IMAPEXT Working Group
Internet Draft: IMAP4 Disconnected Access
Document: draft-melnikov-imap-disc-06.txt
Expires: April 2005

A. Melnikov
Editor

October 2004

Synchronization operations for disconnected IMAP4 clients

Status of this Memo

Abstract

This document attempts to address some of the issues involved in building a disconnected IMAP4 [IMAP4] client. In particular, it deals with the issues of what might be called the "driver" portion of the synchronization tool: the portion of the code responsible for issuing the correct set of IMAP4 commands to synchronize the disconnected client in the way that is most likely to make the human who uses the disconnected client happy.

This note describes different strategies that can be used by disconnected clients as well as shows how to use IMAP protocol in order to minimize the time of synchronization process.

This note also lists IMAP extensions that a server should implement in order to provide better synchronization facilities to disconnected clients.

1. Introduction

Several recommendations presented in this document are generally applicable to all types of IMAP clients. However this document tries to concentrate on disconnected mail clients [IMAP-MODEL]. It also suggests some IMAP extensions* that should be implemented by IMAP servers in order to make the life of disconnected clients easier. In particular, the [UIDPLUS extension was specifically designed to streamline certain disconnected operations, like expunging, uploading and copying messages (see Sections 4.2.1, 4.2.2.1 and 4.2.4).

Readers of this document are also strongly advised to read RFC 2683 [RFC 2683].

* - note, that the functionality provided by the base IMAP protocol [IMAP4] is sufficient to perform basic synchronization.

1.1. Conventions Used in this Document

In examples, "C:" and "S:" indicate lines sent by the client and server respectively.

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 RFC 2119 [KEYWORDS].

Let's call an IMAP command idempotent, if the result of executing the command twice sequentially is the same as the result of executing the command just once.

Editorial comments/questions or missing paragraphs are marked in the text with << and >>.

2. Design Principles

All mailbox state or content information stored on the disconnected client should be viewed strictly as a cache of the state of the server. The "master" state remains on the server, just as it would with an interactive IMAP4 client. The one exception to this rule is that information about the state of the disconnected client's cache (the state includes flag changes while offline and scheduled message

uploads) remains on the disconnected client: that is, the IMAP4 server is not responsible for remembering the state of the disconnected IMAP4 client.

We assume that a disconnected client is a client that, for whatever reason, wants to minimize the length of time that it is "on the phone" to the IMAP4 server. Often this will be because the client is using a dialup connection, possibly with very low bandwidth, but sometimes it might just be that the human is in a hurry to catch an airplane, or some other event beyond our control. Whatever the reason, we assume that we must make efficient use of the network connection, both in the usual sense (not generating spurious traffic) and in the sense that we would prefer not to have the connection sitting idle while the client and/or the server is performing strictly local computation or I/O. Another, perhaps simpler way of stating this is that we assume that network connections are "expensive".

Practical experience with disconnected mail systems has shown that there is no single synchronization strategy that is appropriate for all cases. Different humans have different preferences, and the same human's preference will vary depending both on external circumstance (how much of a hurry the human is in today) and on the value that the human places on the messages being transferred. The point here is that there is no way that the synchronization program can guess exactly what the human wants to do, so the human will have to provide some guidance.

Taken together, the preceding two principles lead to the conclusion that the synchronization program must make its decisions based on some kind of guidance provided by the human by selecting the appropriate options in UI or through some sort of configuration file, but almost certainly should not pause for I/O with the human during the middle of the synchronization process. The human will almost certainly have several different configurations for the synchronization program, for different circumstances.

Since a disconnected client has no way of knowing what changes might have occurred to the mailbox while it was disconnected, message numbers are not useful to a disconnected client. All disconnected client operations should be performed using UIDs, so that the client can be sure that it and the server are talking about the same messages during the synchronization process.

3. Overall picture of synchronization

The basic strategy for synchronization is outlined below. Note that the real strategy may vary from one application to another or may depend on a synchronization mode.

a) Process any "actions" that were pending on the client that were not associated with any mailbox (in particular sending messages composed offline with SMTP. This is not part of IMAP synchronization, but it is mentioned here for completeness);

b) Fetch the current list of "interesting" mailboxes (The disconnected client should allow the user to skip this step completely);

c) "Client-to-server synchronization" - for each IMAP "action" that were pending on the client:

   1) If the action implies opening a new mailbox (any operation that operates on messages) - open the mailbox. Check its UID validity value (see section 4.1 for more details) returned in the UIDVALIDITY response code. If the UIDVALIDITY value returned by the server differs, the client MUST empty the local cache of the mailbox and remove any pending "actions" which refer to UIDs in that mailbox (and consider them failed). Note, this doesn't affect actions performed on client generated fake UIDs (see section 5).

   2) Perform the action. If the action is to delete a mailbox (DELETE), make sure that the mailbox is closed first (see also Section 3.4.12 of [RFC 2683]).

d) "Server-to-client synchronization" - for each mailbox that requires synchronization, do the following:

   1) Check the mailbox UIDVALIDITY (see section 4.1 for more details). with SELECT/EXAMINE/STATUS.
      If UIDVALIDITY value returned by the server differs,
      the client MUST

      * empty the local cache of that mailbox;
      * remove any pending "actions" which refer to UIDs in that mailbox and consider them failed;
      * skip step 2-II;

   2) Fetch the current "descriptors";

      I) Discover new messages.

      II) Discover changes to old messages.

   3) Fetch the bodies of any "interesting" messages that the client doesn't already have.

e) Close all open mailboxes not required for further operations (if staying online) or disconnect all open connections (if going offline).

Terms used:

"Actions" are queued requests that were made by the human to the client's MUA software while the client was disconnected.

Let define "descriptors" as a set of IMAP4 FETCH data items. Conceptually, a message's descriptor is that set of information that allows the synchronization program to decide what protocol actions are necessary to bring the local cache to the desired state for this message; since this decision is really up to the human, this information probably includes a at least a few header fields intended for human consumption. Exactly what will constitute a descriptor depends on the client implementation. At a minimum, the descriptor contains the message's UID and FLAGS. Other likely candidates are the RFC822.SIZE, RFC822.HEADER, BODYSTRUCTURE or ENVELOPE data items.

Comments:

1). The list of actions should be ordered. E.g., if the human deletes message A1 in mailbox A, then expunges mailbox A, then deletes message A2 in mailbox A, the human will expect that message A1 is gone and that message A2 is still present but is now deleted.

By processing all the actions before proceeding with synchronization, we avoid having to compensate for the local MUA's changes to the server's state. That is, once we have processed all the pending actions, the steps that the client must take to synchronize itself will be the same no matter where the changes to the server's state originated.

2). Steps a) and b) can be performed in parallel. Alternatively step a) can be performed after d).

3). On step b) the set of "interesting" mailboxes pretty much has to be determined by the human. What mailboxes belong to this set may vary between different IMAP4 sessions with the same server, client, and human. An interesting mailbox can be a mailbox returned by LSUB command. The special mailbox "INBOX" SHOULD always be considered "interesting".

4). On step d-2-II) the client also finds out about changes to the flags of messages that the client already has in its local cache, as well as finding out about messages in the local cache that no longer exist on the server (i.e., messages that have been expunged).

5). "Interesting" messages are those messages that the synchronization program thinks the human wants to have cached locally, based on the configuration and the data retrieved in step (b).

6). A disconnected IMAP client is a special case of an IMAP client, so it MUST be able to handle any "unexpected" unsolicited responses, like EXISTS and EXPUNGE, at any time.
The disconnected client MAY ignore EXPUNGE response during "client-to-server" synchronization phase (step c)).

The rest of this discussion will focus primarily on the synchronization issues for a single mailbox.

4. Mailbox synchronization steps and strategies

4.1. Checking UID Validity

The "UID validity" of a mailbox is a number returned in an UIDVALIDITY response code in an OK untagged response at mailbox selection time. The UID validity value changes between sessions when UIDs fail to persist between sessions.

Whenever the client selects a mailbox, the client must compare the returned UID validity value with the value stored in the local cache. If the UID validity values differ, the UIDs in the client's cache are no longer valid. The client MUST then empty the local cache of that mailbox and remove any pending "actions" which refer to UIDs in that mailbox. The client MAY also issue a warning to the human. The client MUST NOT cancel any scheduled uploads (i.e. APPENDs) for the mailbox.

Note that UIDVALIDITY is not only returned on a mailbox selection. The COPYUID and APPENDUID response codes defined in the [UIDPLUS] extension (see also 4.2.2) and the UIDVALIDITY STATUS response data item also contain a UIDVALIDITY value for some other mailbox. The client SHOULD behave as described in the previous paragraph (but it should act on the other mailbox' cache), no matter how it obtained the UIDVALIDITY value.

4.2. Synchronizing local changes with the server

4.2.1. Uploading messages to the mailbox

Two of the most common examples of operations resulting in message uploads are:

1) Saving a draft message
2) Copying a message between remote mailboxes on two different IMAP servers or a local mailbox and a remote mailbox.

Message upload is performed with the APPEND command. A message scheduled to uploaded has no UID associated with it, as all UIDs are assigned by the server. The APPEND command will effectively associate a UID with the uploade message that can be stored in the local cache for future reference.

However [IMAP4] doesn't describe a simple mechanism to discover the message
by just performing the APPEND command. In order to discover the UID the clie
do one of the following:

1) Remove the uploaded message from cache. After that use the mechanism desc
   in 4.3 to fetch the information about the uploaded message as if it had b
   by some other client.

2) Try to fetch header information as described in 4.2.2 in order to find a
   that corresponds to the uploaded message. One strategy for doing this is
   in 4.2.2.

Case 1) describes a not particularly smart client.

```
C: A003 APPEND Drafts (\Seen $MDNSent) {310}
S: + Ready for literal data
C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
C: From: Fred Foobar <foobar@Blurdybloop.COM>
C: Subject: afternoon meeting
C: To: mooch@owatagu.siam.edu
C: Message-Id: <B27397-0100000@Blurdybloop.COM>
C: MIME-Version: 1.0
C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
C:
C: Hello Joe, do you think we can meet at 3:30 tomorrow?
C:
S: A003 OK APPEND Completed
```

Fortunately there is a simpler way to discover the message UID in the presen
of the [UIDPLUS] extension:

```
C: A003 APPEND Drafts (\Seen $MDNSent) {310}
S: + Ready for literal data
C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
C: From: Fred Foobar <foobar@Blurdybloop.COM>
C: Subject: afternoon meeting
C: To: mooch@owatagu.siam.edu
C: Message-Id: <B27397-0100000@Blurdybloop.COM>
C: MIME-Version: 1.0
C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
C:
C: Hello Joe, do you think we can meet at 3:30 tomorrow?
C:
S: A003 OK [APPENDUID 1022843275 77712] APPEND completed
```

The UID of the appended message is the second parameter of APPENDUID
response code.

4.2.2. Optimizing "move" and "copy" operations

Practical experience with IMAP, and other mailbox access protocols that support multiple mailboxes suggests that moving a message from one mailbox to another is an extremely common operation.

4.2.2.1. Moving a message between two mailboxes on the same server

In IMAP4 a "move" operation between two mailboxes on the same server is really a combination of a COPY operation and a STORE +FLAGS (\Deleted) operation. This makes good protocol sense for IMAP, but it leaves a simple-minded disconnected client in the silly position of deleting and possibly expunging its cached copy of a message, then fetching an identical copy via the network.

However, the presence of the UIDPLUS extension in the server can help:

```
   C: A001 UID COPY 567,414 "Interesting Messages"
   S: A001 OK [COPYUID 1022843275 414,567 5:6] Completed
```

This tells the client that the message with UID 414 in the current mailbox was successfully copied to the mailbox "Interesting Messages" and was given the UID 5, and that the message with UID 567 was given the UID 6.

In the absence of UIDPLUS extension support in the server the following trick can be used. By including the Message-ID: header and the INTERNALDATE data item as part of the descriptor, the client can check the descriptor of a "new" message against messages that are already in its cache, and avoid fetching the extra copy. Of course, it's possible that the cost of checking to see if the message is already in the local cache may exceed the cost of just fetching it, so this technique should not be used blindly. If the MUA implements a "move" command, it makes special provisions to use this technique when it knows that a copy/delete sequence is the result of a "move" command.

Note, that servers are not required (although they are strongly encouraged with "SHOULD language") to preserve INTERNALDATE when copying messages.

Also note, since it's theoretically possible for this algorithm to find the wrong message (given sufficiently malignant Message-ID headers), implementors should provide a way to disable this optimization, both permanently and on a message-by-message basis.

Example: Copying a message in the absence of UIDPLUS extension.

At some point in time the client has fetch the source message and some information was cached:

```
   C: C021 UID FETCH <uids> (BODY.PEEK[] INTERNALDATE FLAGS)
   ...
   S: * 27 FETCH (UID 123 INTERNALDATE "31-May-2002 05:26:59 -0600"
      FLAGS (\Draft $MDNSent) BODY[] {1036}
   S: ...
   S: Message-Id: <20040903110856.22a127cd@chardonnay>
```

```
S: ...
S: ...message body...
S: )
...
S: C021 OK fetch completed
```

Later on the client decides to copy the message:

```
C: C035 UID COPY 123 "Interesting Messages"
S: C035 OK Completed
```

As the server hasn't provided the COPYUID response code, the client tries the optimization described above:

```
C: C036 SELECT "Interesting Messages"
...
C: C037 UID SEARCH ON 31-May-2002 HEADER
   "Message-Id" "20040903110856.22a127cd@chardonnay"
S: SEARCH 12368
S: C037 OK completed
```

Note, that if the server has returned multiple UIDs in the SEARCH response the client MUST NOT use any of the returned UID.

4.2.2.2. Moving a message from a remote mailbox to a local

Moving a message from a remote mailbox to a local is done with FETCH (that includes FLAGS and INTERNALDATE) followed by UID STORE <uid> +FLAGS.SILENT (\Deleted):

```
C: A003 UID FETCH 123 (BODY.PEEK[] INTERNALDATE FLAGS)
S: * 27 FETCH (UID 123 INTERNALDATE "31-May-2002 05:26:59 -0600"
   FLAGS (\Seen $MDNSent) BODY[]
S: ...message body...
S: )
S: A003 OK UID FETCH completed
C: A004 UID STORE <uid> +FLAGS.SILENT (\Deleted)
S: A004 STORE completed
```

Note, that there is no reason to fetch the message during synchronization if it's already in the client's cache. Also, the client SHOULD preserve delivery date in the local cache.

4.2.2.3. Moving a message from a local mailbox to a remote

Moving a message from a local mailbox to a remote is done with APPEND:

```
C: A003 APPEND Drafts (\Seen $MDNSent) "31-May-2002 05:26:59 -0600" {310}
S: + Ready for literal data
```

```
C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
C: From: Fred Foobar <foobar@Blurdybloop.COM>
C: Subject: afternoon meeting
C: To: mooch@owatagu.siam.edu
C: Message-Id: <B27397-0100000@Blurdybloop.COM>
C: MIME-Version: 1.0
C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
C:
C: Hello Joe, do you think we can meet at 3:30 tomorrow?
C:
S: A003 OK [APPENDUID 1022843275 77712] completed
```

The client SHOULD specify delivery date from the local cache in the APPEND.

If the [LITERAL+] extension is available, the client can save a round trip*:

```
C: A003 APPEND Drafts (\Seen $MDNSent) "31-May-2002 05:26:59 -0600" {310+
C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
C: From: Fred Foobar <foobar@Blurdybloop.COM>
C: Subject: afternoon meeting
C: To: mooch@owatagu.siam.edu
C: Message-Id: <B27397-0100000@Blurdybloop.COM>
C: MIME-Version: 1.0
C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
C:
C: Hello Joe, do you think we can meet at 3:30 tomorrow?
C:
S: A003 OK [APPENDUID 1022843275 77712] completed
```

* - Note that there is a risk that the server will reject the message due to its size. If this happens, the client will waste bandwidth transferring the whole message. If the client wouldn't have used the LITERAL+, this could have been avoided:

```
C: A003 APPEND Drafts (\Seen $MDNSent) "31-May-2004 05:26:59 -0600"
   {16777215}
S: A003 NO Sorry, message is too big
```

4.2.2.4. Moving a message between two mailboxes on different servers

Moving a message between two mailbox on two different servers is a combination of the operations described in 4.2.2.2 followed by the operations described in 4.2.2.3.

4.2.2.5. Uploading multiple messages to a remote mailbox with MULTIAPPEND

When there is a need to upload multiple messages to a remote mailbox (e.g. as per 4.2.2.3), the presence of certain IMAP extensions may significantly improve performance. One of them is [MULTIAPPEND].

For some mail stores opening a mailbox for appending might be expensive. [MULTIAPPEND] tells the server to open mailbox once (instead of opening and closing it "n" times per "n" messages to be uploaded) and keep it open while a group of messages is being uploaded to the server.

Also, if the server supports both [MULTIAPPEND] and [LITERAL+] extensions, the entire upload is accomplished in a single command/response round trip.

Note: Client implementors should be aware, that [MULTIAPPEND] performs append of multiple messages atomically. This means, for example, if there is not enough space to save "n"-th message (or the message has invalid structure and is rejected by the server) after successful upload of "n-1" messages, the whole upload operation fails and no message will be saved in the mailbox. Although, this behavior might be desirable in certain situations, it might not be what you want. Otherwise, the client should use the regular APPEND command (Section 4.2.2.3), possibly utilizing the [LITERAL+] extension.
See also section 5.1 for discussions about error recovery.

Note: MULTIAPPEND can be used together with the UIDPLUS extension in a way similar to what was described in section 4.2.1. [MULTIAPPEND] extends the syntax of the APPENDUID response code to allow for multiple message UIDs in the second parameter.

Example:
 An example below demonstrates the use of MULTIAPPEND together with
 UIDPLUS (synchronization points where the client waits for confirmations
 from the server are marked with "<--->"):

```
   C: A003 APPEND Jan-2002 (\Seen $MDNSent) "31-May-2002 05:26:59 -0600" {31
 <--->
   S: + Ready for literal data
   C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
   C: From: Fred Foobar <foobar@Blurdybloop.COM>
   C: Subject: afternoon meeting
   C: To: mooch@owatagu.siam.edu
   C: Message-Id: <B27397-0100000@Blurdybloop.COM>
   C: MIME-Version: 1.0
   C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
   C:
   C: Hello Joe, do you think we can meet at 3:30 tomorrow?
   C:  (\Seen) " 1-Jun-2002 22:43:04 -0800" {286}
 <--->
   S: + Ready for literal data
   C: Date: Mon, 7 Feb 1994 22:43:04 -0800 (PST)
   C: From: Joe Mooch <mooch@OWaTaGu.siam.EDU>
   C: Subject: Re: afternoon meeting
   C: To: foobar@blurdybloop.com
   C: Message-Id: <a0434793874930@OWaTaGu.siam.EDU>
   C: MIME-Version: 1.0
```

```
   C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
   C:
   C: 3:30 is fine with me.
   C:
   S: A003 OK [APPENDUID 1022843275 77712,77713] completed
```

The upload takes 3 round trips.

Example:
The example above was modified for the case when the server supports MULTIAPPEND, LITERAL+ and UIDPLUS. The upload takes only 1 round trip.

```
   C: A003 APPEND Jan-2002 (\Seen $MDNSent) "31-May-2002 05:26:59 -0600" {31
   C: Date: Mon, 7 Feb 1994 21:52:25 -0800 (PST)
   C: From: Fred Foobar <foobar@Blurdybloop.COM>
   C: Subject: afternoon meeting
   C: To: mooch@owatagu.siam.edu
   C: Message-Id: <B27397-0100000@Blurdybloop.COM>
   C: MIME-Version: 1.0
   C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
   C:
   C: Hello Joe, do you think we can meet at 3:30 tomorrow?
   C:  (\Seen) " 1-Jun-2002 22:43:04 -0800" {286+}
   C: Date: Mon, 7 Feb 1994 22:43:04 -0800 (PST)
   C: From: Joe Mooch <mooch@OWaTaGu.siam.EDU>
   C: Subject: Re: afternoon meeting
   C: To: foobar@blurdybloop.com
   C: Message-Id: <a0434793874930@OWaTaGu.siam.EDU>
   C: MIME-Version: 1.0
   C: Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
   C:
   C: 3:30 is fine with me.
   C:
   S: A003 OK [APPENDUID 1022843275 77712,77713] completed
```

4.2.3. Replaying local flag changes

The disconnected client uses STORE command to synchronize local flag state with the server. The disconnected client SHOULD use +FLAGS.SILENT or -FLAGS. in order to set or unset flags modified by the user while offline. The FLAGS form MUST NOT be used, as there is a risk that this will overwrite flags on the server that has been changed by some other client.

Example:
For the message with UID 15, the disconnected client stores the following flags \Seen and $Highest. The flags were modified on the server by some oth client: \Seen, \Answered and $Highest.
While offline the user requested to remove $Highest flags and to add \Delet The flag synchronization sequence for the message should look like:

```
C: A001 UID STORE 15 +FLAGS.SILENT (\Deleted)
S: A001 STORE completed
C: A002 UID STORE 15 -FLAGS.SILENT ($Highest)
S: A002 STORE completed
```

If the disconnected client is able to store an additional binary state information (or a piece of information that can take a value from a predefin set of values) in the local cache of an IMAP mailbox or in a local mailbox (e.g. message priority), and if the server supports storing of arbitrary keywords, the client MUST use keywords to store this state on the server.

Example:
Imagine a speculative mail client that can mark a message as one of work-re ($Work), personal ($Personal) or spam ($Spam). In order to mark a message a personal the client issues:

```
C: A001 UID STORE 15 +FLAGS.SILENT ($Personal)
S: A001 STORE completed
C: A002 UID STORE 15 -FLAGS.SILENT ($Work $Spam)
S: A002 STORE completed
```

In order to mark the message as neither work, nor personal, not spam, the c issues:

```
C: A003 UID STORE 15 -FLAGS.SILENT ($Personal $Work $Spam)
S: A003 STORE completed
```

4.2.4. Processing mailbox compression (EXPUNGE) requests

A naive disconnected client implementation that supports compressing a mailb while offline may decide to issue an EXPUNGE command to the server in order to expunge messages marked \Deleted. The problem with this command during synchronization is that it permanently erases all messages with the \Deleted i.e. even those messages that were marked as \Deleted on the server while th was offline. Doing this might result in an unpleasant surprise for the user.

Fortunately the [UIDPLUS] extension can help in this case as well. The exten introduces UID EXPUNGE command, that, unlike EXPUNGE, takes a UID set parame that lists UIDs of all messages that can be expunged. When processing this c the server erases only messages with \Deleted flag listed in the UID list. T messages not listed in the UID set will not be expunged even if they have th flag set.

Example: While offline 3 messages with UIDs 7, 27 and 65 were marked \Delete when the user requested to compress the open mailbox. Another client marke a message \Deleted on the server (UID 34). During synchronization the disconnected client issues:

```
C: A001 UID EXPUNGE 7,27,65
S: * ... EXPUNGE
```

```
 S: * ... EXPUNGE
 S: * ... EXPUNGE
 S: A001 UID EXPUNGE completed
```

If another client issues UID SEARCH DELETED command (to find all messages \Deleted flag) before and after the UID EXPUNGE it will get:

Before:

```
 C: B001 UID SEARCH DELETED
 S: * SEARCH 65 34 27 7
 S: B001 UID SEARCH completed
```

After:

```
 C: B002 UID SEARCH DELETED
 S: * SEARCH 34
 S: B002 UID SEARCH completed
```

In the absence of the [UIDPLUS] extension the following sequence of command used as an approximation. Note: It's possible for another client to mark add messages as deleted while this sequence is being performed. In this case, th additional messages will be expunged as well.

1). Find all messages marked \Deleted on the server:

```
 C: A001 UID SEARCH DELETED
 S: * SEARCH 65 34 27 7
 S: A001 UID SEARCH completed
```

2). Find all messages that must not be erased (for the previous example the list will consist of the message with UID 34)

3). Temporary remove \Deleted flag on all messages found in step 2)

```
 C: A002 UID STORE 34 -FLAGS.SILENT (\Deleted)
 S: A002 UID STORE completed
```

4). Expunge the mailbox

```
 C: A003 EXPUNGE
 S: * 20 EXPUNGE
 S: * 7 EXPUNGE
 S: * 1 EXPUNGE
 S: A003 EXPUNGE completed
```

Here message with UID 7 has message number 1; with UID 27 - message number 7 and with UID 65 - message number 20.

5). Restore \Deleted flag on all messages found when performing step 2)

```
 C: A004 UID STORE 34 +FLAGS.SILENT (\Deleted)
 S: A004 UID STORE completed
```

4.2.5. Closing a mailbox

When the disconnected client has to close a mailbox, it should not use CLOSE command, because CLOSE does a silent EXPUNGE (section 4.2.4 explains why EXPUNGE should not be used by a disconnected client). It is safe to use CLOSE only if the mailbox was opened with EXAMINE.

If the mailbox was opened with SELECT, the client can use one of the following commands to implicitly close the mailbox and prevent the silent expunge:

1). UNSELECT - This is a command described in [UNSELECT] that works as CLOSE, but doesn't cause the silent EXPUNGE. This command is supported by the server if it reports UNSELECT in its CAPABILITY list.
2). SELECT <another_mailbox> - SELECT causes implicit CLOSE without EXPUNGE.
3). If the client intends to issue LOGOUT after closing the mailbox, it may just issue LOGOUT, because LOGOUT causes implicit CLOSE without EXPUNGE as well.
4). SELECT <non_existing_mailbox> - if the client knows a mailbox that doesn exist or can't be selected, it MAY SELECT it.

If the client opened the mailbox with SELECT and just wants to avoid implicit EXPUNGE without closing the mailbox, it may also use the following:

5). EXAMINE <mailbox> - reselect the same mailbox in read-only mode.

4.3. Details of "Normal" synchronization of a single mailbox

The most common form of synchronization is where the human trusts the integrity of the client's copy of the state of a particular mailbox, and simply wants to bring the client's cache up to date so that it accurately reflects the mailbox's current state on the server.

4.3.1. Discovering new messages and changes to old messages

Let <lastseenuid> represent the highest UID that the client knows about in this mailbox. Since UIDs are allocated in strictly ascending order, this is simply the UID of the last message in the mailbox that the client knows about. Let <lastseenuid+1> represent <lastseenuid>'s UID plus one. Let <descriptors> represent a list consisting of all the FETCH data item items that the implementation considers to be part of the descriptor; at a minimum this is just the FLAGS data item, but it usually also includes BODYSTRUCTURE and RFC822.SIZE. At this step <descriptors> SHOULD NOT include RFC822.

With no further information, the client can issue the following two commands:

```
   tag1 UID FETCH <lastseenuid+1>:* <descriptors>
   tag2 UID FETCH 1:<lastseenuid> FLAGS
```

The first command will request some information about "new" messages (i.e. messages received by the server since the last synchronization). It will also allow the client to build a message number to UID map (only for new messages). The second command allows the client to
 1) update cached flags for old messages;
 2) find out which old messages got expunged;
 3) build a mapping between message numbers and UIDs (for old messages).

The order here is significant. We want the server to start returning the list of new message descriptors as fast as it can, so that the client can start issuing more FETCH commands, so we start out by asking for the descriptors of all the messages we know the client cannot possibly have cached yet. The second command fetches the information we need to determine what changes may have occurred to messages that the client already has cached. Note, that the former command should only be issued if the UIDNEXT value cached by the client differs from the one returned by the server. Once the client has issued these two commands, there's nothing more the client can do with this mailbox until the responses to the first command start arriving. A clever synchronization program might use this time to fetch its local cache state from disk, or start the process of synchronizing another mailbox.

Example of the first FETCH:

```
   C: A011 UID fetch 131:* (FLAGS BODYSTRUCTURE INTERNALDATE RFC822.SIZE)
```

Note #1: The first FETCH may result in huge volume of data sent by the server. A smart disconnected client should use message ranges (see also section 3.2.1.2 of [RFC 2683]), so that the user is able to execute a different operation between fetching information for a group of new messages.

Example: Knowing the new UIDNEXT returned by the server on SELECT or EXAMINE (<uidnext>), the client can split the UID range <lastseenuid+1>:<uidnext> into groups, e.g. 100 messages. After that the client can issue:

```
   C: A011 UID fetch <lastseenuid+1>:<lastseenuid+100>
      (FLAGS BODYSTRUCTURE INTERNALDATE RFC822.SIZE)
   ...
   C: A012 UID fetch <lastseenuid+101>:<lastseenuid+200>
      (FLAGS BODYSTRUCTURE INTERNALDATE RFC822.SIZE)
   ...
   ...
   C: A0FF UID fetch <lastseenuid+901>:<uidnext>
      (FLAGS BODYSTRUCTURE INTERNALDATE RFC822.SIZE)
```

Note, that without issuing a SEARCH command it is not possible to

determine how many messages will fall into a subrange, as UIDs are not necessarily contiguous.

Note #2: The client SHOULD ignore any unsolicited EXPUNGE responses received during the first FETCH command. EXPUNGE responses contain message numbers which are useless to a client that doesn't have the message-number-to-UID translation table.

The second FETCH command will result in zero or more untagged fetch responses. Each response will have a corresponding UID FETCH data item. All messages that didn't have a matching untagged FETCH response MUST be removed from the local cache.

For example, if the <lastseenuid> had a value 15000 and the local cache contained 3 messages with the UIDs 12, 777 and 14999 respectively, than after receiving the following responses from the server:

```
S: * 1 FETCH (UID 12 FLAGS (\Seen))
S: * 2 FETCH (UID 777 FLAGS (\Answered \Deleted))
```

the client must remove the message with UID 14999 from its local cache.

Note #3: If the client is not interested in flag changes (i.e. the client only wants to know which old messages are still on the server), the second FETCH command can be substituted with:

```
tag2 UID SEARCH UID 1:<lastseenuid>
```

This command will generate less traffic. However an implementor should be aware that in order to build the mapping table from message numbers to UIDs the output of the SEARCH command MUST be sorted first, because there is no requirement for a server to return UIDs in SEARCH response in any particular order.

4.3.2. Searching for "interesting" messages.

This step is either performed entirely on the client (from the information r
in step 4.3.1), entirely on the server or some combination of both.
The decision on what is an "interesting" message is up to the client softwar
and the human. One easy criterion that should probably be implemented in an
client is whether the message is "too big" for automatic retrieval, where "t
is a parameter defined in the client's configuration.

Another commonly used criterion is the age of a message. For example, the cl
may choose to download only messages received in the last week (in this case
would be today's date minus 7 days):

```
tag3 UID SEARCH UID <uidset> SINCE <date>
```

Keep in mind that a date search disregards time and timezone.
The client can avoid doing this search if it specified INTERNALDATE in <desc

on step 4.3.1. If the client did, it can perform the local search on its mes

At this step the client also decides what kind of information about a partic
message to fetch from the server. In particular, even for a message that is
to be "too big" the client MAY choose to fetch some part(s) of it. For examp
if the message is a multipart/mixed containing a text part and a MPEG attach
there is no reason for the client not to fetch the text part. The decision o
part should or should not be fetched can be based on the information receive
the BODYSTRUCTURE FETCH response data item (i.e. if BODYSTRUCTURE was includ
<descriptors> on step 4.3.1).

4.3.3. Populating cache with "interesting" messages.

Once the client has found out which messages are "interesting", it
can start issuing appropriate FETCH commands for "interesting" messages or
parts thereof.

It is important to note that fetching a message into the disconnected
client's local cache does NOT imply that the human has (or even will)
read the message. Thus, the synchronization program for a
disconnected client should always be careful to use the .PEEK
variants of the FETCH data items that implicitly set the \Seen flag.

Once the last descriptor has arrived and the last FETCH command has
been issued, the client simply needs to process the incoming fetch
items, using them to update the local message cache.

In order to avoid deadlock problems, the client must give processing
of received messages priority over issuing new FETCH commands during
this synchronization process. This may necessitate temporary local
queuing of FETCH requests that cannot be issued without causing a
deadlock. In order to achieve the best use of the "expensive" network
connection, the client will almost certainly need to pay careful
attention to any flow-control information that it can obtain from the
underlying transport connection (usually a TCP connection).

Note: The requirement stated in the previous paragraph might result in
an unpleasant user experience, if followed blindly. For example, the
user might be unwilling to wait for the client to finish synchronization
before starting to process the user's requests. A smart disconnected clien
should allow the user to perform requested operations in between IMAP
commands which are part of the synchronization process. See also the
Note #1 in section 4.3.1.

Example: After fetching a message BODYSTRUCTURE the client discovers
a complex MIME message. Than it decides to fetch MIME headers
of the nested MIME messages and some body parts.

```
C: A011 UID fetch 11 (BODYSTRUCTURE)
S: ...
```

```
C: A012 UID fetch 11 (BODY[HEADER] BODY[1.MIME] BODY[1.1.MIME]
    BODY[1.2.MIME] BODY[2.MIME] BODY[3.MIME] BODY[4.MIME] BODY[5.MIME]
    BODY[6.MIME] BODY[7.MIME] BODY[8.MIME] BODY[9.MIME] BODY[10.MIME]
    BODY[11.MIME] BODY[12.MIME] BODY[13.MIME] BODY[14.MIME] BODY[15.MIME]
    BODY[16.MIME] BODY[17.MIME] BODY[18.MIME] BODY[19.MIME] BODY[20.MIME]
    BODY[21.MIME])
S: ...
C: A013 UID fetch 11 (BODY[1.1] BODY[1.2])
S: ...
C: A014 UID fetch 11 (BODY[3] BODY[4] BODY[5] BODY[6] BODY[7] BODY[8]
    BODY[9] BODY[10] BODY[11] BODY[13] BODY[14] BODY[15] BODY[16]
    BODY[21])
S: ...
```

4.3.4. User initiated synchronization

After the client has finished the main synchronization process as described 4.3.1-4.3.3, the user may optionally request additional synchronization step while the client is still online. This is not any different from the process described in 4.3.2 and 4.3.3.

Typical examples are:
1) fetch all messages selected in UI.
2) fetch all messages marked as \Flagged on the server.

4.4. Special case: descriptor-only synchronization

For some mailboxes, fetching the descriptors might be the entire synchronization step. Practical experience with IMAP has shown that a certain class of mailboxes (e.g., "archival" mailboxes) are used primarily for long-term storage of important messages that the human wants to have instantly available on demand but does not want cluttering up the disconnected client's cache at any other time. Messages in this kind of mailbox would be fetched exclusively by explicit actions queued by the local MUA. Thus, the only synchronization desirable on this kind of mailbox is fetching enough descriptor information for the user to be able to identify messages for subsequent download.

Special mailboxes that receive messages from a high volume, low priority mailing list might also be in this category, at least when the human is in a hurry.

4.5. Special case: fast new-only synchronization

In some cases the human might be in such a hurry that s/he doesn't care about changes to old messages, just about new messages. In this case, the client can skip the UID FETCH command that obtains the

flags and UIDs for old messages (1:<lastseenuid>).

4.6. Special case: blind FETCH

In some cases the human may know (for whatever reason) that s/he always wants to fetch any new messages in a particular mailbox, unconditionally. In this case, the client can just fetch the messages themselves, rather than just the descriptors, by using a command like:

```
tag1 UID FETCH <lastseenuid+1>:* (FLAGS BODY.PEEK[])
```

Note, that this example ignores the fact that the messages can be arbitrary long. The disconnected client MUST always check for message size before downloading, unless explicitly told otherwise. A well behaved client should use instead something like the following:

1) Issue "tag1 UID FETCH <lastseenuid+1>:* (FLAGS RFC822.SIZE)"
2) From the message sizes returned in step 1 construct UID set <required_messages>
3) Issue "tag2 UID FETCH <required_messages> (BODY.PEEK[])"

or

1) Issue "tag1 UID FETCH <lastseenuid+1>:* (FLAGS)"
2) Construct UID set <old_uids> from the responses of 1)
3) Issue "tag2 SEARCH UID <old_uids> SMALLER <message_limit>" Construct UID set <required_messages> from the result of the SEARCH command.
4) Issue "tag3 UID FETCH <required_messages> (BODY.PEEK[])"

or

1) Issue "tag1 UID FETCH <lastseenuid+1>:* (FLAGS BODY.PEEK[]<0.<length>>)" where <length> should be replaced with the maximal message size the client is willing to download.
   Note: In response to such a command, the server will only return partial data if the message is longer than <length>. It will return the full message data for any message whose size is smaller than or equal to <length>. In the former case, the client will not be able to extract the full [MIME] structure of the message from the truncated data, so the client should include BODYSTRUCTURE in the UID FETCH command as well.

5. Implementation considerations

Below are listed some common implementation pitfalls that should be considered when implementing a disconnected client.

1) Implementing fake UIDs on the client.

   A message scheduled to be uploaded has no UID, as UIDs are selected by
   the server. The client may implement fake UIDs internally in order to
   reference not yet uploaded messages in further operations. For example,
   a message could be scheduled to be uploaded, but subsequently marked as d
   to another mailbox). Here the client MUST NOT under any circumstances
   sent these fake UIDs to the server. Also, client implementors should
   be reminded that according to [IMAP4] an UID is a 32bit unsigned integer
   excluding 0. So, both 4294967295 and 2147483648 are valid UIDs and 0 and
   are both invalid. Some disconnected mail clients have been known to send
   negative numbers (e.g. "-1") as message UIDs to servers during synchroniz

   Example 1: The user starts composing a new message, edits it, saves it,
   continues to edit and saves it again.

   A disconnected client may record in its replay log (log of operations
   to be replayed on the server during synchronization) the sequence of
   operations as shown below. For the purpose of this example we assume
   that all draft messages are stored in the mailbox called Drafts on an
   IMAP server. We will also use the following conventions:
    <old_uid> UID of the intermediate version of the draft when it was saved
              for the first time. This is a fake UID generated on the client
    <new_uid> UID of the final version of the draft. This is another fake UI
              generated on the client.

    1). APPEND Drafts (\Seen $MDNSent \Drafts) {<nnn>}
        ...first version of the message follows...
    2). APPEND Drafts (\Seen $MDNSent \Drafts) {<mmm>}
        ...final version of the message follows...
    3). STORE <old_uid> +FLAGS (\Deleted)

   Step 1 corresponds to the first attempt to save the draft message,
   step 2 corresponds to the second attempt to save the draft message
   and the step 3 deletes the first version of the draft message saved
   in step 1.

   A naive disconnected client may send the command in step 3 without
   replacing the fake client generated <old_uid> with the value returned
   by the server in step 1. A server will probably reject this command,
   which will make the client believe that the synchronization sequence
   has failed.

2) Section 5.1 talks about common implementation errors related to error
   recovery during playback.

3) Don't assume that the disconnected client is the only client used by
   the user.

   <<Is the example below is generic enough to be moved elsewhere?>>
   Example 2: Some clients may use the \Deleted flag as an indicator that

the message should not appear in the user's view. Usage of the \Deleted flag for this purpose is not safe, as other clients (e.g. online clients) might EXPUNGE the mailbox at any time.

4) Beware of data dependencies between synchronization operations.

It might be very tempting for a client writer to perform some optimizations on the playback log. Such optimizations might include removing redundant operations (for example, see the optimization #2 in section 5.3), or their reordering.

It is not always safe to reorder or remove redundant operations during synchronization, because some operations may have dependencies. So if in doubt, don't do this. The following example demonstrates this:

Example 3: The user copies a message out of a mailbox and then deletes the mailbox.

```
C: A001 SELECT Old-Mail
S: ...
C: A002 UID COPY 111 ToDo
S: A002 OK [COPYUID 1022843345 111 94] Copy completed
...
C: A015 CLOSE
S: A015 OK Completed
C: A016 DELETE Old-Mail
S: A016 OK Mailbox deletion completed successfully
```

If the client performs DELETE (tag A016) first and COPY (tag A002) second, than the COPY fails. Also, the message that the user so carefully copied into another mailbox, has been lost.

5.1. Error recovery during playback

Error recovery during synchronization is one of the trickiest parts to get right. Below, we will discuss certain error conditions and suggest possible choices to handle them:

1). Lost connection to the server.

The client MUST remember the current position in playback (replay) log replay it starting from the interrupted operation (the last command issued by the client, but not acknowledged by the server) next time it successfully connects to the same server. If the connection was lost wh executing a non-idempotent IMAP command (see the definition in Section reconnected the client MUST make sure that the interrupted command was indeed not executed. If it wasn't executed, the client must restart pla from the interrupted command, otherwise from the following command.

When reconnected, care must be taken in order to properly reapply logic

operations that are represented by multiple IMAP commands, e.g. UID EXP
emulation when UID EXPUNGE is not supported by the server (see section

Once the client detects that the connection to the server was lost,
it MUST stop replaying its log. There are existing disconnected clients
that, to the great annoyance of users, pop up an error dialog
for each and every playback operation that fails.

2). Copying/appending messages to a mailbox that doesn't exist.
(The server advertises this condition by sending the TRYCREATE response
code in the tagged NO response to the APPEND or COPY command.)

The user should be advised about the situation and be given
one of the following choices:
a). Try to recreate a mailbox;
b). Copy/upload messages to another mailbox;
c). Skip copy/upload.
d). Abort replay.

3). Copying messages from, rename or get/change ACLs [ACL] on
a mailbox that doesn't exist:

a). Skip operation
b). Abort replay

4). Deleting mailboxes or deleting/expunging messages that no longer exist.

This is actually is not an error and should be ignored by the client.

5). Performing operations on messages that no longer exist.

a). Skip operation
b). Abort replay

In the case of changing flags on an expunged message the client should
silently ignore the error.

Note 1: Several synchronization operations map to multiple IMAP commands
(for example "move" described in 4.2.2). The client must guarantee
atomicity of each such multistep operation. For example,
when performing a "move" between two mailboxes on the same server,
if the server is unable to copy messages, the client MUST NOT attempt to
set the \Deleted flag on the messages being copied, let alone expunge
them. However, the client MAY consider that move operation succeeded
even if the server was unable to set the \Deleted flag on copied messages.

Note 2: Many synchronization operations have data dependencies.
A failed operation must cause all dependent operations to fail as
well. The client should check that and MUST NOT try to perform
all dependent operations blindly (unless the user corrected the original
problem). For example, a message may be scheduled to be appended to

a mailbox on the server and later on the appended message may be copied to another mailbox. If the APPEND operation fails, the client must not attempt to COPY the failed message later on. (See also Section 5, example 3)

5.2. Quality of implementation issues.

Below listed some quality of implementation issues for disconnected clients. They will help to write a disconnected client that works correctly, performs synchronization as quickly as possible (and thus can make the user happier as well as save her some money) and minimizes the server load:

1) Don't lose information.

   No matter how smart your client is in other areas, if it loses informatio users will get very upset.

2) Don't do work unless explicitly asked. Be flexible. Ask all questions BEFORE starting synchronization, if possible.

3) Minimize traffic

   The client MUST NOT issue a command if the client already received the required information from the server.

   The client MUST make use of UIDPLUS extension if it is supported by the server.

   See also optimization #1 in Section 5.3.

4) Minimize number of round trips.

   Round trips kill performance, especially on links with high latency. Sections 4.2.2.5 and 5.2 give some advices how to minimize number of round trips.

   See also optimization #1 in Section 5.3.

5.3. Optimizations

Some useful optimizations are described in this section. A disconnected client that supports the recommendations listed below will give the user a more pleasant experience.

1) The initial OK or PREAUTH responses may contain the CAPABILITY response c as described in section 7.1 of [IMAP4]. This response code gives the same information as returned by the CAPABILITY command(*). A disconnected client that pays attention to this response code can avoid sending CAPABILITY command and will save a round trip.

(*) - Note: Some servers report in the CAPABILITY response code extensions that are only relevant in unauthenticated state or in all states. Such servers usually send another CAPABILITY response code upon successful authentication using LOGIN or AUTHENTICATE command (that negotiates no security layer, see section 6.2.2 of [IMAP4]). The CAPABILITY response code sent upon successful LOGIN/AUTHENTICATE might be different from the CAPABILITY response code in the initial OK response, as extensions only relevant for unauthenticated state will not be advertised and some additional extensions available only in authenticated and/or selected state will be.

Example 1:

```
S: * OK [CAPABILITY IMAP4REV1 LOGIN-REFERRALS STARTTLS AUTH=DIGEST-MD5 AU
   imap.example.com ready
C: 2 authenticate DIGEST-MD5
...
S: 2 OK [CAPABILITY IMAP4REV1 IDLE NAMESPACE MAILBOX-REFERRALS SCAN SORT
   THREAD=REFERENCES THREAD=ORDEREDSUBJECT MULTIAPPEND] User authenticate
   (no layer)
```

2) An advanced disconnected client may choose to optimize its replay log. For example, there might be some operations which are redundant (the list is not complete):

 a) an EXPUNGE followed by another EXPUNGE or CLOSE;
 b) changing flags (other than the \Deleted flag) on a message that gets immediately expunged;
 c) opening and closing the same mailbox.

When optimizing, be careful about data dependencies between commands. For example, if the client is wishing to optimize (see case b) above)

```
 tag1 UID STORE <uid1> +FLAGS (\Deleted)
 ...
 tag2 UID STORE <uid1> +FLAGS (\Flagged)
 ...
 tag3 UID COPY <uid1> "Backup"
 ...
 tag4 UID EXPUNGE <uid1>
```

it can't remove the second UID STORE command, because the message is bein copied before it gets expunged.

In general, it might be a good idea to keep mailboxes open during synchronization (see case c) above), if possible. This can be more easily achieved in conjunction with optimization #3 described below.

3) Perform some synchronization steps in parallel, if possible.

Several synchronization steps don't depend on each other and thus can be performed in parallel. Because the server machine is usually more powerful than the client machine and can perform some operations in parallel, this may speed up the total time of synchronization.

In order to achieve such parallelization the client will have to open more than one connection to the same server. Client writers should not forget about non-trivial cost associated with establishing a TCP connecti and performing an authentication. The disconnected client MUST NOT use one connection per mailbox. In most cases it is sufficient to have two connections. The disconnected client SHOULD avoid selecting the same mailbox in more than one connection, see section 3.1.1 of the [RFC 2683] for more details.

Any mailbox synchronization MUST start with checking of the UIDVALIDITY as described in section 4.1 of this document. The client MAY use STATUS command to check UID Validity of a non selected mailbox. This is preferab to opening many connections to the same server to perform synchronization of multiple mailboxes simultaneously. As described in section 5.3.10 of [IMAP4], this SHOULD NOT be used on the selected mailbox.

6. IMAP extensions that may help

The following extensions can save traffic and/or number of round trips:

1) The use of [UIDPLUS] is discussed in sections 4.1, 4.2.1, 4.2.2.1 and 4.2

2) The use of the MULTIAPPEND and LITERAL+ extensions for uploading messages is discussed in 4.2.2.5.

3) Use the CONDSTORE extension (see section 6.1) for quick flag resynchroniz

6.1. CONDSTORE extension

An advance disconnected mail client should use the [CONDSTORE] extension when it is supported by the server. The client must cache the value from HIGHESTMODSEQ OK response code received on mailbox opening and update it whenever the server sends MODSEQ FETCH data items.

If the client receives NOMODSEQ OK untagged response instead of HIGHESTMODSEQ, it MUST remove the last known HIGHESTMODSEQ value from its cache and follow more general instructions in section 3.

When the client opens the mailbox for synchronization it first compares UIDVALIDITY as described in step d)1) in section 3. If the cached UIDVALIDITY value matches the one returned by the server, the client MUST compare the cached value of HIGHESTMODSEQ with the one returned by the server. If the cached HIGHESTMODSEQ value also matches the one returned by the server, then the client MUST NOT fetch flags for

cached messages, as they hasn't changed. If the value on the server is higher than the cached one, the client MAY use "SEARCH MODSEQ <cached-value>" to find all messages with flags changed since the last time the client was online and had the mailbox opened. Alternatively the client MAY use "FETCH 1:* (FLAGS) (CHANGEDSINCE <cached-value>)". The latter operation combines searching for changed messages and fetching new information.

In all cases the client still needs to fetch information about new messages (if requested by the user), as well as discover which messages have expunged.

Step d) ("Server-to-client synchronization") in section 4 in the presence of the CONDSTORE extension is amended as follows:

```
    d) "Server-to-client synchronization" - for each mailbox that requires
       synchronization, do the following:

      1a) Check the mailbox UIDVALIDITY (see section 4.1 for more details).
          with SELECT/EXAMINE/STATUS.
          If the UIDVALIDITY value returned by the server differs,
          the client MUST

          * empty the local cache of that mailbox;
          * "forget" the cached HIGHESTMODSEQ value for the mailbox;
          * remove any pending "actions" which refer to UIDs in
            that mailbox. Note, this doesn't affect actions performed on
            client generated fake UIDs (see section 5);
          * skip steps 1b and 2-II;

      1b) Check the mailbox HIGHESTMODSEQ. If the cached value is the same
          as the one returned by the server, skip fetching message flags
          on step 2-II, i.e. the client only has to find out which messages
          got expunged.

      2) Fetch the current "descriptors";

          I)  Discover new messages.

          II) Discover changes to old messages using
              "FETCH 1:* (FLAGS) (CHANGEDSINCE <cached-value>)" or
              "SEARCH MODSEQ <cached-value>".

      3) Fetch the bodies of any "interesting" messages that the client
         doesn't already have.

 Example (the UIDVALIDITY value is the same, but the HIGHESTMODSEQ value
          has changed on the server while the client was offline):

      C: A142 SELECT INBOX
      S: * 172 EXISTS
```

```
      S: * 1 RECENT
      S: * OK [UNSEEN 12] Message 12 is first unseen
      S: * OK [UIDVALIDITY 3857529045] UIDs valid
      S: * FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
      S: * OK [PERMANENTFLAGS (\Deleted \Seen \*)] Limited
      S: * OK [HIGHESTMODSEQ 20010715194045007]
      S: A142 OK [READ-WRITE] SELECT completed
```

after that either:

```
      C: A143 UID FETCH 1:* (FLAGS) (CHANGEDSINCE 20010715194032001)
      S: * 2 FETCH (UID 6 MODSEQ (20010715205008000) FLAGS (\Deleted))
      S: * 5 FETCH (UID 9 MODSEQ (20010715195517000) FLAGS ($NoJunk
           $AutoJunk $MDNSent))
         ...
      S: A143 OK FETCH completed
```

or:

```
      C: A143 SEARCH MODSEQ 20010715194032001
      S: * SEARCH 2 5 6 7 11 12 18 19 20 23 (MODSEQ 20010917162500)
      S: A143 OK Search complete
```

7. Security Considerations

It is believed that this document does not raise any new security concerns that are not already present in the base [IMAP] protocol, and these issues are discussed in [IMAP]. Additional security considerations may be found in different extensions mentioned in this document, in particular in [UIDPLUS], [LITERAL+], [CONDSTORE], [MULTIAPPEND] and [UNSELECT].

Implementors are also reminded about the importance of thorough testing.

8. References

8.1. Normative References

[KEYWORDS] Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, Harvard University, March 1997.

[IMAP4] Crispin, M., "Internet Message Access Protocol - Version 4rev1", RFC 3501, University of Washington, March 2003.

[UIDPLUS] Myers, J., "IMAP4 UIDPLUS extension", RFC 2359, June 1988.

[LITERAL+] Myers, J. "IMAP4 non-synchronizing literals", RFC 2088, January 1997.

[CONDSTORE] Melnikov, A., Hole, S., "IMAP Extension for Conditional STORE operation", Work in progress, draft-melnikov-imap-condstore-XX.txt,

Isode Limited, ACI WorldWide/MessagingDirect.

[MULTIAPPEND] Crispin, M., "INTERNET MESSAGE ACCESS PROTOCOL - MULTIAPPEND EXTENSION", RFC 3502, University of Washington, March 2003.

[UNSELECT] Melnikov, A., "Internet Message Access Protocol (IMAP) UNSELECT command", RFC 3691, Isode Limited, February 2004.

[RFC 2683] Leiba, B., "IMAP4 Implementation Recommendations", RFC 2683, September 1999.

8.2. Informative References

[ACL] Myers, J., "IMAP4 ACL Extension", RFC 2086, January 1997.
and
Melnikov, A., "IMAP4 ACL Extension", draft-ietf-imapext-acl-XX.txt, Work in Progress.

[IMAP-MODEL] Crispin, M. "Distributed Electronic Mail Models in IMAP4", RFC 1733, University of Washington, December 1994.

9. Acknowledgment

This document is based on the draft-ietf-imap-disc-01.txt written by Rob Austein in November 1994.

The editor appreciate comments posted by Mark Crispin to the IMAP mailing list and the comments/corrections/ideas received from Grant Baillie, Cyrus Daboo, John G. Myers, Chris Newman and Timo Sirainen.

The editor would also like to thank the developers of Netscape Messenger and Mozilla mail clients for providing examples of disconnected mail clients that served as a base for many recommendations in this document.

10. Editor's Address

Alexey Melnikov
mailto: alexey.melnikov@isode.com

Isode Limited
5 Castle Business Village,
36 Station Road,
Hampton, Middlesex,
United Kingdom, TW12 2BX

Phone: +44 77 53759732

11. Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

12. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.