IMAPEXT Working Group   A. Melnikov
Internet Draft: IMAP4 Disconnected Access   Editor
Document: draft-melnikov-imap-disc-00.txt   June 2002

Synchronization operations for disconnected IMAP4 clients

Status of this Memo

This is a draft document based on the expired draft written by the IETF IMAP Working Group. A revised version of this draft document will be submitted to the RFC editor as an Informational RFC for the Internet Community. Discussion and suggestions for improvement are requested, and should be sent to imap@CAC.Washington.EDU.

1.   Abstract

This document attempts to address some of the issues involved in building a disconnected 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.

2. 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].

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

3. 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 transfered. 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 preceeding 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 throught 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 occured 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.

4. 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;

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 5.1 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.

   2) Perform the action. If the action is to delete a mailbox (DELETE), make sure that the mailbox is closed first.

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

   1) Check its UIDVALIDITY (see section 5.1 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 and remove any pending "actions" which refer to UIDs in

that mailbox.

2) Fetch the current "descriptors";

<< Be more detailed? >>

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

d) Close all open mailboxes not required for further operations (if staying online) or disconnect all open connections (if going offl

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 and BODYSTRUCTURE data items.

Comments:

1). The list of actions should be ordered. Eg, 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. Special mailbox "INBOX" SHOULD always be considered "interesting".

4). On step d2) 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).

6). "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).

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

5. Mailbox synchronization steps and strategies

5.1. 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. COPYUID and APPENDUID response codes defined in [UIDPLUS] extension (see also 5.2.2) and 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.

5.2. Synchronizing local changes with the server

5.2.1. Uploading messages to the mailbox

There are two most typical examples of operations that will result in messag uploads:

1) Saving a draft message
2) Message copy between remote mailboxes on two different IMAP servers

or a local mailbox and a remote mailbox.

Note, that alternatively this step can be performed at the end of the automatic mailbox synchronization.

Message upload is performed with APPEND command. A message scheduled to be 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 a future reference. However [IMAP4] doesn't describe a simple mechanism to discover the message by just performing the APPEND command. In order to discover UID the client c do one of the following:

1) Remove the uploaded message from cache. After that use the mechanism desc in 5.3 to fetch the information about the uploaded message as if it was u by some other client.

2) Try to fetch header information as described in 5.2.2 in order to find a that corresponds to the uploaded message. One strategy of doing that is d in 5.2.2.

Case 1) discribes a non 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 [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 APPEND [APPENDUID 1022843275 77712] completed
```

The UID of the appended message is the second parameter of APPENDUID respon

5.2.2. Optimizing "move" 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.

5.2.2.1. 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 UIDPLUS extension support in the server can help:

```
A001 UID COPY 567,414 "Interesting Messages"
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 "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 make special provisions to use this technique when it knows that a copy/delete sequence is the result of a "move" command.

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 >>

5.2.2.2. 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 (RFC822 INTERNALDATE FLAGS)
S: * 27 FETCH (UID 123 INTERNALDATE "31-May-2002 05:26:59 -0600"
   FLAGS (\Seen $MDNSent) RFC822
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 already in the client's cache. Also, the client SHOULD preserve delivery date in the local cache.

5.2.2.3. 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 APPEND [APPENDUID 1022843275 77712] completed
```

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

5.2.2.4. Moving a message between two mailbox on two different servers

Moving a message between two mailbox on two different servers is a combinati
of the 5.2.2.2 followed by 5.2.2.3.

5.2.3. 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. 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
```

5.2.4. 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 \Deleted fla i.e. even those messages that were marked as \Deleted on the server while th was offline. Doing so will lead the user to an unpleasant surprise.

Fortunately [UIDPLUS] extension can help in this case as well. The extension 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 server erases only messages with \Deleted flag listed in the UID list. Thus, a message not listed in the UID set will not be expunged even if it has \Del 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 [UIDPLUS] extension the following sequence of command can
used as an approximation. Note, that when this sequence is performed, there
possibility that another client marks additional messages as deleted and the
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 on the 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
```

5.2.5. 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 5.2.4 explains why EXPUNGE must 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 implicitely close the mailbox and prevent the silent expunge:

1). UNSELECT - This is an undocumented command that works as CLOSE, but doesn't cause the silent EXPUNGE. This command is supported by the serve if it reports UNSELECT in its CAPABILITY list.
2). EXAMINE <mailbox> - reselect the same mailbox in read-only mode.
3). SELECT <another_mailbox> - SELECT causes implicit CLOSE without EXPUNGE.
4). 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.
5). SELECT <non_existing_mailbox> - if the client knows a mailbox that doesn exist or can't be selected, it MAY SELECT it.

5.3. 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.

5.3.1. 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 issue the following two commands:

```
tag1 UID FETCH <lastseen+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 latter 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)
S: ...
```

The second FETCH command will result in nil 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: 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 the ascendi order.

5.3.2. Searching for "interesting" messages.

This step is either performed entirely on the client (from the information r in step 5.3.1), after performing additional searches on the server or 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 criteria is the age of a message. For example, the cli 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 5.3.1. If the client did, it can perform the search itself.

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 BODYSTRUCTURE FETCH response data item (i.e. if BODYSTRUCTURE was included i <descriptors> on step 5.3.1).

5.3.3. Populating cache with "interesting" messages.

Once the client found out which messages are "interesting", the client can start issuing appropriate FETCH commands for "interesting" messages or bodyparts 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 achive 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).

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: ...
```

5.3.4. User initiated synchronization

After the client finished the main synchronization that was described in 5.3.1-5.3.3 the user may optionally request additional synchronization steps while the client is still online. This is not any different from the process described in 5.3.3.

<< Example: Fetch all flagged messages. >>

<< Example: Fetch all messages selected in UI. >>

5.4. 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 (eg, "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 that is necessary for a mailbox of this kind is fetching the descriptor information that the human will use to identify messages that should be explicitly fetched.

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

5.5. 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>).

5.6. 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 <lastseen+1>:* (FLAGS RFC822.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 explicitely told otherwise. A good behaving client should use instead something like the following:

1) Issue "tag1 UID FETCH <lastseen+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> (RFC822.PEEK)"

or

1) Issue "tag1 UID FETCH <lastseen+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> (RFC822.PEEK)"

6. Implementation considerations

Below are listed some golden rules that should be considered when implementi
a good disconnected IMAP client. They will help to write a disconnected clie
that works correctly, performs synchronization as quickly as possible
(and thus can save the user money) as well as minimizes the load of the serv

1) Don't reorder operations during synchronization.

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

Example 1: 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: A005 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.

<< Describe one case when it is safe to reorder: the disconnected client
doesn't allow to perform DELETE and RENAME while offline and EXPUNGE
is never used (UID EXPUNGE is used instead or its emulation as described
in 5.2.3 >>

2) 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.

3) Minimize number of round trips.

Round trips kill performance, especially on links with high latency.
This section gives some advices how to minimize number of round trips.

4) 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 be aware of the non-trivial cost associated with establishing TCP connection and performing authentication. The disconnected client MUST NOT use a connection per each mailbox. In most cases it is sufficient to have two connections.

Any mailbox synchronization MUST start with checking of the UIDVALIDITY as described in section 5.1 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 6.3.10 of [IMAP4], this MUST NOT be used on the selected mailbox.

Below listed some quality of implementation issues for disconnected clients:

1) Don't loose information.

<< To be added later >>

6.1. Optimizations

<< To be added later >>

6.2. Error recovery during playback

<< To be added later >>

7. IMAP extensions that may help

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

1) The use of [UIDPLUS] was discussed in 5.1, 5.2.1, 5.2.2.1 and 5.2.4.

<< Describe how MULTIAPPEND and LITERAL+ can be used >>

<< Describe how CONDSTORE can be used >>

8. Security Considerations

Security considerations are not discussed in this memo.

9. 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 2060, University of Washington, December 1996.

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

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

<< Add missing references: MULTIAPPEND, CONDSTORE, ACL? >>

10. Aknowledgement

This document is a revision of the draft-ietf-imap-disc-01.txt written by Rob Austein <sra@epilogue.com> in November 1994.

The editor also appreciate comments posted by Mark Crispin to the IMAP maili list and the comments and corrections received from Cyrus Daboo.

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.

11. Editor's Address

Alexey Melnikov
mailto: mel@messagingdirect.com

ACI WorldWide/MessagingDirect
22 The Quadrant, Richmond,
Surrey, United Kingdom, TW9 1BP

12. Full Copyright Statement