

IMAP4 Implementation Recommendations

Status of this Document

This document is an Internet-Draft and is in full conformance with all provisions of [Section 10 of RFC2026](#). Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <http://www.ietf.org/ietf/lid-abstracts.txt>

The list of Internet-Draft Shadow Directories can be accessed at <http://www.ietf.org/shadow.html>.

This document provides information for the Internet community. This document does not specify an Internet standard of any kind. Distribution of this document is unlimited. A revised version of this draft document will be submitted to the RFC editor. Discussion and suggestions for improvement are requested; discussion should be held on the IMAP mailing list, "imap@u.washington.edu" (subscription requests go to "imap-request@u.washington.edu").

1. Abstract

The IMAP4 specification [[RFC-2060](#)] describes a rich protocol for use in building clients and servers for storage, retrieval, and manipulation of electronic mail. Because the protocol is so rich and has so many implementation choices, there are often trade-offs that must be made and issues that must be considered when designing such clients and servers. This document attempts to outline these issues and to make recommendations in order to make the end products as interoperable as possible.

[2.](#) Conventions used in this document

In examples, "C:" indicates lines sent by a client that is connected to a server. "S:" indicates lines sent by the server to the client.

The words "must", "must not", "should", "should not", and "may" are used with specific meaning in this document; since their meaning is somewhat different from that specified in [RFC 2119](#), we do not put them in all caps here. Their meaning is as follows:

must -- This word means that the action described is necessary to ensure interoperability. The recommendation should not be ignored.

must not -- This phrase means that the action described will be almost certain to hurt interoperability. The recommendation should not be ignored.

should -- This word means that the action described is strongly recommended and will enhance interoperability or usability. The recommendation should not be ignored without careful consideration.

should not -- This phrase means that the action described is strongly recommended against, and might hurt interoperability or usability. The recommendation should not be ignored without careful consideration.

may -- This word means that the action described is an acceptable implementation choice. No specific recommendation is implied; this word is used to point out a choice that might not be obvious, or to let implementors know what choices have been made by existing implementations.

[3.](#) Interoperability Issues and Recommendations

[3.1.](#) Accessibility

This section describes the issues related to access to servers and server resources. Concerns here include data sharing and maintenance of client/server connections.

[3.1.1.](#) Multiple Accesses of the Same Mailbox

One strong point of IMAP4 is that, unlike POP3, it allows for multiple simultaneous access to a single mailbox. A user can, thus, read mail from a client at home while the client in the office is still connected; or the help desk staff can all work out of the same inbox, all seeing the same pool of questions. An important point about this capability, though is that NO SERVER IS GUARANTEED TO

SUPPORT THIS. If you are selecting an IMAP server and this facility is important to you, be sure that the server you choose to install, in the configuration you choose to use, supports it.

If you are designing a client, you must not assume that you can access the same mailbox more than once at a time. That means

1. you must handle gracefully the failure of a SELECT command if the server refuses the second SELECT,
2. you must handle reasonably the severing of your connection (see "Severed Connections", below) if the server chooses to allow the second SELECT by forcing the first off,
3. you must avoid making multiple connections to the same mailbox in your own client (for load balancing or other such reasons), and
4. you must avoid using the STATUS command on a mailbox that you have selected (with some server implementations the STATUS command has the same problems with multiple access as do the SELECT and EXAMINE commands).

A further note about STATUS: The STATUS command is sometimes used to check a non-selected mailbox for new mail. This mechanism must not be used to check for new mail in the selected mailbox; [section 5.2 of \[RFC-2060\]](#) specifically forbids this in its last paragraph. Further, since STATUS takes a mailbox name it is an independent operation, not operating on the selected mailbox. Because of this, the information it returns is not necessarily in synchronization with the selected mailbox state.

[3.1.2. Severed Connections](#)

The client/server connection may be severed for one of three reasons: the client severs the connection, the server severs the connection, or the connection is severed by outside forces beyond the control of

the client and the server (a telephone line drops, for example). Clients and servers must both deal with these situations.

When the client wants to sever a connection, it's usually because it has finished the work it needed to do on that connection. The client should send a LOGOUT command, wait for the tagged response, and then close the socket. But note that, while this is what's intended in the protocol design, there isn't universal agreement here. Some contend that sending the LOGOUT and waiting for the two responses (untagged BYE and tagged OK) is wasteful and unnecessary, and that the client can simply close the socket. The server should interpret the closed socket as a log out by the client. The counterargument is that it's useful from the standpoint of cleanup, problem determination, and the like, to have an explicit client log out, because otherwise there is no way for the server to tell the

B. Leiba

[Page 3]

Internet DRAFT

Implementation Recommendations

July 1999

difference between "closed socket because of log out" and "closed socket because communication was disrupted". If there is a client/server interaction problem, a client which routinely terminates a session by breaking the connection without a LOGOUT will make it much more difficult to determine the problem.

Because of this disagreement, server designers must be aware that some clients might close the socket without sending a LOGOUT. In any case, whether or not a LOGOUT was sent, the server should not implicitly expunge any messages from the selected mailbox. If a client wants the server to do so, it must send a CLOSE or EXPUNGE command explicitly.

When the server wants to sever a connection it's usually due to an inactivity timeout or is because a situation has arisen that has changed the state of the mail store in a way that the server can not communicate to the client. The server should send an untagged BYE response to the client and then close the socket. Sending an untagged BYE response before severing allows the server to send a human-readable explanation of the problem to the client, which the client may then log, display to the user, or both (see [section 7.1.5 of \[RFC-2060\]](#)).

Regarding inactivity timeouts, there is some controversy. Unlike POP, for which the design is for a client to connect, retrieve mail,

and log out, IMAP's design encourages long-lived (and mostly inactive) client/server sessions. As the number of users grows, this can use up a lot of server resources, especially with clients that are designed to maintain sessions for mailboxes that the user has finished accessing. To alleviate this, a server may implement an inactivity timeout, unilaterally closing a session (after first sending an untagged BYE, as noted above). Some server operators have reported dramatic improvements in server performance after doing this. As specified in [\[RFC-2060\]](#), if such a timeout is done it must not be until at least 30 minutes of inactivity. The reason for this specification is to prevent clients from sending commands (such as NOOP) to the server at frequent intervals simply to avert a too-early timeout. If the client knows that the server may not time out the session for at least 30 minutes, then the client need not poll at intervals more frequent than, say, 25 minutes.

[3.2.](#) Scaling

IMAP4 has many features that allow for scalability, as mail stores become larger and more numerous. Large numbers of users, mailboxes, and messages, and very large messages require thought to handle efficiently. This document will not address the administrative issues involved in large numbers of users, but we will look at the

other items.

[3.2.1.](#) Flood Control

There are three situations when a client can make a request that will result in a very large response - too large for the client reasonably to deal with: there are a great many mailboxes available, there are a great many messages in the selected mailbox, or there is a very large message part. The danger here is that the end user will be stuck waiting while the server sends (and the client processes) an enormous response. In all of these cases there are things a client can do to reduce that danger.

There is also the case where a client can flood a server, by sending an arbitrarily long command. We'll discuss that issue, too, in this

section.

[3.2.1.1.](#) Listing Mailboxes

Some servers present Usenet newsgroups to IMAP users. Newsgroups, and other such hierarchical mailbox structures, can be very numerous but may have only a few entries at the top level of hierarchy. Also, some servers are built against mail stores that can, unbeknownst to the server, have circular hierarchies - that is, it's possible for "a/b/c/d" to resolve to the same file structure as "a", which would then mean that "a/b/c/d/b" is the same as "a/b", and the hierarchy will never end. The LIST response in this case will be unlimited.

Clients that will have trouble with this are those that use

```
C: 001 LIST "" *
```

to determine the mailbox list. Because of this, clients should not use an unqualified "*" that way in the LIST command. A safer approach is to list each level of hierarchy individually, allowing the user to traverse the tree one limb at a time, thus:

```
C: 001 LIST "" %  
S: * LIST () "/" Banana  
S: * LIST ...etc...  
S: 001 OK done
```

and then

```
C: 002 LIST "" Banana/%  
S: * LIST () "/" Banana/Apple  
S: * LIST ...etc...  
S: 002 OK done
```

Using this technique the client's user interface can give the user full flexibility without choking on the voluminous reply to "LIST *".

Of course, it is still possible that the reply to

```
C: 005 LIST "" alt.fan.celebrity.%
```

may be thousands of entries long, and there is, unfortunately, nothing the client can do to protect itself from that. This has not yet been a notable problem.

Servers that may export circular hierarchies (any server that

directly presents a UNIX file system, for instance) should limit the hierarchy depth to prevent unlimited LIST responses. A suggested depth limit is 20 hierarchy levels.

3.2.1.2. Fetching the List of Messages

When a client selects a mailbox, it is given a count, in the untagged EXISTS response, of the messages in the mailbox. This number can be very large. In such a case it might be unwise to use

```
C: 004 FETCH 1:* ALL
```

to populate the user's view of the mailbox. One good method to avoid problems with this is to batch the requests, thus:

```
C: 004 FETCH 1:50 ALL
```

```
S: * 1 FETCH ...etc...
```

```
S: 004 OK done
```

```
C: 005 FETCH 51:100 ALL
```

```
S: * 51 FETCH ...etc...
```

```
S: 005 OK done
```

```
C: 006 FETCH 101:150 ALL
```

```
...etc...
```

Using this method, another command, such as "FETCH 6 BODY[1]" can be inserted as necessary, and the client will not have its access to the server blocked by a storm of FETCH replies. (Such a method could be reversed to fetch the LAST 50 messages first, then the 50 prior to that, and so on.)

As a smart extension of this, a well designed client, prepared for very large mailboxes, will not automatically fetch data for all messages AT ALL. Rather, the client will populate the user's view only as the user sees it, possibly pre-fetching selected information, and only fetching other information as the user scrolls to it. For example, to select only those messages beginning with the first unseen one:

```

C: 003 SELECT INBOX
S: * 10000 EXISTS
S: * 80 RECENT
S: * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
S: * OK [UIDVALIDITY 824708485] UID validity status
S: * OK [UNSEEN 9921] First unseen message
S: 003 OK [READ-WRITE] SELECT completed
C: 004 FETCH 9921:* ALL
... etc...

```

If the server does not return an OK [UNSEEN] response, the client may use SEARCH UNSEEN to obtain that value.

This mechanism is good as a default presentation method, but only works well if the default message order is acceptable. A client may want to present various sort orders to the user (by subject, by date sent, by sender, and so on) and in that case (lacking a SORT extension on the server side) the client WILL have to retrieve all message descriptors. A client that provides this service should not do it by default and should inform the user of the costs of choosing this option for large mailboxes.

[3.2.1.3.](#) Fetching a Large Body Part

The issue here is similar to the one for a list of messages. In the BODYSTRUCTURE response the client knows the size, in bytes, of the body part it plans to fetch. Suppose this is a 70 MB video clip. The client can use partial fetches to retrieve the body part in pieces, avoiding the problem of an uninterruptible 70 MB literal coming back from the server:

```

C: 022 FETCH 3 BODY[1]<0.20000>
S: * 3 FETCH (FLAGS(\Seen) BODY[1]<0> {20000}
S: ...data...)
S: 022 OK done
C: 023 FETCH 3 BODY[1]<20001.20000>
S: * 3 FETCH (BODY[1]<20001> {20000}
S: ...data...)
S: 023 OK done
C: 024 FETCH 3 BODY[1]<40001.20000>
...etc...

```

[3.2.1.4.](#) BODYSTRUCTURE vs. Entire Messages

Because FETCH BODYSTRUCTURE is necessary in order to determine the number of body parts, and, thus, whether a message has "attachments", clients often use FETCH FULL as their normal method of populating the

Internet DRAFT

Implementation Recommendations

July 1999

user's view of a mailbox. The benefit is that the client can display a paperclip icon or some such indication along with the normal message summary. However, this comes at a significant cost with some server configurations. The parsing needed to generate the FETCH BODYSTRUCTURE response may be time-consuming compared with that needed for FETCH ENVELOPE. The client developer should consider this issue when deciding whether the ability to add a paperclip icon is worth the tradeoff in performance, especially with large mailboxes.

Some clients, rather than using FETCH BODYSTRUCTURE, use FETCH BODY[] (or the equivalent FETCH [RFC822](#)) to retrieve the entire message. They then do the MIME parsing in the client. This may give the client slightly more flexibility in some areas (access, for instance, to header fields that aren't returned in the BODYSTRUCTURE and ENVELOPE responses), but it can cause severe performance problems by forcing the transfer of all body parts when the user might only want to see some of them - a user logged on by modem and reading a small text message with a large ZIP file attached may prefer to read the text only and save the ZIP file for later. Therefore, a client should not normally retrieve entire messages and should retrieve message body parts selectively.

[3.2.1.5](#). Long Command Lines

A client can wind up building a very long command line in an effort to try to be efficient about requesting information from a server. This can typically happen when a client builds a message set from selected messages and doesn't recognise that contiguous blocks of messages may be group in a range. Suppose a user selects all 10,000 messages in a large mailbox and then unselects message 287. The client could build that message set as "1:286,288:10000", but a client that doesn't handle that might try to enumerate each message individually and build "1,2,3,4, [and so on] ,9999,10000". Adding that to the fetch command results in a command line that's almost 49,000 octets long, and, clearly, one can construct a command line that's even longer.

A client should limit the length of the command lines it generates to approximately 1000 octets (including all quoted strings but not

including literals). If the client is unable to group things into ranges so that the command line is within that length, it should split the request into multiple commands. The client should use literals instead of long quoted strings, in order to keep the command length down.

For its part, a server should allow for a command line of at least 8000 octets. This provides plenty of leeway for accepting reasonable length commands from clients. The server should send a BAD response

to a command that does not end within the server's maximum accepted command length.

[3.2.2. Subscriptions](#)

The client isn't the only entity that can get flooded: the end user, too, may need some flood control. The IMAP4 protocol provides such control in the form of subscriptions. Most servers support the SUBSCRIBE, UNSUBSCRIBE, and LSUB commands, and many users choose to narrow down a large list of available mailboxes by subscribing to the ones that they usually want to see. Clients, with this in mind, should give the user a way to see only subscribed mailboxes. A client that never uses the LSUB command takes a significant usability feature away from the user. Of course, the client would not want to hide the LIST command completely; the user needs to have a way to choose between LIST and LSUB. The usual way to do this is to provide a setting like "show which mailboxes?: [] all [] subscribed only".

[3.2.3. Searching](#)

IMAP SEARCH commands can become particularly troublesome (that is, slow) on mailboxes containing a large number of messages. So let's put a few things in perspective in that regard.

The flag searches should be fast. The flag searches (ALL, [UN]SEEN, [UN]ANSWERED, [UN]DELETED, [UN]DRAFT, [UN]FLAGGED, NEW, OLD, RECENT) are known to be used by clients for the client's own use (for instance, some clients use "SEARCH UNSEEN" to find unseen mail and "SEARCH DELETED" to warn the user before expunging messages).

Other searches, particularly the text searches (HEADER, TEXT, BODY) are initiated by the user, rather than by the client itself, and somewhat slower performance can be tolerated, since the user is aware that the search is being done (and is probably aware that it might be time-consuming). A smart server might use dynamic indexing to speed commonly used text searches.

The client may allow other commands to be sent to the server while a SEARCH is in progress, but at the time of this writing there is little or no server support for parallel processing of multiple commands in the same session (and see "Multiple Accesses of the Same Mailbox" above for a description of the dangers of trying to work around this by doing your SEARCH in another session).

Another word about text searches: some servers, built on database back-ends with indexed search capabilities, may return search results that do not match the IMAP spec's "case-insensitive substring"

requirements. While these servers are in violation of the protocol, there is little harm in the violation as long as the search results are used only in response to a user's request. Still, developers of such servers should be aware that they ARE violating the protocol, should think carefully about that behaviour, and must be certain that their servers respond accurately to the flag searches for the reasons outlined above.

In addition, servers should support CHARSET UTF-8 [UTF-8] in searches.

[3.3](#) Avoiding Invalid Requests

IMAP4 provides ways for a server to tell a client in advance what is and isn't permitted in some circumstances. Clients should use these features to avoid sending requests that a well designed client would know to be invalid. This section explains this in more detail.

[3.3.1](#). The CAPABILITY Command

All IMAP4 clients should use the CAPABILITY command to determine what version of IMAP and what optional features a server supports. The client should not send IMAP4rev1 commands and arguments to a server that does not advertize IMAP4rev1 in its CAPABILITY response. Similarly, the client should not send IMAP4 commands that no longer exist in IMAP4rev1 to a server that does not advertize IMAP4 in its CAPABILITY response. An IMAP4rev1 server is NOT required to support obsolete IMAP4 or IMAP2bis commands (though some do; do not let this fact lull you into thinking that it's valid to send such commands to an IMAP4rev1 server).

A client should not send commands to probe for the existence of certain extensions. All standard and standards-track extensions include CAPABILITY tokens indicating their presense. All private and experimental extensions should do the same, and clients that take advantage of them should use the CAPABILITY response to determine whether they may be used or not.

[3.3.2.](#) Don't Do What the Server Says You Can't

In many cases, the server, in response to a command, will tell the client something about what can and can't be done with a particular mailbox. The client should pay attention to this information and should not try to do things that it's been told it can't do.

Examples:

- * Do not try to SELECT a mailbox that has the \Noselect flag set.

- * Do not try to CREATE a sub-mailbox in a mailbox that has the \Noinferiors flag set.
- * Do not respond to a failing COPY or APPEND command by trying to CREATE the target mailbox if the server does not respond with a [TRYCREATE] response code.
- * Do not try to expunge a mailbox that has been selected with the [READ-ONLY] response code.

[3.4.](#) Miscellaneous Protocol Considerations

We describe here a number of important protocol-related issues, the misunderstanding of which has caused significant interoperability

problems in IMAP4 implementations. One general item is that every implementer should be certain to take note of and to understand [section 2.2.2](#) and the preamble to [section 7](#) of the IMAP4rev1 spec [RFC-2060].

[3.4.1.](#) Well Formed Protocol

We cannot stress enough the importance of adhering strictly to the protocol grammar. The specification of the protocol is quite rigid; do not assume that you can insert blank space for "readability" if none is called for. Keep in mind that there are parsers out there that will crash if there are protocol errors. There are clients that will report every parser burp to the user. And in any case, information that cannot be parsed is information that is lost. Be careful in your protocol generation. And see "A Word About Testing", below.

In particular, note that the string in the INTERNALDATE response is NOT an [RFC-822](#) date string - that is, it is not in the same format as the first string in the ENVELOPE response. Since most clients will, in fact, accept an [RFC-822](#) date string in the INTERNALDATE response, it's easy to miss this in your interoperability testing. But it will cause a problem with some client, so be sure to generate the correct string for this field.

[3.4.2.](#) Special Characters

Certain characters, currently the double-quote and the backslash, may not be sent as-is inside a quoted string. These characters must be preceded by the escape character if they are in a quoted string, or else the string must be sent as a literal. Both clients and servers must handle this, both on output (they must send these characters properly) and on input (they must be able to receive escaped

characters in quoted strings). Example:

```
C: 001 LIST "" %
S: * LIST () "" INBOX
```

```
S: * LIST () "\\\" TEST
S: * LIST () "\\\" {12}
S: "My" mailbox
S: 001 OK done
C: 002 LIST "" "\"My\" mailbox\\%"
S: * LIST () "\\\" {17}
S: "My" mailbox\Junk
S: 002 OK done
```

Note that in the example the server sent the hierarchy delimiter as an escaped character in the quoted string and sent the mailbox name containing imbedded double-quotes as a literal. The client used only quoted strings, escaping both the backslash and the double-quote characters.

The CR and LF characters may be sent ONLY in literals; they are not allowed, even if escaped, inside quoted strings.

And while we're talking about special characters: the IMAP spec, in the section titled "Mailbox International Naming Convention", describes how to encode mailbox names in modified UTF-7 [[UTF-7](#) and [RFC-2060](#)]. Implementations must adhere to this in order to be interoperable in the international market, and servers should validate mailbox names sent by client and reject names that do not conform.

As to special characters in userids and passwords: clients must not restrict what a user may type in for a userid or a password. The formal grammar specifies that these are "astrings", and an astring can be a literal. A literal, in turn can contain any 8-bit character, and clients must allow users to enter all 8-bit characters here, and must pass them, unchanged, to the server (being careful to send them as literals when necessary). In particular, some server configurations use "@" in user names, and some clients do not allow that character to be entered; this creates a severe interoperability problem.

[3.4.3](#). [UIDs](#) and [UIDVALIDITY](#)

Servers that support existing back-end mail stores often have no good place to save [UIDs](#) for messages. Often the existing mail store will not have the concept of [UIDs](#) in the sense that IMAP has: strictly increasing, never re-issued, 32-bit integers. Some servers solve this by storing the [UIDs](#) in a place that's accessible to end users,

allowing for the possibility that the users will delete them. Others solve it by re-assigning UIDs every time a mailbox is selected.

The server should maintain UIDs permanently for all messages if it can. If that's not possible, the server must change the UIDVALIDITY value for the mailbox whenever any of the UIDs may have become invalid. Clients must recognize that the UIDVALIDITY has changed and must respond to that condition by throwing away any information that they have saved about UIDs in that mailbox. There have been many problems in this area when clients have failed to do this; in the worst case it will result in loss of mail when a client deletes the wrong piece of mail by using a stale UID.

It seems to be a common misunderstanding that "the UIDVALIDITY and the UID, taken together, form a 64-bit identifier that uniquely identifies a message on a server". This is absolutely NOT TRUE. There is no assurance that the UIDVALIDITY values of two mailboxes be different, so the UIDVALIDITY in no way identifies a mailbox. The ONLY purpose of UIDVALIDITY is, as its name indicates, to give the client a way to check the validity of the UIDs it has cached. While it is a valid implementation choice to put these values together to make a 64-bit identifier for the message, the important concept here is that UIDs are not unique between mailboxes; they are only unique WITHIN a given mailbox.

Some server implementations have attempted to make UIDs unique across the entire server. This is inadvisable, in that it limits the life of UIDs unnecessarily. The UID is a 32-bit number and will run out in reasonably finite time if it's global across the server. If you assign UIDs sequentially in one mailbox, you will not have to start re-using them until you have had, at one time or another, 2^{32} different messages in that mailbox. In the global case, you will have to reuse them once you have had, at one time or another, 2^{32} different messages in the entire mail store. Suppose your server has around 8000 users registered (2^{13}). That gives an average of 2^{19} UIDs per user. Suppose each user gets 32 messages (2^5) per day. That gives you 2^{14} days ($16000+$ days = about 45 years) before you run out. That may seem like enough, but multiply the usage just a little (a lot of spam, a lot of mailing list subscriptions, more users) and you limit yourself too much.

What's worse is that if you have to wrap the UIDs, and, thus, you have to change UIDVALIDITY and invalidate the UIDs in the mailbox, you have to do it for EVERY mailbox in the system, since they all share the same UID pool. If you assign UIDs per mailbox and you have a problem, you only have to kill the UIDs for that one mailbox.

Under extreme circumstances (and this is extreme, indeed), the server may have to invalidate UIDs while a mailbox is in use by a client -

that is, the UIDs that the client knows about in its active mailbox are no longer valid. In that case, the server must immediately change the UIDVALIDITY and must communicate this to the client. The server may do this by sending an unsolicited UIDVALIDITY message, in the same form as in response to the SELECT command. Clients must be prepared to handle such a message and the possibly coincident failure of the command in process. For example:

```
C: 032 UID STORE 382 +Flags.silent \Deleted
S: * OK [UIDVALIDITY 12345] New UIDVALIDITY value!
S: 032 NO UID command rejected because UIDVALIDITY changed!
C: ...invalidates local information and re-fetches...
C: 033 FETCH 1:* UID
...etc...
```

At the time of the writing of this document, the only server known to do this does so only under the following condition: the client selects INBOX, but there is not yet a physical INBOX file created. Nonetheless, the SELECT succeeds, exporting an empty INBOX with a temporary UIDVALIDITY of 1. While the INBOX remains selected, mail is delivered to the user, which creates the real INBOX file and assigns a permanent UIDVALIDITY (that is likely not to be 1). The server reports the change of UIDVALIDITY, but as there were no messages before, so no UIDs have actually changed, all the client must do is accept the change in UIDVALIDITY.

Alternatively, a server may force the client to re-select the mailbox, at which time it will obtain a new UIDVALIDITY value. To do this, the server closes this client session (see "Severed Connections" above) and the client then reconnects and gets back in synch. Clients must be prepared for either of these behaviours.

We do not know of, nor do we anticipate the future existence of, a server that changes UIDVALIDITY while there are existing messages, but clients must be prepared to handle this eventuality.

[3.4.4.](#) FETCH Responses

When a client asks for certain information in a FETCH command, the server may return the requested information in any order, not necessarily in the order that it was requested. Further, the server may return the information in separate FETCH responses and may also return information that was not explicitly requested (to reflect to the client changes in the state of the subject message). Some examples:

```
C: 001 FETCH 1 UID FLAGS INTERNALDATE
S: * 5 FETCH (FLAGS (\Deleted))
S: * 1 FETCH (FLAGS (\Seen) INTERNALDATE "... " UID 345)
S: 001 OK done
```

(In this case, the responses are in a different order. Also, the server returned a flag update for message 5, which wasn't part of the client's request.)

```
C: 002 FETCH 2 UID FLAGS INTERNALDATE
S: * 2 FETCH (INTERNALDATE "... ")
S: * 2 FETCH (UID 399)
S: * 2 FETCH (FLAGS ())
S: 002 OK done
```

(In this case, the responses are in a different order and were returned in separate responses.)

```
C: 003 FETCH 2 BODY[1]
S: * 2 FETCH (FLAGS (\Seen) BODY[1] {14}
S: Hello world!
S: )
S: 003 OK done
```

(In this case, the FLAGS response was added by the server, since fetching the body part caused the server to set the \Seen flag.)

Because of this characteristic a client must be ready to receive any FETCH response at any time and should use that information to update its local information about the message to which the FETCH response refers. A client must not assume that any FETCH responses will come in any particular order, or even that any will come at all. If after

receiving the tagged response for a FETCH command the client finds that it did not get all of the information requested, the client should send a NOOP command to the server to ensure that the server has an opportunity to send any pending EXPUNGE responses to the client (see [[RFC-2180](#)]).

[3.4.5.](#) [RFC822](#).SIZE

Some back-end mail stores keep the mail in a canonical form, rather than retaining the original MIME format of the messages. This means that the server must reassemble the message to produce a MIME stream when a client does a fetch such as [RFC822](#) or BODY[], requesting the entire message. It also may mean that the server has no convenient way to know the [RFC822](#).SIZE of the message. Often, such a server will actually have to build the MIME stream to compute the size, only to throw the stream away and report the size to the client.

When this is the case, some servers have chosen to estimate the size, rather than to compute it precisely. Such an estimate allows the

client to display an approximate size to the user and to use the estimate in flood control considerations (q.v.), but requires that the client not use the size for things such as allocation of buffers, because those buffers might then be too small to hold the actual MIME stream. Instead, a client should use the size that's returned in the literal when you fetch the data.

The protocol requires that the [RFC822](#).SIZE value returned by the server be EXACT. Estimating the size is a protocol violation, and server designers must be aware that, despite the performance savings they might realize in using an estimate, this practice will cause some clients to fail in various ways. If possible, the server should compute the [RFC822](#).SIZE for a particular message once, and then save it for later retrieval. If that's not possible, the server must compute the value exactly every time. Incorrect estimates do cause severe interoperability problems with some clients.

[3.4.6.](#) Expunged Messages

If the server allows multiple connections to the same mailbox, it is often possible for messages to be expunged in one client unbeknownst to another client. Since the server is not allowed to tell the client about these expunged messages in response to a FETCH command, the server may have to deal with the issue of how to return information about an expunged message. There was extensive discussion about this issue, and the results of that discussion are summarized in [\[RFC-2180\]](#). See that reference for a detailed explanation and for recommendations.

[3.4.7.](#) The Namespace Issue

Namespaces are a very muddy area in IMAP4 implementation right now (see [\[NAMESPACE\]](#) for a proposal to clear the water a bit). Until the issue is resolved, the important thing for client developers to understand is that some servers provide access through IMAP to more than just the user's personal mailboxes, and, in fact, the user's personal mailboxes may be "hidden" somewhere in the user's default hierarchy. The client, therefore, should provide a setting wherein the user can specify a prefix to be used when accessing mailboxes. If the user's mailboxes are all in "~/mail/", for instance, then the user can put that string in the prefix. The client would then put the prefix in front of any name pattern in the LIST and LSUB commands:

```
C: 001 LIST "" ~/mail/%
```

(See also "Reference Names in the LIST Command" below.)

[3.4.8.](#) Creating Special-Use Mailboxes

It may seem at first that this is part of the namespace issue; it is not, and is only indirectly related to it. A number of clients like to create special-use mailboxes with particular names. Most commonly, clients with a "trash folder" model of message deletion want to create a mailbox with the name "Trash" or "Deleted". Some clients want to create a "Drafts" mailbox, an "Outbox" mailbox, or a "Sent Mail" mailbox. And so on. There are two major interoperability problems with this practice:

1. different clients may use different names for mailboxes with

similar functions (such as "Trash" and "Deleted"), or may manage the same mailboxes in different ways, causing problems if a user switches between clients and

2. there is no guarantee that the server will allow the creation of the desired mailbox.

The client developer is, therefore, well advised to consider carefully the creation of any special-use mailboxes on the server, and, further, the client must not require such mailbox creation - that is, if you do decide to do this, you must handle gracefully the failure of the CREATE command and behave reasonably when your special-use mailboxes do not exist and can not be created.

In addition, the client developer should provide a convenient way for the user to select the names for any special-use mailboxes, allowing the user to make these names the same in all clients used and to put them where the user wants them.

[3.4.9.](#) Reference Names in the LIST Command

Many implementers of both clients and servers are confused by the "reference name" on the LIST command. The reference name is intended to be used in much the way a "cd" (change directory) command is used on Unix, PC DOS, Windows, and OS/2 systems. That is, the mailbox name is interpreted in much the same way as a file of that name would be found if one had done a "cd" command into the directory specified by the reference name. For example, in Unix we have the following:

```
> cd /u/jones/junk
> vi banana           [file is "/u/jones/junk/banana"]
> vi stuff/banana     [file is "/u/jones/junk/stuff/banana"]
> vi /etc/hosts       [file is "/etc/hosts"]
```

In the past, there have been several interoperability problems with this. First, while some IMAP servers are built on Unix or PC file systems, many others are not, and the file system semantics do not make sense in those configurations. Second, while some IMAP servers

expose the underlying file system to the clients, others allow access only to the user's personal mailboxes, or to some other limited set

of files, making such file-system-like semantics less meaningful. Third, because the IMAP spec leaves the interpretation of the reference name as "implementation-dependent", in the past the various server implementations handled it in vastly differing ways.

The following recommendations are the result of significant operational experience, and are intended to maximize interoperability.

Server implementations must implement the reference argument in a way that matches the intended "change directory" operation as closely as possible. As a minimum implementation, the reference argument may be prepended to the mailbox name (while suppressing double delimiters; see the next paragraph). Even servers that do not provide a way to break out of the current hierarchy (see "breakout facility" below) must provide a reasonable implementation of the reference argument, as described here, so that they will interoperate with clients that use it.

Server implementations that prepend the reference argument to the mailbox name should insert a hierarchy delimiter between them, and must not insert a second if one is already present:

```
C: A001 LIST ABC DEF
S: * LIST () "/" ABC/DEF    <=== should do this
S: A001 OK done

C: A002 LIST ABC/ /DEF
S: * LIST () "/" ABC//DEF   <=== must not do this
S: A002 OK done
```

On clients, the reference argument is chiefly used to implement a "breakout facility", wherein the user may directly access a mailbox outside the "current directory" hierarchy. Client implementations should have an operational mode that does not use the reference argument. This is to interoperate with older servers that did not implement the reference argument properly. While it's a good idea to give the user access to a breakout facility, clients that do not intend to do so should not use the reference argument at all.

Client implementations should always place a trailing hierarchy delimiter on the reference argument. This is because some servers prepend the reference argument to the mailbox name without inserting a hierarchy delimiter, while others do insert a hierarchy delimiter if one is not already present. A client that puts the delimiter in will work with both varieties of server.

Internet DRAFT

Implementation Recommendations

July 1999

Client implementations that implement a breakout facility should allow the user to choose whether or not to use a leading hierarchy delimiter on the mailbox argument. This is because the handling of a leading mailbox hierarchy delimiter also varies from server to server, and even between different mailstores on the same server. In some cases, a leading hierarchy delimiter means "discard the reference argument" (implementing the intended breakout facility), thus:

```
C: A001 LIST ABC/ /DEF
S: * LIST () "/" /DEF
S: A001 OK done
```

In other cases, however, the two are catenated and the extra hierarchy delimiter is discarded, thus:

```
C: A001 LIST ABC/ /DEF
S: * LIST () "/" ABC/DEF
S: A001 OK done
```

Client implementations must not assume that the server supports a breakout facility, but may provide a way for the user to use one if it is available. Any breakout facility should be exported to the user interface. Note that there may be other "breakout" characters besides the hierarchy delimiter (for instance, UNIX filesystem servers are likely to use a leading "~" as well), and that their interpretation is server-dependent.

[3.4.12.](#) Mailbox Hierarchy Delimiters

The server's selection of what to use as a mailbox hierarchy delimiter is a difficult one, involving several issues: What characters do users expect to see? What characters can they enter for a hierarchy delimiter if it is desired (or required) that the user enter it? What character can be used for the hierarchy delimiter, noting that the chosen character can not otherwise be used in the mailbox name?

Because some interfaces show users the hierarchy delimiters or allow users to enter qualified mailbox names containing them, server implementations should use delimiter characters that users generally expect to see as name separators. The most common characters used

for this are "/" (as in Unix file names), "\" (as in OS/2 and Windows file names), and "." (as in news groups). There is little to choose among these apart from what users may expect or what is dictated by the underlying file system, if any. One consideration about using "\" is that it's also a special character in the IMAP protocol. While the use of other hierarchy delimiter characters is permissible,

A DESIGNER IS WELL ADVISED TO STAY WITH ONE FROM THIS SET unless the server is intended for special purposes only. Implementers might be thinking about using characters such as "-", "_", ";", "&", "#", "@", and "!", but they should be aware of the surprise to the user as well as of the effect on URLs and other external specifications (since some of these characters have special meanings there). Also, a server that uses "\" (and clients of such a server) must remember to escape that character in quoted strings or to send literals instead. Literals are recommended over escaped characters in quoted strings in order to maintain compatibility with older IMAP versions that did not allow escaped characters in quoted strings (but check the grammar to see where literals are allowed):

```
C: 001 LIST "" {13}
S: + send literal
C: this\\%\\%\\%\\h*
S: * LIST () "\\\" {27}
S: this\\is\\a\\mailbox\\hierarchy
S: 001 OK LIST complete
```

In any case, a server should not use normal alpha-numeric characters (such as "X" or "0") as delimiters; a user would be very surprised to find that "EXPENDITURES" actually represented a two-level hierarchy. And a server should not use characters that are non-printable or difficult or impossible to enter on a standard US keyboard. Control characters, box-drawing characters, and characters from non-US alphabets fit into this category. Their use presents interoperability problems that are best avoided.

The UTF-7 encoding of mailbox names also raises questions about what to do with the hierarchy delimiters in encoded names: do we encode each hierarchy level and separate them with delimiters, or do we encode the fully qualified name, delimiters and all? The answer for IMAP is the former: encode each hierarchy level separately, and insert delimiters between. This makes it particularly important not

to use as a hierarchy delimiter a character that might cause confusion with IMAP's modified UTF-7 [UTF-7 and [RFC-2060](#)] encoding.

To repeat: a server should use "/", "\", or "." as its hierarchy delimiter. The use of any other character is likely to cause problems and is STRONGLY DISCOURAGED.

[3.4.11.](#) ALERT Response Codes

The protocol spec is very clear on the matter of what to do with ALERT response codes, and yet there are many clients that violate it so it needs to be said anyway: "The human-readable text contains a special alert that must be presented to the user in a fashion that calls the user's attention to the message." That should be clear

B. Leiba

[Page 20]

Internet DRAFT

Implementation Recommendations

July 1999

enough, but I'll repeat it here: Clients must present ALERT text clearly to the user.

[3.4.12.](#) Deleting Mailboxes

The protocol does not guarantee that a client may delete a mailbox that is not empty, though on some servers it is permissible and is, in fact, much faster than the alternative of deleting all the messages from the client. If the client chooses to try to take advantage of this possibility it must be prepared to use the other method in the even that the more convenient one fails. Further, a client should not try to delete the mailbox that it has selected, but should first close that mailbox; some servers do not permit the deletion of the selected mailbox.

That said, a server should permit the deletion of a non-empty mailbox; there's little reason to pass this work on to the client. Moreover, forbidding this prevents the deletion of a mailbox that for some reason can not be opened or expunged, leading to possible denial-of-service problems.

Example:

[User tells the client to delete mailbox BANANA, which is currently selected...]


```
C: 008 CLOSE
S: 008 OK done
C: 009 DELETE BANANA
S: 009 NO Delete failed; mailbox is not empty.
C: 010 SELECT BANANA
S: * ... untagged SELECT responses
S: 010 OK done
C: 011 STORE 1:* +FLAGS.SILENT \DELETED
S: 011 OK done
C: 012 CLOSE
S: 012 OK done
C: 013 DELETE BANANA
S: 013 OK done
```

3.5. A Word About Testing

Since the whole point of IMAP is interoperability, and since interoperability can not be tested in a vacuum, the final recommendation of this treatise is, "Test against EVERYTHING." Test your client against every server you can get an account on. Test your server with every client you can get your hands on. Many clients make limited test versions available on the Web for the downloading. Many server owners will give serious client developers

B. Leiba

[Page 21]

Internet DRAFT

Implementation Recommendations

July 1999

guest accounts for testing. Contact them and ask. NEVER assume that because your client works with one or two servers, or because your server does fine with one or two clients, you will interoperate well in general.

In particular, in addition to everything else, be sure to test against the reference implementations: the PINE client, the University of Washington server, and the Cyrus server.

See the following URLs on the web for more information here:

IMAP Products and Sources: <http://www.imap.org/products.html>

IMC MailConnect: <http://www.imc.org/imc-mailconnect>

4. Security Considerations

This document describes behaviour of clients and servers that use the IMAP4 protocol, and as such, has the same security considerations as described in [[RFC-2060](#)].

[5](#). References

[[RFC-2060](#)]; Crispin, M.; "Internet Message Access Protocol - Version 4rev1"; [RFC 2060](#); University of Washington; December 1996.

[[RFC-2119](#)]; Bradner, S.; "Key words for use in RFCs to Indicate Requirement Levels"; [RFC 2119](#); Harvard University; March 1997.

[[RFC-2180](#)]; Gahrns, M.; "IMAP4 Multi-Accessed Mailbox Practice"; [RFC 2180](#); Microsoft; July 1997.

[UTF-8]; Yergeau, F.; " UTF-8, a transformation format of Unicode and ISO 10646"; [RFC 2044](#); Alis Technologies; October 1996.

[UTF-7]; Goldsmith, D. & Davis, M.; "UTF-7, a Mail-Safe Transformation Format of Unicode"; [RFC 2152](#); Apple Computer, Inc. & Taligent, Inc.; May 1997.

[NAMESPACE]; Gahrns, M. & Newman, C.; "IMAP4 Namespace"; draft document <[draft-gahrns-imap-namespace-01.txt](#)>; Microsoft & Innosoft; June 1997.

[6](#). Author's Address

Barry Leiba
IBM T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532

Phone: 1-914-784-7941
Email: leiba@watson.ibm.com

This document will expire at the end of January 2000.