DS Protocol

Approved Version 2.0 – 19 Jul 2011

Open Mobile Alliance

OMA-TS-DS_Protocol–V2_0-20110719-A

Contents

1. Scope

This document specifies the message flows between data synchronization client and server in order to ensure an inter-operable solution across all devices.

Please refer to [DSCONCEPTS] for further information on the OMA DS organization and history.

2. References

2.1 Normative References

[AES] "Specification for the Advanced Encryption Standard (AES)", National Institute of Standards and Technology, FIPS 197. November 26, 2001.
URL:http://www.nist.gov

[DEVINF] "OMA DS Device Information ", Open Mobile Alliance™, OMA-TS-DS-DevInf-V2_0,
URL:http://www.openmobilealliance.org/

[DM_ERELD] "OMA Device Management", Version 1.2, Open Mobile Alliance™, OMA-DM-V1_2, URL: http://www.openmobilealliance.org

[DS_MO] "OMA DS Management Object", Open Mobile Alliance™, OMA-TS-DS_MO-V1_0,
URL:http://www.openmobilealliance.org/

[DSCONCEPTS] "Data Synchronization Concepts and Definitions", Open Mobile Alliance™, OMA-TS-DS_Concepts-V2_0,
URL:http://www.openmobilealliance.org/

[DSHISTORY] "OMA DS Standards Change History", Open Mobile Alliance™, OMA-WP-SyncML_ChangeHistory,
URL:http://www.openmobilealliance.org/

[DSHTTPBINDING] "SyncML HTTP Binding Specification", Version 1.2.1, Open Mobile Alliance™, OMA-TS-SyncML_HTTPBinding-V1_2_1,
URL:http://www.openmobilealliance.org/

[DSOBEXBINDING] "OMA DS OBEX Binding Specification", Version 1.2, Open Mobile Alliance™, OMA-TS-SyncML_OBEXBinding_V1_2,
URL:http://www.openmobilealliance.org/

[DSSYNTAX] "Data Synchronization Syntax", Open Mobile Alliance™, OMA-TS-DS_Syntax-V2_0,
URL:http://www.openmobilealliance.org/

[DSWSPBINDING] "OMA DS OBEX Binding Specification", Version 1.2, Open Mobile Alliance™, OMA-TS-SyncML_WSPBinding-V1_2,
URL:http://www.openmobilealliance.org/

[IMCVCAL] "vCalendar – The electronic calendaring and scheduling exchange format – Version 1.0",
URL:http://www.imc.org/pdi/vcal-10.doc

[IMCVCARD] "vCard - The electronic business card - Version 2.1",
URL:http://www.imc.org/pdi/vcard-21.doc

[IMEI] "Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); Numbering, addressing and identification" (3G TS 23.003 Version 3.4.0 Release 1999),
URL:http://webapp.etsi.org/action/PU/20000523/ts_123003v030400p.pdf

[RFC2045] "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies". N. Borenstein, N. Freed, November 1996.
URL:http://www.ietf.org/rfc/rfc2045.txt

[RFC2104] "HMAC: Keyed-Hashing for Message Authentication". H. Krawczyk, M. Bellare, R. Canetti, February 1997.
URL:http://www.ietf.org/rfc/rfc2104.txt

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997,
URL:http://www.ietf.org/rfc/rfc2119.txt

[RFC2234] "Augmented BNF for Syntax Specifications: ABNF", D. Crocker, Ed., P. Overell, November 1997,
URL:http://www.ietf.org/rfc/rfc2234.txt

[RFC2279] "UTF-8, a transformation format of ISO 10646", F. Yergeau, January 1998,
URL:http://www.ietf.org/rfc/rfc2279.txt

[RFC2396] "Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, et al., August 1998,

URL:http://www.ietf.org/rfc/rfc2396.txt

[RFC2426] "vCard MIME Directory Profile", F. Dawson, T. Howes, September, 1998,
URL:http://www.ietf.org/rfc/rfc2426.txt

[RFC2437] "RSA Cryptography Specifications Version 2.0". B. Kaliski, J. Staddon, October 1998.
URL:http://www.ietf.org/rfc/rfc2437.txt

[RFC2445] "Internet Calendaring and Scheduling Core Object Specification (iCalendar)", F. Dawson, D. Stenerson, November 1998,
URL:http://www.ietf.org/rfc/rfc2445.txt

[RFC2616] "Hypertext Transfer Protocol -- HTTP/1.1". R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, June 1999.
URL:http://www.ietf.org/rfc/rfc2616.txt

[RFC4825] "The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)", J. Rosenberg. May, 2007,
URL:http://www.ietf.org/rfc/rfc4825.txt

2.2 Informative References

None.

Please refer to [DSCONCEPTS] for the other Informative References.

3. Terminology and Conventions

3.1 Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

All sections and appendixes, except "Scope" and "Introduction", are normative, unless they are explicitly indicated to be informative.

Any reference to components of the Data Synchronization XML Schema or XML snippets is specified in this `typeface`.

3.2 Definitions

Please refer to the [DSCONCEPTS] document.

3.3 Abbreviations

Please refer to the [DSCONCEPTS] document.

4. Introduction

This specification defines a synchronization protocol between the DS Client and DS Server.

4.1 Version History

For a detailed change history of OMA-DS, refer to **[DSHISTORY]**.

Specific Protocol Document changes include:

4.1.1 Version 1.0.1

This was the initial definition of the SyncML Protocol, between a client device and a server device.

4.1.2 Version 1.1.x

The protocol was enhanced to support Large Objects, indicate UTC support, Maximum object size supported, and convey the Number of Changes.

The Core specifications were split into SyncML Common, SyncML DS, and SyncML DM.

With the release of 1.1.2, SyncML was renamed OMA-DS.

4.1.3 Version 1.2.x

Suspend and Resume functionality was added, Server Alerted Notification was extensively redone, Data Sync Record and Field Level filtering was added, Synchronization of hierarchical data objects, and field level replace was implemented. Additionally, information specific to particular data formats was moved into separate data object definition specifications.

4.1.4 Version 2.0

Version 2.0 re-designed the syntax and semantics, using XML Schema, simpler syntax, and much stronger type checking. A variety of things were done to reduce traffic. Meta Information was re-integrated into the core specifications, generally as attributes modifying existing elements (See `[DSSYNTAX]`). Sync Types were replaced with a specific SyncAlert, which allowed for "Fingerprints" (see section 5.2) to be exchanged for a number of cases that used to result in Slow Syncs, to improve sync interruption recovery. Device information was allowed to be returned partially, rather than always required to be fully returned (See section 5.10.5.2). Synchronzation of sub-parts was enabled (See Sections 5.9, 5.10 and 5.16). Security was enhanced with addition encryption capabilities for the data being synchronized (See Encryption related items in `[DSSYNTAX]`). Continuous Sync was created (See Section 6.2).

5. Protocol Fundamentals

5.1 Sync Type Negotiation Parameters

Prior to the data synchronization process, the DS Client and DS Server MUST negotiate the data synchronization mechanisms. The negotiation process aims to determine the synchronization direction, synchronization behavior, and to identify what part of data needs to be sent.

In the sync mechanism negotiation process, the DS Client or DS Server sends data synchronization negotiation parameters to the other side. The parameters are shown below.

5.1.1 Direction

This indicates which sides will send data. The possible values are “fromClient”, “fromServer”, “twoWay” or “NoWay”. “NoWay” means that the direction is not specified. If “NoWay” is specified for the “*Direction*”, the “*Behavior*” parameter will have no meaning. During the “NoWay” synchronization, the server or the client can exchange data freely and without the restriction of the sync type. The server or the client can send data items to the other side or retrieve data items or retrieve hierarchy information from the other side.

Direction may only be modified to subtract from the requested data transfer. For example, if both sides are sending data, it may be negotiated so that only one side will send data. If only one side is sending data, then it cannot be forced for the other side to send data. The allowed transitions are thus from “twoWay” to “fromServer”, “twoWay” to “fromClient”, or to keep the requested value.

The detailed transition rules are listed in the table below:

From \ To	twoWay	fromClient	fromServer	NoWay
twoWay	Allowed	Allowed	Allowed	Allowed
fromClient	Not allowed	Allowed	Not allowed	Allowed
fromServer	Not allowed	Not allowed	Allowed	Allowed
NoWay	Not allowed	Not allowed	Not allowed	Allowed

5.1.2 Behavior

This indicates what behavior is expected to occur with existing data. The possible values are “Refresh” or “Preserve”.

OMA DS provides the capability for refreshing the entire data on the DS Client with the equivalent synchronization data on the DS Server. This could be necessary if the DS Client and the DS Server versions are no longer "in sync" with each other due to hardware or power failure in the mobile device, or if the version on the DS Client has become corrupted or erased from memory. This capability is provided by the DS Client issuing a "refresh" *Behaviour* attribute value in `SyncAlert` command to the DS Server.

Behavior may only be modified to reduce the amount of data being preserved. For example, if the client is clearing its data store, the server may not request that the client keep its data store. The allowed transitions are thus from “Preserve” to “Refresh”, or to keep the requested value.

The detailed transition rules are listed in the table below:

From \ To	Preserve	Refresh
Preserve	Allowed	Allowed
Refresh	Not allowed	Allowed

The *Behavior* parameter will be used with *Direction* parameter.

If *Direction* is “fromServer” and *Behavior* is “Refresh”, it would indicate that the client is to operate as if it has cleared its data store prior to further data being transferred, such as when reloading the client data stores. If *Direction* is “fromClient” and *Behavior* is “Refresh”, it would indicate the same for the server, such as the client performing a backup

operation. Note that there is no need to be able to specify refreshing both sides – the requester could clear its own data, and just request the other side to clear its data. Also note that this does not require the actual clearing of data before the sync – in many instances, knowledge of what data is presently available can prevent having to transfer that data again. Any data identified during the sync would not be deleted.

The detailed combination rules for *Direction* and *Behavior* are listed in the table below:

Combination	Refresh	Preserve
twoWay	Not allowed	Allowed
fromClient	Allowed	Allowed
fromServer	Allowed	Allowed
NoWay	Not allowed	Allowed

5.1.3 IDValidity

This indicates whether the IDs of data objects can be trusted. The possible values are "true" or "false". "False" indicates that some kind of reset or renumbering operation has occurred, or a loss of mapping information has occurred.

IDValidity may only be reduced. The only allowed transition is from "true" to "false", which could occur if the initiator of the sync had valid IDs, but the recipient did not.

The detailed transition rules are listed in the table below:

From \ To	true	false
true	Allowed	Allowed
false	Not allowed	Allowed

5.1.4 ChangeLogValidity

This indicates whether accurate knowledge of what has changed is available. The possible values are "true" or "false". The value "false" indicates that some kind of reset or loss of change data, or a loss of an accurate starting point for changes (such as a sync anchor mismatch) has occurred. Without accurate information about what data has changed since a particular point in time, some information about all data objects will generally need to be transferred. This also applies for a very first sync, since all data objects will have to be examined. Note that as soon as one data object has been successfully transferred, *ChangeLogValidity* of subsequent syncs could theoretically be considered "true", but since the sync anchors are not referring to the same point in time it must stay "false" until valid sync anchors are available.

Note that if *IDValidity* is "false", then even if *ChangeLogValidity* is "true", the change log information.cannot be trusted by the other side.

ChangeLogValidity is only applicable for parties sending data. E.g. if the *Behavior* parameter is "Refresh" and the Direction is "fromServer", then only the *ChangeLogValidity* of the server is relevant.

ChangeLogValidity may only be reduced. The only allowed transition is from "true" to "false", which could occur if the initiator of the sync has valid change log data, but the recipient does not.

The detailed transition rules are listed in the table below:

From \ To	true	false
true	Allowed	Allowed
false	Not allowed	Allowed

5.2 Fingerprint Introduction

Fingerprints are values associated with particular data item contents. They may be specified in a number of different methods, but they must change if any part of a data item changes. Fingerprint size should not exceed the data item size. It may be desirable to have the fingerprint size a predefined length, such as 4 bytes. If the data item size is smaller than the predefined fingerprint size, it is not desirable to create fingerprints for the data item.

Fingerprints are compared to detect if particular data items have changed, and may contain additional information, such as details of what has changed.

Both the DS Client and the DS Server MUST support fingerprints. The sending of fingerprints is optional. During the synchronization, the DS Client MAY send fingerprints to the DS Server for data item comparison purpose, and the DS Client MAY send the data items directly.

Fingerprints are always transmitted as paired values – ID and fingerprint.

5.2.1 Fingerprint Generation Method

Basically, there are two ways to generate fingerprints. The first, the client generates fingerprints for the data item contents and sends them to the server. The second, both the client and server generate the fingerprints for the data item contents separately.

How to choose a specific method is based on the fingerprint characteristic information. Client generated fingerprints SHOULD be used. For a particular data object, if part of the data item content is unique and can be used as fingerprints for comparison purpose, mutually generated fingerprints MAY be used.

5.2.1.1 Client Generated Fingerprints

In this case, fingerprints are defined in a client specific fashion. The client can store the fingerprints with the data item contents or the client can generate the fingerprints in real time, that is, just before the client determines to send fingerprints to the server. The associated fingerprints MUST be re-generated if the data item contents are modified on the client side.

After receiving the client generated fingerprints, the server SHOULD store the received fingerprints with the data items. Once the server modifies (Add / Update / Delete) some data items, the associated fingerprints MUST be treated as opaque values by the server, and the associated fingerprints can be deleted or marked as "invalid" or reset. If the server reset the fingerprints, the reset value MUST be unique and does not conflict with other valid fingerprint values. If the server marks the fingerprints as "invalid", the server SHOULD keep the old values of the fingerprints.

The server can compare each received fingerprint sent by client for a particular data item with the stored fingerprint on server side to determine if the data item has been changed. If the fingerprints are unique, then the received fingerprints MAY be compared agaist all the available fingerprints as a form of duplicate detection in case the data item identifiers are not valid (*IDValidity*).

In case that the server marks the fingerprints as "invalid" and keeps the old values of the fingerprints:

- If the client sends an unchanged fingerprint for one data item, the server can identify that the client has not changed the item, but the server has, and thus the server does not need to receive the data item from the client.
- If the client sends a new fingerprint for one data item, the server can identify that the client has changed the item since the last time it was in sync, and the server has changed it, and thus the server needs to receive the data item from the client to be able to merge the changes.

5.2.1.2 Mutually Generated Fingerprints

In some situations, usually specific to a particular data object type, part of the data item content can be used as fingerprints. For example, a zip file already contains a CRC (Cyclic Redundancy Check) for each file within it, so it is easy to retrieve / generate a consistent value. In this case, the fingerprints can be generated by the client and by the server with the same result. If the client or server modified the data item, the client or server MUST re-generate the fingerprint.

It is expected that this method will only apply if it is fully specified in a data object definition.

5.2.2 Fingerprint Generation Algorithm (Informative)

The DS Client and DS Server MAY choose the fingerprint generation algorithm according to the predefined local policy.

Some possible fingerprint generation algorithm choices are shown below.

5.2.2.1 Changed Flag

Clients may choose to implement their fingerprint calculation as a simple changed (or dirty) flag. In this case, the fingerprint would be a constant value (such as 0), to indicate that the data item has not changed since the last sync. If the data item changes, the fingerprint would be changed (perhaps to another constant value, such as 1). When the fingerprint of a data item is compared with its previous value, the fact that it has changed can be detected, allowing that particular data item to be transferred during a sync session.

Advantages include that it is simple to calculate, and requires minimal client storage.

Drawbacks include that flags must be maintained per server, or else when syncing to a different server, the changed state will be unknown, and thus all data items must be transferred. Preserving the accuracy of the flag across device resets (that do not wipe data) may also be an issue. Additionally, software changes may be required to properly record when changes occur.

5.2.2.2 Changed Count

Clients may choose to implement their fingerprint calculation as a simple count of how many times a data item has changed. Each time a data item is changed, the count can be incremented. When the fingerprint of a data item is compared with the value from a previous sync, that fact that it has changed can be detected, allowing that particular data item to be transferred during a sync session.

Advantages include that it is simple to calculate, and requires minimal client storage.

Drawbacks include slightly larger storage requirements than a simple flag, and that preserving the accuracy of the value across device resets (that do not wipe data) may also be an issue. Additionally, software changes may be required to properly record when changes occur.

5.2.2.3 Hash

Clients may choose to implement their fingerprint calculation as some kind of hash function. When the fingerprint of a data item is compared with the value from a previous sync, that fact that it has changed can be detected, allowing that particular data item to be transferred during a sync session.

Advantages include that it is specific to the data item, and thus is not affected by device reset. It also may be calculated as needed if detection of changes when they occur is not feasible. Clients typically support at least an MD5 hash already.

Drawbacks include slightly larger storage requirements than a simple flag, and that it is likely to be more computationally intensive than simpler methods.

5.2.2.4 Sub-Item specific

Clients may choose to implement their fingerprint calculation in a manner to allow detection of what pieces of a data item has changed, such as a flag per field (or other sub-item). When the fingerprint of a data item is compared with the value from a previous sync, that fact that it has changed can be detected, as well as what sub-items have changed. This allows field level updates to be performed, rather than requiring the entire record to be transferred.

Note that because the fingerprint is only generated by the client, the client will only be able to detect if the entire data item has changed on the server. This may lead to certain inefficiencies, such as situations where the data on the server will override the data on the client, and the client will not be able to determine if the changes on the client will be relevant.

Advantages include that field level updates may be used.

Drawbacks include slightly larger storage requirements than a simple flag, and that it is likely to be more computationally intensive than simpler methods. It may also have some of the drawbacks of simpler functions, in that there may be requirements to record when changes occur, and issues across device reset.

5.3 Change Log Information

This protocol requires that DS Client and DS Server are able to keep tracks of changes that have happened between two sync processes for one datastore. i.e., they are responsible for maintaining the change log information about the modifications associated with data items of a datastore. The types of the modifications can be e.g., replace, addition, and deletion. This protocol does not specify in which format this change log information is maintained. However, when synchronization is started, the DS Client or DS Server MUST be able to specify, which data items have changed. To specify the changed data items, the unique identifiers are used. To indicate the type of a modification, the different sync commands (e.g., `Replace`) MUST be used.

5.4 Multiple devices

If a device synchronizes data items with multiple devices, the change log information MUST be able to indicate all modifications since a previous synchronization with each device. This allows the data items in different devices to be kept up-to-date.

When synchronizing multiple DS Clients with the same DS Server, the DS Server MUST maintain different ID mapping tables for different DS Clients. If the DS Server stores only one set of data items for different DS Clients, the GUIDs of the data items on the DS Server for each DS Client MAY or MAY not be the same. If the DS Server uses different GUIDs for the same data item to different DS Clients, the DS Server MUST be able to know that the different GUIDs refer to the same data item.

For the client generated fingerprints, the DS Server SHOULD store fingerprints of the data items separately for each DS Client if the fingerprint generation algorithms of multiple DS Clients are different. In the case that the same fingerprint generation algorithm is used and the fingerprints generated for the same data item are the same, the DS Server MAY store only one set of fingerprints for different DS Clients. The DS Server MAY use the stored fingerprints received from one DS Client to compare with the fingerprints received from an other DS Client in subsequent synchronizations with different DS Clients. If the received fingerprint from the DS Client for the same data item is detected to be the same as the stored fingerprint on the DS Server, then the data item and the associated fingerprint does not need to be sent by the DS Client.

For mutually generated fingerprints, the DS Server SHOULD use the same set of fingerprints to all DS Clients in the multiple devices synchronization environment if the fingerprint generation algorithms of multiple DS Clients are the same.

For both client and mutually generated fingerprints, the DS Server may have knowledge of the fingerprint generation algorithms of multiple DS Clients, but the mechanism is out of scope of OMA DS enabler.

5.5 Usage of Sync Anchors

5.5.1 Sync Anchors for Datastores

To enable sanity checks of synchronization, this protocol uses sync anchors (See Definitions) associated with datastores (e.g., a calendar datastore). There are two sync anchors, *`Last`* and *`Next`* (See [DSSyntax]), which are always sent at the initialization of sync. The *`Last`* sync anchor describes the last event (e.g., time) when the datastore was synchronized from the point of sending device and the *`Next`* sync anchor describes the current event of sync from the point of sending device. Thus, both the DS Client and the DS Server send their own anchors to each other. The sync anchors are sent within the `SyncAlert` operation by using the Syntax Schema as defined by the DS Syntax specification. The receiving device MUST echo the *`Next`* sync anchor back to the transmitting device in the `Status` for the `SyncAlert` command.

The utilization of sync anchors is implementation specific but in order to enable the utilization, the *`Next`* sync anchor of the other device needs to be stored until the next synchronization. The DS Server MUST store the *`Next`* sync anchor sent by the DS Client to enable this utilization.

If the device stores the *`Next`* sync anchor, it is able to compare during the next synchronization whether the sync anchor is the same as the *`Last`* sync anchor sent by another device. If they are matching, the device is able to conclude that no failures

have happened since last sync. If they are not matching, the device can request a special action from another device (e.g., recovery sync).

The stored sync anchors MUST NOT be updated before the synchronization process is finished.

The synchronization process is finished after a device has finished the last step, that is, Pkg #5 for the DS Client or Pkg #6 for the DS Server, and the synchronization was successful on the `Sync` command level (i.e. no other than 200-class statuses has been returned for `Sync` commands). Also if the session is ended, the transport level (directly under SyncML level) communication has to be properly ended before synchronization can be seen as finished. If the communication between synchronizing devices is not ended properly according to transport level specification, devices MUST NOT update their sync anchors.

5.5.1.1 Example of Datastore Sync Anchor Usage

In this example, a sync client and server synchronize twice (sync process #1 and #2) with each other. After the sync process #1, the persistent memory of the sync client is reset. Because of that, the datastore anchors do not match at the sync process #2, the sync server notifies this, and it initiates the slow sync with the client.

The sync process #1 is started at 10:10:10 AM on the 10th of October 2001. The previous synchronization (before the sync process #1) was started at 09:09:09 AM on the 9th of September 2001. At this synchronization process, the recovery sync is not initiated because the sync anchors match. I.e., the sync server has the sync event (09:09:09 AM on the 9th of September, 2001).

The sync process #2 is started at 11:11:11 AM on the 11th of November 2001. Because the memory of the sync client was reset after the sync process #2, the sync server initiates the recovery sync.

In the figure below, both the sync processes are depicted. Only the initialization phases and the client sync anchors are shown in the figure.

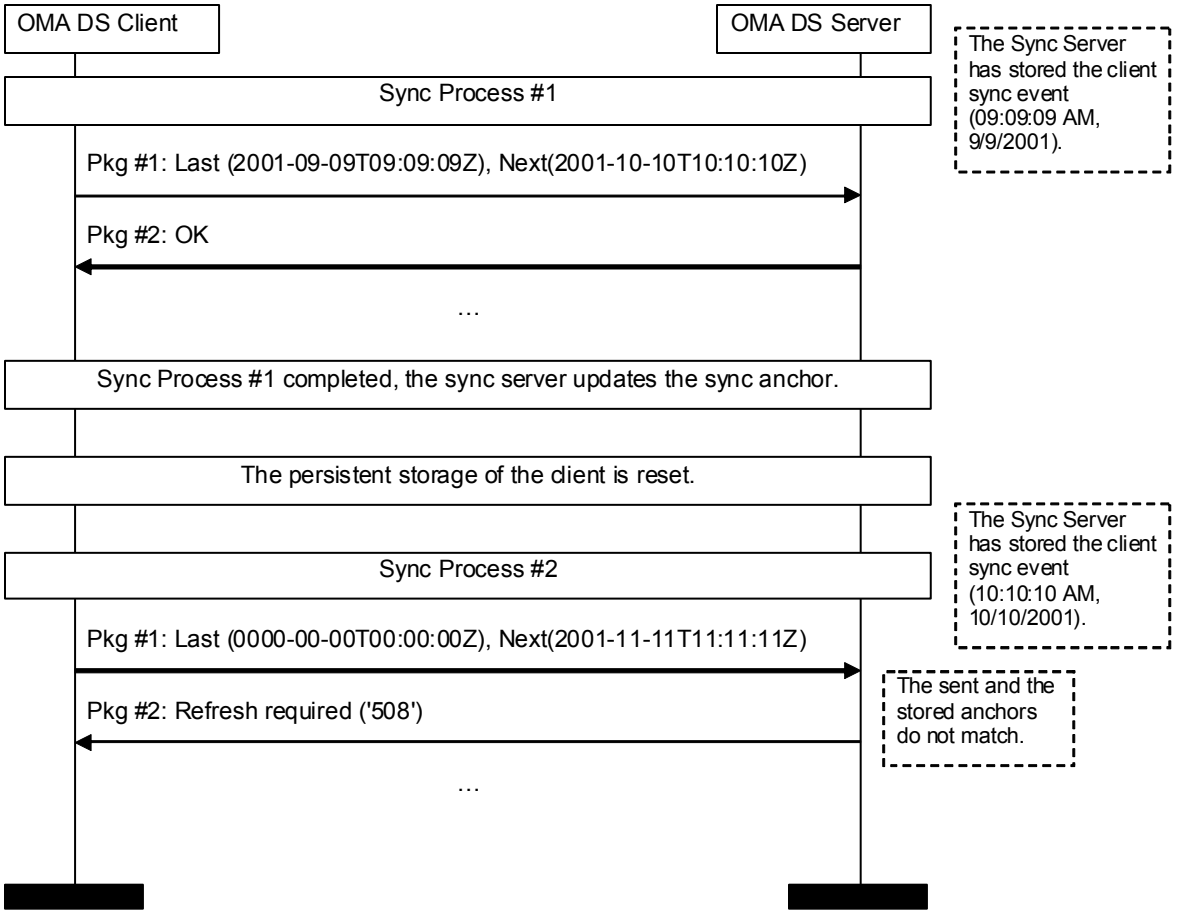

Figure 1 - Example of Sync Anchor Usage

5.5.2 Sync Anchors for Data Items

This protocol does not specify the functionality to transfer the sync anchors associated with individual data items.

5.6 ID Mapping of Data Items

This protocol is based on the principle that the DS Client and the DS Server can have their own IDs for data items in their datastores. These IDs MAY or MAY NOT match with each other. The DS Client uses local unique identifier (LUID) to identify data items, and the DS Server uses globally unique identifier (GUID) to identify data items. Usually, GUID is larger than LUID. Because the IDs can be different, the DS Server MUST maintain the ID mapping table for data items. That is, the DS Server knows which LUID and which GUID points to the same data item.

Figure 2 shows an example of an ID mapping table after synchronization. In this example the mapping table in the DS Server is depicted as a separate from the actual datastore.

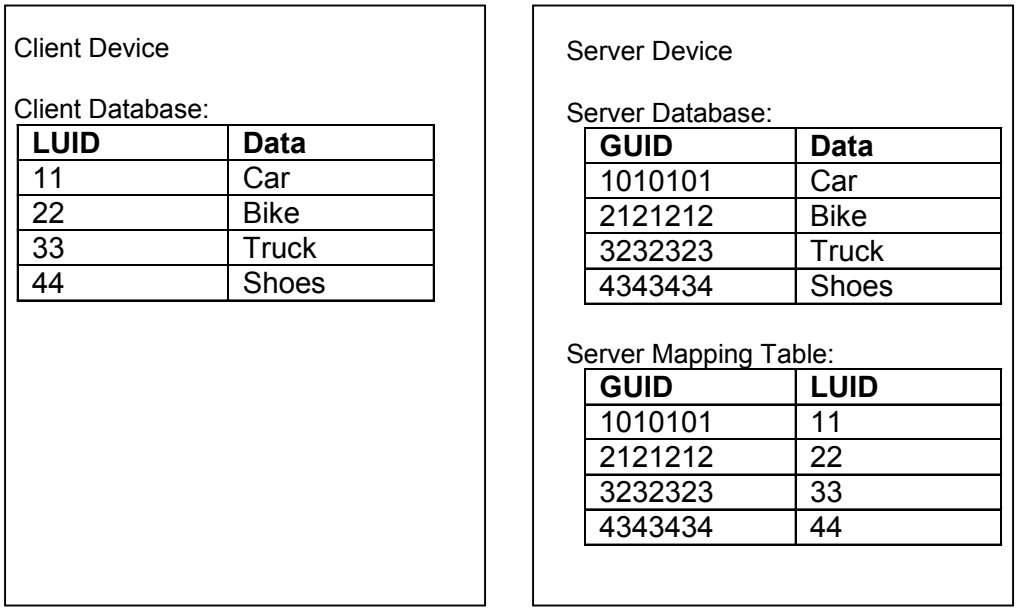

Figure 2 - Example: ID Mapping of Data Items

The LUIDs MUST be assigned by the DS Client. This means that even if the DS Server adds an item to the DS Client, the DS Client assigns a LUID for this item. In this case, the DS Client returns the LUID of the new item to the DS Server. The `Map` operation is used for this. After the `Map` operation is sent by the DS Client, the DS Server is able to update its mapping table with the client LUID.

When a DS Server is adding a new item to a DS Client, it MUST NOT send its actual GUID if the size of the actual GUID is exceeding the maximum size of the temporary GUID defined by the DS Client. If the actual GUID size exceeds the maximum size, the DS Server MUST use a smaller temporary GUID when adding an item to the client. The maximum size of the temporary GUID is defined in the device information document of the DS Client.

If the DS Server has modified an existing item (i.e., an item which is on both the devices), the DS Server MUST identify the item by using the client LUID for this item, when the modification (e.g., replace or deletion) is synchronized with the DS Client. In the case of the client modifications, items are also identified with LUIDs, when the modifications are sent to the DS Server. The DS Server can map a LUID to its own GUID by utilizing the mapping table.

5.6.1 Caching of Map Operations

After a DS Server has requested one or more additions to be done by the DS Client, and the DS Client has completed these additions to its datastore and allocated LUIDs for them, the DS Client MAY cache the Map operations associated with these LUIDs. The DS Client MAY cache the `Map` operations, if the DS Server has explicitly indicated that it does not require a response to its sync message. However, the DS Client is always allowed to send the `Map` operations back to the DS Server immediately after adding the items to the DS Client datastore. This is the case even if the DS Server has indicated that it does not require a response.

If the map items are cached, the `Map` operations are sent back to the DS Server at the beginning of a subsequent synchronization session (in Pkg #3 from the DS Client to the DS Server). That is, the DS Server MUST receive the `Map` operations before it is able to process any client updates related to the items with which the `Map` operations are associated.

If the DS Server has the control of a transport protocol (e.g., acting as an OBEX client), it MUST always request a response to the `Sync` command, which it has sent to the DS Client. Thus, the DS Server MUST NOT disconnect before it has got a response to the `Sync` command from the DS Client.

5.7 Conflict Resolution

Conflicts happen because of modifications on the same items on the server and the client datastores. (For example the same calendar item has been manually updated on the both sides.) In general, the client sends the conflict detection data to the server, and the server performs the conflict detection and resolution according to the pre-configured conflict resolution policy. After conflict resolution, the server sends back the `Status` command to the client to indicate the conflict resolution result. After that, the server and the client can synchronize the conflicted data according to the conflict resolution result.

There are multiple policies to resolve the conflicts and the DS Syntax protocol provides the status codes (See "Response Status Codes" chapter in [DSSYNTAX]) for some common policies. The table below shows four common policies and the corresponding server and client behaviors:

Policy	Server behavior
Server wins	Server sends back the server side modified data item to the client.
Client wins	Server replaces the server side data item with the received client data item.
Merge	Server merges the client and server instances of the data item. In addition, server will send a `Replace` command to the client with the merged data.
Duplicate	Server creates in the server datastore a duplication of the client data item. In addition, server will send an `Add` command to the client with the duplicate data item.

The example below depicts a case that the server sends a status to the client.

```
<Status CmdID="1" MsgRef="1" CmdRef="2" Cmd="Replace" Code="208"/>
<!-- Conflict, originator wins -->
```

Although the server is in general assumed to include the conflict resolution functionality, the possibility that the client would also provide the conflict resolution functionality is not excluded (See Section 10.10). If the client desires, the DS Client MAY retrieve the data items from the DS Server and perform conflict resolution. This will be left to the implementation of the DS Client.

The administration, and how the conflict resolution policy is configured, is out of the scope of this protocol and the DS Syntax protocol.

5.8 Identifiers

Identifiers in OMA DS, such as in the `Source` or `Target` elements, can be a combination of URI as defined by [RFC2396], URN as defined by [RFC2396] or UID. UID is the unique identifier for the data item. For the DS Client, UID is the LUID (Local Unique Identifier) and for the DS Server, UID is the GUID (Global Unique Identifier).

In OMA DS, all URI and URN values are specified as parsable character in elements or attributes. If URI scheme or URN scheme is used, the DS Client and DS Server MUST specify a valid URI or URN value using the appropriate URI scheme or URN scheme. The addressing scheme on the transport level (e.g. HTTP) is independent from the addressing scheme used at the data sync protocol layer and the two schemes do not need to match.

For further information about URI and URN, please refer to **[RFC2396]**.

The following is a list of common URI schemes:

URI Scheme Type	Description
HTTP	Hypertext Transfer Protocol
OBEX	IrDA Object Exchange Protocol
WSP	Wireless Session Protocol

Table 1 – Common URI Schemes

The following is a list of common URN schemes:

URI Scheme Type	Description
IMEI	International Mobile Equipment Identifier [IMEI]. The IMEI URN specifies a valid, 15 digit

	IMEI. The format is 'IMEI: ###############'
ESN	Electronic Serial Number. The ESN URN specifies a valid, 8 digit ESN. The format is 'ESN: ########'
MEID	Mobile Equipment Identity. The MEID URN specifies a valid, 15 digit MEID. The format is 'MEID: ###############'

Table 2 – Common URN Schemes

Other URI or URN schemes MAY be used as well.

5.9 Hierarchical URI Construction

This section illustrates the URI construction syntax. The following figure shows an example of hierarchy.

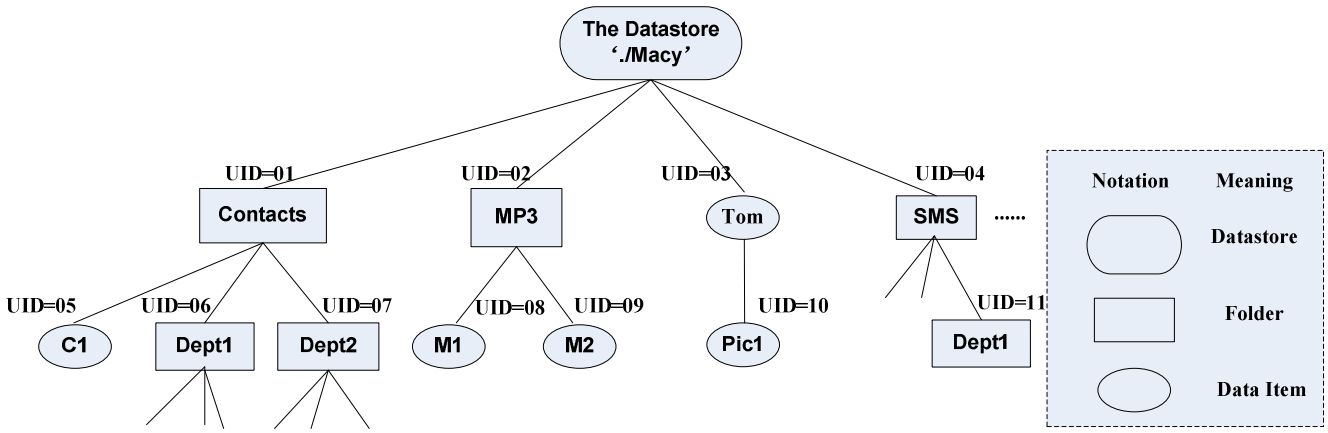

Figure 3 - Example of hierarchy

In hierarchical manner, all the nodes MUST be uniquely addressed with an URI. A node can be a datastore name or a data item identifier. For datastore node, the datastore name is the client datastore name and it MUST be unique in the device. For data items, the node identifiers constructed in URI MUST be the UID (Unique Identifier). For both client and server, the UID is LUID (Local Unique Identifier). Any data object without a LUID can not be addressed. URI used in OMA DS SHOULD be treated and interpreted as case sensitive.

The URI for a node is constructed by starting at the device root and can be addressed using absolute URI or relative URI. Within the absolute URI for a node, the root node MUST be '.' and the datastore node name MUST follow the root node and be delimited by '/' from the root node. There MUST be only one datastore node in the URI. The absolute URI for a node MUST be constructed by starting at the root and, as the tree is traversed down to the node in sequence, each node name is appended to the previous ones using "/" as the delimiting character. If the Base URI is presented in the context, the relative URI SHOULD be used for convenience.

For the MP3 folder and M1 data item in the tree above, a DS Client or DS Server would present the addresses as './Macy/02' and './Macy/02/08' respectively.

There are two kinds of nodes: interior node and leaf node. The interior node is a container. A container could be a datastore, data item or logical folder which contains other containers or leaf nodes.

The following restrictions on URI syntax are intended to simplify the parsing of URI.

- A URI MUST NOT end with the delimiter "/". Note that this implies that the root node MUST be denoted as "." and not "./".
- A URI MUST NOT be constructed using the character sequence "../". The character sequence "./" MUST NOT be used anywhere else but in the beginning of a URI.
- A node identifier in the URI MUST escape the reserved characters described in [RFC2396].

5.9.1 Hierarchical Structure Exchange

In order to effectively fulfil data store partial sync, the DS Client or Server should be able to retrieve relevant parts of the hierarchical structure of a datastore in the other side. With knowledge of the hierarchical structure, the DS Client or Server would be able to specify a particular node as the start point on both sides and hence effectively perform data store partial sync.

5.9.1.1 Request for hierarchical structure information

The DS Client or Server can use the `Get` command with an attribute appending in the URI to retrieve the hierarchical structure information. The attribute is added to the URI specified in the `Item` element inside the `Get` command. There are three kinds of attributes: All_Nodes, Direct_Child_Nodes, Interior_Nodes.

The following table shows the formats and meanings for the `Get` command and the URI in it with the different attributes:

Format	Meaning
`GET` <*URI*>**?list=** All_Nodes	Retrieve the hierarchical structure information for all the nodes under the specified URI
`GET` <*URI*>**?list=** Direct_Child _Nodes	Retrieve the hierarchical structure information for all the direct child nodes under the specified URI
`GET` <*URI*>**?list=** Interior _Nodes	Retrieve the hierarchical structure information for all the interior nodes under the specified URI

The example below depicts a case that the server retrieves the client's hierarchical structure information for all the nodes:

```
<Get CmdID="4">
   <Item>
      <TargetClientURI>./Macy/02?List=All_Nodes</TargetClientURI>
   </Item>
</Get>
```

5.9.1.2 Response carrying hierarchical structure information

The recipient side of DS Server or Client MUST send the hierarchical structure information related to the URI specified by the sender. The recipient SHALL return only the URIs of the requested nodes, without the content of the nodes.

Requested information from the node (URI in the `Get` command with *`Struct`* attribute) is embedded into an `Item` inside a `Results` command. `Meta` MUST be used to indicate the *`Type`* and *`Format`* of the node, unless the *`Type`* and *`Format`* have the default values. The `SourceClientURI/TargetClientURI` MUST indicate the URI of the node.

The hierarchical structure information from the requested node can be included into multiple `Item` elements inside a `Results` command or there can be multiple `Results` commands with single `Item`.

According to Figure 3, the partial data store specified by URI './Macy/02' is as the following:

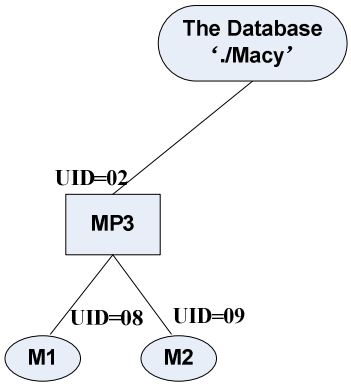

Figure 4 **- Example of hierarchy**

According to the above structure, for the retrieval request example, specified in the above section, DS Client returns the hierarchical structure corresponding to the specified URI in `Results` element as the following:

```
<Results CmdID="5" CmdRef="4 ">
   ...
   <Item>
      <SourceClientURI>./Macy/02</SourceClientURI>
      <Meta Format="xml" Type="application/vnd.omads-folder"/>
   </Item>
   <Item>
      <SourceClientURI>./Macy/02/08</SourceClientURI>
      <Meta Format="xml" Type="application/vnd.omads-file"/>
   </Item>
   <Item>
       <SourceClientURI>./Macy/02/09</SourceClientURI>
       <Meta Format="xml" Type="application/vnd.omads-file"/>
   </Item>
</Results>
```

For each `Item` element:

1) The `SourceClientURI`/`TargetClientURI` element specifies the node's absolute URI;

2) The *`Type`* attribute specifies the node's MIME type.

5.10 Addressing

There are five types of addressing which is illustrated by the following figure and further described by the following sections:

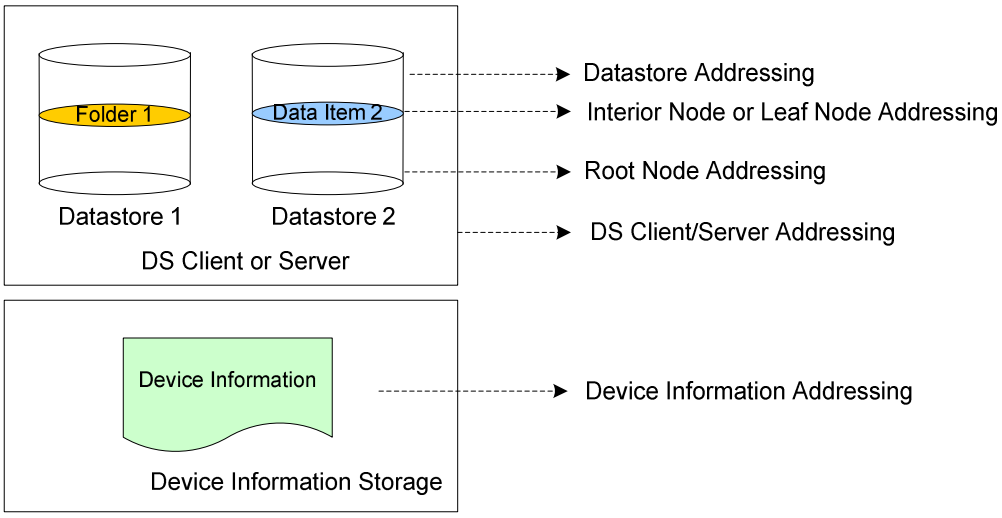

Figure 5 - Addressing Types Illustration

5.10.1 DS Client and DS Server Addressing

The DS Client MUST be addressed either using absolute URI or an URN. The DS Server MUST be addressed using absolute URI. The appropriate URI scheme or URN scheme MUST be used and the URI or URN MUST be valid to correctly address the DS Client or Server.

Example 1: DS Server Addressing using URI

```
<SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
```

Example 2: DS Client Addressing using URN

```
<SourceClientURI>IMEI:493005100592800</SourceClientURI>
```

5.10.2 Usage of RespURI and Re-direction Status Codes

The DS Client MUST support receiving the `RespURI` element as specified in [DSSYNTAX]. After receiving it, the DS Client MUST send all subsequent messages against the address specified in `RespURI` element.

The support of the re-direction related status codes is OPTIONAL.

5.10.3 Datastore Addressing

The datastore within DS Client or Server MUST be addressed using datastore URI.

The datastore URI SHOULD be represented using a textual name. The character '/' SHOULD NOT be used in the datastore name. The datastore itself MUST NOT have hierarchy structure.

For example, in the URI './ContactDB/101', the './ContactDB' will be considered as the datastore URI.

Example 1: Contact Datastore Addressing using URI:

```
<Sync>
   ......
   <TargetClientURI>./ContactDB</TargetClientURI>
   ......
</Sync>
```

5.10.4 Interior Node or Leaf Node Addressing

The interior node or leaf node is the data item or logical folder which MUST be addressed either using the relative URI or the absolute URI as described in section 5.8.

Example 1: Interior Node or Leaf Node Addressing using UID

```
<Item>
   ......
   <SourceClientURI>101</SourceClientURI>
   ......
</Item>
```

Example 2: Interior Node or Leaf Node Addressing using URI

```
<Sync>
    .....
    <TargetClientURI>./Macy/101</TargetClientURI>
    ......
</Sync>
```

5.10.5 Device Information Addressing

5.10.5.1 External Device Information Addressing

If the device information is stored in an external storage, the device information MUST be specified and retrieved based on absolute external URI. The external URI can be stored as the value of `DevInf/ExtURI` element [DEVINF]. The sender can send the external URI as the value of `DevInf/ExtURI` element, and the recipient can retrieve the external device information document based on the external URI received in `DevInf/ExtURI` element. The appropriate URI scheme MUST be used for the external device information URI and the URI MUST be valid to correctly address the external device information document.

Note: The retrieval mechanism can be XCAP [RFC4825] or others, and this is out of scope of OMA DS enabler.

Example 1: Device Information Addressing using URI

```
<Put CmdID="2">
   <Meta Type="application/vnd.syncml-devinf+xml"/>
   <Item>
     <SourceClientURI>./devinf20</SourceClientURI>
     <Data><![CDATA[
       <DevInf xmlns='syncml:devinf'>
         <ExtURI>http://www.vendorwebsite.example.com/deviceinfo/model.xml</ExtURI>
         <DevCap> ... </DevCap>
         <DataStore> ... </DataStore>
       </DevInf>]]>
     </Data>
   </Item>
</Put>
```

5.10.5.2 Internal Device Information Addressing

If the device information is stored in the DS Client or DS Server, it MUST be addressed using absolute URI. The absolute URI MUST start from the device information datastore URI, and the absolute URI for an Element MUST be constructed by starting at the root Element specified in [DEVINF] specification.

For example, in the specified URI, './devinfo20/DevInf/DataStore/CTCap/Property/PropParam', './devinfo20' is the device information datastore URI, and '/DevInf' is the root Element of the device information, and '/DataStore/CTCap/Property/PropParam' is the URI relative to the root Element.

The DS Client or Server should be able to retrieve parts of the device information in the other side. The URI can be used to indicate one Element or Attribute specified in [DEVINF] specification. If the URI is appointed to one Element within the `Get` command, all the sub-Elements and Attributes belonging to the specified Element MUST be returned.

In case that there are multiple Elements or Attributes with the same name, the condition matching expression can be used to select the specified one. If no condition matching expression is present, all the Elements or Attributes with the same name will be returned.

The `Get` command and the URI in it have the following format:

`GET` <*URI*>**[Element='Element_Value']**

Then the URI is addressed with the condition that whose sub-Element has the specified element value.

Example 1: Retrieve the selected DataStore information whose SourceRef is './contacts'

```
<Get>
    ......
    <TargetClientURI> ./devinfo20/DevInf/Datastore[SourceRef='./contacts'] </TargetClientURI>
    ......
</Get>
```

Example 2: Retrieve the selected Property information whose DataStore/SourceRef is './Contacts' and the PropName is 'PHOTO'

```
<Get>
    ......
    <TargetClientURI>
./devinfo20/DevInf/Datastore[SourceRef='./Contacts']/CTCap/Property[PropName='PHOTO']
    </TargetClientURI>
    ......
</Get>
```

5.11 Sync Scope Indication

The synchronization scope is specified by the target and source address within the `SyncAlert` element. The target and source address could point to a node as described in section 5.10.

If multiple target and source addresses need to be synchronized within one session, the DS Client or DS Server MUST use `SyncAlert` command for each target and source pair to negotiate the sync types. Accordingly the DS Client or Server MUST use `Sync` command for each target and source pair to convey the synchronization data.

Example 1: Initiate to synchronize a datastore

```
<SyncAlert CmdID="1">
   <Anchor Last="234" Next="276" />
   <Cred>……</Cred>
   <TargetServerURI>./ServerMacy</TargetServerURI>
   <SourceClientURI>./ClientMacy</SourceClientURI>
   <SyncType Direction="fromClient" Behaviour="Preserve"/>
</SyncAlert>
```

Example 2: Initiate to synchronize a part of a datastore

```
<SyncAlert CmdID="1">
   <Anchor Last="234" Next="276" />
   <Cred>……</Cred>
   <TargetServerURI>./ServerMacy/1000002</TargetServerURI>
   <SourceClientURI>./ClientMacy/02</SourceClientURI>
   <SyncType Direction="fromClient" Behaviour="Preserve"/>
```

```
</SyncAlert>
```

5.12 Exchange of Device Information

This protocol provides the functionality to exchange the device information during the initialization of a data synchronization session (See Chapter 8) or during a separate device information exchange session. The exchange can be requested by the DS Client or the DS Server.

During a separate device information exchange session, the DS Client or the DS Server can exchange their device information without performing data synchronization. During a data synchronization session or device information exchange session, the DS Client or the DS Server can use the `Put` command to send its device information to the other side and use the `Get` command to retrieve the other side's device information.

The DS Client or Server MAY store their device information externally (see Section 5.9.5.1). If DS Client or DS Server stores their device information externally, they MUST also provide default device information internally.

If there is no active session, the DS Server can initiate a DS Notification message to the DS Client, indicating the DS Client to initiate a session to send all or updated device information. Also, the DS Server can use a DS Notification message to indicate the DS Client to initiate an empty session, and the DS Server can use the `Get` command to retrieve the device information from the DS Client.

The DS Client MUST send its device information to the server during the first synchronization session or the first device information exchange session.The DS Client SHOULD send the updated device information when the device information has been updated in the DS Client since the last session. The DS Client MUST also be able to transmit its device information if it is asked by the Server. The DS Client SHOULD also support the receiving of the Server device information.

The DS Server MUST be able to send its device information if requested by the DS Client. The DS Server MUST support the functionality of receiving and processing the Client device information when sent by the DS Client or requested by the DS Server itself.

Implementation consideration. The exchange of the device information can require that a quite large amount of data is transferred over the air. Thus, the devices SHOULD avoid requesting the exchange at every time when sync is initialized. In addition, the devices SHOULD consider whether they need to send all device specific data if it is clear that another device cannot utilize it. E.g., if the client indicates that it does not support the vCard 3.0 content format, the server SHOULD NOT send the supported properties of vCard 3.0 if it supports it.

5.13 Device Memory Management

This protocol with the Device Information Schema provides possibility to specify the dynamic memory capabilities for datastores of a device or for persistent storage on a device. The capabilities specify how much memory there is left for usage. The dynamic capabilities can be specified every time when the synchronization is done. The static memory capabilities are exchanged when the sync initialization is done (See Chapter 5.12 and Chapter 8).

Although the sending of persistent memory capabilities is optional for both DS Clients and DS Servers, DS Clients SHOULD send them and DS Servers MAY send them.

The usage of different types of memory capabilities is dependent on the persistent storage model on a device. Below there is an example how the dynamic memory capabilities of a calendar datastore on a device are represented, when the `Sync` command is sent.

```
<Sync CmdID="1" FreeID="81" FreeMem="8100" >
   <TargetServerURI>./calendar/james_bond</TargetServerURI>
   <SourceClientURI>./dev-calendar</SourceClientURI>
   <Replace>
      ...
   </Replace>
   ...
</Sync>
```

The datastore-specific *FreeID* and *FreeMem* attributes in the Sync command MUST be associated with the source datastore specified in the SourceClientURI / SourceServerURI element of the Sync command.

5.14 Multiple Messages in Package

This protocol provides the functionality to transfer one SyncML package in multiple SyncML messages. This might be necessary if one SyncML package is too large to be transferred in one SyncML message. This limitation might be caused e.g., by the transport protocol or by the limitations of a small footprint device.

If a SyncML package is transferred in multiple SyncML messages, the last message in the package MUST include the Final element (See [DSSYNTAX]).

Other messages belonging to the package MUST NOT include the Final element.

The Final element can only be included when all necessary commands belonging to a specific package have been sent.

The Final element MUST NOT be included if the other end has not closed the preceding package. E.g., if the server is still sending the package #4 to the client, the client MUST NOT close the package #5 prior to receiving the last message belonging to the package #4. The exclusion of the Final element is not to be used to indicate that a logical phase is not completed if an error occurs.

If a device receives a message in which the Final flag is missing and a Sync element for a datastore is included, the device MUST be able to handle the case that in the next message, there is another Sync element for the same datastore.

The device, which receives the SyncML package containing multiple messages, MUST be able to ask more messages. This happens by sending an Alert command with a specific alert code, 222 back to the originator of the package, or if there are other SyncML commands to be sent as a response, the Alert command with the 222 alert code MAY be omitted. After receiving the message containing the Final element, the Alert command MUST NOT be used anymore.

More messages are not desirable if errors, which prevent the continuation of synchronization, have occurred.

The recipient of a package MAY start to send its next package at the same time when asking more messages from the originator if this makes sense. Thus, in Chapters 9 - 11, it is specified which commands or elements are allowed to be sent before receiving the final message belonging to a package.

Below, there is depicted an example that the client is sending Package #3 in multiple messages (2 messages) and the server also sends Package #4 in multiple messages (2 messages).

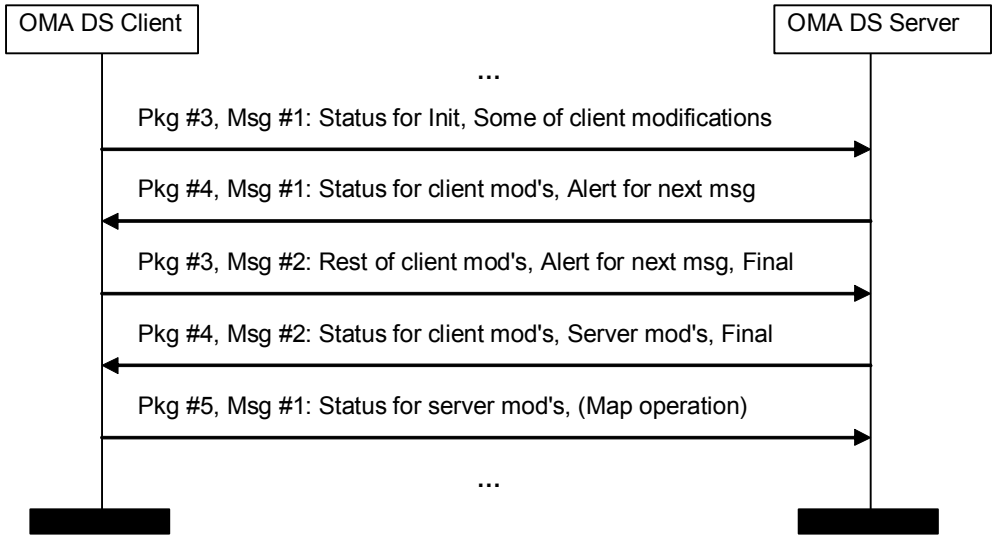

Figure 6 : Example of Sending Multiple Messages in a Package

5.15 Large Object Handling

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

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

Note: This mechanism does not allow sending multiple large objects in the same time. A new data object MUST NOT be added by a sender to any message until the previous data object has been completed. If a data object is chunked across multiple messages, the chunks MUST be sent in contiguous messages. New Sync commands (i.e. `Add`, `Replace`, `Delete` or `Copy`) or new Items MUST NOT be placed between chunks of a data object.

5.15.1 Conformance statements:

Clients SHOULD support receiving Large Objects and servers MUST support receiving Large Objects.

Supporting Sending Large Objects is optional for both clients and servers.

A client supporting receiving Large Object MUST declare the <SupportLargeObjs/> tag in its DevInf.

Supporting receiving or sending Large Objects implies conformance constraints for several tags.

TAGS:

- **SupportLargObjs** [DEVINF]
- **MaxObjSize** [DSSYNTAX]
- **MaxMsgSize** [DSSYNTAX]
- **Size** [DSSYNTAX]
- **MoreData** [DSSYNTAX]

STATUS CODES AND ALERTS:

- **Status 213** Chunked item accepted and buffered [DSSYNTAX]
- **Alert 222** Next Message [DSSYNTAX]
- **Alert 223** End of Data for chunked object not received [DSSYNTAX]
- **Status 424** Size Mismatch [DSSYNTAX]
- **Status 416** Request entity too large [DSSYNTAX]
- **Status 411** Size Required [DSSYNTAX]

If a device supports receiving Large Objects it MUST declare the maximum size of object (`MaxObjSize` attribute) it is capable of receiving as information within the `SyncAlert` or `Sync` command, as specified in [DSSYNTAX].

The device MUST also declare and fill the `MaxMsgSize` tag. This tag, also declared in `SyncHdr` element, specifies the maximum byte size of any response message to a given request. Knowledge of both `MaxObjSize` and `MaxMsgSize` allows to compute appropriate data chunk size.

5.15.2 Large Object exchange sequence:

This section illustrates and details the process of Large Object Handling. The following figure depicts a normal flow when handling Large Objects between 2 entities: the "Sending Large Object Device" and the "Receiving Large Object Device". Note that these 2 entities can represent a Client or a Server: a Client can send Large Objects to a Server, but a Server can send also Large Objects to a Client.

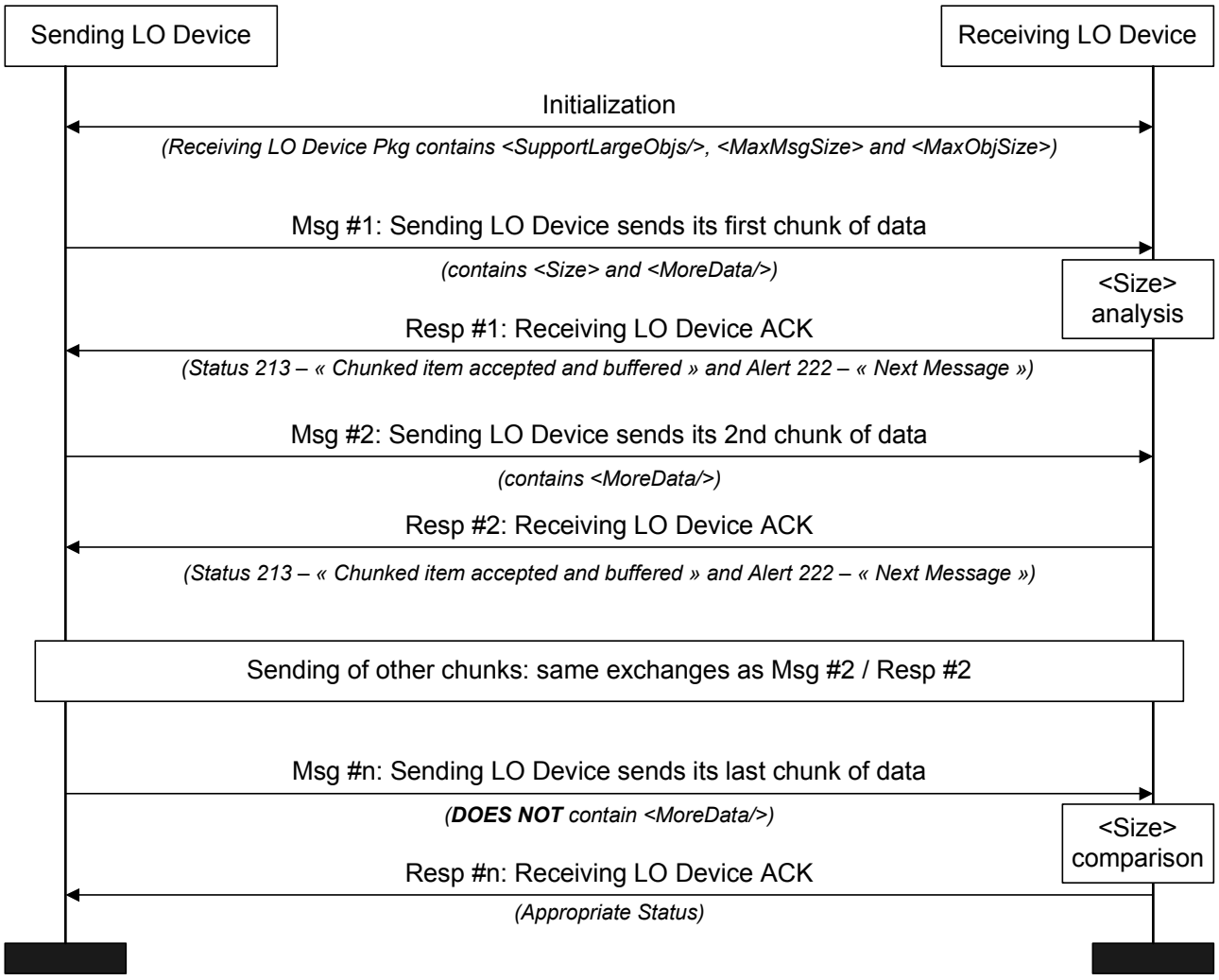

Figure 7 - Example of Sending a Large Object (normal case)

Note 1: In the previous diagram, LO means "Large Object".
Note 2: Please refer to the section 5.14 for the use of the Alert 222.

Exchange of a Large Object can be summarized with the following sequence:

1. During initialization:

1.1. On the sending device side

1.1.1. Sending device SHOULD use knowledge of the recipient's **`MaxMsgSize`** to determine at what size segmentation occurs.

1.2. On the receiving device side

1.2.1. Receiving device MUST NOT have specified SupportLargeObjs="false" in its DevInf. It MUST also specify the value of its ***`MaxMsgSize`*** and its ***`MaxObjSize`***.

2. When the first chunk of data is transmitted:

2.1. On the sending device side (Msg #1)

2.1.1. The sender MUST declare in the command element (e.g. `Add`, `Replace`) the overall size of the data element content that is going to be sent, using the ***Size*** attribute of a `Meta` element.

Note: The ***Size*** attribute MUST only be specified for the first chunk of data.

2.1.2. A **<MoreData/>** empty element MUST be added after the <Data> element.

2.2. On the receiving device side (Resp #1)

2.2.1. On receipt of a data chunk with the **<MoreData/>** element, the recipient MUST respond with a **"Status 213 – Chunked item accepted and buffered"** and ask for the next message using the **Alert 222** mechanism as defined in section 6.9

Error case behavior:

1- If the Size exceeds the ***MaxObjSize*** of the recipient, the recipient MUST respond with a **"Status 416 - Requested size too big"** (the request failed because the specified byte size in the request was too big). The recipient MUST NOT commit the command.

2- If the recipient gets the first chunk with a **<MoreData/>** element, but no ***Size*** attribute, or non filled ***Size*** attribute, it MUST respond with a **"Status 411 - Size required"**. The recipient MUST NOT commit the command. The sender MAY attempt to retransmit the entire data object.

3. When extra chunks of data are transmitted:

3.1. On the sending device side (Msg #2)

3.1.1. Meta and Item information SHOULD be repeated on each subsequent message containing chunks of the same data object.

3.1.2. A **<MoreData/>** empty element MUST be added after the `Data` element.

3.2. On the receiving device side (Resp #2)

3.2.1. On receipt of a data chunk with the **<MoreData/>** element, the recipient MUST respond with a **"Status 213 – Chunked item accepted and buffered"** and ask for the next message using the **Alert 222** mechanism as defined in section 6.9

Error Case Behavior:

If the recipient detects a new data object or command before the previous item has been completed (by the chunk without the **<MoreData/>** Element), the recipient MUST respond with an **"Alert 223 – End of Data for chunked object not received"**. The **`Alert`** SHOULD contain the complete source and/or target information from the original command to enable the sender to identify the failed command.

Note: a **`Status`** would not suffice here because there would not necessarily be a command ID to refer to. The recipient MUST NOT commit the new and original commands. The sender MAY attempt to retransmit the entire original data object.

4. When the last chunk of data is transmitted:

4.1. On the sending device side (Msg #n)

4.1.1. The last chunk of data MUST NOT be followed with **<MoreData/>** element.

4.2. On the receiving device side (Resp #n)

4.2.1. On receipt of the last chunk of the data object, the recipient reconstructs the data object from its constituent chunks. It MUST validate that the size of re-constituted object matches the object ***Size*** supplied in the Meta information by the sender, then apply the requested command. The appropriate status MUST then be sent to the originator.

Error case behavior:

If the sizes do not match then a **"Status 424 – Size mismatch"** MUST be sent and the recipient MUST NOT commit the command. The sender MAY attempt to retransmit the entire data object.

5.15.3 Large Object exchange sequence example:

In this example the client sends a large object (for addition) to the server. The server has declared supporting Large Objects handling in its DevInf.

Client initializes a sync session

```
<SyncML Version="2.0">
     <SyncHdr SessionID="272244708" MsgID="1" MaxMsgSize="3000" >
          <TargetServerURI>http://Syncserver.com/sync</TargetServerURI>
          <SourceClientURI>IMEI_number</SourceClientURI>
          <Cred>
               <Meta Format="b64" Type="syncml:auth-basic" />
               <Data>dGVzdDp0ZXN0cHc=</Data>
          </Cred>
     </SyncHdr>
     <SyncBody>
          <SyncAlert CmdID="1" MaxObjSize="10000000" >
               <!-- … -->
          </SyncAlert>
          <Final/>
     </SyncBody>
</SyncML>
```

Server Response

```
<SyncML Version="2.0">
   <SyncHdr SessionID="272244708" MsgID="1" MaxMsgSize="30000">
          <TargetClientURI>IMEI_number</TargetClientURI>
          <SourceServerURI>http://Syncserver.com/sync </SourceServerURI>
   </SyncHdr>
   <SyncBody>
          <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="212">
               <ServerURI> http://Syncserver.com/sync </ServerURI>
               <ClientURI>IMEI_number</ClientURI>
          </Status>
          <Status CmdID="2" MsgRef="1" CmdRef="1" Cmd="SyncAlert" Code="200">
               <!-- … -->
          </Status>
          <SyncAlert CmdID="3" MaxObjSize="10000000">
               <!-- … -->
          </SyncAlert>
          <Final/>
   </SyncBody>
</SyncML>
```

The client begins to send a large object

```
<SyncML Version="2.0">
     <SyncHdr SessionID="272244708" MsgID="2" MaxMsgSize="3000">
         <!-- … -->
     </SyncHdr>
     <SyncBody>
```

```
          <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="200">
              <!-- … -->
          </Status>
          <Status CmdID="2" MsgRef="1" CmdRef="3" Cmd="SyncAlert" Code="200">
              <!-- … -->
          </Status>
          <Sync CmdID="3" >
               <!-- … -->
               <Add CmdID="4" >
                    <Meta Type="application/vnd.omads-file+xml" Size="2304"/>
                    <Item>
                         <SourceClientURI>1001</SourceClientURI>
                         <Data>
                            <!-- -->
                            <!-- Big Block Of Data Takes Place Here -->
                            <!-- -->
                         </Data>
                         <MoreData/>
                    </Item>
               </Add>
          </Sync>
     </SyncBody>
</SyncML>
```

Server response: Data chunk is accepted

```
<SyncML Version="2.0">
     <SyncHdr SessionID="272244708" MsgID="2">
       <!-- … -->
     </SyncHdr>
     <SyncBody>
          <Status CmdID="1" MsgRef="2" CmdRef="0" Cmd="SyncHdr" Code="200">
               <!-- … -->
          </Status>
          <Status CmdID="2" MsgRef="2" CmdRef="3" Cmd="Sync" Code="200">
               <!-- … -->
          </Status>
          <Status CmdID="3" MsgRef="2" CmdRef="4" Cmd="Add" Code="213">
               <!-- … -->
          </Status>
          <Alert CmdID="4" Code="222">
               <Item>
                    <Data>Next Message Please</Data>
               </Item>
          </Alert>
     </SyncBody>
</SyncML>
```

Client sends the last chunk of the large object

```
<SyncML Version="2.0">
     <SyncHdr SessionID="272244708" MsgID="3" MaxMsgSize="3000">
          <!-- … -->
     </SyncHdr>
     <SyncBody>
          <Status CmdID="1" MsgRef="2" CmdRef="0" Cmd="SyncHdr" Code="200">
               <!-- … -->
          </Status>
          <Status CmdID="2" MsgRef="2" CmdRef="4" Cmd="Alert" Code="200">
               <!-- … -->
          </Status>
          <Sync CmdID="3">
```

```
                <!-- … -->
                <Add CmdID="4">
                     <Meta Type="application/vnd.omads-file+xml"/>
                     <Item>
                          <SourceClientURI>1001</SourceClientURI>
                          <Data>
                              <!-- -->
                              <!-- Large Object Ends Here -->
                              <!-- -->
                          </Data>
                     </Item>
                </Add>
           </Sync>
           <Final/>
     </SyncBody>
</SyncML>
```

Further interactions continue here and the examples are omitted.

5.16 Hierarchical synchronization

Hierarchical synchronization consists in synchronizing a hierarchical data structure on a server and its equivalent on the client. A hierarchical data structure can be compared to a tree structure that is composed of:

- Branches that are links between a node and its children
- Nodes that contain the information : There are three kind of nodes :
  - the root node which has no parent
  - interior node which has a unique parent
  - leaf node which has no children

One of the most known tree structure is the filesystem where folders act as nodes and files as leaves

Regarding OMA-DS protocol, hierarchical synchronization mechanism is based on the use of `TargetClientParentURI`, `SourceClientParentURI or SourceServerParentURI` element.

The client uses `SourceClientParentURI` to specify the LUID for the parent of the client's side item.

Depending on the existence of the parent item on the client, the server will use `TargetClientParentURI` or `SourceServerParentURI`:

- If the parent exists on the client , the server MUST use the `TargetClientParentURI` tag that will contain the client's LUID for the parent of the server side's item
- If the parent item doesn't exist on the client, the server MUST use the `SourceServerParentURI` tag that will contain a temporary GUID for the parent of the server side's item. This case occurs when the server send items that the client has not mapped yet (e.g. moving a file into a newly created folder)

As an example we propose in the following a possible filesystem synchronization scenario

We suppose the client and server are synchronized. Items mapping table is

Client's LUID	Server's GUID	Object name

990	ABCD990	Urgent
995	ABCD995	Work
1000	ABCD1000	Image1.jpg
1001	ABCD1001	Pictures
1002	ABCD1002	Friends

After the last synchronization the following modifications were made on the client:

- The folder "NewFolder" has been created at the root of the filesystem.
- A file named "NewDocument.doc" has been created in the "NewFolder" folder
- The file "Image1.jpg" has been moved to the "Pictures" folder

Regarding the server modifications made are:

- A new folder "ToBeDone" has been created in the folder "Work"
- A file "ToDoList.doc" has been created in the folder "ToBeDone"
- Subfolder "Urgent" has been moved to "ToBeDone"

For those new resources the partial mapping table can be expressed as:

Client's LUID	Server's GUID	Object name
	ABCD997	ToBeDone
	ABCD998	ToDoList.doc
1003		NewFolder
1004		NewDocument.doc

A subsequent synchronization will produce the following package snippets:

Package 3 : Client modifications

```
<Sync CmdID="7" >
   <TargetServerURI>./FileSystem</TargetServerURI>
   <SourceClientURI>./Files</SourceClientURI>
   <Add CmdID="8" >
      <Meta Type="application/vnd.omads-folder+xml"/>
      <Item>
         <SourceClientURI>1003</SourceClientURI>
         <SourceClientParentURI>/</SourceClientParentURI>
         <Data> Data containing DataObjFolder should be placed here </Data>
      </Item>
   </Add>
   <Add CmdID="9">
      <Meta Type="application/vnd.omads-file+xml"/>
      <Item>
```

```
        <SourceClientURI>1004</SourceClientURI>
        <SourceClientParentURI>1003</SourceClientParentURI>
        <Data>Data containing DataObjFile should be placed here</Data>
      </Item>
   </Add>
   <Move CmdID="10">
      <Meta Type="application/vnd.omads-file+xml"/>
      <Item>
        <SourceClientURI>1000</SourceClientURI>
        <SourceClientParentURI>1001</SourceClientParentURI>
      </Item>
   </Move>
</Sync>
```

The client modification package contains:

- An `Add` command with `SourceClientParentURI` containing "/" and `SourceClientURI` "1003" (creation of "NewFolder" at the root)
- An `Add` command with `SourceClientParentURI` containing "1003" and `SourceClientURI` "1004" (creation of "NewDocument.doc" in the "NewFolder" folder)
- A `Move` command with `SourceClientParentURI` containing "1001" and `SourceClientURI` "1000" (Moving of "image1.jpg" in "Pictures")

Package 4: server modifications

```
<Sync CmdID="17">
   <TargetClientURI>./Files</TargetClientURI>
   <SourceServerURI>./FileSystem </SourceServerURI>
   <Add CmdID="18">
      <Meta Type="application/vnd.omads-folder+xml"/>
      <Item>
        <SourceServerURI>ABCD997</SourceServerURI>
        <TargetClientParentURI>995</TargetClientParentURI>
        <Data> Data containing DataObjFolder should be placed here </Data>
      </Item>
   </Add>
   <Add CmdID="19">
      <Meta Type="application/vnd.omads-file+xml"/>
      <Item>
        <SourceServerURI>ABCD998</SourceServerURI>
        <SourceServerParentURI>ABCD997</SourceServerParentURI>
        <Data> Data containing DataObjFile should be placed here </Data>
      </Item>
   </Add>
   <Move CmdID="20">
      <Meta Type="application/vnd.omads-folder+xml"/>
      <Item>
        <TargetClientURI>990</TargetClientURI>
        <SourceServerParentURI>ABCD997</SourceServerParentURI>
      </Item>
   </Move>
</Sync>
```

The server modifications package contains:

- An `Add` command with `TargetClientParentURI` containing "995" (the client's LUID for the parent of the "ToBeDone" item)
- An `Add` command with `SourceServerParentURI` containing "ABCD997" (Since the client has not mapped "ToBeDone" yet we use `SourceServerParentURI` and the server side's parent GUID of "ToDoList.doc")
- A `Move` command with `SourceServerParentURI` containing "ABCD997" and `TargetClientURI` containing 990 (Moving "Urgent" to "ToBeDone")

5.17 Suspend and Resume of synchronization session

Interruption can occur in two different ways:

1. User initiated interruption/Pause (Can also be viewed as an intentional pause):

   This kind of interruption occurs when user requests to pause the current session and thereby resulting into a negotiation phase between client and server to pause the session.

   In order to interrupt the sync session, the client MAY send a message containing 'Interrupt Sync Session' with no client side data items and MAY contain statuses to server's data items. This message can be sent before receiving the complete package from the server.

2. Loss of network coverage or phone malfunction (Can also be viewed as an unintentional pause):

   This kind of interruption can be due to loss of network coverage or phone malfunction or for other unknown reasons that lead to an immediate interruption of the sync session and thus not needing a special alert code for interruption.

The interruption MAY occur during any of the synchronization phases - initialization, or during synchronization, or during mapping phase.

In case that the session is interrupted, the DS Client or DS Server SHOULD initiate recovery sync.

5.18 Busy Signaling

If the server is able to receive the data from the client but it is not able to process the request(s) at a reasonable time after receiving the modifications from the client, the server SHOULD send information about that to the client. Note that this processing time is dependent e.g. on the transport protocol transferring SyncML messages. This happens by sending the Busy Status package back to the client.

After the client has received a busy signal from the server, the client MAY ask for the sync results later or start the synchronization from the beginning. If the client starts the synchronization from the beginning its *`Last`* sync anchor MUST NOT be updated.

If the server has sent the busy status to the client and it does not get a request from the client (i.e., Result `Alert`), the server MUST assume that the client has stopped the synchronization and start the synchronization from the beginning. The server MUST NOT update its *`Last`* sync nor the client *`Next`* sync anchors.

5.18.1 Busy Status from Server

Informing the client that the server is busy happens by sending the Busy Status package to the client. This can be sent before any package is completely received. The Busy Status package MUST NOT be used to return status information related to any individual data items or command which are in `SyncBody` of the client request.

The requirements for the elements within the Busy Status package are:

1. Mandatory elements within the `SyncHdr` element.
2. The `Status` element for the `SyncHdr` MUST be included in `SyncBody`.
   - The status code (101, in progress) MUST be returned within the `Status` for the `SyncHdr` command sent by the client.
3. The `Final` element MUST NOT be used for the message.

Example of Busy Status package:

```
<SyncML Version="2.0">
   <SyncHdr SessionID="1" MsgID="2">
      <TargetClientURI>IMEI:493005100592800</TargetClientURI>
      <SourceServerURI>http://www.syncml.org/sync-server</SourceServerURI>
   </SyncHdr>
   <SyncBody>
      <Status CmdID="1" MsgRef="2" CmdRef="0" Cmd="SyncHdr" Code="101">
         <!--Status code for Busy-->
         <ServerURI>http://www.syncml.org/sync-server</ServerURI>
         <ClientURI>IMEI:493005100592800</ClientURI>
      </Status>
   </SyncBody>
</SyncML>
```

5.18.2 Result Alert from Client

The result alert is sent to ask results to the last message which was sent to the server. This is done by sending a Result Alert package from the client to the server. A message within this package has the following requirements.

1. Mandatory elements within the `SyncHdr` element.
2. The `Alert` element MUST be included in `SyncBody`. There are the following requirements for this `Alert` element.
   - *`CmdID`* MUST be used.
   - The *`Item`* element is used to specify the server and the client device.
   - The *`Code`* attribute is used to include the `Alert` code. The alert code is '221' (See [DSSYNTAX]).
3. The `Final` element MUST NOT be used for the message.

If the server is still busy, when it receives this Result Alert from the client, it MUST again return the Busy Status with the '101' status code back to client. The status code is associated with the `SyncHdr` and the `Alert` command sent by the client.

Example of Result Alert package:

```
<SyncML Version="2.0">
   <SyncHdr SessionID="1" MsgID="3">
      <TargetServerURI>http://www.syncml.org/sync-server</TargetServerURI>
      <SourceClientURI>IMEI:493005100592800</SourceClientURI>
   </SyncHdr>
   <SyncBody>
<!--Status for SyncHdr is omited here-->
      <Alert CmdID="1" Code="221">
         <Item>
            <TargetServerURI>http://www.syncml.org/sync-server</TargetServerURI>
            <SourceClientURI>IMEI:493005100592800</SourceClientURI>
         </Item>
      </Alert>
```

```
   </SyncBody>
</SyncML>
```

5.19 OMA DS Data Formats

OMA DS not only provides for a common set of commands, but also identifies a small set of common data formats. The data formats provide a common set of media types for exchanging common accepted information, such as contacts, calendars and messages. Support for these data formats is mandatory for conformance to this specification. In addition to these common formats, OMA DS allows for the identification of any other registered format. OMA DS utilizes the MIME content type framework for identifying data formats, called MIME media types.

5.19.1 MIME Usage

There are two MIME content types for the OMA Data Synchronization Message. The MIME content type of `application/vnd.syncml+xml` identifies the clear-text XML representation for the SyncML Message. The MIME content type of `application/vnd.syncml+wbxml` identifies the WBXML binary representation for the SyncML Message. Section 8 of this specification specifies the MIME content type registration for these two MIME media types.

One of these two MIME content types MUST be used for identifying OMA Data Synchronization Messages within transport and session level protocols that support MIME content types.

5.20 Soft and Hard Data Deletion

The SyncML `Delete` command provides the capability for a SyncML request to delete data from the recipient's data store. Two forms of deletion are supported. Normally, when a `Delete` command is specified, it conveys a request to completely delete the specified data from the recipient's data store. The deleted data SHOULD no longer be associated with the originator's synchronization data. This is the semantics of a "Hard Delete". In addition, SyncML provides support for a "Soft Delete" command.

The rationale for a "Soft Delete" is based on the possibility of limited storage resources in a client device. The data is deleted to free-up storage for other, higher priority data on the client device. After sending/receiving the "Soft Delete" command to/from the DS Server, the DS Client MUST delete the data items on the DS Client side according to the specified LUIDs and free up storage resources, and the DS Server MUST maintain the “Soft Deleted” data items. The LUIDs associated with the “Soft Deleted” data items MUST be maintained by the DS Client so that the DS Client MAY request the “Soft Deleted” data items later or the DS Server MAY send the stored data items to the DS Client later.

The operation of “Soft Delete” is defined in the SftDel chapter (see [DSSYNTAX]).

On occasions, an exception can occur when the DS Client specifies "Soft Delete" for a data element that has already been "Hard Deleted" on the DS Server. This condition will cause a "Soft-Delete Conflict" for that event when a two-way synchronization is attempted. This version of OMA DS does not specify how to negotiate the resolution of such "Soft-Delete Conflicts". However, it does provide status codes to identify Soft-Delete Conflict conditions and to also identify how the conflict might have been resolved.

Note that the DS Server should internally note the "Soft Deleted" data items to avoid synchronization these data items from the DS Server to the DS Client. When the DS Server receives the “Soft Deletion” request from the DS Client, the DS Server SHOULD mark the requested data items as “Soft Deleted” according to the specified LUIDs. The "Soft Deleted" data items SHOULD be considered outside of the sync scope during a normal synchronization, except that the DS Client specifies the LUIDs for the "Soft Deleted" data items on the DS Server side. After receiving the retrieval request for the “Soft Deleted” LUIDs from the DS Client, the DS Server sends the requested “Soft Deleted” data items to the DS Client according to the LUIDs.

5.21 Replacing Data

The SyncML `Replace` command provides the capability for the originator to replace existing data. The command can also be the cause for an "Update Conflict".

5.21.1 Field-level Replace

The SyncML Replace command also provides the capability for the originator to send an update to the recipient without having to transfer the entire item. This technique is also called Field-level changes. This feature is extremely useful for the data types in which relatively concise attributes (for example the "read" status of the e-mail) are more likely to change, than substantially larger attributes like the body of the message or the attachments.

Not all data types are equally suited for being used with Field-level replace. It is the responsibility of the sender to compose the partial items in the corresponding data format in such a manner that they are unambiguously interpreted by the receiver. Also it is the responsibility of the sender to compose the partial items in the corresponding data format ensuring that the format remains valid. If the sender cannot meet these criteria then it MUST send a replace for the entire item instead of a field-level replace.

Example:

It is ambiguous to send the field-level change containing the following vCard

```
BEGIN:VCARD
VERSION:2.1
N:Doe;John;;;
TEL;HOME:(321) 654-987
END:VCARD
```

In this case if the receiving side supports more than one HOME phone number, it will have an ambiguity understanding which one was changed.

Example 2:

It is improper to send the field-level change containing the following vCard

```
BEGIN:VCARD
TITLE:Worker
END:VCARD
```

The vCard format mandates the VERSION and N attributes to be present within the item.

On occasions, an exception can occur where the same data element on both the OMA DS client and the OMA DS server have been updated or replaced. For example, the start and end date/time for the same event might have been changed to different values on the OMA DS client compared to the description on the OMA DS server. This condition will cause an "Update Conflict" for that event when a two-way synchronization is attempted. This version of OMA DS does not specify how to negotiate the resolution of such Update Conflicts. However, it does provide status codes to identify Update Conflict conditions and to also identify how the conflict might have been resolved.

5.22 Data Sync Record and Field Level Filtering

Server data stores frequently contain much more data than can fit into small devices. Other aspects of the protocol enable clients and servers to indicate data store capacity and therefore avoid data overflow conditions, however it is often the case that small devices only want to synchronize a particular, prioritised subset of the data that resides in the server's data store (referred to from this point forth as record filtering). Devices could also allow users to override the level of support for certain properties previously defined in the device info structure (referred to from this point forth as field filtering).

Support for receiving filters MUST be indicated in the device info for each data store. Support MUST be indicated by the inclusion of the `Filter-Rx` element within the `Datastore` element. The `Filter-Rx` element MUST contain a `CTType` and a `VerCT` element. The `CTType` element specifies the filtering grammar supported. For every `Filter-Rx` element a corresponding `FilterCap` element MUST be included in the `Datastore` element specifying any keywords or property names that can be filtered on.

A filter is specified by including the Filter element for the target data store in a SyncAlert command. The recipient stores the specified filter criteria, and performs synchronization based on the filter criteria. The specified filter criteria will be applicable only for this data store sync within this synchronization session.

When a Filter element is present, the Filter Meta *Type* attribute MUST be included and MUST correspond to the MIME type the filter applies to. Within the Filter element, the Record, Field element and *FilterType* attribute MAY be included and all MAY be present.

The Record element MUST contain an Item containing a Meta *Type* attribute representing the filter type used, and one Data element representing the query data. The Data element MUST be a logical expression whereby the expression MUST only contain values defined in the FilterCap element.

The Field element MUST contain an Item containing a Meta *Type* attribute representing the device information MIME type and one Data element containing one or more Property elements. The mark-up characters of the Data element content MUST be properly escaped according to [XML] specification rules or the CDATA sections MUST be used. The Property elements override the corresponding property in the CTCap element for the current synchronisation session. Only the properties that differ from the properties specified in the CTCap element MAY be specified.

If WBXML encoding is used, no more than one property MAY be specified in the Data element. Specifying more than one property in WBXML document violates the rules for well-formed WBXML documents.

The *FilterType* attribute MUST contain a keyword that indicates the type of behavior that the sender is requesting. If the *FilterType* attribute is not present, then the *FilterType* value of "EXCLUSIVE" MUST be assumed.

If an implementation receives a filter record request for a data store that does not support filtering, a status code of 406 (OPTIONAL feature not supported) MUST be returned for the command containing the Filter element. If a filter record request specifying a filter type that is not supported by the data store is received, a status code 415 (unsupported media type or format) MUST be returned for the command containing the Filter element. If a filter record request is received which is syntactically incorrect or contains a query that is not supported then a status code of 422 (bad CGI or filter query) MUST be returned for the command containing the Filter element. If any of those error conditions occur, the sender of the filter MAY attempt to resend a new query. If the second query fails as well, a sender SHOULD either remove the filter query or terminate the synchronization.

If an implementation received a filter field request for a data store containing properties not previously defined in the corresponding CTCap element, then a status code of 400 (bad request) SHOULD be returned. Otherwise, the recipient of the filter field request MUST override any properties previously retrieved in the CTCap element in the device info with the properties present in the filter field request. The properties MUST only be overridden for the current synchronization session only.

5.22.1 Filter Behavior Definition

Filtering allows an implementation (most often a client) to constrain the set of items in a data store it wishes to synchronize against and to further constrain the data returned.

The filter only applies to the recipient, that is, an implementation that sends a filter for a synchronisation session is not constrained in the set of items it might send. Using a Filter will allow synchronization with a subset of the data in the data store.

When a Filter is being used during a Sync, the set of data that is defined by the Filter MUST be fully synchronized during a normal synchronization operation.

If a subsequent SyncAlert command is sent for the same datastore but with a different Filter command, the set of data that is defined by the new Filter command MUST be fully synchronized during a normal synchronization operation. The recipient of the Filter command MUST no longer send items that were part of the previous filter if those items are no longer part of the new filter.

A recipient MAY choose how to insure that this expectation is met. For example, it might require requesting a slow sync, or it might require re-sending records that have been previously synchronized with a different set of fields.

5.22.2 Filter Query Syntax

The filter query is a logical expression contained in the `Filter Record` element and is applied to each item in the recipient's datastore. Often, the values of properties in the data items are compared to literal values supplied by the requestor. Items for which the expression evaluates to true are the set of items for that synchronization session. The filter query is expressed according to a particular grammar. The `Record Item Meta` *`Type`* indicates the grammar of the filter query supplied in the `Data` element. This enables the protocol to support additional filter grammars without sacrificing interoperability. The list of grammar types an implementation is capable of receiving MUST be indicated through the use of the device info `Filter-Rx` element.

Comparison items MUST be valid property names or keywords specified in the `FilterCap` element for the particular filter query grammar being used and all comparison operators MUST be supported for each comparison item. Literal values used in comparisons MUST be valid for the property or keyword according to the content type being used for the query. Comparisons are performed using the character encoding specified in the content type, where appropriate.

A grammar MAY provide logical operators for conjoining sub-expressions (e.g. AND, OR, NOT) to create arbitrarily complex expressions. A grammar MAY provide mechanisms for selecting items based on the presence of properties.

5.22.2.1 Content type requirements

If an implementation supports receiving filters on a given data store, all expressions that test the values of certain base media object properties for that data store, regardless of the query grammar used, are OPTIONAL. If the expression is unsupported by the recipient, one of the previously listed status codes MUST be returned to the sender. The sender SHOULD then modify the expression based on the device info obtained from the recipient. If no expression can be agreed upon between the sender and the recipient then it is up to the sender to determine if the synchronization can be sent without any Filter element or if the synchronization SHOULD be aborted.

Filtering for all contents types is OPTIONAL and MAY be supported.

5.22.2.1.1 Contacts Media Object Filter

Filtering for vCard 2.1 [IMCVCARD] and vCard 3.0 [RFC2426] objects can be specified using both `Record` and `Field` elements. In the case of `Record` elements, the set of recommended keywords to support are as follows:

```
ct-filter-keyword = "CATEGORIES" | "GROUP"
```

If one chooses to filter based on a property name, some properties like Name ("N") might have several fields. Individual fields could be indicated using a subscript notation. Thus, "N[1]" refers to the family name and "N[2]" refers to the given name.

5.22.2.1.2 Calendar Media Object Filter

Filtering for vCalendar 1.0 [IMCVCAL] and iCalendar 2.0 [RFC2445] objects can be specified using both `Record` and `Field` elements. In the case of `Record` elements, the set of recommended keywords to support are as follows:

```
ct-filter-keyword = "SINCE" | "BEFORE" | "STATUS"
```

The format of the "SINCE" and "BEFORE" keywords MUST be a date-time or date format as specified below.

```
date-time     = date "T" time
date          = date-value
date-value    = date-fullyear date-month date-mday
date-fullyear = 4DIGIT
date-month    = 2DIGIT ;01-12
date-mday     = 2DIGIT ;01-28, 01-29, 01-30, 01-31 (based on month/year)
time          = time-hour time-minute time-second time-utc
```

```
time-hour     = 2DIGIT        ;00-23
time-minute   = 2DIGIT        ;00-59
time-second   = 2DIGIT        ;00-59
time-utc      = "Z"
```

The “SINCE” keyword represents items that are within or later than the specified date while the “BEFORE” keyword represents items that are earlier than the specified date.

Implementations that support receiving filters for calendar media objects are responsible for expanding recurrence rules (“RRULE” properties) to determine if any instances match the filter conditions. Also, when specifying a date-time format, the use of UTC MUST be used in order to avoid time zone ambiguities. Implementations that cannot provide a UTC date-time value MUST provide a date value instead.

5.22.2.2 CGI Syntax

This section specifies a CGI-like filtering syntax. When using this syntax, the `Filter Item Meta Type` element MUST be ‘syncml:filtertype-cgi’. All implementations that support receiving filter requests MUST support the “syncml:filtertype-cgi” grammar.

The format for the CGI scripting is defined here in an ABNF notation [RFC2234] and the grammar defined here is largely the same as the grammar defined in the OMA DS 1.1.2 and OMA DS 1.2.1 specification.

If the CGI syntax is supported for a data store, then all of the logical CGI scripting primitives in the following table MUST be supported.

CGI syntax queries may contain at most 1 type of logical separator, but they MAY contain several logical separators of the same type. For example, they MAY contain several “&OR;” logical operators but a query cannot contain both an “&AND;” and an “&OR;” logical operator in the same expression.

The SPACE character MUST be specified by the hexadecimal encoding as stated by the format specification for Uniform Resource Identifiers.

Since the use of lexographic comparison operators is locale specific (for example the use of the “<” operator), devices SHOULD NOT use these operators when specifying a filter for free form text properties. Instead, the following logical operators SHOULD be used in place: “&EQ;”, “&iEQ;”, “&NE;”, “&iNE;”, “&CON;”, “&iCON;”.

Queries using value-based properties (properties that may only contain specific pre-defined values) SHOULD only use the following operators: “&EQ;”, “&iEQ;”, “&NE;”, “&iNE;”.

```
VCHAR = %x20-7E   ;Visible latin characters within UTF-8 or SPACE character

string-value = 1*VCHAR  ;Case sensitive string value

log-equalitycomp = "&EQ;"  ;Equal To (case sensitive)

/                  "&iEQ;" ;Equal To (case insensitive)

/                  "&NE;"  ;Not Equal To (case sensitive)

/                  "&iNE;" ;Not Equal To (case insensitive)


log-op = log-equalitycomp

/ "&GT;"                ;Greater Than (case sensitive)

/ "&iGT;"               ;Greater Than (case insensitive)

/ "&GE;"                ;Greater Than Or Equal To (case sensitive)
```

```
/ "&iGE;"               ;Greater Than Or Equal To (case insensitive)

/ "&LT;"                ;Less Than (case sensitive)

/ "&iLT;"               ;Less Than (case insensitive)

/ "&LE;"                ;Less Than Or Equal To (case sensitive)

/ "&iLE;"               ;Less Than Or Equal To (case insensitive)

/ "&CON;                ;Contains the value (case sensitive)

/ "&iCON;               ;Contains the value (case insensitive)

/ "&NCON;               ;Does Not Contain the value (case sensitive)

/ "&iNCON;              ;Does Not Contain the value (case insensitive)


log-sep = "&OR;"        ;Logical OR

/         "&AND;"       ;Logical AND

luid-expression = "&LUID;" log-equalitycomp string-value

ct-no-value = "&NULL;"   ; No property value for the item

ct-filter-keyword = string-value   ; Valid content-type specific filter keywords

ct-filter-value = string-value  ; Valid content-type specific property value

ct-expression = ct-filter-keyword (log-op ct-filter-value | log-equalitycomp ct-no-value)

filter-expression =  ct-expression | luid-expression filter-query = filter-expression *(log-
sep filter-expression)
```

5.22.3 Indicating Filter Support

Implementations that support filtering MUST indicate their support in the device info. For each data store, a device indicates the list of filter grammars it is capable of receiving in the `Filter-Rx` element.

5.22.3.1 Minimum Requirements for Filtering support

Implementations that support filtering MUST support receiving the "syncml:filtertype-cgi" grammar. Specifically, they MUST include at least one `Filter-Rx` element specifying the receiving of the "syncml:filtertype-cgi" grammar.

5.22.4 Handling Data Outside Filter Criteria

The following outlines how to handle synchronizing data outside the filter criteria

1. When using an exclusive filter type, the DS Server not only sends all of its changes to the client, but the DS Server MUST also send `Delete` commands for all client items that are outside the filter criteria. If the DS Client supports the OPTIONAL Soft Delete, the DS Server MAY send Soft Deletes; otherwise, the DS Server MUST send Hard Deletes.

2. When using an inclusive filter type, before the DS Client sends any of its changes to the DS Server, the DS Client MAY choose to send Soft Deletes for all items that were outside the filter criteria after the previous synchronization.

Note that if the data items about to be Soft Deletes have never been sent to the DS Server, the DS Client SHOULD send them prior to issuing the Soft Deletes. Note that if a DS Client chooses to delete the items outside of the filter query instead of archiving them, it SHOULD do so immediately after a synchronization in order to prevent the user from modifying the items between synchronizations and thus potentially losing changes made by users.

3. When both the DS Client and DS Server support equivalent filter queries, the DS Client and DS Server exchange only the data within the filter criteria.

5.22.5 Examples

The following examples are provided to further illustrate the usage of filtering in OMA DS 2.0.

5.22.5.1 Contact Media Objects

5.22.5.1.1 Example 1

In this scenario, the client wishes to sync only Contact items that fall into the "business" or "personal" group.

1. During the initial sync, the client and server exchange their device info.
2. The client analyses the server's device info, and the client notes that the server supports receiving filters on the Contacts data store for queries using the "syncml:filtertype-cgi" grammar.
    a. The server includes in its device info the `Filter-Rx` and `FilterCap` elements that it supports.
    b. The client doesn't require filtering on any additional fields, so it determines that this server supports the filter it wishes to send.

```
<DataStore DisplayName="Contacts DB" >
   <SourceRef>./contacts</SourceRef>
      ...
   <RxTx-CT>
      ...
   </RxTx-CT>
   <CTCap>
      ...
   </CTCap>
   <SyncCap FPUnique="true"/>
   <Filter-Rx>
      <CTType>syncml:filtertype-cgi</CTType>
      <VerCT>1.0</VerCT>
   </Filter-Rx>
   <FilterCap>
      <CTType>syncml:filtertype-cgi</CTType>
      <VerCT>1.0</VerCT>
      <FilterKeyword>GROUP</FilterKeyword>
      <PropName>CATEGORIES</PropName>
   </FilterCap>
</DataStore>
```

3. The client sends a `SyncAlert` for the Contacts data store with a filter.
    a. It includes the `Filter Meta` *`Type`* attribute to indicate the content type desired (vCard in this example).
    b. It includes a `Filter Record` element with a `Meta` *`Type`* value of "syncml:filtertype-cgi" to indicate the grammar being used.
    c. The filter query in the `Item Data` element contains a value of "GROUP&iCON;business&OR; GROUP &iCON;personal" to constrain the items synchronized to those that fall into the "business" or "personal" group (case insensitive).

```
<SyncAlert CmdID="5">
   <!--  -->
   <Filter>
        <Meta Type="text/x-vcard" />
       <Record>
            <Item>
               <Meta Type="syncml:filtertype-cgi" />
               <Data> GROUP&iCON;business&OR;GROUP&iCON;personal</Data>
            </Item>
        </Record>
</Filter>
</SyncAlert>
```

4. The server receives the `SyncAlert` with the `Filter Record` element.
    a. It determines that it supports the filter operation for the data store, content type, filter grammar, and properties.
    b. It replies with a status code of 200 for the `SyncAlert`, indicating that it can satisfy the request to sync with filtering.
5. The synchronization process continues normally.
    a. The client sends all of its changes to the server (the filter constraint is not imposed on it in this scenario).
    b. The server sends changes only for items that satisfy the filter query.

5.22.5.1.2 Example 2

In this scenario, the client wishes to sync only Contact items that fall into the "business" or "personal" group. Additionally the client has indicated in its device info that it supports the PHOTO property, but it does not wish to receive the PHOTO property from the server for this synchronization request.

1. During the initial sync, the client and server exchange their device info.
2. The client analyses the server's device info, and the client notes that the server supports receiving filters on the Contacts data store for queries using the "syncml:filtertype-cgi" grammar.
    a. The server includes in its device info the `Filter-Rx` and `FilterCap` elements that it supports.
    b. The client does not require filtering on any additional fields, so it determines that this server supports the filter it wishes to send.
3. The client sends a `SyncAlert` for the Contacts data store with a filter.
    a. It includes the `Filter Meta` *`Type`* element to indicate the content type desired (vCard in this example).
    b. It includes a `Filter Record` element with a `Meta` *`Type`* value of "syncml:filtertype-cgi" to indicate the grammar being used.
    c. The filter query in the `Item Data` element contains a value of "GROUP&iCON;business&OR;GROUP&iCON;personal" to constrain the items synchronized to those that fall into the "business" or "personal" group (case insensitive).
    d. It includes a `Filter Field` element containing a Property element set to "PHOTO" containing a MaxSize element set to 0 (zero).

```
<SyncAlert CmdID="5">
   <!--    -->
   <Filter>
      <Meta Type="text/x-vcard" />
```

```
      <Field>
            <Item>
                  <Meta Type="application/vnd.syncml-devinf+xml" />
                  <Data>
                     <![CDATA[
                        <Property>
                           <PropName>PHOTO</PropName>
                           <MaxSize Truncate="false">0</MaxSize>
                        </Property>
                     ]]>
                  </Data>
               </Item>
      </Field>
      <Record>
            <Item>
                  <Meta Type="syncml:filtertype-cgi" />
                  <Data><![CDATA[GROUP&iCON;business&OR;GROUP&iCON;personal]]></Data>
            </Item>
      </Record>
   </Filter>
</SyncAlert>
```

4. The server receives the `SyncAlert` with the `Filter Record` and `Field` elements.
    a. It determines that it supports the filter operations for the data store, content type, filter grammar, and properties.
    b. It replies with a status code of 200 for the `SyncAlert`, indicating that it can satisfy the request to sync with filtering.
5. The synchronization process continues normally.
    a. The client sends all of its changes to the server (the filter constraint is not imposed on it in this scenario).
    b. The server sends changes only for items that satisfy the filter query. The server does not send any PHOTO properties since the client has requested that it wishes to receive only 0 bytes of this property for this synchronization request and the value SHOULD not be truncated.

5.22.5.2 Calendar Media Objects

5.22.5.2.1 Example 1

In this scenario, the client wishes to synchronize calendar items that fall within a two week window of time (starting with the current date).

1. During the initial sync, the client and server exchange their device info.
2. The client analyses the server's device info, and the client notes that the server supports receiving filters on the Calendar data store for queries using the "syncml:filtertype-cgi" grammar.
    a. The server includes in its device info the `Filter-Rx` and `FilterCap` elements that it supports. This includes the SINCE and BEFORE keywords.
    b. The client doesn't require filtering on any additional fields, so it determines that this server supports the filter it wishes to send.

```
<DataStore DisplayName="Calendar Agenda DB">
   <SourceRef>./calendar/events</SourceRef>
   ...
   <CTCap>
      ...
   </CTCap>
   <SyncCap FPUnique="true"/>
```

```
   <Filter-Rx>
      <CTType>syncml:filtertype-cgi</CTType>
      <VerCT>1.0</VerCT>
   </Filter-Rx>
   <FilterCap>
      <CTType>syncml:filtertype-cgi</CTType>
      <VerCT>1.0</VerCT>
      <FilterKeyword>BEFORE</FilterKeyword>
      <FilterKeyword>SINCE</FilterKeyword>
   </FilterCap>
</DataStore>
```

3. The client sends a `SyncAlert` for the Calendar data store with a filter.

    a. It includes the `Filter Meta` *`Type`* element to indicate the content type desired (iCalendar in this example).

    b. It includes a `Filter Record` element with a `Meta Type` value of "syncml:filtertype-cgi" to indicate the grammar being used.

    c. The filter query in the `Item Data` element contains a value of "SINCE&EQ;20030606T000000Z&AND;BEFORE&EQ;20030620T000000Z" to constrain the items synchronized to those that occur between June 6, 2003 and June 19, 2003 inclusive, using the "SINCE" and "BEFORE" keywords.

```
<SyncAlert>
   <!--   -->
   <Filter>
      <Meta Type="text/x-vcalendar"/>
      <Record>
         <Item>
            <Meta Type="syncml:filtertype-cgi" />
            <Data><![CDATA[SINCE&EQ;20020707T000000Z&AND;BEFORE&EQ;20020728T000000Z]]></Data>
         </Item>
      </Record>
   </Filter>
</SyncAlert>
```

4. The server receives the `SyncAlert` with the `Filter Record` element.

    a. It determines that it supports the filter operation for the data store, content type, filter grammar, and properties.

    b. It replies with a status code of 200 for the `SyncAlert`, indicating that it can satisfy the request to sync with filtering.

5. The synchronization process continues normally.

    a. The client sends all of its changes to the server (the filter constraint is not imposed on it in this scenario).

    b. The server sends changes only for items that satisfy the filter query.

5.22.5.2.2 Example 2

In this scenario, the client wishes to synchronize task items (iCalendar vTODO components) that have not been completed and are due within the next 2 weeks (using the DUE iCalendar property). The client also wishes to limit the size of the DESCRIPTION property to 100 bytes and only receive ATTACH properties that are less than 1000 bytes.

1. During the initial sync, the client and server exchange their device info.

2. The client analyses the server's device info, and the client notes that the server supports receiving filters on the Calendar data store for queries using the "syncml:filtertype-cgi" grammar.

a. The server includes in its device info the `Filter-Rx` and `FilterCap` elements that it supports. This includes the STATUS and DUE keywords.

b. The client doesn't require filtering on any additional fields, so it determines that this server supports the filter it wishes to send.

```
<DataStore DisplayName="Task DB">
   <SourceRef>./calendar/tasks</SourceRef>
      ...
   <CTCap>
      ...
   </CTCap>
   <SyncCap FPUnique="true"/>
   <Filter-Rx>
      <CTType>syncml:filtertype-cgi</CTType>
      <VerCT>1.0</VerCT>
   </Filter-Rx>
   <FilterCap>
      <CTType>syncml:filtertype-cgi</CTType>
      <VerCT>1.0</VerCT>
      <PropName>DUE</PropName>
      <PropName>STATUS</PropName>
   </FilterCap>
</DataStore>
```

3. The client sends a `SyncAlert` for the Calendar data store with a filter.

a. It includes the `Filter Meta` *`Type`* element to indicate the content type desired (iCalendar in this example).

b. It includes a `Filter Record` element with a `Meta` *`Type`* value of "syncml:filtertype-cgi" to indicate the grammar being used.

c. The filter query in the `Item Data` element contains a value of "DUE&LE;20030620T000000Z&AND;STATUS&NE;COMPLETED" to constrain the items synchronized to those that are due before June 20, 2003 and have not been completed.

d. It includes a `Filter Field` element containing a Property element set to "DESCRIPTION" containing a MaxSize element set to 100 and a second property element set to "ATTACH" containing a MaxSize of 1000 with the Truncate flag set to false.

```
<SyncAlert>
   <!--   -->
   <Filter>
      <Meta Type="text/x-vcalendar" />
      <Field>
         <Item>
            <Meta Type="application/vnd.syncml-devinf+xml" />
            <Data>
                    <![CDATA[
                       <Property>
                          <PropName>DESCRIPTION</PropName>
                          <PropInfo>
                             <MaxSize>100</MaxSize>
                          </PropInfo>
                       </Property>
                       <Property>
                          <PropName>ATTACH</PropName>
                          <PropInfo>
                              <MaxSize Truncate="false">1000</MaxSize>
                           </PropInfo>
                       </Property>
                    ]]>
            </Data>
```

```
        </Item>
      </Field>
      <Record>
        <Item>
          <Meta Type="syncml:filtertype-cgi" />
          <Data><![CDATA[DUE&LE;20030728T000000Z&AND;STATUS&NE;COMPLETED]]></Data>
        </Item>
      </Record>
   </Filter>
</SyncAlert>
```

4. The server receives the `SyncAlert` with the `Filter Record` element.

    a. It determines that it supports the filter operation for the data store, content type, filter grammar, and properties.

    b. It replies with a status code of 200 for the `SyncAlert`, indicating that it can satisfy the request to sync with filtering.

5. The synchronization process continues normally.

    a. The client sends all of its changes to the server (the filter constraint is not imposed on it in this scenario).

    b. The server sends changes only for items that satisfy the filter query. The server has truncated the DESCRIPTION properties to 100 bytes and has also not sent any ATTACH properties that are larger than 1000 bytes.

6. Overall Data Synchronization Flow

6.1 Overview

One data synchronization process is a complete one-time data exchange process between the DS Client and DS Server. It includes sync initialization phase (Pkg #1 and Pkg#2), data exchange phase (Pkg #3 and Pkg #4) and mapping phase (Pkg #5 and Pkg#6). Detailed procedures of the sync process are described in section 7.

During one data synchronization session, the DS Client and DS Server can perform one or more data synchronization processes. After one complete sync process, the DS Client and DS Server can maintain the session and perform data synchronization continuously. This is called continuous sync. DS Client and DS Server MUST support continuous sync functionality.

With this functionality, the DS Client and DS Server can perform data synchronization continuously. This will be very helpful in some circumstances, for example, the data items are changed frequently, real time sync is required, or the DS Client and DS Server are connected through cables, not OTA. And this can avoid the effort to initiate a session frequently.

The overall sync flow is depicted in the figure below.

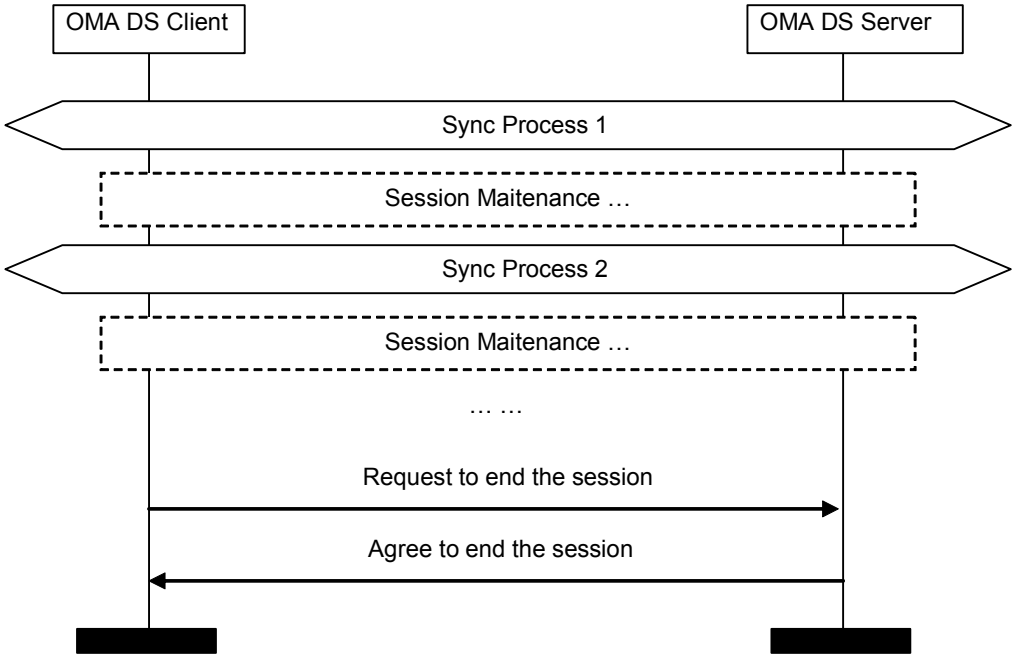

Figure 8: Overall Sync Flow

Between different sync processes, the DS Client or DS Server MUST send session maintenance commands (`Alert` 212 or `Alert` 213) to the other side to maintain the session.

For the sync process after session maintenance, the requirements for the sync flow are the same as the first sync process during the session. The sync type parameters negotiation is needed, but the device information negotiation SHOULD be omitted.

The DS Client or DS Server SHOULD send session end command (`Alert` 211) to the other side to end the session, or the session can be ended unintentionally, for example, loss of network coverage or phone malfunctions.

The session maintenance commands and the session end command SHOULD be sent in the last package of the sync process. That is, Pkg #5 for the DS Client or Pkg #6 for the DS Server. It is recommended that DS Server opens the session and waits for the DS Client to end the session.

6.2 Session Maintenance

There are two possible ways to maintain the session. They are called “Alert Poll” and “Alert Idle”.

The DS Client and DS Server MUST support both of them.

The DS Client and DS Server MAY choose to use either Alert Poll or Alert Idle to maintain the session. During one session, the same one SHOULD be used.

Either the DS Client or the DS Server can send session maintenance commands to the other side to request to maintain the session. The recipient SHOULD agree to continue the session. But the recipient MAY choose to refuse and end the session.

During the session maintenance period, either DS Client or DS Server MAY send new SyncML commands to perform data synchronization.

The session maintenance request message and response message MUST include `SyncML` element and the corresponding required elements according to the DS Syntax specification. For security reason, the authentication information SHOULD be included in session maintenance messages.

6.2.1 Alert Poll

The alert code for Alert Poll is 212.

The session maintenance flow using Alert Poll is depicted in the figure below.

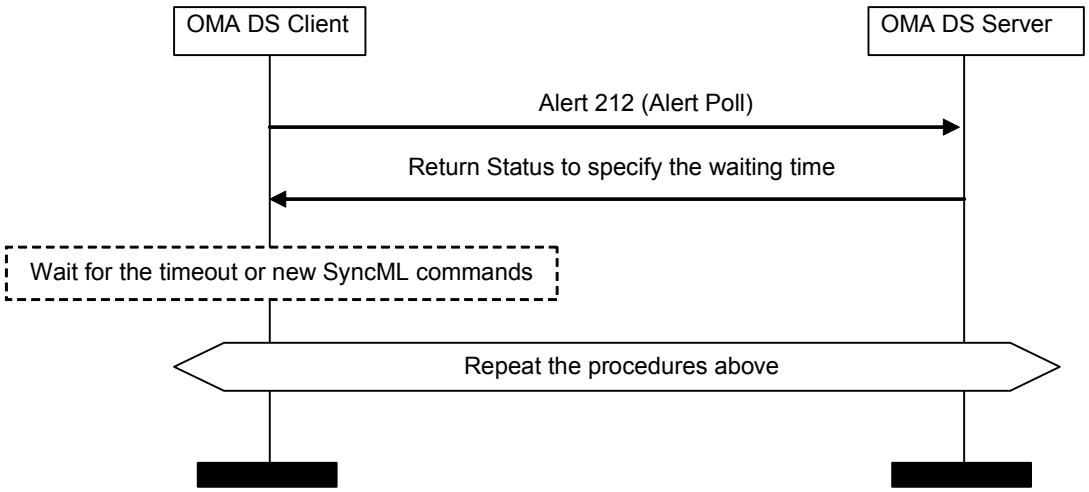

Figure 9: Alert Poll

The requestor sends `Alert` 212 command to the recipient, and the recipient responds with the expected waiting time in `Status/Item/Data`. The waiting time is specified in seconds.

The requestor waits for the timeout or new SyncML commands. If the requestor does not receive any SyncML commands from the other side and the requestor has no new SyncML commands to send during the waiting time, after the timeout, the requestor SHOULD send the next `Alert` 212 command. And the procedures will repeat until either side has new SyncML commands to send or either side determines to end the session.

Examples:

Alert poll request from client:

```
<Alert CmdID="2" Code="212" />                <!--Alert Poll-->
```

Alert poll response from server:

```
<Status CmdID="2" MsgRef="12" CmdRef="2" Cmd="Alert" Code="200">
  <!--Agree to maintain-->
   <Item>
       <Meta Format="int" />
       <Data>120</Data>    <!--Waiting time, specified in seconds-->
   </Item>
</Status>
```

6.2.2 Alert Idle

The alert code for Alert Idle is 213.

The session maintenance flow using Alert Idle is depicted in the figure below.

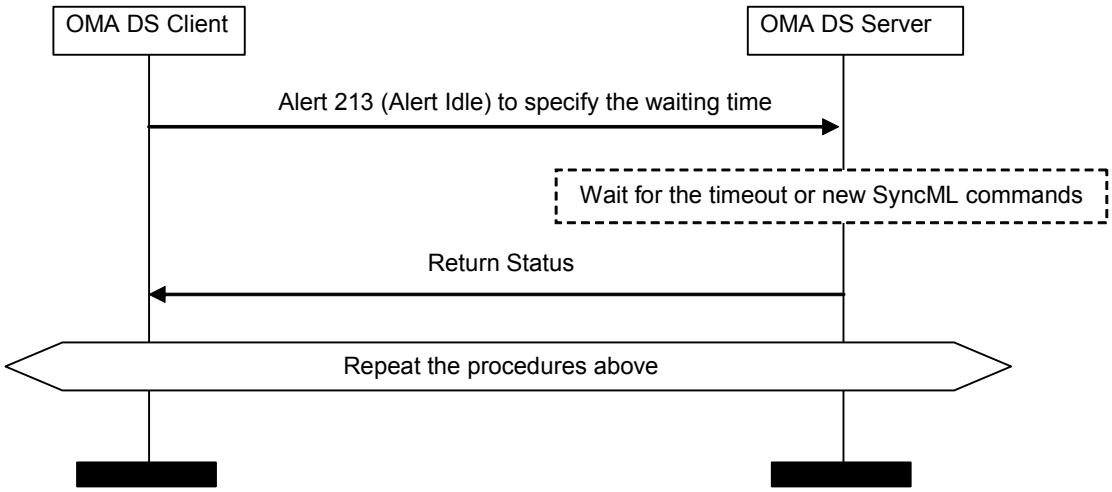

Figure 10: Alert Idle

The requestor sends `Alert` 213 command to specify the waiting time to the recipient. The expected waiting time is specified in `Status/Item/Data`. And the waiting time is specified in seconds.

The recipient waits for the timeout or new SyncML commands. If the recipient does not receive any SyncML commands from the other side and the recipient has no new SyncML commands to send during the waiting time, after the timeout, the recipient sends back the `Status` command.

The requestor sends `Alert` 213 again. And the procedures will repeat until either side has new SyncML commands to send or either side determines to end the session.

Examples:

Alert idle request from client:

```
<Alert CmdID="2" Code="213">              <!--Alert idle-->
   <Item>
       <Meta Format="int" />
       <Data>120</Data>                   <!--Waiting time, specified in seconds-->
   </Item>
</Alert>
```

Alert idle response from server:

```
<Status CmdID="2" MsgRef="13" CmdRef="2" Cmd="Alert" Code="200" />   <!--Agree to maintain-->
```

6.3 Session End

Either the DS Client or the DS Server MAY send session end commands to the other side to request to end the session. The recipient MUST agree to end the session. If the recipient wants to continue the session, the recipient MAY initiate a new session.

The requestor MAY specify the `NoStatus` attribute in the `Alert` command, then no response will be sent by the recipient.

Examples:

Session end request from client:

```
<Alert CmdID="2" Code="211" />                <!--Session end-->
```

Session end response from server:

```
<Status CmdID="2" MsgRef="14" CmdRef="2" Cmd="Alert" Code="200" /> <!--Agree to end session--
>
```

6.4 Error Cases

If the recipient refuses to continue the session, status code "`(518) refuse to continue the session`" is created by the session maintenance commands.

7. Data Synchronization Flow for Single Sync Process

7.1 Overview

Individual data synchronization processes occur within the scope of an active data synchronization session (See Chapter 6, and Figure 8. While a DS Session is open and not already in a Sync process, a Sync Process may be initiated by either the DS Client or the DS Server by sending a `SyncAlert` command.

Historically, the steps during synchronization were broken up into packages, numbered Pkg #0 through Pkg #6. Since Continuous Sync allows for these steps to be repeated, as well as split up slightly differently, text descriptions will be used here along with the former package number. Also note that Pkg #1 formerly included authentication, whereas with Continuous Sync that may occur at any time prior to or during the current sync process.

For ease of implementation, all sync processes have the same core logical flow, shown in Figure 11. A single sync process is allowed to synchronize multiple data stores, but all data stores MUST proceed through packages in unison – e.g. there is a single Pkg #1 containing the SyncAlerts for all data stores sent by the DS Client, and so on.

Note that packages are a logical concept, and actual message flow may overlap. E.g. the first message of a session may include authentication, `SyncAlert` from Client (Pkg #1), and Sync Package from Client (Pkg #3), where each piece assumes the prior pieces were successful, or additional sync processes may be started in Pkg #6. Additionally, Sync Packages that require more than one message will typically receive statuses immediately (Pkg #4 partially overlapping with Pkg #5, and Pkg #5 partially overlapping with Pkg #6).

Beyond the implicit order requirements, such as being unable to send a status for a command that has not yet been sent, there is an explicit requirement that the Sync Package from the server for each data store MUST NOT be sent before the corresponding Sync Package from the client for that data store has been completed.

The typical sync flow for a single sync process is depicted in the figure below.

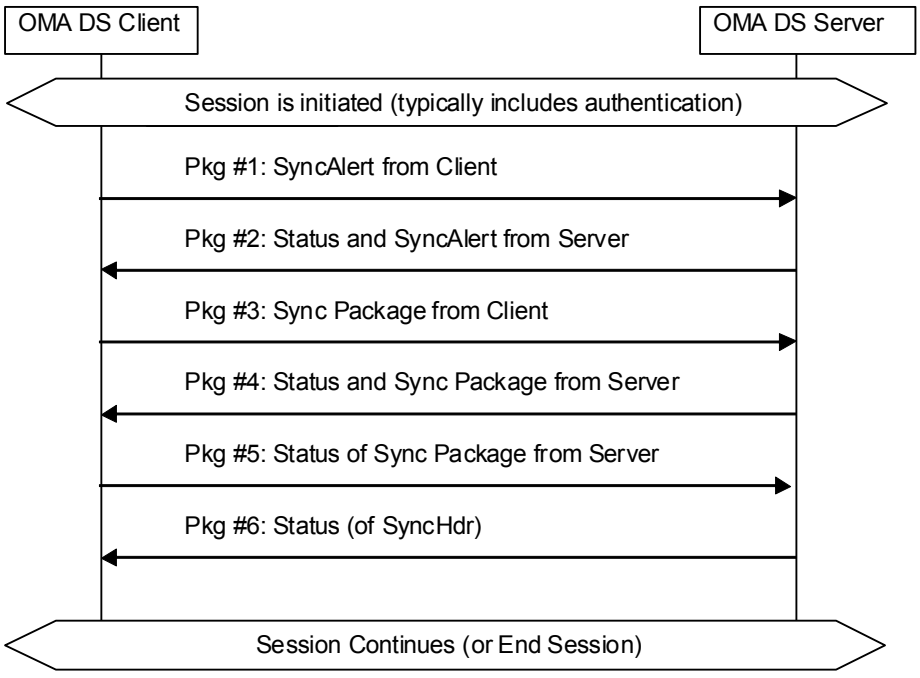

Figure 11: Typical Sync Flow for Single Sync Process

Descriptions for the above sync flows are as follows:

Pkg #1: The DS Client sends the `SyncAlert` command with the client requested sync type parameters to the DS Server. If this is the first sync process in the session, Pkg #1 typically also includes authentication information and possibly device information. Optionally, the `SyncAlert` will include fingerprints of the data items in the case of a recovery sync. (See section 8)

Pkg #2: The DS Server responds and sends statuses for the Pkg #1 commands, and a `SyncAlert` command with server specified sync type parameters to the DS Client. It also may include authentication information or device information. Optionally, the `SyncAlert` will include the data item identifiers that the client should send in the case of a recovery sync. (See section 8)

Pkg #3: The DS Client sends statuses for the Pkg #2 commands, and sends the sync package with the modified or specified data items and the associated fingerprints to the DS Server.

Pkg #4: The DS Server sends statuses for the Pkg #3 commands and sends back the server side modified data items to the DS Client.

Pkg #5: The DS Client sends statuses for the Pkg #4 commands. Optionally, the DS Client sends mapping information and/or fingerprints.

Pkg #6: The DS Server sends back the status for the client sent package (Pkg #5).

Then the whole sync process completes. The session MAY be closed or continued according to the session maintenance commands or session end command.

Note that the contents of the sync packages may be different or absent according to the different sync type parameters.

7.2 Sync Flows

While there are many possible specific combinations of things that can be done, there are three general sync flows: normal sync, client initiated recovery sync and server initiated recovery sync, show below.

Specific packages have the following requirements:

A sync process initiated by a DS Client starts with the DS Client sending a `SyncAlert` to the DS Server (Pkg #1). This MAY contain an `IDContainer` containing `SourceClientURI`s (LUIDs) with *FP* Attributes. The DS Server responds with a status for the DS Client's `SyncAlert`, and if the DS Server agrees to perform the requested sync, a `SyncAlert` after the status for the DS Client's `SyncAlert` (Pkg #2). A `SyncAlert` from the DS Server MAY contain an `IDContainer`, in which case the listed `SourceClientURI`s are to be used as the set of data for the DS Client to send to the DS Server. If no `IDContainer` is sent, then the set of data for the DS Client to send will be based upon the DS Client's change log. If the status indicates that a recovery sync is required, the client SHOULD request a new sync (and thus a new Pkg #1), including an `IDContainer` containing `SourceClientURI`s (LUIDs) with *FP* Attributes.

A sync process initiated by a DS Server starts with the DS Server sending a `SyncAlert` to the DS Client (No previous package identifier). This SHOULD trigger the DS Client to initiate a sync process as shown above, with the DS Server specified sync parameters. After the server has sent the `SyncAlert`, and if the client does not agree with the sync anchor in that `SyncAlert`, then the Client MUST start a recovery sync. This is done by sending back a `Status` on that `SyncAlert` with 'Recovery Required' (508). In this same message, the client SHOULD start the recovery sync with a new `SyncAlert` containing an `IDContainer`. In this case, this package can be considered as a repeated Pkg #1. Note that it is not necessary for the client to compare the sync anchor from the server, and an extended exchange of `SyncAlert`s is not recommended due to the possibility of looping.

Once initialization is complete, if the sync parameters specify that client data should be sent, the DS Client would begin sending data to the DS Server. This is called the Sync Package from Client (Pkg #3). The last message containing DS Client data items from the specified datastore MUST include the Final flag. Note that the statuses for these commands were previously considered part of Pkg #4. If the sync parameters specify that server data should be sent, the DS Server would

then send its data to the DS Client. This is called the Sync Package from Server (Pkg #4). The DS Server data items MUST NOT be sent before the Final flag is received from the client, if the client was to send any data items. The last message containing DS Server data items from the specified datastore MUST include the `Final` flag. The statuses from the DS Client (which may include new client identifiers and FPs) are considered Pkg #5. The statuses from the DS Server of messages containing Pkg #5 information are considered Pkg #6.

Once the DS Server has received and sent statuses for all the data it expects from the DS Client (e.g. Pkg #3 and Pkg #4, if the sync parameters specify the client should send data), and sends all the data it expects to send and receives matching statuses (e.g. Pkg #4 and Pkg #5), it should update its sync anchors, and consider the sync process as complete.

Once the DS Client has sent and received statuses for all the data it expects to send (e.g. Pkg #3 and Pkg #4, if the sync parameters specify the client should send data), and receives all the data it expects from the DS Server (e.g. Pkg #4, if the sync parameters specify the server should send data), sends matching statuses (Pkg #5), and receives a status for the last message (Pkg #6), it should update its sync anchors, thus completing the sync process.

7.2.1 Normal Sync

7.2.1.1 Overview

When both sides have no problems, that is, the change log is valid (*`ChangeLogValidity`* set as "true"), the sync anchors match, and the item identifiers and mapping table are valid (*`IDValidity`* set as "true"), the DS Client and DS Server can exchange the changed data items directly.

The two-way preserve sync flow (*`Direction`* set as "twoWay", *`Behavior`* set as "Preserve") is depicted in the figure below.

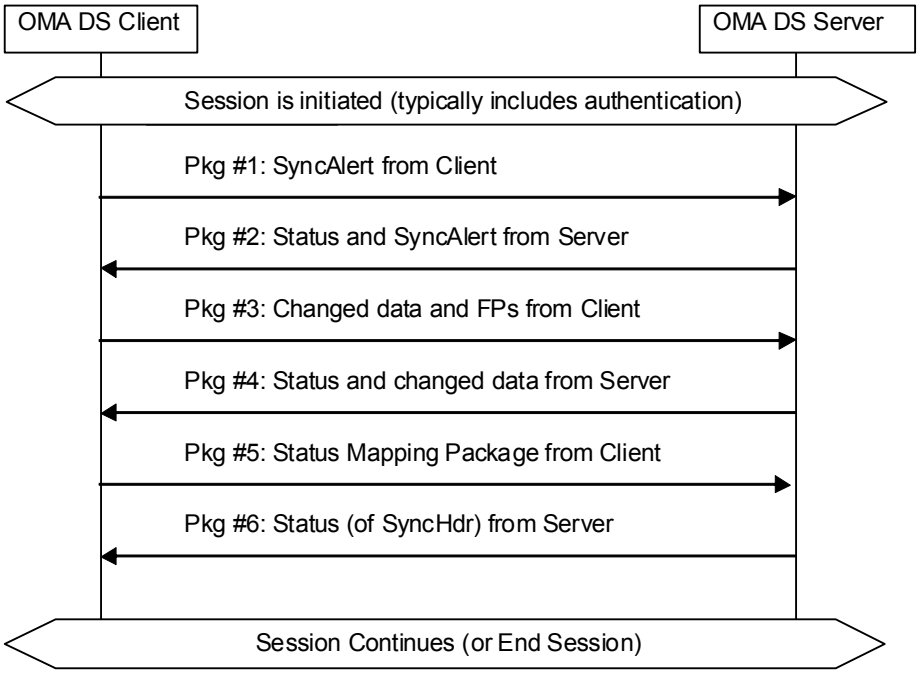

Figure 12: Normal Sync Flow

Descriptions and examples for the above sync flows are as the follows:

Pkg #1: The DS Client requests a sync process using the `SyncAlert` command.

Pkg #2: The DS Server compares the received *`Last`* anchor with the stored anchor value and sends back the *`Next`* Anchor. The DS Server sends back the response to indicate the sync type parameters using the `SyncAlert` command.

The sync type parameters MUST conform to the sync type transition rules specified in section 5.1.

To avoid round trips, after receiving the sync type parameters from the DS Server, the DS Client SHOULD accept the parameters. Otherwise, the DS Client SHOULD initiate another sync process.

Pkg #3: The DS Client sends the changed data items and the associated fingerprints to the DS Server.

Pkg #4: The DS Server returns status for the data items the client sent and sends the server side changed data items to the DS Client.

Note that the DS Server does not include fingerprints with its data items, because fingerprints are generated by the DS Client. Even if it is mutually generated fingerprints, the server does not need to send them back to the client.

Pkg #5: The DS Client returns status for the server modifications. The DS Client returns fingerprints for the server added or modified data items, and returns new assigned client identifiers for the server added data items.

Pkg #6: The DS Server returns statuses for the messages in Pkg #5.

Detailed requirements for Pkg #3, Pkg #4, Pkg #5 and Pkg #6 are specified in the following sections.

Note that if the direction is one way, that is, fromClient or fromServer, some packages can be simplified or omitted. If the direction is one way from client, the server does not need to send back the server modifications. If it is one way from server, the client does not need to send client modifications to the server. Packages not sent require no statuses.

7.2.1.2 Client Modifications to Server

To enable sync, the client needs to inform the server about all client data modifications, which have happened since the previous sync package with modifications has been sent from the client to the server. These modifications include also modifications which have happened during the previous sync process after the client has sent its modifications to the server. Any client modification, which is done after sending this package, MUST be reported to the server during the next sync process. It is not allowed to put them inside subsequent messages of the same sync process from the client to the server.

The requirements for the sync package from the DS Client to the DS Server are described as the following:

1. The `Sync` element MUST be included in `SyncBody` with the following clarifications.

   `NoStatus` SHOULD NOT be specified with the `Sync` command.

   The *`FreeMem`* attribute SHOULD be specified on the `Sync` element. If supplied this information MUST be sent in the first message of this package.

   `NumberOfChanges` MAY be used to indicate the number of changes in the source node.

2. If there are modifications in the client, the operational elements (e.g., `Add`, `Copy`, `Delete`, `Move`, and `Replace`) have the following requirements.

   `NoStatus` SHOULD NOT be specified with all these operations.

   The `SourceClientURI` element MUST be included to indicate the LUID of the data item within the `Item` element. If the fingerprint generation method is client generated fingerprint, the fingerprint of the data item MUST be included in the *`FP`* attribute of the `SourceClientURI` element within `Replace` and `Add` commands.

   The *`Type`* attribute MUST be included in the `Meta` element of items containing data to indicate the type of the data item (E.g., MIME type). The `Meta` element inside an operation or inside an item can be used.

3. The `Final` element MUST be used for the message, which is the last in this package. After the server has received the final message of the package, it can complete the sync analysis and send its modifications back to client.

7.2.1.3 Server Modifications to Client

The sync package (Refer Pkg #4) from the DS Server to the DS Client has the following purposes:

- To inform the client about the results of sync analysis.
- To inform about all data modifications, which have happened in the server since the previous sync process.

Any server modifications, which are done after sending this package, MUST be reported to the client during the next sync process. It is not allowed to put them inside subsequent messages of the same sync process from the server to the client.

The requirements for messages within this sync package are following.

1. The `Status` element for the Sync element received from the client is used to indicate the general status of the sync analysis and the status information related to data items sent by the client (e.g., a conflict has happened.). Status information for data items MAY be sent before Pkg #3 is completely received.
2. The `Sync` element MUST be included in `SyncBody`, if earlier there were no occurred errors, which could prevent the server to process the sync analysis and to send its modifications back to the client. For the `Sync` element the following clarifications apply:

   `NoStatus` SHOULD NOT be specified for the `Sync` command.

   `NumberOfChanges` MUST be used to indicate the number of changes in the server node if the client has indicated that it supports *`NumberOfChanges`*.

3. If there are modifications in the server, the operational elements (e.g., `Add`, `Copy`, `Delete`, `Move`, and `Replace`) have the following requirements

   `NoStatus` SHOULD NOT be specified with all these operations.

   The *`Type`* attribute MUST be included in the `Meta` element of items containing data to indicate the type of the data item (E.g., MIME type). The `Meta` element inside an operation or inside an item MAY be used.

4. The `Final` element MUST be used for the message, which is the last in this package.

7.2.1.4 Data Mapping Status from Client

The data mapping status package from the client to the server is needed to transport the information about the result of the data update on the client side. In addition, it is used to indicate the LUID's of the new data items, which have been added in the client, i.e., the `Status` including mapping LUID's and temporary GUID's is sent to the server.

This package SHOULD be sent if the server has added or modified data items.

The messages in this package have the following requirements.

1. The `Status` element for the Sync element received from the server is used to indicate the results of data update in the client and the status information related to the individual data items is transferred to the server. The status information for data items MAY be sent before Package #4 is completely received.
2. The `Status` elements for the operational elements (e.g., `Add`, `Copy`, `Delete`, `Move`, and `Replace`) received from the server have the following requirements

   `StatusItem` MUST be included in the `Status` element for each item of the original command if the client has new client identifiers for any of the data items, or the client has changed client generated fingerprints for any of the data items, or the client has different result codes for any of the data items.

   If the fingerprint generation method is client generated fingerprint, the *`FP`* attribute MUST be included in the `ClientURI` element in `StatusItem` element for the server added or modified data items.

If there are new client identifiers for any data items, the new LUID MUST be included in the `ClientURI` element of the `StatusItem` element.

3. The `Final` element MUST be used for the message, which is the last in this package.

7.2.1.5 Status Package from Server

The status package from the server to the client is needed to inform the client that the server has received the status information of the data items. This status package is sent back to the client even if there were no mapping operations in last package from the client to the server.

The `Final` element MUST be used for the message, which is the last in this package.

7.2.1.6 Examples

The normal sync examples are described as below. Note that the examples are informative and some elements (for example, SyncHdr) are omitted here for simplicity reason.

Pkg #1: The DS Client requests a two-way preserve sync. The client IDs are valid, he client change log is valid, and the client has sync anchors associated with this server.

```
<SyncBody>
  <!-- -->
  <SyncAlert CmdID="3">
     <Anchor Last="001" Next="002" />
     <TargetServerURI>./contacts</TargetServerURI>
     <SourceClientURI>./dev-contacts</SourceClientURI>
     <SyncType Behaviour="Preserve" Direction="twoWay"
               ChangeLogValidity="true" IDValidity="true"/>
  </SyncAlert>
  <Final/>
</SyncBody>
```

Pkg #2: The Last anchor matches, the server side change log is valid and the server side mapping table is also valid. The server agrees to sync as requested.

```
<SyncBody>
  <!-- -->
  <Status CmdID="1" MsgRef="1" Cmd="SyncAlert" CmdRef="3" Code="200" >
    <ServerURI>./contacts</ServerURI>
    <ClientURI>./dev-contacts</ClientURI>
  </Status>
  <SyncAlert CmdID="3" NoStatus="true">
     <Anchor Next="002" />
     <TargetClientURI>./dev-contacts</TargetClientURI>
     <SourceServerURI>./contacts</SourceServerURI>
     <SyncType Behaviour="Preserve" Direction="twoWay"
               ChangeLogValidity="true" IDValidity="true"/>
  </SyncAlert>
  <Final/>
</SyncBody>
```

Pkg #3: The DS Client sends the modifications associated with fingerprints to the server.

```
<SyncBody>
    <!-- -->
    <Sync CmdID="3" NumberOfChanges="1" FreeID="81" FreeMem="8100">
        <TargetServerURI>./contacts</TargetServerURI>
        <SourceClientURI>./dev-contacts</SourceClientURI>
        <Replace CmdID="4" >
           <Meta Type="text/x-vcard" />
```

```
            <Item>
               <SourceClientURI FP="1234">1012</SourceClientURI>
               <Data>
                  <!--The vCard data would be placed here.-->
               </Data>
            </Item>
         </Replace>
    </Sync>
    <Final/>
</SyncBody>
```

Pkg #4: The DS Server returns status for client modifications and sends the server modifications to the DS Client.

```
<SyncBody>
    <!-- -->
    <Status CmdID="2" MsgRef="2" Cmd="Sync" CmdRef="3" Code="200" >
        <ServerURI>./contacts</ServerURI>
        <ClientURI>./dev-contacts</ClientURI>
    </Status>
    <Status CmdID="3" MsgRef="2" Cmd="Replace" CmdRef="4" Code="200" />
    <Sync CmdID="4" NumberOfChanges="2" FreeID="81" FreeMem="8100">
        <TargetClientURI>./dev-contacts</TargetClientURI>
        <SourceServerURI>./contacts</SourceServerURI>
        <Replace CmdID="5" >
           <Meta Type="text/x-vcard" />
           <Item>
              <TargetClientURI>1023</TargetClientURI>
              <Data>
                 <!--The vCard data would be placed here.-->
              </Data>
           </Item>
        </Replace>
        <Add CmdID="6" >
           <Meta Type="text/x-vcard" />
           <Item>
              <SourceServerURI>ABCD12345_1024</SourceServerURI>
              <Data>
                 <!--The vCard data would be placed here.-->
              </Data>
           </Item>
        </Add>
    </Sync>
    <Final/>
</SyncBody>
```

Pkg #5: The DS Client returns data mapping status for the server modifications to the DS Server.

```
<SyncBody>
    <!-- -->
    <Status CmdID="2" MsgRef="2" Cmd="Sync" CmdRef="4" Code="200" >
        <ServerURI>./contacts</ServerURI>
        <ClientURI>./dev-contacts</ClientURI>
    </Status>
    <Status CmdID="3" MsgRef="2" Cmd="Replace" CmdRef="5" Code="200" >
       <StatusItem>
          <ClientURI FP="1234">1023</ClientURI>
       </StatusItem>
    </Status>
    <Status CmdID="4" MsgRef="2" Cmd="Add" CmdRef="6" Code="200" >
       <StatusItem>
          <ServerURI>ABCD12345_1024</ServerURI>
          <ClientURI FP="2345">1024</ClientURI>
       </StatusItem>
    </Status>
    <Final/>
```

```
</SyncBody>
```

Pkg #6: The DS Server returns status to the DS Client for the data mapping status package from client.

```
<SyncBody>
    <!-- -->
    <Status CmdID="1" MsgRef="3" Cmd="SyncHdr" CmdRef="1" Code="200" >
      <!-- -->
      </Status>
    <Final/>
</SyncBody>
```

7.2.2 Client Initiated Recovery Sync

7.2.2.1 Overview

The recovery sync can be desired for many reasons, e.g., the client or the server has lost its change log information, the LUID's have wrapped around in the client, the GUID's have wrapped around in the server, the mapping table has problem in the server, or the sync anchors mismatch.

The recovery sync is a form of synchronization in which fingerprints and data item identifiers of the data items in one or more datastores are sent by the DS Client to the DS Server, and the received fingerprints are compared with the stored fingerprints by the DS Server to determine the data items that the DS Client should send. After fingerprints comparison and analysis, the DS Server returns either a list of identifiers of data items for the client to send to the server, or a SyncAlert without an IDContainer of data item identifiers to indicate the client should send all the data items referred to in the client's SyncAlert.

Either the DS Client or the DS Server may indicate a need for a recovery sync. If the DS Client receives a 508 ('Recovery required') to a previous SyncAlert, or detects another problem in data returned from the DS Server (such as SyncAlert parameters, or sync anchors), the DS Client SHOULD return the appropriate statuses, and initiate a recovery sync.

If the DS Client has a problem, that is, the data item identifiers are invalid, the change log is invalid, or there is no last anchor, the DS Client SHOULD initiate a recovery sync. The DS Client specifies the sync type parameters in the SyncAlert command. The presence of an IDContainer holding data item identifiers and fingerprints indicates that a sync is a recovery sync. This is called client initiated recovery sync.

If the client's state information indicates the devices are synchronizing with each other for the first time, a recovery sync MUST be initiated. In this case, the *ChangeLogValidity* MUST be set as "false". *IDValidity* MAY be set as "true" if the client implementation would not have different data item identifiers if there had been a previous sync with the server, that the client just does not have information about. *IDValidity* MUST be set as "false" if the client implementation could have had different data item identifiers in a previous sync with the server, that the client just does not have information about. For example, if uninstalling and reinstalling the client would lose all previous state information, but the data item identifiers were stored with the address book and thus would be unchanged, *IDValidity* could be set as "true", whereas if reinstalling the client generates new data item identifiers *IDValidity* MUST be set as "false".

7.2.2.2 Flows

The two-way preserve sync flow (*Direction* set as "twoWay", *Behavior* set as "Preserve", *IDValidity* set as "true", *ChangeLogValidity* set as "false") is depicted in the figure below.

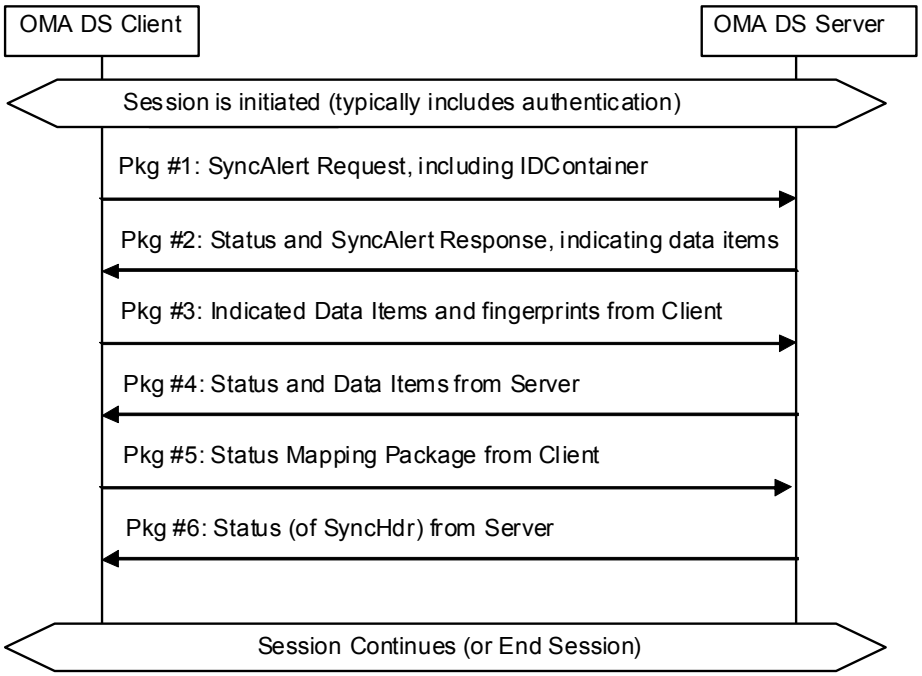

Figure 13: Client Initiated Recovery Sync Flow

Descriptions and examples for the above sync flows are as the follows:

Pkg #1: The DS Client requests a sync process using the `SyncAlert` command, including an IDContainer to indicate the data item identifiers and fingerprints.

Pkg #2: The DS Server compares the received fingerprints with the stored fingerprints for the data items, and then sends back either a list of identifiers of data items for the client to send to the server, or a SyncAlert without an IDContainer of data item identifiers to indicate the client should send all the data items referred to in the client's SyncAlert.

Note that if the client *`IDValidity`* was "false", indicating the client data item identifiers might not be the same as in a previous sync, the client SHOULD initiate a recovery sync, including sending the new data item identifiers and corresponding fingerprints in the SyncAlert command to the server, and the server MAY use fingerprints to match data items instead of data item identifiers. For example, if the server has a list of data items and fingerprints, and receives a new list of data items and fingerprints with *`IDValidity`* set to "false", the server only knows that the identifiers may have changed. If the device information indicates fingerprints are typically unique (FPUnique), an efficient approach to rebuild a mapping table would be to first compare the fingerprint for a given new client identifier with a stored fingerprint for that identifier on the server (if there is one). If a match is found, the mapping is correct for that identifier. If the fingerprints do not match, the server could scan through the entire list of fingerprints for data items that the server has for a match. If there is exactly one match found, then the server may update its mapping table to associate its data item with the new client identifier. If no match or multiple matches are found, the server would need to send that client identifier back to the client in the server's SyncAlert to request the client to send the entire data item for further analysis.

Pkg #3: the DS Client sends data items and associated fingerprints indicated by the server in Pkg #2.

Note that when the client receives a list of identifiers that the server indicates the client to send, the client does not have information as to if those data items are currently on the server or not. Thus the client should use the Replace command, instead of the Add command. Also note that when given a specific list of items to send, there is no need for the client to send Delete commands.

Pkg #4~Pkg #6 are similar to normal sync.

Note that server conflict resolution may encounter the following cases, which have the listed typical behaviors (for two-way sync):

If IDValidity is “true” for both the client and the server, and:

- The identifiers and fingerprints match, then the mapping has been established, and the data item does not need to be sent by the client.
- The client identifier and server identifier match, but the fingerprints do not, then there appears to have been a change in the data, and the server SHOULD request the data item be transferred by the client, unless the server is expected to overwrite or ignore any client changes.
- A client identifier does not match any server identifier, then the client appears to have data the server does not, and the server SHOULD request the data item be transferred by the client.
- A server identifier does not match any client identifier, then the server appears to have data the client does not, and if this still applies after Pkg #3 and been received and further conflict analysis made, the server would typically Add the data item to the client in Pkg #4 (it could choose to send a Delete if it has information that the client used to have the item, and the item should now be deleted).

If IDValidity was “false” for either the client or the server, and FPUnique was specified, and:

- The identifiers and fingerprints match, then a possible mapping has been established. The server may scan through all the fingerprints it has to check for a rare case of multiple matches (and if found, request the data item), or may use the mapping, and thus not request the data item be transferred by the client.
- The client identifier and server identifier match, but the fingerprints do not. The server may scan through the entire list of fingerprints for data items that the server has for a match. If exactly one match is found, then the server has found a match, and does not need to request the data item be transferred by the client.
- A client identifier does not match any server identifier, and the corresponding fingerprint does not match any server fingerprint, then the client appears to have data the server does not, and the server SHOULD request the data item be transferred by the client.
- A server identifier does not match any client identifier, or a server fingerprint does not match any client fingerprint, then the server appears to have data the client does not, and if this still applies after Pkg #3 and been received and further conflict analysis made, the server would typically Add the data item to the client in Pkg #4 (it could choose to send a Delete if it has information that the client used to have the item, and the item should now be deleted).

If IDValidity was “false” for either the client or the server, but FPUnique was not specified, then the server SHOULD always ask for all data items (typically by not returning an IDContainer)., and all conflict analysis would need to be done after the receipt of Pkg #3.

7.2.2.3 Examples

The client initiated recovery sync examples are described as below. Note that the examples are informative and some elements (for example, SyncHdr) are omitted here for simplicity reason.

Pkg #1: The DS Client requests a two-way preserve recovery sync. The client IDs are valid and the client change log is invalid.

```
<SyncBody>
  <!-- -->
  <SyncAlert CmdID="3">
    <Anchor Last="001" Next="002" />
    <TargetServerURI>./contacts</TargetServerURI>
    <SourceClientURI>./dev-contacts</SourceClientURI>
    <SyncType Behaviour="Preserve" Direction="twoWay"
              ChangeLogValidity="false" IDValidity="true"/>
    <IDContainer>
      <ID FP="0001">LUID001</ID>
      <ID FP="0002">LUID002</ID>
      <ID FP="0003">LUID003</ID>
```

```
    </IDContainer>
  </SyncAlert>
</SyncBody>
```

Pkg #2: The Last anchor matches, the server side change log is valid and the server side mapping table is also valid. The server agrees to sync. The server compares the fingerprints and determines the data items the client should send. In this example, the server includes its last known fingerprint for the data item it desires from the client. This is optional. Some clients may be able to take advantage of this information, such as to send a field level replace instead of a full replace.

```
<SyncBody>
  <!-- -->
  <Status CmdID="1" MsgRef="1" Cmd="SyncAlert" CmdRef="3" Code="200" >
    <ServerURI>./contacts</ServerURI>
    <ClientURI>./dev-contacts</ClientURI>
  </Status>
  <SyncAlert CmdID="3" NoStatus="true">
    <Anchor Next="002" />
    <TargetClientURI>./dev-contacts</TargetClientURI>
    <SourceServerURI>./contacts</SourceServerURI>
    <SyncType Behaviour="Preserve" Direction="twoWay"
              ChangeLogValidity="true" IDValidity="true"/>
    <IDContainer>
      <ID FP="0012">LUID002</ID>
    </IDContainer>
  </SyncAlert>
  <Final/>
</SyncBody>
```

Pkg #3: The DS Client sends the modifications associated with fingerprints to the server.

```
<SyncBody>
    <!-- -->
    <Sync CmdID="3" NumberOfChanges="1" FreeID="81" FreeMem="8100">
        <TargetServerURI>./contacts</TargetServerURI>
        <SourceClientURI>./dev-contacts</SourceClientURI>
        <Replace CmdID="4" >
           <Meta Type="text/x-vcard" />
           <Item>
              <SourceClientURI FP="0002">LUID002</SourceClientURI>
              <Data>
                 <!--The vCard data would be placed here.-->
              </Data>
           </Item>
        </Replace>
    </Sync>
    <Final/>
</SyncBody>
```

Pkg #4: The DS Server returns status for client data items and sends the server side data items to the DS Client.

```
<SyncBody>
    <!-- -->
    <Status CmdID="2" MsgRef="2" Cmd="Sync" CmdRef="3" Code="200" >
      <ServerURI>./contacts</ServerURI>
      <ClientURI>./dev-contacts</ClientURI>
    </Status>
    <Status CmdID="3" MsgRef="2" Cmd="Replace" CmdRef="4" Code="200" />
    <Sync CmdID="4" NumberOfChanges="2" FreeID="81" FreeMem="8100">
        <TargetClientURI>./dev-contacts</TargetClientURI>
        <SourceServerURI>./contacts</SourceServerURI>
        <Replace CmdID="5" >
           <Meta Type="text/x-vcard" />
           <Item>
              <SourceServerURI> ABCD12345_1023</SourceServerURI>
              <Data>
```

```
                <!--The vCard data would be placed here.-->
             </Data>
          </Item>
       </Replace>
       <Add CmdID="6" >
          <Meta Type="text/x-vcard" />
          <Item>
             <SourceServerURI>ABCD12345_1024</SourceServerURI>
             <Data>
                <!--The vCard data would be placed here.-->
             </Data>
          </Item>
       </Add>
    </Sync>
    <Final/>
</SyncBody>
```

Pkg #5: The DS Client returns data mapping status for the server modifications to the DS Server.

```
<SyncBody>
    <!-- -->
    <Status CmdID="2" MsgRef="2" Cmd="Sync" CmdRef="4" Code="200" >
      <ServerURI>./contacts</ServerURI>
      <ClientURI>./dev-contacts</ClientURI>
    </Status>
    <Status CmdID="3" MsgRef="2" Cmd="Replace" CmdRef="5" Code="200" >
       <StatusItem>
          <ClientURI FP="3456">LUID1023</ClientURI>
       </StatusItem>
    </Status>
    <Status CmdID="4" MsgRef="2" Cmd="Add" CmdRef="6" Code="200" >
       <StatusItem>
          <ServerURI>ABCD12345_1024</ServerURI>
          <ClientURI FP="4567">LUID1024</ClientURI>
       </StatusItem>
    </Status>
    <Final/>
</SyncBody>
```

Pkg #6: The DS Server returns status to the DS Client for the data mapping status package from client.

```
<SyncBody>
    <!-- -->
    <Status CmdID="1" MsgRef="3" Cmd="SyncHdr" CmdRef="1" Code="200" >
      <!-- -->
    </Status>
    <Final/>
</SyncBody>
```

7.2.3 Server Initiated Recovery Sync

7.2.3.1 Overview

In the case where the DS Client attempts a normal sync, but the DS Server has some issues that require recovery, the DS Server can convert the sync into a Recovery Sync. Issues that might require this include lost change log information, mismatched anchors, lost mapping information, lost data item identifiers, and so on. To force a Recovery Sync, the DS Server returns an error status code to the DS Client's `SyncAlert`, and sends back the desired server side sync type parameters in a `SyncAlert` command. The error code which is used in this case MUST be 508 ('Recovery required'). This is done at the Sync Initialization (Refer Pkg #2). After the DS Client has received the status and the `SyncAlert` command for the recovery sync, the DS Client sends a new `SyncAlert` with an `IDContainer` including data items identifiers and fingerprints to the DS Server, and this package can be considered as a repeated Pkg #1. The DS Server then sends back either a list of identifiers of data items for the client to send to the server, or a `SyncAlert` without an `IDContainer` of data

item identifiers to indicate the client should send all the data items referred to in the client's `SyncAlert`. This package can be considered as a repeated Pkg #2. Sync can be thought to start when the DS Client sends data items in Pkg #3.

Note that if the DS Client had already attempted to start Pkg #3 with the normal sync request, the server could remove any included data items from the list of items for the client to send in subsequent messages.

Note that the DS Client could choose to guess which data items to send, and start to send them immediately after the SyncAlert with fingerprints, in the same message. The server could then return a reduced list of data items for the client to send in subsequent messages.

Note that after the first exchange, this is virtually identical to the Client Initiated Recovery Sync.

7.2.3.2 Flows

The two-way preserve sync flow (*`Direction`* set as "twoWay", *`Behavior`* set as "Preserve", server *`IDValidity`* set as "false", server *`ChangeLogValidity`* set as "true") is depicted in the figure below.

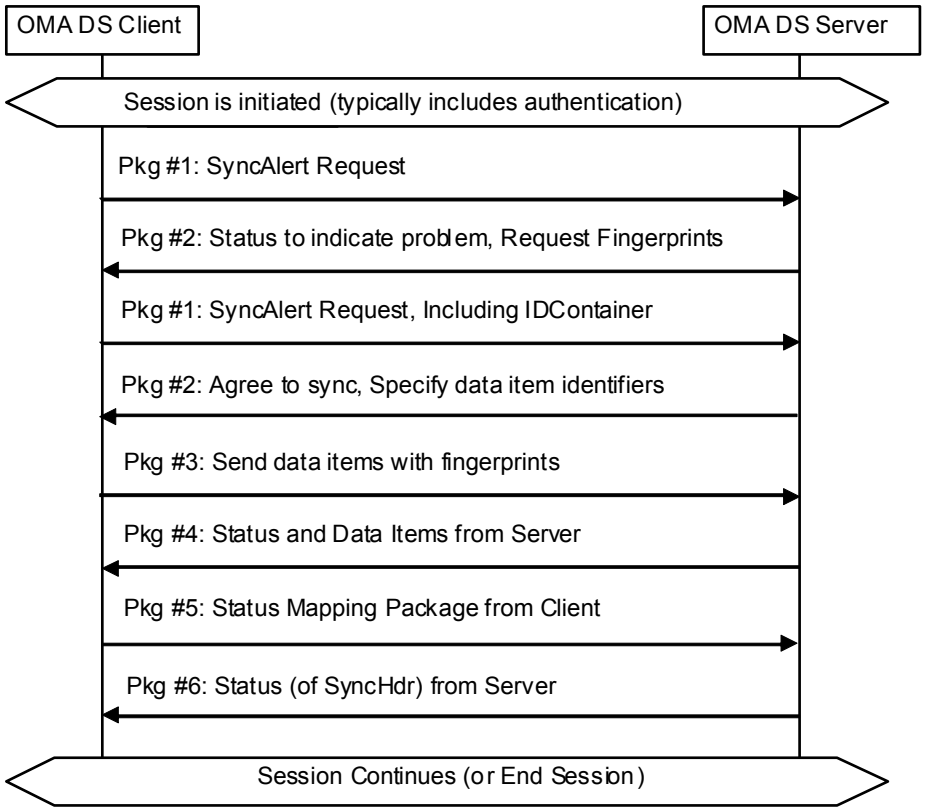

Figure 14: Server Initiated Recovery Sync Flow

Descriptions and examples for the above sync flows are as the follows:

Pkg #1: The DS Client requests a normal sync process using the `SyncAlert` command.

Pkg #2: The DS Server indicates problem by status code 508 "Recovery Required", and sends back the server side sync type parameters by using the `SyncAlert` command.

Pkg #1: The DS Client requests a recovery sync process using the `SyncAlert` command, including `IDContainer` to indicate the data item identifiers and fingerprints.

Pkg #2: The DS Server compares the received fingerprints with the stored fingerprints for the data items, and sends back the status for the `SyncAlert` command and sends back the `IDContainer` in `SyncAlert` command to include data item identifiers to indicate the data items the client should send.

Pkg #3: the DS Client sends data items and associated fingerprints indicated by the server in Pkg #2.

Pkg #4~Pkg #6 are similar as the normal sync.

7.2.3.3 Examples

The client initiated recovery sync examples are described as below. Note that the examples are informative and some elements (for example, `SyncHdr`) are omitted here for simplicity reason.

Pkg #1: The DS Client requests a two-way preserve sync. The client identifiers are valid and the client change log is valid.

```
<SyncBody>
  <!-- -->
  <SyncAlert CmdID="3">
    <Anchor Last="001" Next="002" />
    <TargetServerURI>./contacts</TargetServerURI>
    <SourceClientURI>./dev-contacts</SourceClientURI>
    <SyncType Behaviour="Preserve" Direction="twoWay"
              ChangeLogValidity="true" IDValidity="true"/>
  </SyncAlert>
  <Final/>
</SyncBody>
```

Pkg #2: The Last anchor matches. The server side change log is valid but the server side mapping table is lost. The server indicates the problem, and sends back the `SyncAlert` with the server side sync type parameters.

```
<SyncBody>
  <!-- -->
  <Status CmdID="1" MsgRef="1" Cmd="SyncAlert" CmdRef="3" Code="508" >
   <ServerURI>./contacts</ServerURI>
   <ClientURI>./dev-contacts</ClientURI>
  </Status>
  <SyncAlert CmdID="3">
    <Anchor Next="002" />
    <TargetClientURI>./dev-contacts</TargetClientURI>
    <SourceServerURI>./contacts</SourceServerURI>
    <SyncType Behaviour="Preserve" Direction="twoWay"
              ChangeLogValidity="true" IDValidity="false"/>
  </SyncAlert>
  <Final/>
</SyncBody>
```

Pkg #1: The DS Client requests a two-way preserve recovery sync, and uses `IDContainer` to indicate data item identifiers and fingerprints.

```
<SyncBody>
  <!-- -->
  <Status CmdID="2" MsgRef="2" Cmd="SyncAlert" CmdRef="3" Code="200" >
   <ServerURI>./contacts</ServerURI>
   <ClientURI>./dev-contacts</ClientURI>
  </Status>
  <SyncAlert CmdID="3">
    <Anchor Last="001" Next="002" />
    <TargetServerURI>./contacts</TargetServerURI>
    <SourceClientURI>./dev-contacts</SourceClientURI>
    <SyncType Behaviour="Preserve" Direction="twoWay"
              ChangeLogValidity="true" IDValidity="false"/>
    <IDContainer>
      <ID FP="1234">0001</ID>
      <ID FP="2345">0002</ID>
      <ID FP="3456">0003</ID>
    </IDContainer>
  </SyncAlert>
</SyncBody>
```

Pkg #2: The Last anchor matches, server side change log is valid, and the server side mapping table is invalid. The server agrees to sync. The server compares the fingerprints to recreate its mapping table, and determines the data items the client should send.

```
<SyncBody>
  <!-- -->
  <Status CmdID="1" MsgRef="1" Cmd="SyncAlert" CmdRef="3" Code="200" >
   <ServerURI>./contacts</ServerURI>
   <ClientURI>./dev-contacts</ClientURI>
  </Status>
  <SyncAlert CmdID="3" NoStatus="true">
     <Anchor Next="002" />
     <TargetClientURI>./dev-contacts</TargetClientURI>
     <SourceServerURI>./contacts</SourceServerURI>
     <SyncType Behaviour="Preserve" Direction="twoWay"
               ChangeLogValidity="true" IDValidity="false"/>
     <IDContainer>
       <ID>0002</ID>
     </IDContainer>
  </SyncAlert>
  <Final/>
</SyncBody>
```

Pkg #3: The DS Client sends the modifications associated with fingerprints to the server.

```
<SyncBody>
    <!-- -->
    <Sync CmdID="3" NumberOfChanges="1" FreeID="81" FreeMem="8100">
         <TargetServerURI>./contacts</TargetServerURI>
         <SourceClientURI>./dev-contacts</SourceClientURI>
         <Replace CmdID="4" >
            <Meta Type="text/x-vcard" />
            <Item>
               <SourceClientURI FP="5678">0002</SourceClientURI>
               <Data>
                  <!--The vCard data would be placed here.-->
               </Data>
            </Item>
         </Replace>
    </Sync>
    <Final/>
</SyncBody>
```

Pkg #4: The DS Server returns status for client data items and sends the server side data items to the DS Client.

```
<SyncBody>
    <!-- -->
    <Status CmdID="2" MsgRef="2" Cmd="Sync" CmdRef="3" Code="200" >
      <ServerURI>./contacts</ServerURI>
      <ClientURI>./dev-contacts</ClientURI>
    </Status>
    <Status CmdID="3" MsgRef="2" Cmd="Replace" CmdRef="4" Code="200" />
    <Sync CmdID="4" NumberOfChanges="2" FreeID="81" FreeMem="8100">
         <TargetClientURI>./dev-contacts</TargetClientURI>
        <SourceServerURI>./contacts</SourceServerURI>
         <Replace CmdID="5" >
            <Meta Type="text/x-vcard" />
            <Item>
               <TargetClientURI>0001</TargetClientURI>
               <Data>
                  <!--The vCard data would be placed here.-->
               </Data>
            </Item>
         </Replace>
         <Add CmdID="6" >
```

```
          <Meta Type="text/x-vcard" />
          <Item>
             <SourceServerURI>ABCD12345_1024</SourceServerURI>
             <Data>
                <!--The vCard data would be placed here.-->
             </Data>
          </Item>
        </Add>
    </Sync>
    <Final/>
</SyncBody>
```

Pkg #5: The DS Client returns data mapping status for the server modifications to the DS Server.

```
<SyncBody>
    <!-- -->
    <Status CmdID="2" MsgRef="2" Cmd="Sync" CmdRef="4" Code="200" >
      <ServerURI>./contacts</ServerURI>
      <ClientURI>./dev-contacts</ClientURI>
    </Status>
    <Status CmdID="3" MsgRef="2" Cmd="Replace" CmdRef="5" Code="200" >
       <StatusItem>
          <ClientURI FP="6789">0001</ClientURI>
       </StatusItem>
    </Status>
    <Status CmdID="4" MsgRef="2" Cmd="Add" CmdRef="6" Code="200" >
       <StatusItem>
          <ServerURI>ABCD12345_1024</ServerURI>
          <ClientURI FP="7890">1024</ClientURI>
       </StatusItem>
    </Status>
    <Final/>
</SyncBody>
```

Pkg #6: The DS Server returns status to the DS Client for the data mapping status package from client.

```
<SyncBody>
    <!-- -->
    <Status CmdID="1" MsgRef="3" Cmd="SyncHdr" CmdRef="1" Code="200" >
      <!-- -->
    </Status>
    <Final/>
</SyncBody>
```

7.2.4 Error Case Behaviors

In this chapter, the recommended behaviors are defined in the cases of different error types.

7.2.4.1 No Packages from Server after Initialization

If the client has sent its modifications to the server and it does not get the status associated with those modifications, the client MUST assume that the server has not received those client modifications. At the next time when synchronization is started, the modifications, to which the status was not received, MUST be sent to the server again.

In case of one way sync from client, if the client has sent the empty sync command to the server, it does not get any complete response to it (new modifications), the client SHOULD abort this sync process and try to get the modifications later by starting the sync from the beginning.

7.2.4.2 No Data Update Status from Client

If the server has sent its modifications to the client and it does not get the status associated with those server modifications, the server MUST assume that the client has not received those server modifications. Thus, at the next time when synchronization is started, the server modifications in addition to new ones MUST be sent to the client.

7.2.4.3 No Data Map Acknowledge from Server

If the client has sent the `Status` command containing mapping information to the server and it does not get response to this package, the client SHOULD assume that the server has not received the mapping information. Thus, the server SHOULD compare the sync anchors at the next time and indicate recovery sync in case of anchor mismatch.

7.2.4.4 Errors with Defined Error Codes

If the device receives a defined error code (see [DSSYNTAX]), it MUST act according to that error type.

8. Sync Initialization

The sync initialization implies that the actual synchronization (See Chapters 7), i.e., the `sync` commands, can also be transmitted and processed. Prior to the sync initialization, the DS server MAY send the Notification Package to the DS Client to trigger synchronization with it (See Chapter xx) but this does not remove the need for the initialization. The sync initialization has the following purposes:

- To process the authentication between the DS Client and the DS Server on the SyncML level.
- To indicate which datastores could be synchronized and which sync type parameters could be used.
- To enable the exchange of device information of the DS Client and DS Server.

The two first ones MUST be supported by the DS Client and the DS Server.

The authentication is done by using the `Cred` and `Chal` element of the DS Syntax protocol.

The sync type parameters negotiation is done by using the `SyncAlert` command of the DS Syntax protocol.

The exchange of device information is done by utilizing the `Put` and `Get` commands of the DS Syntax protocol and the Device Information Schema (See also Chapter 5.12).

The initialization procedure is depicted in the figure below. Some parts of the procedure (some responses) can be included in the actual synchronization messages if it is necessary.

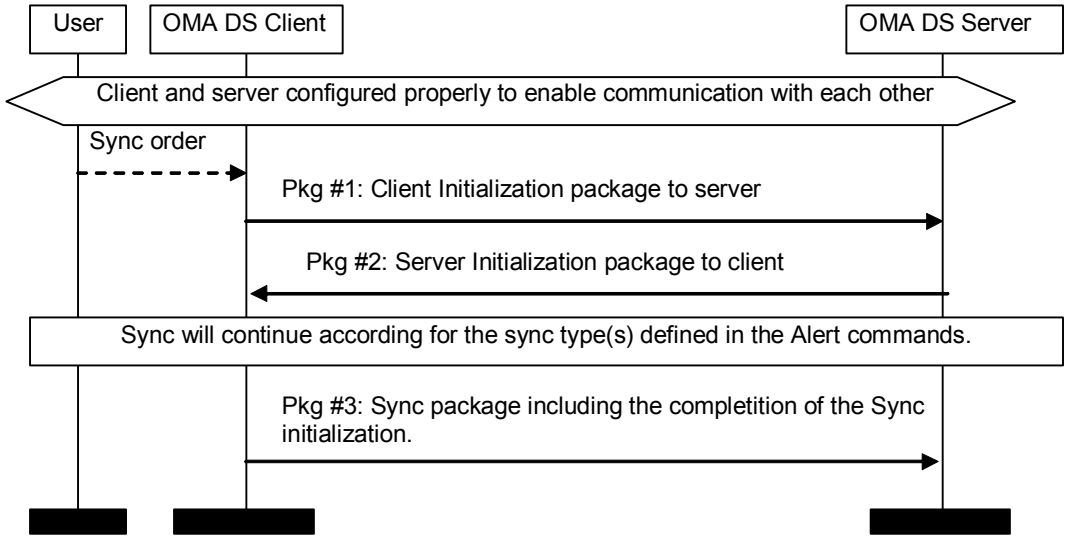

Figure 15 MSC of Synchronization Initialization

The arrows in all figures in this document represent SyncML packages, which can include one or more messages. The package flow presented above is one OMA DS session that means that all messages have the same OMA DS session ID.

The purpose and the requirements for each of the packages in the figure above are considered in the next sections.

8.1 Initialization Requirements for Client

As described in the previous chapter, the DS Client needs to inform the DS Server which datastores or which part of datastores it wants to synchronize and which sync type parameters are desired. The DS Client MAY also include the authentication information and the client device information into this initialization.

The datastores or part of the datastores, which are desired to be synchronized, are indicated in the separate `SyncAlert` commands. I.e., for each datastore or part of datastore, a separate `SyncAlert` command MUST be included in the `SyncBody`. In addition, the `SyncAlert` command is used to exchange the sync anchors.

The sync type parameters are indicated in the `SyncAlert` command. See Section 5.1 for sync type negotiation parameters.

The authentication information, if it is included, MUST be placed inside the `Cred` element in the `SyncHdr`. Either the Basic or the SHA-256 Digest credential type can be used.

The device information, if it is included, MUST be sent by using the `Put` command in the `SyncBody` element. The device information MUST conform to the Device Information Schema. The DS Client can also ask the device information of the DS Server. The `Get` command is used for this operation.

The detailed requirements for the sync initialization package (Pkg #1 in Figure 15) from the DS Client to the DS Server are:

1. Requirements for the elements within the `SyncML` element.
   - The *`Version`* attribute MUST be included to specify the version of the protocol. The value MUST be '2.0' when complying with this specification.
2. Requirements for the elements within the `SyncHdr` element.
   - *`SessionID`* MUST be included to indicate the ID of a sync session.
   - *`MsgID`* MUST be used to unambiguously identify the message belonging to a sync session and traveling from the DS Client to the DS Server.
   - The `TargetServerURI` element MUST be used to identify the DS Server.
   - The `SourceClientURI` element MUST be used to identify the DS Client.
   - The `Cred` element MUST be included if the authentication is needed.
3. The `SyncAlert` element(s) for each datastore or part of datastore to be synchronized MUST be included in `SyncBody`. The requirements for the elements or attributes within the `SyncAlert` element(s) are described as the following:
   - *`CmdID`* MUST be used.
   - *`NoStatus`* SHOULD NOT be specified within the `SyncAlert` command.
   - `SyncType` element MUST be used to specify the sync type parameters requested by the DS Client.
   - `IDContainer` element MAY be used to send the data item identifiers and fingerprints to the DS Server. In case of recovery sync, the DS Client can send the data item identifiers and fingerprints to the DS Server. After comparison, the DS Server can send back the identifiers and fingerprints for the data items that the DS Server requests the DS Client to send.
   - `TargetServerURI` MUST be used to specify the address of the DS Server interior node, and it can point to a datastore or a folder.
   - `SourceClientURI` MUST be used to specify the address of the DS Client interior node, and it can point to a datastore or a folder.
   - The sync anchors of the DS Client MAY be included to specify the previous and current (*`Last`* and *`Next`*) sync anchors (See also Chapter 5.5.1). The sync anchors are carried inside the `Anchor` element.
   - `Fiter` element MAY be used to specify the filtering criteria.

- *Correlator* attribute MAY be used to correlate the SyncAlert request sent by the DS Client and the SyncAlert response sent by the DS Server.

4. If the client device information is sent from the DS Client to the DS Server, the following requirements for the Put command in the SyncBody exist.
   - *CmdID* MUST be used.
   - The *Type* attribute MUST be included in the Meta element of the Put command to indicate that the type of the data is the type of the Device Information Schema.
   - The SourceClientURI element in the Item element MUST be used to specify the URI of the client device information.
   - The Data element in the Item element is used to carry the device information data.
5. If the DS Client requests the device information from the DS Server, the following requirements for the Get command in the SyncBody exist.
   - *CmdID* MUST be used.
   - The *Type* attribute MUST be included in the Meta element of the Get command to indicate that the type of the data is the type of the Device Information Schema.
   - The TargetServerURI element in the Item element is used to specify the URI of the server device information.
6. The Final element MUST be used for the message, if this message is the last one in this package.

8.1.1 Example of Sync Initialization Package from Client

```
<SyncML Version="2.0">
   <SyncHdr SessionID="4" MsgID="1" MaxMsgSize="5000">
      <TargetServerURI>http://www.syncml.org/sync-server</TargetServerURI>
      <SourceClientURI>IMEI:493005100592800</SourceClientURI>
      <Cred> <!--The authentication is optional.-->
         <Meta Type="syncml:auth-basic"/>
         <Data>QnJ1Y2UyOk9oQmVoYXZl</Data> <!--base64 formatting of "userid:password"-->
      </Cred>
   </SyncHdr>
   <SyncBody>
      <SyncAlert CmdID="1" >
        <Anchor Last="234" Next="276"/>
        <TargetServerURI>./contacts/james_bond</TargetServerURI>
         <SourceClientURI>./dev-contacts</SourceClientURI>
         <SyncType Direction="twoWay" Behaviour="Preserve"
                   ChangeLogValidity="true" IDValidity="true"/>
      </SyncAlert>
      <Put CmdID="2">
         <Meta Type="application/vnd.syncml-devinf+xml"/>
         <Item>
            <SourceClientURI>./devinf20</SourceClientURI>
            <Data><![CDATA[
               <DevInf xmlns='syncml:devinf20' Version="2.0">
                  <DevCap UTC="true" SupportLargeObjs="true" SupportNumberOfChanges="true">
                     <Man>Big Factory, Ltd.</Man>
                     <Model>4119</Model>
                     <OEM>Jane's phones</OEM>
                     <FwV>2.0e</FwV>
```

```
                <SwV>2.0</SwV>
                <HwV>1.22I</HwV>
                <DevID>1218182THD000001-2</DevID>
                <DevType>phone</DevType>
              </DevCap>
              <DataStore DisplayName="Phonebook" MaxGUIDSize="32">
                <SourceRef>./contacts</SourceRef>
                <RxTx-CT>
                  <Rx-Pref>
                    <CTType>text/x-vcard</CTType>
                    <VerCT>2.1</VerCT>
                  </Rx-Pref>
                  <Tx-Pref>
                    <CTType>text/x-vcard</CTType>
                    <VerCT>2.1</VerCT>
                  </Tx-Pref>
                 </RxTx-CT>
                <CTCap>
                  <CTType>text/x-vcard</CTType>
                  <VerCT>2.1</VerCT>
                  <Property>
                    <PropName>BEGIN</PropName>
                    <ValEnum>VCARD</ValEnum>
                  </Property>
                  <Property>
                    <PropName>END</PropName>
                    <ValEnum>VCARD</ValEnum>
                  </Property>
                  <Property>
                    <PropName>VERSION</PropName>
                    <ValEnum>2.1</ValEnum>
                  </Property>
                  <Property>
                    <PropName>N</PropName>
                  </Property>
                  <Property>
                    <PropName>TEL</PropName>
                    <PropInfo>
                      <MaxOccur>5</MaxOccur>
                    </PropInfo>
                    <PropParam ParamName="TYPE">
                    <ValEnum>VOICE, CELL</ValEnum>
                    <ValEnum>VOICE, HOME</ValEnum>
                  </PropParam>
                </Property>
                <Property>
                  <PropName>NOTE</PropName>
                  <PropInfo>
                     <MaxOccur>1</MaxOccur>
                     <MaxSize Truncate="false">255</MaxSize>
                  </PropInfo>
                </Property>
                <Property>
                  <PropName>PHOTO</PropName>
                  <PropInfo>
                    <MaxOccur>1</MaxOccur>
                  </PropInfo>
                  <PropParam ParamName="TYPE">
                    <ValEnum>JPEG</ValEnum>
                  </PropParam>
                </Property>
              </CTCap>
              <SyncCap SupportHierarchicalSync="true"/>
            </DataStore>
```

```
            </DevInf>]]>
          </Data>
        </Item>
      </Put>
      <Get CmdID="3" >
        <Meta Type="application/vnd.syncml-devinf+xml"/>
        <Item>
          <TargetServerURI>./devinf20</TargetServerURI>
        </Item>
      </Get>
      <Final/>
   </SyncBody>
</SyncML>
```

8.2 Initialization Requirements for Server

When the DS Server has received the sync initialization package from the DS Client, it completes the initialization phase by responding to the DS Client from the DS Server perspective. To complete the initialization, the DS Server sends its authentication information, sync anchors, and device information back to the client. The DS Server MAY change the client indicated sync type parameters and send the new sync type parameters back to DS Client. The DS Client MAY accept the sync type parameters sent by the DS Server, or the DS Client can initiate another new session.

The detailed requirements for the sync initialization package (Pkg #2 in Figure 4) from the server to the client are:

1. Requirements for the elements within the `SyncHdr` element are the same as the requirements for the SyncHdr element sent by the DS Client, except the message direction.

2. The `Status` MUST be returned for the `SyncAlert` command sent by the DS Client if the DS Client requested the response. This can be sent before Package #1 is completely received (See Chapter 5.14).

   - If the DS Client is not authenticated to use the service, the sync type parameter is not desired (e.g., recovery sync needed), or some other error occurs, the DS Server MUST return an error for that.

3. If the DS Client sent the device information to the DS Server, the DS Server MUST be able to receive it and the `Status` MUST be returned for the `Put` command. This can be sent before Package #1 is completely received.

4. If the DS Client requested the device information of the DS Server, the `Status` element MUST be returned and the `Results` element MAY be returned. This can be sent before Package #1 is completely received. The requirements for the `Results` element are described as the following:

   - The *`Type`* attribute MUST be included in the `Meta` element in the `Results` element to indicate that the type of the data is the type of the Device Information Schema.

   - The `SourceServerURI` element in the `Item` element is used to specify the URI of the device information of the DS Server.

   - The `Data` element in the `Item` element is used to carry the device information of the DS Server.

5. The `SyncAlert` element(s) for each datastore to be synchronized MUST be included in `SyncBody` and the following requirements exist for the `SyncAlert` command.

   - *`CmdID`* MUST be used.

   - *`NoStatus`* SHOULD NOT be specified within the `SyncAlert` command.

   - The `SyncType` element MUST be used to specify the sync type parameters determined by the DS Server. If the sync type parameters are different than the sync type parameters requested by the DS Client, the DS Client SHOULD follow these new sync type parameters when synchronization is continued.

- `IDContainer` element MAY be used to send to the DS Client the identifiers and fingerprints for data items that the DS Server requests the DS Client to send.
- `TargetClientURI` MUST be used to specify the address of the DS Client interior node, and it can point to a datastore or a folder.
- `SourceServerURI` MUST be used to specify the address of the DS Server interior node, and it can point to a datastore or a folder..
- The sync anchors of the DS Server MAY be included to specify the previous and current (*`Last`* and *`Next`*) sync anchors of the DS Server (See also Chapter 5.5.1).
- *`Correlator`* attribute MUST be used, if the `SyncAlert` request includes the *`Correlator`* attribute. The *`Correlator`* attribute within the `SyncAlert` response sent by the DS Server MUST be the same as the *`Correlator`* attribute within the `SyncAlert` request sent by the DS Client.

6. If the server device information was not requested by the DS Client, the DS Server MAY send it to the DS Client by using the `Put` command. The requirements for the `Put` command in the `SyncBody` are the same as the requirements for the `Put` command sent by the DS Client, except the message direction.
7. If the DS Client did not send its device information and the DS Server needs to retrieve them, the DS Server can request those by using the `Get` command. The requirements for the `Get` command in the `SyncBody` are the same as the requirements for the `Get` command sent by the DS Client, except the message direction.
8. The `Final` element MUST be used for the message, if this message is the last one in this package.

To complete the sync initialization from the client side, the DS Client MUST respond to the commands (`SyncAlert`, possible `Put` and `Get`) sent by the DS Server, if *`NoStatus`* attributes are not specified for the commands. The `Status` elements and the `Result` element associated with the commands can be returned in the first package occurring in actual synchronization (that is, Package #3).

8.2.1 Example of Sync Initialization Package from Server

```
<SyncML Version="2.0">
   <SyncHdr SessionID="4" MsgID="1" >
     <TargetClientURI>IMEI:493005100592800</TargetClientURI>
     <SourceServerURI>http://www.syncml.org/sync-server</SourceServerURI>
     <Cred> <!--The authentication is optional.-->
        <Meta Type="syncml:auth-basic"/>
        <Data>QnJ1Y2UyOk9oQmVoYXZl</Data> <!--base64 formatting of "userid:password"-->
     </Cred>
   </SyncHdr>
   <SyncBody>
     <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="212">
         <!--Statuscode for OK, authenticated for session-->
        <ServerURI>http://www.syncml.org/sync-server</ServerURI>
        <ClientURI>IMEI:493005100592800</ClientURI>
     </Status>
     <Status CmdID="2" MsgRef="1" CmdRef="1" Cmd="SyncAlert" Code="200">
        <ServerURI>./contacts/james_bond</ServerURI>
        <ClientURI>./dev-contacts</ClientURI>
     </Status>
     <Status CmdID="3" MsgRef="1" CmdRef="2" Cmd="Put" Code="200" />
     <Status CmdID="4" MsgRef="1" CmdRef="3" Cmd="Get" Code="200" />
     <Results CmdID="5" MsgRef="1" CmdRef="3">
        <Meta Type="application/vnd.syncml-devinf+xml"/>
        <Item>
           <SourceServerURI>./devinf20</SourceServerURI>
           <Data><![CDATA[
              <DevInf xmlns='syncml:devinf20' Version="2.0">
```

```
<DevCap UTC="true" SupportLargeObjs="true" SupportNumberOfChanges="true">
   <Man>Small Factory, Ltd.</Man>
   <Model>Tiny Server</Model>
   <OEM>Tiny Shop</OEM>
   <FwV>1.0.1</FwV>
   <SwV>2.0</SwV>
   <HwV>1.0</HwV>
   <DevID>485749KR</DevID>
   <DevType>Server</DevType>
</DevCap>
<DataStore DisplayName="Addressbook">
   <SourceRef>./contacts</SourceRef>
   <RxTx-CT>
     <Rx-Pref>
      <CTType>text/x-vcard</CTType>
      <VerCT>2.1</VerCT>
     </Rx-Pref>
     <Rx>
      <CTType>text/vcard </CTType>
      <VerCT>3.0</VerCT>
     </Rx>
     <Tx-Pref>
      <CTType>text/x-vcard</CTType>
      <VerCT>2.1</VerCT>
     </Tx-Pref>
     <Tx>
      <CTType>text/vcard</CTType>
      <VerCT>3.0</VerCT>
     </Tx>
   </RxTx-CT>
   <CTCap>
      <CTType>text/x-vcard</CTType>
      <VerCT>2.1</VerCT>
      <Property>
         <PropName>BEGIN</PropName>
         <ValEnum>VCARD</ValEnum>
      </Property>
      <Property>
         <PropName>END</PropName>
         <ValEnum>VCARD</ValEnum>
      </Property>
      <Property>
         <PropName>VERSION</PropName>
         <ValEnum>2.1</ValEnum>
      </Property>
      <Property>
         <PropName>N</PropName>
      </Property>
      <Property>
         <PropName>TEL</PropName>
         <PropInfo>
           <MaxOccur>8</MaxOccur>
         </PropInfo>
         <PropParam ParamName="TYPE">
            <ValEnum>VOICE, CELL</ValEnum>
            <ValEnum>VOICE, HOME</ValEnum>
            <ValEnum>FAX, HOME</ValEnum>
         </PropParam>
      </Property>
      <Property>
         <PropName>NOTE</PropName>
         <PropInfo>
           <MaxOccur>1</MaxOccur>
           <MaxSize Truncate="false">1024</MaxSize>
```

```
                    </PropInfo>
                  </Property>
                  <Property>
                    <PropName>PHOTO</PropName>
                    <PropInfo>
                        <MaxOccur>1</MaxOccur>
                    </PropInfo>
                    <PropParam ParamName="TYPE">
                      <ValEnum>JPEG</ValEnum>
                      <ValEnum>GIF</ValEnum>
                    </PropParam>
                  </Property>
                </CTCap>
                <CTCap>
                  <CTType>text/vcard</CTType>
                  <VerCT>3.0</VerCT>
                  <Property>
                    <PropName>BEGIN</PropName>
                    <ValEnum>VCARD</ValEnum>
                  </Property>
                  <Property>
                    <PropName>END</PropName>
                    <ValEnum>VCARD</ValEnum>
                  </Property>
                  <Property>
                    <PropName>VERSION</PropName>
                    <ValEnum>3.0</ValEnum>
                  </Property>
                  <Property>
                    <PropName>N</PropName>
                  </Property>
                  <Property>
                    <PropName>TEL</PropName>
                    <PropInfo>
                      <MaxOccur>8</MaxOccur>
                    </PropInfo>
                    <PropParam ParamName="TYPE">
                      <ValEnum>VOICE, CELL</ValEnum>
                      <ValEnum>VOICE, HOME</ValEnum>
                      <ValEnum>FAX, HOME</ValEnum>
                    </PropParam>
                  </Property>
                  <Property>
                    <PropName>NOTE</PropName>
                    <PropInfo>
                       <MaxOccur>1</MaxOccur>
                       <MaxSize Truncate="false">1024</MaxSize>
                    </PropInfo>
                  </Property>
                  <Property>
                    <PropName>PHOTO</PropName>
                    <PropInfo>
                       <MaxOccur>1</MaxOccur>
                    </PropInfo>
                    <PropParam ParamName="TYPE">
                      <ValEnum>JPEG</ValEnum>
                      <ValEnum>GIF</ValEnum>
                    </PropParam>
                  </Property>
                </CTCap>
                <SyncCap/>
              </DataStore>
            </DevInf>]]>
          </Data>
```

```
        </Item>
      </Results>
      <SyncAlert CmdID="6">
        <Anchor Next="276"/>
        <TargetClientURI>./dev-contacts</TargetClientURI>
        <SourceServerURI>./contacts/james_bond </SourceServerURI>
        <SyncType Direction="twoWay" Behaviour="Preserve"
                  ChangeLogValidity="true" IDValidity="true"/>
      </SyncAlert>
      <Final/>
   </SyncBody>
</SyncML>
```

8.3 Sync without Separate Initialization

Synchronization can be started without a separate initialization (See the initialization in Chapter 8). This means that the initialization is done simultaneously with sync. This can be done to decrease the number of SyncML messages to be sent over the air.

The DS Client MAY support the feature "Sync without separate initialization". The DS Server MUST support the feature "Sync without separate initialization".

When the DS Client initiates the synchronization session, it combines the Package #1 within the Package #3. The `SyncAlert` command(s) (from the DS Client) in Package #1 is sent within Package #3, in which the `Sync` command(s) are also placed. The DS Server MUST combine Package #2 within Package #4. The `SyncAlert` command(s) (from DS Server) in Package #2 is sent within Package #4, in which the `Sync` command(s) are also placed.

See the example in 8.3.2.

8.3.1 Robustness and Security Considerations

If the DS Client implementation decides to use sync without a separate initialization, the following considerations SHOULD be taken into account:

- The DS Client sends its modifications to the DS Server before the DS Server gets the sync anchors from the client. Because of this, the DS Client MAY need to send all data again if the DS Server has a need to request a recovery sync (Section 9.5).
- Server sync anchor are not received before sending the client modifications. Thus, if the DS Client needs to request a recovery sync, earlier data, which was sent in Package #3 to the server, was unnecessarily sent and all data needs to be sent to DS Server.
- The DS Client sends its modifications to the DS Server before there is any possibility for the DS Server to send its credentials (if requested) to the DS Client. That is, the DS Client cannot be sure whether it is communicating with the correct DS Server.

8.3.2 Example of Sync without Separate Initialization

Here is shown an example, how the DS Client starts sync without a separate sync initialization. Only two packages are shown here (combination of Packages #1 and #3 and the combination of Packages #2 and #4). Package #5 and #6 can follow as defined in the specification.

Combination of Package #1 and #3

```
<SyncML Version="2.0">
   <SyncHdr SessionID="4" MsgID="1" >
      <TargetServerURI>http://www.syncml.org/sync-server</TargetServerURI>
      <SourceClientURI>IMEI:493005100592800</SourceClientURI>
      <Cred> <!--The authentication is optional.-->
         <Meta Type="syncml:auth-basic" Format="b64"/>
         <Data>QnJ1Y2UyOk9oQmVoYXZl</Data> <!--base64 formatting of "userid:password"-->
      </Cred>
```

```
   </SyncHdr>
   <SyncBody>
      <SyncAlert CmdID="1" >
        <Anchor Last="234" Next="276"/>
        <TargetServerURI>./contacts/james_bond</TargetServerURI>
         <SourceClientURI>./dev-contacts</SourceClientURI>
         <SyncType Direction="twoWay" Behaviour="Preserve"
                   ChangeLogValidity="true" IDValidity="true"/>
      </SyncAlert>
      <Sync CmdID="2" FreeMem="8100" FreeID="81" NumberOfChanges="1" >
        <!--Free memory (bytes) in Calendar datastore on a device -->
        <!--Number of free records in Calendar datastore-->
         <TargetServerURI>./contacts/james_bond</TargetServerURI>
         <SourceClientURI>./dev-contacts</SourceClientURI>
         <Replace CmdID="3" >
            <Meta Type="text/x-vcard" />
            <Item>
               <SourceClientURI>1012</SourceClientURI>
               <Data><!--The vCard data would be placed here.--></Data>
            </Item>
         </Replace>
      </Sync>
      <Final/>
   </SyncBody>
</SyncML>
```

Combination of Package #2 and #4

```
<SyncML Version="2.0">
   <SyncHdr SessionID="4" MsgID="1" >
      <TargetClientURI>IMEI:493005100592800</TargetClientURI>
      <SourceServerURI>http://www.syncml.org/sync-server</SourceServerURI>
   </SyncHdr>
   <SyncBody>
      <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="212">
        <!--Status code for OK, authenticated for session-->
         <ServerURI>http://www.syncml.org/sync-server</ServerURI>
         <ClientURI>IMEI:493005100592800</ClientURI>
      </Status>
      <Status CmdID="2" MsgRef="1" CmdRef="1" Cmd="SyncAlert" Code="200">
         <ServerURI>./contacts/james_bond</ServerURI>
         <ClientURI>./dev-contacts</ClientURI>
      </Status>
      <Status CmdID="3" MsgRef="1" CmdRef="2" Cmd="Sync" Code="200">
         <!--Status code for Success-->
         <ServerURI>./contacts/james_bond</ServerURI>
         <ClientURI>./dev-contacts</ClientURI>
      </Status>
      <Status CmdID="4" MsgRef="1" CmdRef="3" Cmd="Replace" Code="200" />
      <SyncAlert CmdID="5" >
         <Anchor Last="234" Next="276"/>
         <TargetClientURI>./dev-contacts</TargetClientURI>
         <SourceServerURI>./contacts/james_bond</SourceServerURI>
         <SyncType Direction="twoWay" Behaviour="Preserve"
                   ChangeLogValidity="true" IDValidity="true"/>
      </SyncAlert>
      <Sync CmdID="6" NumberOfChanges="2" >
         <TargetClientURI>./dev-contacts</TargetClientURI>
         <SourceServerURI>./contacts/james_bond</SourceServerURI>
         <Replace CmdID="7" >
            <Meta Type="text/x-vcard" />
            <Item>
               <TargetClientURI>1023</TargetClientURI>
               <Data><!--The vCard data would be placed here.--></Data>
            </Item>
```

```
        </Replace>
        <Add CmdID="8" >
           <Meta Type="text/x-vcard" />
           <Item>
              <SourceServerURI>10536681</SourceServerURI>
              <Data><!--The vCard data would be placed here.--></Data>
           </Item>
        </Add>
     </Sync>
     <Final/>
  </SyncBody>
</SyncML>
```

8.4 Separate Device Information Negotiation

Device information negotiation can be done separately, without performing data synchronization. In case that the device has either been upgraded, or the user has selected additional data sections to synchronize, the DS Client and the DS Server can negotiate their device information without performing data synchronization.

Figure X shows the device information negotiation flow:

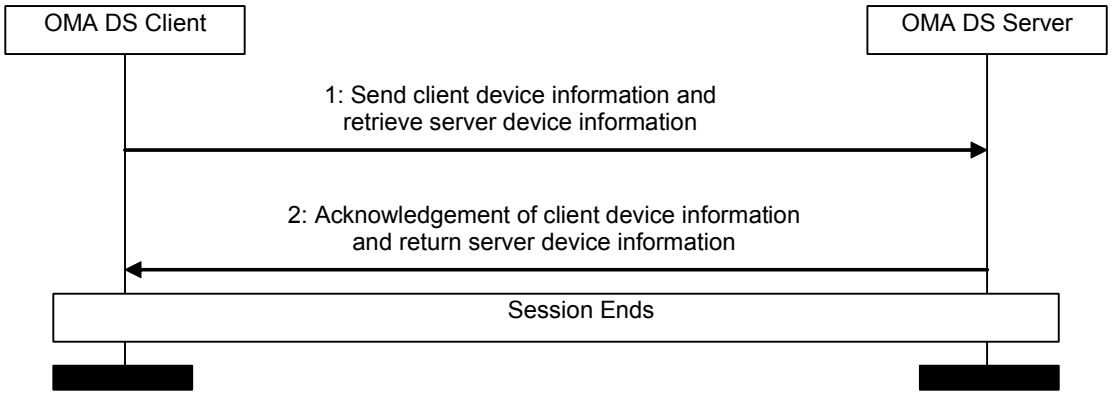

Figure 16: Separate Device Information Negotiation Flow

The detailed flow is as the following:

Step 1: The DS Client sends its device information to the DS Server, and retrieves the server device information.

Step 2: The DS Server sends back acknowledgement of client device information and returns the server device information.

Then the session ends.

In the separate device information negotiation session, the Alert request and response is not necessary because the DS Client and DS Server don't need to perform data synchronization on the data stores. In the device information negotiation session request, the Type element MUST be included in the Meta element of the Put command to indicate that the type of the data is the type of the Device Information Schema.

8.5 Error Case Behaviors

In this chapter, the recommended behaviors are defined in the cases of different error types, which can occur during the sync initialization.

8.5.1 No Packages from Server

If the client has sent its sync initialization package to the server and it does not get any complete response to it, the client MUST assume that the server has not received the sync initialization package of the client. The client MUST send its sync initialization package again later.

8.5.2 No Initialization Completion from Client

If the server has sent its sync initialization package to the client and it does not get any complete response to it (Refer Pkg #3), the server MUST assume that the client has not received the sync initialization package of the server. The server can drop the session and the sync initialization MUST be started from the beginning when synchronization is started at the next time.

8.5.3 Initialization Failure

If the initialization fails, a defined error code in [DSSYNTAX] is sent, the devices MUST act according that error type.

9. Security

9.1 Credentials

Two examples of credentials exchanged between DS Client and DS Server are shown in the following list.

1. Server Identifier (this is a unique ID that identifies the DS Server), a shared secret.
2. User Identifier (this is a unique ID that identifies the DS Client), a shared secret.

When the DS Server authenticates the DS Client, the DS Server SHALL use a different shared secret for each DS Client it serves, in order that a DS Client can not pose effectively as this DS Server in a interaction with another DS Client.

In order for the DS Client to authenticate with a specific DS Server, the credentials for the DS Server MUST be initially provisioned to the DS Client. The credentials MAY be managed with the OMA DM enabler [DM_ERELD]. The DS MO specification [DS_MO] documents the use of OMA DS data such as credentials in the OMA DM management tree.

9.2 Authentication

In this chapter, the authentication procedures are defined for the any of the hash-function based authentication schemes described in [DSSYNTAX]

9.2.1 Authentication Challenge

If the response code to a request is '(407) Credential required', it means that a credential is required for authentication purpose. In this case, the `Status` command to the request MUST include a `Chal` element which contains a challenge applicable to the requested resource. The DS Client MAY repeat the request with a suitable credential contained in a `Cred` element.

If the status code to a request is '(401) Unauthorized', it means that authorization has been refused for those credentials. The DS Client need not repeat the request with the same credentials.

Both the DS Client and the DS Server can challenge for authentication.

If the response code to a request is '(212) Authentication accepted', no further authentication is needed for the remainder of the synchronization session.

If a request includes credentials and the response code to the request is '(200) Command completed successfully', the same credentials MAY be sent within the next request.

In the case of the hash-function based authentication scenario, whether the response code is 212 or 200, the `Chal` element can be returned. Then, the next nonce in `Chal` MUST be used for computing the digest when the next sync request is sent.

Once authentication has occurred, the authentication scheme for a security layer SHOULD be kept same for the whole session.

In case of authentication failure:

- The response message indicating the authentication failure on the protocol layer SHOULD contain only `Status` commands. A `Status` command MUST be provided for every command received in the request.
- If the session is continued, the next message:
    - SHOULD contain the proper credentials.
    - MUST contain a `Status` for the `SyncHdr` of the message indicating the authentication failure.
    - MUST have the same session ID as the previous messages.

- MUST be sent to the `RespURI`, if a `RespURI` was specified.

9.2.2 Authorization

The `Cred` element MUST be included in requests (message or command), which are sent after receiving the 401 or 407 responses if the request is repeated. In addition, it can be sent in the first request from a device if the authentication is mandated through pre-configuration. The content of the `Cred` element is specified in [DSSYNTAX]. The authentication type is dependent on the challenge (See the previous chapter) or the pre-configuration.

9.2.3 Protocol Layer Authentication

When the authentication is considered, the DS Client MUST support the protocol layer authentication by specifing the credentials in the `SyncHdr` element. The challenge for the authentication is carried within the `Status` command. The authentication can happen in both directions, i.e., the DS Client can authenticate itself to the DS Server and vice versa.

9.2.4 Datastore Layer Authentication

The DS Client MAY support datastore layer authentication. The datastore layer authentication is accomplished by using the `Cred` element in the `SyncAlert` and `Sync` commands. Within the `Status` element, the challenge for the authentication MAY be specified. The authentication can happen in both directions, i.e., the DS Client can authenticate itself to the DS Server and vice versa.

9.2.5 Authentication Examples

9.2.5.1 SHA-256 authentication with a challenge

At this example, the DS Client tries to initiate sync with the DS Server without any credentials (Pkg #1). The DS Server challenges the DS Client (Pkg #2) for the protocol layer authentication using the SHA-256 authentication scheme. The DS Client sends Pkg #1 again with the credentials. The DS Server accepts the credentials and the session is authenticated (Pkg #2). Then the DS Server sends the next nonce to the DS Client, which the DS Client will use when the next sync request is started. In the example, commands in `SyncBody` are not shown although in practice, they would be there.

Pkg #1 from Client

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
        Version="2.0">
   <SyncHdr SessionID="1" MsgID="1">
       <TargetServerURI>http://www.openmobilealliance.org/sync-server</TargetServerURI>
       <SourceClientURI>IMEI:493005100592800</SourceClientURI>
   </SyncHdr>
   <SyncBody>
        ......
   </SyncBody>
</SyncML>
```

Pkg #2 from Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
   <SyncHdr SessionID="1" MsgID="2">
       <TargetClientURI>IMEI:493005100592800 </TargetClientURI>
       <SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
   </SyncHdr>
   <SyncBody>
       <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="407" >
          <!-- Credential required-->
```

```
            <Chal>
                <Meta Type="syncml:auth-sha256" Format="b64" />
                <NextNonce>Tm9uY2U=</NextNonce>
            </Chal>
        </Status>
         ......
    </SyncBody>
</SyncML>
```

Pkg #1 (with credentials) from Client

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
   <SyncHdr SessionID="1" MsgID="2">
       <TargetServerURI>http://www.openmobilealliance.org/sync-server</TargetServerURI>
       <SourceClientURI>IMEI:493005100592800</SourceClientURI>
       <Cred AuthName="Bruce2">
          <Meta Type="syncml:auth-sha256" Format="b64"/>
          <Data>BlahBlahBlahBlah==</Data>
       </Cred>
          <!-- Base64 coded SHA-256 digest for user "Bruce2", shared secret
             "OhBehave", nonce "Nonce" -->
    </SyncHdr>
    <SyncBody>
         ......
    </SyncBody>
</SyncML>
```

Pkg #2 from Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
  <SyncHdr SessionID="1" MsgID="2">
      <TargetClientURI>IMEI:493005100592800 </TargetClientURI>
      <SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
  </SyncHdr>
  <SyncBody>
      <Status CmdID="1" MsgRef="2" CmdRef="0" Cmd="SyncHdr" Code="212" >
        <!--Authenticated for session-->
          <Chal>
             <Meta Type="syncml:auth-sha256" Format="b64" />
              <NextNonce>BlahBlahBlah=</NextNonce>
       <!--This nonce is used when the next session is started-->
          </Chal>
      </Status>
        ......
  </SyncBody>
</SyncML>
```

9.3 Integrity

The OMA DS provides the messages integrity protection by specifying the hashed message authentication code (HMAC) in the underlying transport header. The message authentication code is computed according to the procedure as described in section 7.3.1.

Both, the DS Client and the DS Server can challenge the other side for integrity protection. The sender SHALL specify the integrity protection challenge in the underlying transport header.

Once integrity protection has occurred, the hashed message authentication code SHALL be used on every message transferred between the DS Client and DS Server.

9.3.1 How the HMAC is computed

The HMAC value SHALL be computed according to the HMAC mechanism specified in [RFC2104]. The HMAC value SHALL be computed as following:

Let H = the Hashing function.
Let Digest = the output of the Hashing function.

Let B64 = the base64 encoding function.

Let userid = User Identifier.

Let secret = Secrete known by the sender and recipient.

Let nonce = Challenge specified by the authenticator

Digest = H(B64(H(userid:secret)):nonce:B64(H(message body)))

The DS Client and DS Server SHALL support SHA-256 hashing function. And other hashing functions are not excluded, e.g. SHA-1.

9.3.2 How the HMAC is specified in the OMA DS message

The HMAC value SHALL be transported along with the original OMA DS message. This is achieved by inserting the HMAC value into a transport header called x-syncml-hmac. This mechanism works identically on HTTP, WAP, OBEX and other transport protocols. The HMAC is computed initially by the sender against the entire message body independent with the message format. Upon receiving a message, the recipient SHALL use the same procedure to compute its own HMAC value and verify whether they are identical in order to ensure the authenticity of the sender, and also the integrity of the message. If the 'userid' is incorrect or the HMAC values are not identical, then an authentication failure results SHALL be returned to the sender. Once the integrity protection mechanism is used, the `NextNonce` element SHALL be sent and used for the next HMAC credential check.

The header x-syncml-hmac contains multiple parameters, including the HMAC value, the user or server identifier, and an optional indication of which HMAC algorithm is in use.

The value of the x-syncml-hmac header is defined as a comma separated list of attribute-values pairs. The rule '#rule' and the terms 'token' and 'quoted-string' are used in accordance to the definition in the HTTP 1.1 specified in [RFC2616].

Here is the formal definition:

syncml-hmac = #syncml-hmac-param

syncml-hmac-param = (algorithm | userid | mac)

algorithm = "algorithm" "=" ("SHA-256" | token)

userid = "userid" "=" userid-value

mac = "mac" "=" mac-value

userid-value = quoted-string

mac-value = base64-string

Note that a base64-string is any concatenation of the characters belonging to the base64 Alphabet, as defined in [RFC2045].

The following is an example:

```
x-syncml-hmac: algorithm=SHA-256, userid="Robert Jordan",
mac=NTI2OTJhMDAwNjYxODkwYmQ3NWUxN2RhN2ZmYmJlMzk
```

9.4 Confidentiality

Confidentiality in OMA DS has two major aspects: the protocol layer encryption and transport layer encryption which can be used either separately or together.

9.4.1 Protocol Layer Encryption

OMA DS supports the protocol layer encryption by using the Symmetric Cryptography in which the same key is used for both encryption and decryption. The DS Client generates the symmetric key and transports it to the DS Server under the protection of the Public-Key Cryptography.

When the encryption is considered, the DS Client SHALL support the protocol layer encryption by specifies the `EncryptedKey` element in the `SyncHdr` element.

9.4.1.1 Symmetric Key Transport

During the initialization phase, the DS Client generates the symmetric key and encrypts the key using the DS Server's public key. The DS Client should obtain this public key beforehand. After receiving the initialization request, the DS Server determining whether this request comprises an encryption indicator, if yes, the DS Server obtains the encrypted key from the `EncryptedKey` element. Then the DS Server decrypts it using its private key to get the symmetric key and meanwhile initialization operations being performed.

9.4.1.2 Encryption Challenge

If there is no encryption indicator in the initialization request, and the DS Server requires encryption, the status code '(432) Encryption Required' MUST be returned to the DS Client.

The DS Server and Client MUST support the RSAES-PKCS1-v1_5 algorithm specified in [RFC2437] as the symmetric key transport algorithm. However, other algorithms outside of this specification are not excluded. If other algorithms are specified and the DS Server doesn't support them, the status code '(429) Key Exchange Algorithm not supported' MUST be returned to the DS Client.

The DS Server and Client MUST support the AES-128-CBC algorithm as the symmetric key algorithm. If other algorithms are specified and the DS Server doesn't support them, the status code '(430) Data Encryption Algorithm not supported' MUST be returned to the DS Client. If the DS Server doesn't support the key length, the status code '(431) Key length not supported' MUST be returned to the DS Client.

If encryption indicator is not specified, or algorithm is not supported, or key length is not supported, the `Status` command to the request MUST include a `Chal` element contains a challenge applicable to the requested source. The DS Client SHOULD repeat the request with a suitable encryption indicator contained in an `EncryptedKey` element.

9.4.1.3 Data Encryption

During the synchronization phase, the DS Client transmits synchronization data encrypted by the symmetric key to the DS Server. The DS Server decrypts the data by the acquired symmetric key before performing synchronization operations, after that, it encrypts its own data with the symmetric key and transmits the encrypted data to the DS Client. Also the DS Client decrypts the data by the symmetric key before performing synchronization operations.

During the synchronization phase, both the DS Client and Server MUST use the same symmetric key for encryption and decryption. In this procedure the DS Client and DS Server can use the *`Encrypted`* attribute for each sync commands (e.g. `Add`, `Replace`) to indicate whether the specific content is encrypted. B64 encoding attribute SHOULD apply prior to encryption.

9.4.1.4 Certificate

The DS Server distributes its public key contained in a certificate used by client to encrypt the symmetric key. Only the server possesses the private key that can decrypt the cryptograph and obtain the original symmetric key.

The provisioning of certificates and the certificates management is specified in [DS_MO]. In practice, the DS Client can obtain the DS Server's certificate from the Certificate Authority (CA) or the trusted DS Server.

9.4.1.5 Examples

9.4.1.5.1 Encryption with a challenge

At this example, the DS Client tries to initiate sync with the DS Server without any encryption indicator (Pkg #1). The DS Server challenges the DS Client (Pkg #2) for the protocol layer encryption. The DS Client sends Pkg #1 again with the encryption indicator. The DS Server accepts the encryption request and the symmetric key is obtained by the DS Server (Pkg #2). In the example, commands in `SyncBody` are not shown although in practice, they would be there.

Pkg #1 from Client

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
   <SyncHdr SessionID="1" MsgID="1">
       <TargetServerURI>http://www.openmobilealliance.org/sync-server</TargetServerURI>
       <SourceClientURI>IMEI:493005100592800</SourceClientURI>
   </SyncHdr>
   <SyncBody>
      ......
   </SyncBody>
</SyncML>
```

Pkg #2 from Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
  <SyncHdr SessionID="1" MsgID="2">
      <TargetClientURI>IMEI:493005100592800 </TargetClientURI>
      <SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
  </SyncHdr>
  <SyncBody>
     <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="432" >
           <!-- Encryption required-->
        <Chal>
           <Meta Type="AES-128-CBC" Format="b64" Size="128" />
        </Chal>
     </Status>
     ......
   </SyncBody>
</SyncML>
```

Pkg #1 (with encrypted key) resent from Client

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
    <SyncHdr SessionID="1" MsgID="1">
        <TargetServerURI>http://www.openmobilealliance.org/sync-server</TargetServerURI>
        <SourceClientURI>IMEI:493005100592800</SourceClientURI>
        <EncryptedKey>
```

```
            <Meta Type="AES-128-CBC" Format="b64" Size="128"/>
            <Data>Zz6EivR3yeaaENcRN6lpAQ==</Data>
          </EncryptedKey>
      </SyncHdr>
      <SyncBody>
         ......
      </SyncBody>
</SyncML>
```

Pkg #2 from Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
    <SyncHdr SessionID="1" MsgID="2">
         <TargetClientURI>IMEI:493005100592800 </TargetClientURI>
         <SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
    </SyncHdr>
    <SyncBody>
         <Status CmdID="1" MsgRef="2" CmdRef="0" Cmd="SyncHdr" Code="200" />
   <!--Encryption request accepted-->
            ......
    </SyncBody>
</SyncML>
```

9.4.1.5.2 Symmetric key algorithm not supported challenge

At this example, the client initiates the synchronization by sending encryption indicator with the Pkg#1, and server determines that it doesn't support the symmetric key algorithm and includes the encryption challenge into Pkg#2.

Pkg #1 (with encrypted key) from Client

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
    <SyncHdr SessionID="1" MsgID="1">
        <TargetServerURI>http://www.openmobilealliance.org/sync-server</TargetServerURI>
        <SourceClientURI>IMEI:493005100592800</SourceClientURI>
        <EncryptedKey>
            <Meta Type="3DES" Format="b64" Size="128" />
             <Data> … </Data>
         </EncryptedKey>
    </SyncHdr>
    <SyncBody>
      ......
    </SyncBody>
</SyncML>
```

Pkg #2 (with challenge) from Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
    <SyncHdr SessionID="1" MsgID="2">
        <TargetClientURI>IMEI:493005100592800 </TargetClientURI>
        <SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
    </SyncHdr>
    <SyncBody>
     <Status CmdID="1" MsgRef="1" CmdRef="0" Cmd="SyncHdr" Code="429" >
      <!--Data Encryption Algorithm not supported-->
```

```
        <Chal>
           <Meta Type="AES-128-CBC" Format="b64" Size="128" />
        </Chal>
      </Status>
      ......
    </SyncBody>
</SyncML>
```

9.4.1.5.3 Encrypted data transmitting

At this example, the client transmits the encrypted data with the Pkg#3, and server includes its encrypted data into Pkg#4.

Pkg #3 (with encrypted data) from Client

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
    <SyncHdr SessionID="1" MsgID="2">
        <TargetServerURI>http://www.openmobilealliance.org/sync-server</TargetServerURI>
        <SourceClientURI>IMEI:493005100592800</SourceClientURI>
    </SyncHdr>
    <SyncBody>
       ......
       <Sync CmdID="3" NumberOfChanges="10">
           <TargetServerURI>./contacts</TargetServerURI>
           <SourceClientURI>./dev-contacts</SourceClientURI>
           <Add CmdID="4">
               <Item>
                 <SourceClientURI>1012</SourceClientURI>
                 <Meta Type="text/x-vcard" Format="b64" />
                    <Data Encrypted="true">......</Data><!-- This data is encrypted -->
               </Item>
           </Add>
           <Replace CmdID="5">
               <Item>
                    <SourceClientURI>1012</SourceClientURI>
                       <Meta Type="text/x-vcard" Format="b64" />
                    <Data Encrypted="true">......</Data>  <!-- This data is encrypted -->
               </Item>
           </Replace>
         ......
      </Sync>
      <Final/>
  </SyncBody>
</SyncML>
```

Pkg #4 (with encrypted data) from Server

```
<?xml version="1.0" encoding="UTF-8"?>
<SyncML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="OMA-SUP-XSD_DS_Syntax_Schema-V2_0.xsd"
       Version="2.0">
   <SyncHdr SessionID="1" MsgID="2">
       <TargetClientURI>IMEI:493005100592800 </TargetClientURI>
       <SourceServerURI>http://www.openmobilealliance.org/sync-server</SourceServerURI>
   </SyncHdr>
   <SyncBody>
            ......
           <Sync CmdID="4" NumberOfChanges="5">
               <TargetClientURI>./dev-contacts</TargetClientURI>
               <SourceServerURI>./contacts</SourceServerURI>
               <Add CmdID="5">
                   <Item>
```

```
                        <SourceServerURI>10536681</SourceServerURI>
                        <Meta Type="text/x-vcard" Format="b64" />
                        <Data Encrypted="true">......</Data>  <!-- This data is encrypted -->
                    </Item>
                </Add>
                <Replace CmdID="6">
                    <Item>
                        <TargetClientURI>1023</TargetClientURI>
                        <Meta Type="text/x-vcard" Format="b64" />
                        <Data Encrypted="true">......</Data>  <!-- This data is encrypted -->
                    </Item>
                </Replace>
                ......
            </Sync>
            <Final/>
    </SyncBody>
</SyncML>
```

9.4.2 Transport Layer Encryption

The use of a transport layer encryption is also allowed.

Transport specific security is documented in the transport binding documents [DSHTTPBINDING], [DSOBEXBINDING] and [DSWSPBINDING].

10.Examples

10.1 WBXML Example

Here is an example of a combined Package #1 and #3 from section 8.3.2Example of Sync without Separate Initialization, in tokenized form (numbers in hexadecimal). This example uses inline strings. The example also assumes that the character encoding is UTF-8.

```
# Ethernet II
0000  00 40 05 32 06 16 00 16 b6 89 63 a4 08 00         .@.2......c...

# IP
000e  45 80                                           E.
0010  03 f8 d1 8a 40 00 6c 06 b5 12 4b df 76 29 c0 a8   ....@.l...K.v)..
0020  01 32                                           .2

# TCP
0022  0e bd 00 50 63 3e ef 26 27 b1 4d cf 50 18         ...Pc>.&'.M.P.
0030  80 c4 00 e2 00 00                               ......

# HTTP
0036  50 4f 53 54 20 2f 53 79 6e 63                   POST /Sync
0040  4d 4c 20 48 54 54 50 2f 31 2e 31 0d 0a 41 63 63   ML HTTP/1.1..Acc
0050  65 70 74 2d 4c 61 6e 67 75 61 67 65 3a 20 65 6e   ept-Language: en
0060  2d 75 73 0d 0a 43 6f 6e 74 65 6e 74 2d 54 79 70   -us..Content-Typ
0070  65 3a 20 61 70 70 6c 69 63 61 74 69 6f 6e 2f 76   e: application/v
0080  6e 64 2e 73 79 6e 63 6d 6c 2b 77 62 78 6d 6c 0d   nd.syncml+wbxml.
0090  0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 44 65 76   .User-Agent: Dev
00a0  49 6e 66 2d 45 78 61 6d 70 6c 65 0d 0a 48 6f 73   Inf-Example..Hos
00b0  74 3a 20 64 63 68 61 6d 70 61 67 6e 65 2e 63 6f   t: dchampagne.co
00c0  6d 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67 74   m..Content-Lengt
00d0  68 3a 20 36 30 39 0d 0a 43 6f 6e 6e 65 63 74 69   h: 609..Connecti
00e0  6f 6e 3a 20 4b 65 65 70 2d 41 6c 69 76 65 0d 0a   on: Keep-Alive..
00f0  43 61 63 68 65 2d 43 6f 6e 74 72 6f 6c 3a 20 6e   Cache-Control: n
0100  6f 2d 63 61 63 68 65 0d 0a 0d 0a                 o-cache....

# SyncML
010b  02 00 00 6a 21                                  ...j!
0110  2d 2f 2f 53 59 4e 43 4d 4c 2f 2f 53 63 68 65 6d   -//SYNCML//Schem
0120  61 20 53 79 6e 63 4d 4c 20 32 2e 30 2f 2f 45 4e   a SyncML 2.0//EN
0130  00 ed 60 01 ec 4a 03 34 00 3d 03 31 00 01 64 03   ..`..J.4.=.1..d.
0140  68 74 74 70 3a 2f 2f 77 77 77 2e 73 79 6e 63 6d   http://www.syncm
0150  6c 2e 6f 72 67 2f 73 79 6e 63 2d 73 65 72 76 65   l.org/sync-serve
0160  72 00 01 56 03 49 4d 45 49 3a 34 39 33 30 30 35   r..V.IMEI:493005
0170  31 30 30 35 39 32 38 30 30 00 01 4e 9a 56 03 61   100592800..N.V.a
0180  75 74 68 2d 62 61 73 69 63 00 2a 01 4f 03 51 6e   uth-basic.*.O.Qn
0190  4a 31 59 32 55 79 4f 6b 39 6f 51 6d 56 6f 59 58   J1Y2UyOk9oQmVoYX
01a0  5a 6c 00 01 01 01 6b dc 19 03 31 00 01 87 3a 03   Zl....k...1...:.
01b0  32 33 34 00 3f 03 32 37 36 00 01 64 03 2e 2f 63   234.?.276..d../c
01c0  6f 6e 74 61 63 74 73 2f 6a 61 6d 65 73 5f 62 6f   ontacts/james_bo
01d0  6e 64 00 01 56 03 2e 2f 64 65 76 2d 63 6f 6e 74   nd..V../dev-cont
01e0  61 63 74 73 00 01 9d 22 08 0b 39 01 01 ea 19 03   acts..."..9.....
01f0  32 00 37 03 38 31 30 30 00 36 03 38 31 00 47 03   2.7.8100.6.81.G.
0200  31 00 01 64 03 2e 2f 63 6f 6e 74 61 63 74 73 2f   1..d../contacts/
0210  6a 61 6d 65 73 5f 62 6f 6e 64 00 01 56 03 2e 2f   james_bond..V../
0220  64 65 76 2d 63 6f 6e 74 61 63 74 73 00 01 e0 19   dev-contacts....
0230  03 33 00 01 9a 5e 01 54 56 03 31 30 31 32 00 01   .3...^.TV.1012..
0240  4f 03 54 68 65 20 76 43 61 72 64 20 64 61 74 61   O.The vCard data
0250  20 77 6f 75 6c 64 20 62 65 20 70 6c 61 63 65 64    would be placed
0260  20 68 65 72 65 2e 00 01 01 01 01 12 01 01          here.........
```

In an expanded and annotated form:

Token Stream	Description
02	Version number – WBXML v1.2
00	FPI for DTD in string table
00	index into string table for the identifier
6A	Charset is UTF-8
21	String table length
2d 2f 2f 53 59 4e 43 4d 4c 2f 2f 53 63 68 65 6d 61 20 53 79 6e 63 4d 4c 20 32 2e 30 2f 2f 45 4e 00	"-//SYNCML//Schema SyncML 2.0//EN"
ED	<SyncML (With Content and Attributes)
6C	Version="2.0"
01	> (End of Attributes)
EC	<SyncHdr
4A	SessionID=
03	Inline String Follows
34 00	"4"
3D	MsgID=
03	Inline String Follows
31 00	"1"
01	> (End of Attributes)
64	<TargetServerURI>
03	Inline String Follows
68 74 74 70 3a 2f 2f 77 77 77 2e 73 79 6e 63 6d 6c 2e 6f 72 67 2f 73 79 6e 63 2d 73 65 72 76 65 72 00	"http://www.syncml.org/sync-server"
01	</TargetServerURI>
56	<SourceClientURI>
03	Inline String Follows
49 4d 45 49 3a 34 39 33 30 30 35 31 30 30 35 39 32 38 30 30 00	"IMEI:493005100592800"
01	</SourceClientURI>
4E	<Cred>
9A	<Meta
56	Type="SyncML:"
03	Inline String Follows
61 75 74 68 2d 62 61 73 69 63 00	"auth-basic"
2A	Format="b64"
01	/>
4F	<Data>
03	Inline String Follows
51 6e 4a 31 59 32 55 79 4f 6b 39 6f 51 6d 56 6f 59 58 5a 6c 00	"QnJ1Y2UyOk9oQmVoYXZl"
01	</Data>
01	</Cred>
01	</SyncHdr>
6B	<SyncBody>
DC	<SyncAlert
19	CmdID=
03	Inline String Follows
31 00	"1"
01	>
87	<Anchor
3A	Last=
03	Inline String Follows
32 33 34 00	"234"
3F	Next=
03	Inline String Follows
32 37 36 00	"276"
01	>

64	<TargetServerURI>
03	Inline String Follows
2e 2f 63 6f 6e 74 61 63 74 73 2f 6a 61 6d 65 73 5f 62 6f 6e 64	"./contacts/james_bond"
01	</TargetServerURI>
56	<SourceClientURI>
03	Inline String Follows
2e 2f 64 65 76 2d 63 6f 6e 74 61 63 74 73 00	"./dev-contacts"
01	</SourceClientURI>
9d	<SyncType
22	Direction="twoWay"
08	Behaviour="false"
0B	ChangeLogValidity="true"
39	IDValidity="true"
01	/>
01	</SyncAlert>
EA	<Sync
19	CmdID=
03	Inline String Follows
32 00	"2"
37	FreeMem=
03	Inline String Follows
38 31 30 30 00	"8100"
36	FreeID=
03	Inline String Follows
38 31 00	"81"
47	NumberOfChanges=
03	Inline String Follows
31 00	"1"
01	>
64	<TargetServerURI>
03	Inline String Follows
2e 2f 63 6f 6e 74 61 63 74 73 2f 6a 61 6d 65 73 5f 62 6f 6e 64	"./contacts/james_bond"
01	</TargetServerURI>
56	<SourceClientURI>
03	Inline String Follows
2e 2f 64 65 76 2d 63 6f 6e 74 61 63 74 73 00	"./dev-contacts"
01	</SourceClientURI>
E0	<Replace
19	CmdID=
03	Inline String Follows
31 00	"3"
01	>
9A	<Meta
5E	Type="text/x-vcard"
01	/>
54	<Item>
54 68 65 20 76 43 61 72 64 20 64 61 74 61 20 77 6f 75 6c 64 20 62 65 20 70 6c 61 63 65 64 20 68 65 72 65 2e 00	"The vCard data would be placed here."
01	</Data>
01	</Item>
01	</Replace>
01	</Sync>
12	<Final/>
01	</SyncBody>
01	</SyncML>

Appendix A. Static Conformance Requirements (Normative)

A.1 Conformance Requirements for OMA DS Client

Table 1 – Client Features

Item	Functionality	Reference	Status	Requirement
SCR-DS-CLIENT-001	Support of 'twoWay' Direction	5.1	M	
SCR-DS-CLIENT-002	Support of 'fromClient' Direction	5.1	M	
SCR-DS-CLIENT-003	Support of 'fromServer' Direction	5.1	M	
SCR-DS-CLIENT-004	Support of 'NoWay' Direction	5.1	O	
SCR-DS-CLIENT-005	Support of 'Refresh' Behavior	5.1	O	
SCR-DS-CLIENT-006	Support of 'Preserve' Behavior	5.1	M	
SCR-DS-CLIENT-007	Support of 'Client Generated Fingerprint'	5.2	M	
SCR-DS-CLIENT-008	Support of 'Mutually Generated Fingerprint'	5.2	O	
SCR-DS-CLIENT-009	Support for 'Multiple Messages'	5.14	M	
SCR-DS-CLIENT-010	Support of 'Large Objects'	5.15	O	
SCR-DS-CLIENT-011	Support for 'Hierarchical Synchronization'	5.16	O	
SCR-DS-CLIENT-012	Support of 'Busy Signaling'	5.18	O	
SCR-DS-CLIENT-013	Support for 'Filtering'	5.22	O	
SCR-DS-CLIENT-014	Support for 'Session Maitenance'	6	M	
SCR-DS-CLIENT-015	Support for 'Session End'	6	M	
SCR-DS-CLIENT-016	Support of 'Normal Sync'	7.2	M	
SCR-DS-CLIENT-017	Support of 'Recovery Sync'	7.2	O	
SCR-DS-CLIENT-018	Support for 'Sync Initialization'	8	M	
SCR-DS-CLIENT-019	Support of 'Sync Without Separate Initialization'	8.3	O	
SCR-DS-CLIENT-020	Support of 'Authentication'	9.2	M	
SCR-DS-CLIENT-021	Support for 'Data Intergrity'	9.3	O	
SCR-DS-CLIENT-022	Support for 'Data Encryption'	9.4	O	

A.2 Conformance Requirements for OMA DS Server

Table 2 – Server Features

Item	Functionality	Reference	Status	Requirement
SCR-DS-SERVER-001	Support of ‘twoWay’ Direction	5.1	M	
SCR-DS-SERVER-002	Support of ‘fromClient’ Direction	5.1	M	
SCR-DS-SERVER-003	Support of ‘fromServer’ Direction	5.1	M	
SCR-DS-SERVER-004	Support of ‘NoWay’ Direction	5.1	O	
SCR-DS-SERVER-005	Support of ‘Refresh’ Behavior	5.1	M	
SCR-DS-SERVER-006	Support of ‘Preserve’ Behavior	5.1	M	
SCR-DS-SERVER-007	Support of ‘Client Generated Fingerprint’	5.2	M	
SCR-DS-SERVER-008	Support of ‘Mutually Generated Fingerprint’	5.2	O	
SCR-DS-SERVER-009	Support for ‘Multiple Messages’	5.14	M	
SCR-DS-SERVER-010	Support of ‘Large Objects’	5.15	O	
SCR-DS-SERVER-011	Support for ‘Hierarchical Synchronization’	5.16	O	
SCR-DS-SERVER-012	Support of ‘Busy Signaling’	5.18	O	
SCR-DS-SERVER-013	Support for ‘Filtering’	5.22	O	
SCR-DS-SERVER-014	Support for ‘Session Maitenance’	6	M	
SCR-DS-SERVER-015	Support for ‘Session End’	6	M	
SCR-DS-SERVER-016	Support of ’Normal Sync’	7.2	M	
SCR-DS-SERVER-017	Support of ’Recovery Sync’	7.2	M	
SCR-DS-SERVER-018	Support for ‘Sync Initialization’	8	M	
SCR-DS-SERVER-019	Support of ‘Sync Without Separate Initialization’	8.3	M	
SCR-DS-SERVER-020	Support of ‘Authentication’	9.2	M	
SCR-DS-SERVER-021	Support for ‘Data Intergrity’	9.3	O	
SCR-DS-SERVER-022	Support for ‘Data Encryption’	9.4	M	

Appendix B. Change History (Informative)

B.1 Approved Version 2.0 History

Reference	Date	Description
OMA-TS-DS_Protocol-V2_0-20110719-A	19 Jul 2011	Status changed to Approved by TP: OMA-TP-2011-0258-INP_DS_V2_0_ERP_for_final_Approval