mobile phone. When a user connects for the first time, the device automatically receives branded wallpaper, ringtones and a welcome message. The MVNO can also provide personal management services such as frequent flyer applications with flight schedule updates, currency converters with up-to-date rates, infotainment distribution of sports, news, movie schedules and much more.

Insurance. Brokers need to access a vast amount of information on customers as well as pricing and features of their products. Such information can change daily, hence the advantage of an application that can be updated in the morning and used throughout the day, whether online or offline.

Real Estate. Agents need to be out of the office most of the time to show houses and meet customers. At the same time, they need a variety of information with them: searchable lists of houses on the market, pictures, maps and client information. This information needs to be updated often. After every appointment, the agent has to insert into the company system new customer requirements, final contract items and download an updated list of properties that may change every few hours.

Sales Force Automation. Salespeople are primary users of mobile applications. To enhance their productivity, it is critical to relay orders to corporate as well as to receive product catalog and pricelist changes while in the field. They cannot wait to get back to the office to have the most current information.

Field Maintenance. Crew productivity at big industrial sites can be greatly enhanced by mobile applications. Crews need all the information about the tickets they are going to work on. When they are done, they need to update the status of the job so it can be closed or redirected to somebody else. Less time spent on paperwork means more time solving problems. Up to the minute status means fewer truck rolls to problems that have already been solved.

Intranets. The use of portals and intranets as a source of organizational information and policies in everyday corporate life has become common practice. Unfortunately, access to them usually stops at the entrance to the building. By extending their reach to mobile devices outside of the office and onto personal mobile phones, the corporation's resources are leveraged.

Vending Machine Automation. Vending machines management benefits greatly from remote synchronization. A SIM can be installed in each machine and programmed to store all the transactions, inventory levels and error conditions. Daily, the machine can sync with a central server, allowing the main office to be aware of its status and to tune maintenance and re-stocking schedules. This is the last mile of just-in-time manufacturing and distribution.

Delivery Fleet Management. Delivery personnel need to track their trips and the packages they are delivering and picking up. Often, deliveries are made to locations outside of data connectivity range. The tracking system needs to be always available, yet updated when possible.

3. SyncML: The Open Mobile Alliance Standard

In December, 2000, Ericsson, IBM, Lotus, Motorola, Nokia, Palm, Psion and Starfish Software formed the SyncML initiative to accelerate the market's vision of ubiquitous data access from any networked device. Their goal was to create a common synchronization protocol that could be used industry-wide. They worked with end users, device manufacturers, data providers, infrastructure developers, application developers, and service providers to define a common mobile data synchronization protocol which would satisfy the resource constraints of mobile devices and wireless networks and still provide the extensibility to support a range of applications and data types. The result was SyncML.

In 2002, the original SyncML initiative was consolidated into the Open Mobile Alliance (OMA) under the Data Synchronization Working Group (OMA DS) and Device Management Working Group (OMA DM). The OMA is "the leading industry forum for developing market driven, interoperable mobile service enablers". OMA was formed in June 2002 by nearly 200 companies including the world's leading mobile operators, device and network suppliers, information technology companies and content and service providers.

The mission of the Open Mobile Alliance is to facilitate global user adoption of mobile data services by specifying market driven mobile service enablers that ensure service interoperability across devices, geographies, service providers, operators, and networks, while allowing businesses to compete through innovation and differentiation.

SyncML Data Synchronization (OMA DS)

SyncML Data Synchronization is defined as "the process of making two sets of data look identical"[**Error! Reference source not found.**]. To achieve this goal, starting from a coherent initial state (for example after synchronization). the two sets of data need to be modified accordingly and a conflict resolution policy must be followed. A conflict arises when the same data element is modified in both sets in an inconsistent way.

What is a synchronization protocol? First of all, it is a way to communicate the modifications between different data sets over a period of time.

The primary characteristics of a synchronization protocol are the addressing of the data sets (or database) and single records (in order to identify a particular database or a single record in a database), the definition of the standard commands needed to communicate modifications and the definition of the ways these modifications can be exchanged (required features, transport protocols and so on).

SyncML is "an initiative to develop and promote a single, common data synchronization protocol that can be used industry-wide". Moreover, SyncML is a specification for a common data synchronization framework based on the exchange of XML-based

messages between networked devices. SyncML is sponsored by companies like Motorola, Nokia, Ericsson, IBM and many others.

The SyncML specifications include:
- A representation protocol, which defines the XML messages format
- A synchronization protocol which defines how SyncML messages shall be combined together in order to accomplish a synchronization session
- The binding for different transport protocols (HTTP, OBEX)
- A protocol for device management
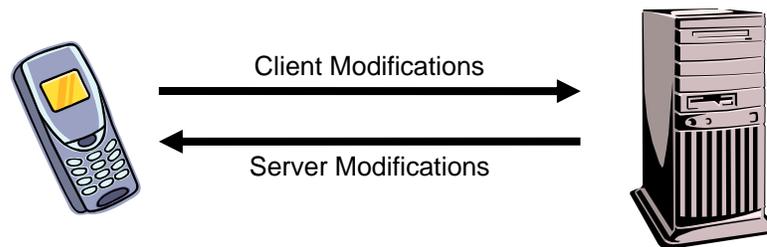
Figure 1 - SyncML Client and Server

SyncML Client and Server

Figure 1 shows a simple case where a cellular phone is a client connecting to a server. The phone transmits to the server a SyncML message, including the most recent modifications; the server synchronizes its data set (based on the changes received by the client, in terms of commands like Add, Delete, and Replace) and responds with a SyncML message including its own modifications. This is a trivial example but it is useful to show the role assumed by the two devices during synchronization.

The SyncML Client contains a synchronization agent and typically sends its modifications first. The SyncML Server is the device implementing both the server-side synchronization agent and the synchronization logic including the procedures to interpret the modifications, to discover and manage conflicts and to generate return messages.

Synchronization Modes

SyncML defines seven synchronization modes.

Two-way sync: the most common synchronization mode whereby client and server exchange modifications on their data. The client sends the modifications first and then the server returns its own.

Slow sync: this is a special case of two-way sync whereby all the client database records are compared, field by field, with the ones on the server. That means the client sends the full database to the server which in turn analyzes all records detecting

missing items that need to be sent back to the client. It is typically used to recover from a situation where it is impossible to resolve synchronization conflicts and to re-set to a consistent state between the client and the server. Another case when slow sync is used occurs when client and server detect that they are out of sync so that a correct state must be recreated from scratch.

One-way sync from the client only: the client sends its modifications to the server but no records from the server are expected or returned.

Refresh sync from client only: the client sends its entire database to the server which replaces its database with the data sent from the client.

One-way sync from the server only: the server sends its modifications to the client but no client modifications are expected in return.

Refresh sync from server only: the server sends its entire database to the client. The client replaces the local set of data with the received one.

Server alerted sync: the server sends a synchronization alert to the client notifying it of the synchronization mode to be used.

SyncML Basics

In the following, we will discuss the basic elements needed to understand data synchronization in general and SyncML synchronization in particular. The following concepts will be presented:
• Sync Anchors
• ID Mapping
• Conflicts
• Security
• Addressing
• Devices and Services Addressing
• Databases Addressing
• Item Addressing
• Device Capabilities

Sync Anchors: to allow a sanity check on the synchronization state between two devices (was the last synchronization successfully completed?), SyncML makes use of two synchronization anchors, next and last. Each database "owns" the anchors that are sent to the other device during the synchronization initialization. Last represents the last synchronization successfully performed. Usually it is a timestamp, but it could be a different identifier such as a session identifier. Next is a tag representing the current synchronization. Client and server exchange their anchors at the beginning of the synchronization process and are required to store the next anchor of the counterpart at the end of the sync process. Using this information, the server can detect if the last synchronization encountered problems and did not end correctly. For example, if the

given Last is equal to the stored anchor, the last synchronization was successfully terminated; if the anchors do not match, client and server are out of sync and need to take action to resolve the issue. For this to work, it is important that the next anchors are saved into the local repository only at the end of a successful synchronization.

ID Mapping: in a multi-device scenario, client and server must be able to create and deal with local item ids independently. Therefore, the server and client may use a different identifier for the same item. Meanwhile, the server needs to keep a mapping table between the client Local Unique Ids (LUID) and the server Global Unique Ids (GUID) (Figure 2).
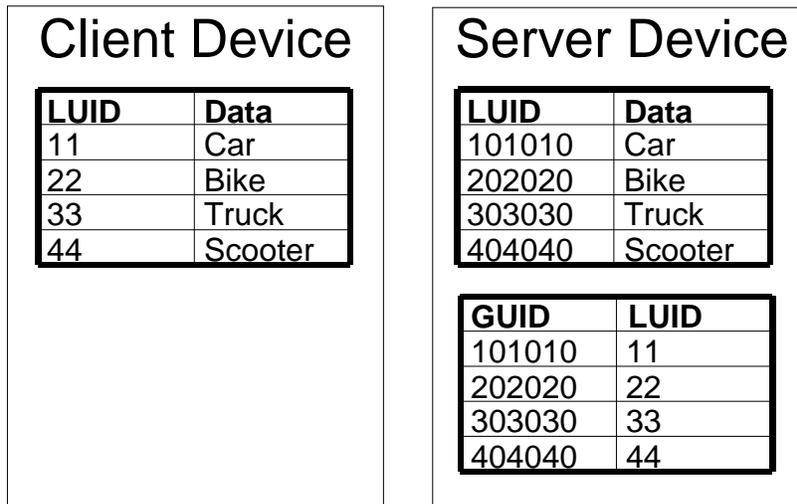
Client Device

LUID	Data
11	Car
22	Bike
33	Truck
44	Scooter

Server Device

LUID	Data
101010	Car
202020	Bike
303030	Truck
404040	Scooter

GUID	LUID
101010	11
202020	22
303030	33
404040	44

Figure 2 - LUIG-GUID mapping example

Note that LUIDs are always created by the client device and only later communicated to the server with the SyncML command Map.

Conflicts: A modification conflict arises when the same item has been modified on both server and client. When this happens, the synchronization engine has the responsibility to resolve the conflict. The client is notified of the error condition by means of a status code. For example, the following message represents a case where the server notifies the client about a conflict resolution:
<Status>
<CmdID>1</CmdID>
<MsgRef>1</MsgRef>
<CmdRef>2</CmdRef>
<Cmd>Replace</Cmd>
<SourceRef>1212</SourceRef>
<Data>208</Data> <!-- Conflict, originator wins -->
</Status>

Security: SyncML mandates two authentication methods, basic and MD5.

Addressing: Defines the way entities involved in synchronization (databases, items, etc.) can be addressed. This is achieved using a naming convention based on URIs.

For example a typical server addressing URI might be:
```
<Source>
<LocURI>http://sync.syncml.org/sync-server</LocURI>
</Source>
```

A client could use a different addressing identifier such as its IMEI number:
```
<Source>
<LocURI>IMEI:493005100592800</LocURI>
</Source>
```

A database is addressed with a URI as described in the SyncML specifications ([**Error! Reference source not found.**] and [**Error! Reference source not found.**]). Note that the URIs can be either absolute or relative:
```
<Sync>...
<Target>
<LocURI>./calendar/james_bond</LocURI>
</Target>
...</Sync>
<Sync>
<Target>
<LocURI>http://www.syncml.org/sync-server/calendar/james_bond</LocURI>
</Target>
...</Sync>
```

Finally, even a single item of a database can be identified with the same method:
```
<Item>...
<Source>
<LocURI>101</LocURI>
</Source>
...</Item>
```

Device Capabilities: SyncML defines how devices can exchange information about their capabilities during the initialization phase. Note that the synchronizing devices are not required to send their capabilities unless requested by their counterpart. The device capabilities exchange is very useful at the server side, because the server can apply proper optimizations according to the remote device resources and features (e.g. database types, available memory, transmission speed). One thing to consider in this context is that capabilities information can be quite large. For this reason, the device capabilities should be sent only if requested. In addition, the information should conform to the capabilities of the counterpart. For example, if the client notifies the server that it

does not support the data format vCard3.0, the server should not insert in its response the vCard3.0 properties it supports.

SyncML Synchronization

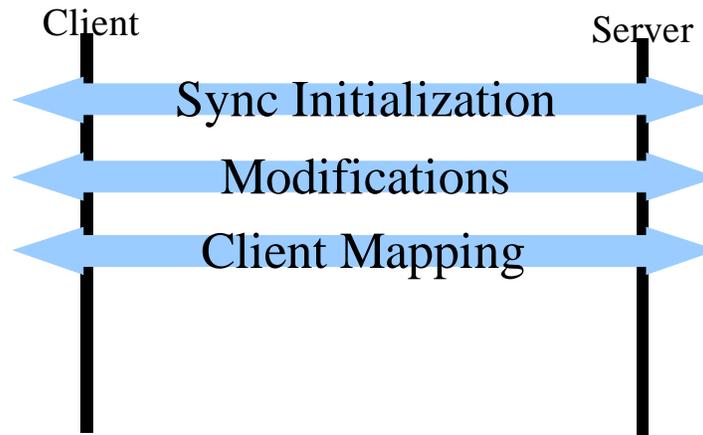The complete SyncML synchronization process is shown in Figure 3.

Figure 3 - Complete SyncML synchronization process

The process consists of a three step sequence: initialization, modifications and client mapping. Each step of the synchronization process is a packet, which can be composed of multiple messages. This is necessary to meet the particular requirements of the transport protocol and to support connection and device limitations.

SyncML specifies that before sending data modifications, the synchronizing counterparts exchange an initialization packet specifying the device characteristics and the synchronization requirements (for example, which databases need to be synchronized, which type of sync is needed or which protocol features are supported). Note that the initialization packet can be merged with the modification package, thereby reducing the chattiness of the communication. The drawback is that the synchronization process cannot be optimized as it could be if the information capabilities were available before performing the synchronization analysis.

A SyncML message is a well formed XML document (but not necessarily valid) with the element <SyncML> as root element and container for all the other tags. A single message is represented by a header within the element <SyncHdr> and a body enclosed in a <SyncBody> element (Figure 4).

```
<SyncML>
<SyncHdr>
…
</SyncHdr>
<SyncBody>
…
</SyncBody>
</SyncML>
```

Figure 4 - A SyncML message

Note that since the XML message is not validated, the XML declaration and prolog can be omitted. The header contains routing information, database addressing and protocol versioning (Figure 5), while the body contains one or more synchronization commands.

```
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>GWQKAKA</SessionID>
<MsgID>msg21</MsgID>
…
</SyncHdr>
```

Figure 5 - SyncHeader example

Each command is represented by a specific XML element, which can be a container for sub-elements, data items or meta information. SyncML commands can be divided into request commands and response commands (Figure 6).

```
<SyncBody>
<Add>
<CmdID>cmdABC</CmdID>
…
<Item>…</Item>
</Add>
</SyncBody>
```

Figure 6 - SyncBody example

Request commands include Add, Alert, Atomic, Copy, Delete, Exec, Get, Map, Put, Replace, Search, Sequence and Sync. Response commands include Status and Results.

Note that the protocol does not specify any semantic for these operations. As a result, an Add operation could have a different meaning on a different device.

For a brief description of each command, see the table below:

Add	Add items to a database
Alert	Notifies counterpart of intention to synchronize a particular database
Atomic	All items included in an Atomic command must either all succeed or fail
Copy	Copy the items into the destination database
Delete	Delete the included items from the destination database
Exec	Execute a program on the destination device
Get	Request information from the remote device
Map	Update LUID-GUID mapping
Put	Send information (such as capabilities) to the device
Replace	Replace the included items into the destination database
Search	Specify a search on the other device
Sequence	The included commands must be executed sequentially
Sync	The container for the modification commands
Status	Status notification for execution & interpretation of any other command
Results	Contains Get or Search results

Example Synchronization. Here is a request message for a two-way sync:

```
<SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>2</MsgID>
<Target><LocURI>http://www.syncml.org/sync-server</LocURI></Target>
<Source><LocURI>IMEI:493005100592800</LocURI></Source>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>1</MsgRef><CmdRef>0</CmdRef><Cmd>SyncHdr</Cmd>
<TargetRef>IMEI:493005100592800</TargetRef>
<SourceRef> http://www.syncml.org/sync-server </SourceRef>
<Data>212</Data> <!--Statuscode for OK, authenticated for session-->
</Status>
<Status>
<CmdID>2</CmdID>
<MsgRef>1</MsgRef><CmdRef>5</CmdRef><Cmd>Alert</Cmd>
<TargetRef>./dev-contacts</TargetRef>
<SourceRef>./contacts/james_bond</SourceRef>
<Data>200</Data> <!--Statuscode for Success-->
<Item>
<Data>
<Anchor xmlns='syncml:metinf'><Next>200005022T093223Z </Next></Anchor>
</Data>
</Item>
</Status>
<Sync>
<CmdID>3</CmdID>
<Target><LocURI>./contacts/james_bond</LocURI></Target>
<Source><LocURI>./dev-contacts</LocURI></Source>
<Meta>
<Mem xmlns='syncml:metinf'>
<FreeMem>8100</FreeMem>
<!--Free memory (bytes) in Calendar database on a device -->
<FreeId>81</FreeId>
<!--Number of free records in Calendar database-->
</Mem>
</Meta>
<Replace>
<CmdID>4</CmdID>
<Meta><Type xmlns='syncml:metinf'>text/x-vcard</Type></Meta>
<Item>
<Source><LocURI>1012</LocURI></Source>
<Data><!--The vCard data would be placed here.--></Data>
</Item>
</Replace>
</Sync>
<Final/>
</SyncBody>
</SyncML>
```

Notes:

The request header contains the version number of the protocol, the DTD utilized, the session and message ids and the addressing of the target server. Target indicates the destination server and Source indicates the client identifier.

The body starts with a <Status> command. Note that the reported message is a modification message; we are assuming the initialization already took place.

The next <Status> responds to an <Alert> command requesting the synchronization of the server database ./contact/james_bond.

Note the use of the next anchor as described earlier.

The next command is a <Sync> command containing the synchronization operations to perform. <Target> and <Source> specify the databases, while the <Meta> element includes information regarding the client status.

Finally, the command <Replace> represents the required operation -- replace the inner items with the data contained in <Data>. Note that the <Meta> element specifies the data type of the data to be stored.

This is an example of a response:

```
<SyncML>
<SyncHdr>
<VerDTD>1.1</VerDTD>
<VerProto>SyncML/1.1</VerProto>
<SessionID>1</SessionID>
<MsgID>2</MsgID>
<Target><LocURI>IMEI:493005100592800</LocURI></Target>
<Source><LocURI>http://www.syncml.org/sync-server</LocURI></Source>
</SyncHdr>
<SyncBody>
<Status>
<CmdID>1</CmdID>
<MsgRef>2</MsgRef><CmdRef>0</CmdRef><Cmd>SyncHdr</Cmd>
<TargetRef>http://www.syncml.org/sync-server</TargetRef>
<SourceRef>IMEI:493005100592800</SourceRef>
<Data>200</Data>
</Status>
<Status><!--This is a status for the client modifications to the server.-->
<CmdID>2</CmdID>
<MsgRef>2</MsgRef><CmdRef>3</CmdRef><Cmd>Sync</Cmd>
<TargetRef>./contacts/james_bond</TargetRef>
<SourceRef>./dev-contacts</SourceRef>
<Data>200</Data> <!--Statuscode for Success-->
</Status>
<Status>
<CmdID>3</CmdID>
<MsgRef>2</MsgRef><CmdRef>4</CmdRef><Cmd>Replace</Cmd>
<SourceRef>1012</SourceRef>
<Data>200</Data> <!--Statuscode for Success-->
</Status>
<Sync>
<CmdID>4</CmdID>
<Target><LocURI>./dev-contacts</LocURI></Target>
<Source><LocURI>./contacts/james_bond</LocURI></Source>
<Replace>
<CmdID>5</CmdID>
<Meta><Type xmlns='syncml:metinf'>text/x-vcard</type></Meta>
<Item>
<Target><LocURI>1023</LocURI></Target>
<Data><!--The vCard data would be placed here.--></Data>
</Item>
</Replace>
<Add>
<CmdID>6</CmdID>
<Meta><Type xmlns='syncml:metinf'>text/x-vcard</type></Meta>
<Item>
<Source><LocURI>10536681</LocURI></Source>
<Data><!--The vCard data would be placed here.--></Data>
</Item>
</Add>
</Sync>
<Final/>
</SyncBody>
</SyncML>
```

Notes:

The structure of the message is similar to the request message.

The <Status> command with <CmdId> equals to 2: it is the status of the synchronization process, 200 in this case, which indicates that the synchronization completed successfully.

The server modifications (one <Replace> and one <Add>) are embedded in the <Sync> command addressed to the database ./dev-contacts.

The Final command indicates that the message is the last in the packet. The server therefore expects no further messages from the client.

SyncML Device Management (OMA DM)

Configuring a mobile device is a difficult task, even for the expert user. Network parameters change from phone to phone, visual interfaces are difficult to use and understand, and mobile operators all have different configurations. Moreover, mobile devices are becoming "smarter" every day. New features are preinstalled on the device and new applications are downloaded and installed by the user. As a consequence, the task of configuring a mobile device is getting even more complex. Consumers, IT personnel and operators are looking for a platform that allows remote management of devices in a convenient and effective way.

To address this need, some companies are starting to offer device management implementations for the mobile world. However, the available solutions are based on proprietary implementations. A proprietary platform exposes end users, mobile operators, device manufacturers, and IT departments to a serious risk of insufficient interoperability and associated cost implications. In the long run, it means being locked-in with a vendor and a technology, while the market is moving towards adopting standards.

The OMA Device Management (DM) protocol is an open, universal industry standard for remote device management of networked devices. Device management is the generic term used for a technology that allows third parties to carry out the difficult procedure of configuring mobile devices on behalf of the end user. Third parties would typically be wireless operators, service providers or corporate IT departments.

Through device management, an external party can remotely set parameters, troubleshoot service problems and install or upgrade software. In broad terms, device management consists of three parts:
- The protocol used between a management server and a mobile device
- The data made available for remote manipulation e.g. browser and mail settings
- The policy specifying who can manipulate a particular parameter or update a particular object in the device

In a wireless environment, the crucial element for a device management protocol is the need to efficiently and effectively address the characteristics of mobile devices, including low bandwidth and high latency. The OMA DM protocol has been built with these requirements in mind.

OMA DM device management scope includes device configuration (i.e. modify or read operating parameters); software maintenance; inventory (i.e. read from a device its current operating parameters, list installed or running software, determine hardware configurations); and diagnostics (i.e. listen for alerts sent from a device, invoke local diagnostics on a device).

OMA DM Protocol

The OMA DM Protocol is relatively simple from a messaging sequence standpoint. The message sequence is essentially broken into three parts (Figure 6):
- Alert Phase: Used only for server initiated management sessions
- Setup Phase: Authentication and Device Information Exchange
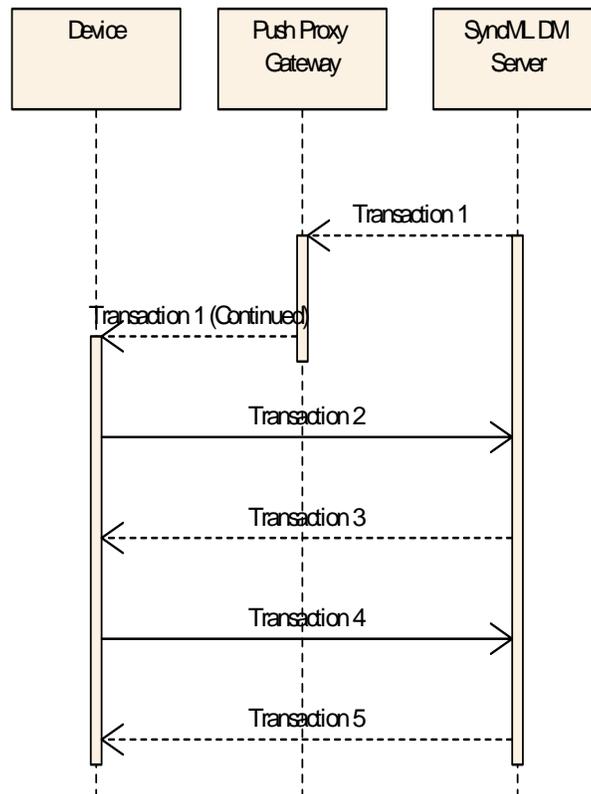- Management Phase

Figure 1 - DM session sequence diagram

Transaction 1: Alert Phase. This phase is optional, with data flowing only from server to client. The server sends a notification package to the client requesting it to start a new management session. This is usually done with a WAP push message sent by a Push Proxy Gateway on behalf of the DM server which acts as the WAP Push initiator agent.

Transaction 2: Set-up Phase (from Client). This phase is always required, with data flowing from client to server. It consists of a request from the client and the response from the server. The initial client request contains three primary pieces of information:
- Device Information: Data such as device id, manufacturer, model tag, phone language and DM protocol version
- Client Credentials: Used for authentication purposes
- Session Alert: Specifies whether the incoming session is client or server initiated

Transaction 3: Setup Phase (from server). The server responds to the initial client request with server credentials with the goal of identifying the server to the client for authentication and identification purposes. The server also sends user interaction and the first management command with the response.

Transactions 4 and 5: Device Management. These two transactions represent the core of the management session. If the server sends a management command to the client in Transaction 3, the client responds with data and status. Afterwards, the server can issue new management commands or simply return status information. The management session ends when the server sends neither additional management or user interaction commands.

Device Management Tree

Device configuration data is organized in a hierarchical structure called the device management tree. Sub-trees are called device management nodes and a leaf, usually a single configuration parameter, is called a manageable object. Device objects can be anything from a single parameter to a splash screen GIF file to an entire application. The device management tree is essentially mapped to permanent or dynamic objects as an addressing schema to manipulate these objects. Permanent objects can be thought of as objects that are built into the device at the time of manufacture and typically cannot be deleted. Dynamic objects are objects that can be added or deleted (e.g. network parameters, ring tones or wallpaper).

The Device Management Node ./DevInfo

As previously mentioned, the initial request from the client always contains device information whereby the data is retrieved from the ./DevInfo sub-tree. The ./DevInfo node is only part of the overall device management tree structure, and it maps to basic device parameters that will allow initial operations and inspection of the device by customer service personnel. The ./DevInfo object (Figure 7) is a standardized object, so that any compliant device exposes the same basic information in the same structure.
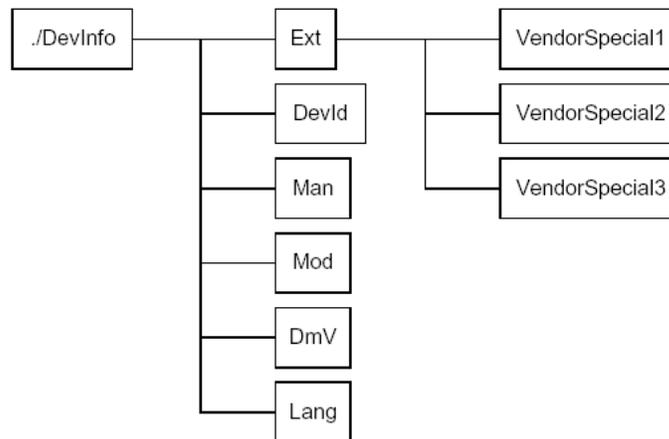
Figure 2 - The ./DevInfo standardized objects

The management sub-tree in the figure is a good example of how management objects are organized. ./DevInfo is the main node, which includes sub-trees (other management nodes) and leafs (manageable objects). Ext, for example, is a sub-tree where any vendor implementation can expose vendor specific information. All the other entities represent manageable objects: DevId contains the device id (such as the IMEI code for a phone device); Man is the device manufacturer identifier; Mod represents the device model identifier; DmV specifies the OMA DM client version; finally, Lang is the current language setting of the device. In addition to the manageable object value, each manageable object has properties. These properties define metadata information about an object to allow things such as access control etc. These properties are:

- Access Control List (ACL): Defines who can manipulate the object. (required)
- Format: Specifies how object should be interpreted e.g. if object is a URL for a management server, the Format may be defined as character (required)
- Name: The name of the object in the tree (required)
- Size: Size of object in bytes (required for Leaf Objects, not for Interior Nodes)
- Title: User-friendly name of the object (optional)
- Tstamp: The time stamp of the last modification (optional)
- Type: MIME type of object (required for Leaf Objects, optional for Interior Nodes)
- VerNo: The version Number of the object (optional)

Management objects can be manipulated via SyncML messages with these commands:
- Add: Add an Object (Node) to a tree
- Get: Returns a Node name based on the URI passed with the GET request
- Replace: Replaces an Object on the tree
- Delete: Deletes an Object on the tree
- Copy: Copies an Object on the tree
- Exec: Execute a device-defined command on an Object on the tree

4. Funambol: The Open Source Mobile Application Platform

Funambol includes both a SyncML Data Synchronization (DS) and SyncML Device Management (DM) solution.

Funambol Data Synchronization

Funambol DS includes three main components: Funambol Server, Funambol Connectors and Funambol Clients

Funambol Server

Funambol Server is implemented as a standard Java 2 Enterprise Edition (J2EE) application so that it can be deployed on top of any J2EE compliant application server, such as JBoss, IBM WebSphere or BEA WebLogic. Since many organizations have already standardized on J2EE for building applications, in many cases the Funambol Server can simply be added to the infrastructure that is already in place. If no application server is already deployed, the Funambol Server can be delivered bundled with the JBoss open source application server, avoiding high initial licenses costs for the application server . Thanks to these architectural choices, the server can run on Windows and Linux/Unix operating systems. Scalability, reliability, and speed are key aspects of the solution.

A version of the Funambol Server, to be run in a Servlet container (such as Tomcat), is also provided primarily for use in a development environment.
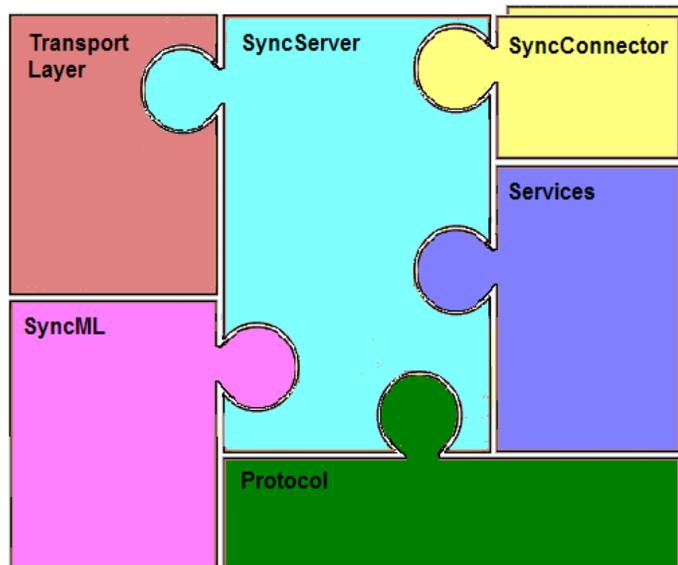

Figure 3: Funambol Server architecture

The Funambol Server is designed with modularity and flexibility in mind, as this is a key requirement for application deployment.

Figure 3 shows the main modules that comprise the Funambol Server.

The core is represented by the Funambol Server engine which makes use of add-on pluggable modules. The Transport Layer module implements transport specific SyncML bindings. In the case of the HTTP protocol, it is a J2EE web module. Other transports have their specific implementations. The SyncML module is responsible for the encoding/decoding of SyncML messages. The Protocol module implements the SyncML synchronization protocol which describes how SyncML messages are combined to represent a complete synchronization session. The Services module provides a variety of horizontal services such as authentication, security, configuration, logging and so forth. Funambol Connectors facilitate integration with external and legacy systems.

Funambol Connectors

The SyncSource API allows customers to build Funambol Connectors to external systems. Funambol includes a number of company and community supported connectors to external data sources e.g.:

- Exchange Connector (community project): Provides access to Microsoft Exchange Personal Information Management (PIM) data such as contacts, calendar, tasks and notes.
- Domino Connector (community project): Provides access to Lotus Domino PIM data.
- File System Connector: Transfers data from and to the Funambol Server in XML or ASCII files (e.g. CSV format) with a flexible transformation mechanism.
- Web Services Connector: Provides access to Web Services using the SOAP protocol. Data is exchanged with the Funambol Server in XML format.
- LDAP Connector: Provides access to a standard LDAP server, extracting contacts information.

Funambol Clients

Leveraging open standards, the Funambol is able to communicate out-of-the-box with a wide variety of SyncML compliant clients. The majority of the latest mobile phones from Nokia, Motorola and Sony Ericsson include clients capable of transferring common personal data such as contacts, calendar and tasks. For devices without a standard SyncML client, Funambol provides a variety of stand-alone clients for Windows mobile (Pocket PC and Smartphones), Java-enabled devices and email clients such as Outlook. The Funambol community also provides plug-ins for BlackBerries and Palms that sync PIM data.

Below are descriptions of some of the Funambol clients and plug-ins.

The Funambol Java client API allows developers to build vertical Java applications that seamlessly synchronize with backend systems. The API supports two methods of access for maximum flexibility: full data access where the Funambol Client takes care of updating the data in the local database and no data access, where the Funambol Client requires the application to access the data, thus allowing the developer to post-process the data before storage (e.g. filtering or combining records). Thanks to the portability of the Java language, the API can be used wherever a Java Virtual Machine (JVM) is available, such on laptops, Tablet PCs, PDAs and Java-enabled (Java ME) mobile phones.

Funambol also provides a Java ME client that enables Java-enabled handsets to get push email and to synchronize contacts and calendar data with a Funambol Server. The Funambol Server, in turn, can use connectors to access email from Yahoo! Gmail, AOL and POP/IMAP email servers and services.

The Funambol C++ client API is designed to be embeddable on mobile devices where size and speed matter most; it can be recompiled on different platforms.

The Funambol Outlook plug-in is an end user application which when installed on a PC allows Outlook PIM data to be synchronized with a Funambol Server.

The Funambol Windows Mobile plug-in enables an end user using a PocketPC or SmartPhone to synchronize all PIM data and email with a Funambol Server.

The Funambol Palm plug-in is a community project for Palm OS devices that enables PIM data including contacts and calendar to be synchronized with a Funambol Server.

The Funambol BlackBerry plug-in is another community project that enables PIM data such as the address book and calendar to sync with a Funambol Server.

The Funambol iPod plug-in is A PIM application for iPod devices which updates the contacts and calendar on the device. The data is synchronized with a Funambol Server.

Funambol Device Management (DM) Framework

Funambol DM has two components, a DM Server and a DM Client API.

Funambol DM Server

The Funambol DM Server implements the OMA DM protocol and an extensible framework for the development of DM applications. The server is based on the Funambol platform and an open source implementation of the SyncML protocol.

Funambol DM Server Architecture. The architecture of the Funambol DM Server is depicted in Figure 4. The system implements DM functionality and supports a variety of extensions to the management engine with custom features. In the diagram, the orange colored blocks represent functionality that developers can extend and plug into the framework. The main components are described below.
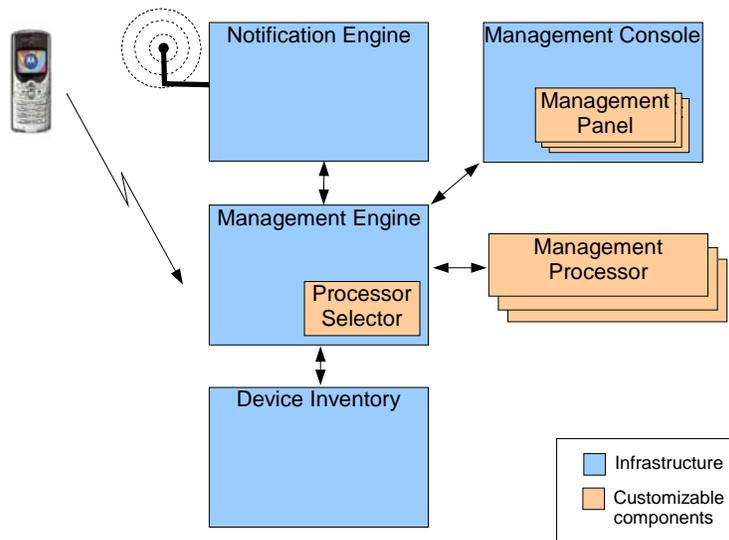

Figure 4 - Funambol Device Management Server architecture

The management engine is the core of the management server. It is responsible for understanding and interpreting the OMA DM protocol. It delegates the management logic to an external (and customizable) management processor component. Since many management processor components can be configured in the server, which one will be used is chosen by the processor selector at runtime. The selector itself is customizable, allowing logic to be adjusted based on the specific situation.

The notification engine implements management server notifications of mobile devices to start a new DM session. For example, in the case of WAP push enabled devices, it builds a binary WAP push message which is delivered to the device as specified by OMA specifications. The notification engine supports both bootstrap and device management notification messages. The former are used to store in the device initial connection and authentication data, the latter is used to trigger a server initiated OMA DM session.

The device Inventory is the repository of Device Descriptor Framework (DDF) descriptors. The DDF framework is part of the OMA DM specification and is meant to be a dynamic discovery mechanism that a server can use to gather information on the device's management tree objects. This is particularly useful to discover a device's specific management tree and objects. The management engine makes use of this repository to store the DDF read from the devices and to gather information about already discovered phones.

The management console is used by helpdesk staff to interact with the management server and, therefore, with the devices.

A management panel controls each management processor. It is plugged into the Funambol SyncAdmin console. The panel implements the user interface through which staff interact with the device, using a specific management processor object.

The management processor is responsible for the specific management code that implements particular device management logic. When the server receives an OMA DM message, the management processor selects the processor best suited to handle the request. This task is performed by a customizable Processor Selector component, which implements the logic for the selection logic for the DM application. Examples are selecting a processor based on:
- device details (e.g. device id, manufacturer, model)
- the state of the current DM session
- external configuration/interaction/events

Management processors can be as complex or simple as a DM application requires. To simplify the work of developers, an easy-to-use management processor based on a scripting language is provided. With it, writing a management processor is as simple as writing a BeanShell (http://www.beanshell.org) script.

Funambol DM Client APIs

Funambol DM Client APIs support the development of OMA DM applications that can be remotely managed by an OMA DM server. It consists of several layers:

Database Layer: represents the device's local database.

SyncPlatform Store (SPS) Layer: abstracts database access. It is not specifically related to DM but it can be used by the framework to access the local database in a generic way. SPS is an optional component in a Funambol application.

SyncPlatform Data Synchronization (SPDS) Layer: abstracts the complexity associated with the OMA DS protocol from extension developers, who simply need to call a few methods of the synchronization manager to kick off the synchronization process.

SyncPlatform DM (SPDM) Layer: the core layer of the OMA DM implementation. The main component in this layer is called device manager. It is responsible for storing and reading the management trees and nodes and for exposing to the application code hooks to start and handle the DM session. All protocol details and complexity are abstracted for the mobile application developer.

5. References

1. SyncML – "Building an Industry-Wide Mobile Data Synchronization Protocol", White Paper, 2000

2. SyncML – "SyncML Sync Protocol, version 1.1", 2002

3. SyncML – "SyncML Representation Protocol, version 1.1", 2002

About Funambol

Funambol is the leading provider of mobile 2.0 messaging software, powered by open source. Funambol open source software has been downloaded over one million times by 10,000 contributors in 200 countries. The software is used is used by thousands of organizations around the world, and the company's commercial software is used by mobile operators, OEMs, ISVs and device manufacturers, including Earthlink and Computer Associates. Funambol is backed by top-tier venture firms. The company has partnerships with leading open source and commercial technology organizations.

Funambol is based in Redwood City, CA, with an R&D center near Milano, Italy. To learn more, visit www.funambol.com or call +1 650 701-1450 (U.S.)
or +49 30-700140-411 (Germany/Europe).