Z-POP v1.0 Functional Specification

Bob Glickstein, NCD Software
bobg@z-code.com

3 Oct 1995

1 Introduction

Z-POP is NCD Software's enhanced POP server, based on RFC-1460.[1] POP, the Post-Office Protocol, is a simple client-server model for downloading mail from a network host and for performing rudimentary folder-management operations on the host's mail spool.

Z-POP is a superset of RFC-1460 POP. It adds four families of features: folder synchronization; real-name service; configuration downloading; and user preferences. The synchronization feature is by far the most intricate.

Z-Mail, NCD Software's e-mail client application, can speak standard POP to ordinary POP servers, but can also exploit the extra features of Z-POP when available.

All new server commands described in this document operate only in the normal transaction state of the server; i.e., after the user is authenticated. Furthermore, all new commands which use a dot alone on a line to indicate the end of some transmitted data also use the conventional simple algorithm for quoting lines of data which begin with a dot, to wit, those dots are doubled.

This document uses the terms "spool" and "folder" interchangeably.

2 The synchronization feature

Z-POP includes several commands oriented toward *folder synchronization*. This is the ability to detect changes between two copies of the spool—the server's copy and the client's copy—and resolve the differences by making changes to one or both copies. A significant amount of the functionality for achieving this lives in the client. In this section I'll talk about client and server functionality being sure to distinguish one from the other.

[1] RFC-1460 has been superseded by RFC-1725—which is still a draft—but Z-POP has not yet implemented the changes between the two drafts.

2.1 Folder differencing

The first phase of folder synchronization is *folder differencing*, in which differences between the server spool and the client spool are computed. Once the differences are known, we can derive the actions needed to make both copies the same. Certain kinds of difference we deemed uninteresting. For example, if both copies contain the same messages and merely the sequences differ, we wish to treat the folders as identical. Certain minor differences (in unimportant message headers, for example) between a pair of otherwise identical messages should similarly be passed over.

It is desirable to minimize the amount of information that must be exchanged between client and server for computing folder differences. Certainly it would be possible for a naïve implementation of folder differencing to transfer a verbatim copy of the client spool to the server, or vice versa, and compute differences therefrom; but particularly in light of the possibilities of large folders and slow serial connections, the fewer bytes exchanged, the better.

We have therefore devised two new concepts: that of a message's *key digest*; and that of a folder's *meta-digest*. The key digest is a compact representation of a message, implemented using the MD5 algorithm defined in RFC-1321. The result is a 128-bit value which uniquely[2] identifies the message. The representation is insensitive to minor changes in the message, such as the addition or removal of trailing newlines in the body. The meta-digest is computed from the key digests of the messages in the folder. The meta-digest is insensitive to reordering of the folder's messages and to multiple occurrences of the same message.

At the beginning of folder differencing, the client requests the meta-digest of the server's spool. If it matches the meta-digest of the client's spool, the two folders are deemed identical and folder differencing ends (as does synchronization, there being no differences to resolve). On the other hand, if the meta-digests do not match, the task is to narrow down the differences to two sets: $\mathcal{S}$, the set of messages in the server spool that don't appear in the client spool; and $\mathcal{C}$, the set of messages in the client spool that don't appear in the server spool. To do this we introduce our third new concept, the *partitioned meta-digest*.

The partitioned meta-digest (PMD for short) is a meta-digest constructed from a subset of the key digests that comprise the "full" meta-digest. A PMD is said to be partitioned "on" some number of bits. When the number of bits is b, there are 2^b possible PMDs, denoted $\mathrm{PMD}_{b,0}$ through $\mathrm{PMD}_{b,2^b-1}$. $\mathrm{PMD}_{0,0}$ is the same as the "full" meta-digest, also referred to as the "top-level" meta-digest.

To construct a folder's PMD on b bits, one partitions the key digests which are the constituents of that folder's $\mathrm{PMD}_{0,0}$ based on the first b bits of each key

[2]It is possible for two messages to have the same key digest; mapping arbitrarily-long streams of bits to 128-bit digests is necessarily many-to-one. However, the odds that two messages "collide" in this way are about 3.4×10^{38} to 1, so don't worry so much.

digest. Only those key digests whose first b bits match the b-bit pattern k are included in $\mathrm{PMD}_{b,k}$.

$$\underbrace{00110}_{\text{First 5 bits} = 6}10011101101000\ldots$$

Determining whether a digest is a member of $PMD_{5,6}$. The number 6 has a 5-bit binary representation of "00110."

After the client finds the server's $\mathrm{PMD}_{0,0}$ not to match its own, it may then choose to try to narrow the search for differences by requesting the server's $\mathrm{PMD}_{1,0}$ and $\mathrm{PMD}_{1,1}$. Since $\mathrm{PMD}_{0,0}$ didn't match, at least one of $\mathrm{PMD}_{1,0}$ and $\mathrm{PMD}_{1,1}$ will also not match, at which point the client may request the server's $\mathrm{PMD}_{2,0}$ and $\mathrm{PMD}_{2,1}$, and/or the server's $\mathrm{PMD}_{2,2}$ and $\mathrm{PMD}_{2,3}$. Generally speaking, when $\mathrm{PMD}_{b,k}$ is found not to match, the search can be narrowed by looking at that PMD's two *children*, which are $\mathrm{PMD}_{b+1,2k}$ and $\mathrm{PMD}_{b+1,2k+1}$.

Note that this procedure partitions the meta-digest in a way that is impervious to differences in message sequencing between the two folders. In other words, if a message is counted as part of the client spool's $\mathrm{PMD}_{1,0}$ (for instance), it will necessarily also be counted as part of the server spool's $\mathrm{PMD}_{1,0}$ if present; never as part of the server spool's $\mathrm{PMD}_{1,1}$.

At some point, the differencing process must stop partitioning meta-digests into smaller and more numerous pieces (which could continue long past the point of diminishing returns) and must directly compare the constituent messages of the client's and the server's PMD. This cutoff should be based on a maximum value for b, chosen according to the size of the client and server spools. Statistically, for an n-message folder, each $\mathrm{PMD}_{b,k}$ represents $n/2^b$ messages. $b_{\max}$ should be chosen such that $n/2^{b_{\max}}$ is acceptably small. The Z-POP implementation considers 8 to be "sufficiently small": on average, when the differencing procedure stops refining smaller and smaller PMDs and begins looking inside them, it will need to compare 8 messages per PMD.

Example: Given a 232-message folder, what's a good value for $b_{\max}$? This reduces to solving $8 = 232/2^x$ for x, or $x = \log_2 \frac{232}{8} \approx 4.86$. Then $b_{\max} = \lceil x \rceil = 5$.

2.1.1 The key digest

The key digest for a message is constructed by invoking the MD5 algorithm on a canonicalized version of the message's stream. Canonicalizing proceeds as follows. "Newline" is here used to mean any of the myriad line-terminating conventions, while CRLF refers to the line-terminating sequence "carriage return, linefeed" (ASCII 13 followed by ASCII 10).

1. All RFC-822 headers are discarded except for the *key headers*, which are:

- Apparently-To
- Cc
- Date
- From
- Message-Id
- Resent-Cc
- Resent-Date
- Resent-From
- Resent-To
- Subject
- To

2. For each remaining header, the header name is canonicalized by capitalizing it as in the preceding list.
3. Each newline-whitespace sequence (that is, each newline followed by one or more spaces or tabs) in a header body is replaced with a single space.
4. Each header body is terminated with a single CRLF.
5. The key headers are reordered to appear in the same sequence as in the preceding list (the "canonical" header ordering). Multiple occurrences of identically named headers retain the original ordering with respect to each other. In other words, a stable sort is performed on the headers based on the ordering dictated by the key header table above.
6. A CRLF is added after the last header (in addition to the CRLF terminating it).
7. All but one of the message body's trailing newlines are removed.
8. All newlines in the message body are canonicalized to CRLFs.

2.1.2 The partitioned meta-digest

The partitioned meta-digest $\mathrm{PMD}_{b,k}$ is computed by selecting those key digests whose first b bits match the b-bit pattern k. For this purpose, bits are considered from least significant to most significant in each octet of the digest, like so:

$$\underbrace{7\,6\,5\,4\,3\,2\,1\,0}_{\text{Octet 0}}\,\underbrace{15\,14\,13\,12\,11\,10\,9\,8}_{\text{Octet 1}}\ldots\,\underbrace{127\,126\,125\,124\,123\,122\,121\,120}_{\text{Octet 15}}$$

The selected digests are sorted into numerically ascending order and duplicates are removed. Then they are concatenated and fed as input to the MD5 algorithm.

2.1.3 The differencing algorithm

The client chooses a value $b_{\max}$ such that $n/2^{b_{\max}}$ is acceptably small, where n is the larger of the client's spool size or the server's spool size (in messages).[3] Then,

$\mathcal{P} \leftarrow \{0\}$
FOR $0 \leq b < b_{\max}$
  $\mathcal{P}' \leftarrow \mathcal{P}$
  $\mathcal{P} \leftarrow \emptyset$
  FOREACH $p \in \mathcal{P}'$
    IF server's $\mathrm{PMD}_{b,p} \neq$ client's $\mathrm{PMD}_{b,p}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{2p, 2p+1\}$

followed by

FOREACH $p \in \mathcal{P}$
  IF server's $\mathrm{PMD}_{b_{\max},p} \neq$ client's $\mathrm{PMD}_{b_{\max},p}$
    $\mathcal{S} \leftarrow \mathcal{S} \cup [\mathrm{ServerMembers}(b_{\max}, p) - \mathrm{ClientMembers}(b_{\max}, p)]$
    $\mathcal{C} \leftarrow \mathcal{C} \cup [\mathrm{ClientMembers}(b_{\max}, p) - \mathrm{ServerMembers}(b_{\max}, p)]$

"ServerMembers" is a function which, given b and p, yields the set of messages in the server spool which are members of $\mathrm{PMD}_{b,p}$. "ClientMembers" is the analogous function for the client.

2.1.4 Server commands

`ZPSH` *b p* `1 1-`*n*. Requests PMDs from the server partitioned on *b* bits. Requested partitions are given by *p*, which is a set of numbers denoted by comma-separated ranges; each range is either a number or a hyphen-separated range of numbers. Example: `2,4-7`. The "1" must appear literally; this is explained below. The final argument 1-*n* denotes the messages which should participate. This should always be all the messages in the folder, except as explained later in this document.

The server responds with a hexadecimal representation (see above for bit ordering) for each PMD, each on a separate line, followed by a line containing a dot. Characters that are not hexadecimal digits may be included—for example, spaces for human legibility. They are ignored.

Example:

C: ZPSH 3 0-3,5 1 1-62
S: +OK

[3] Note that $b_{\max}$ cannot exceed 128, the total number of bits in an MD5 digest and therefore the maximum number of bits that can be considered for distinguishing two digests' partition membership.

S: 0289 8ea6 47a2 4b9d 073f 790d c7d7 5bb0
S: 36d4 772d 0b79 3503 433d 4ef3 b161 b1a7
S: 8692 08ce 0dcd 0479 8ae3 30da 0199 68dc
S: bd20 4ed7 7331 9d6c bee1 eae4 2569 123a
S: 55e8 62aa fbd6 c3a3 5dd6 f077 0a22 3187
S: .

Here the client is asking for PMDs partitioned on 3 bits. Out of the 2^3, or 8, possible PMDs numbered 0 through 7, the client is only interested in $\text{PMD}_{3,0}$, $\text{PMD}_{3,1}$, $\text{PMD}_{3,2}$, $\text{PMD}_{3,3}$, and $\text{PMD}_{3,5}$. The client knows the server spool contains 62 messages.

The server responds with '`+OK`' followed by the five requested PMD values (in the order described).

`ZHB2` *b p* `1-`*n*. Requests the members of the server spool's $\text{PMD}_{b,p}$. The final argument 1-*n* denotes the messages which should participate. This should always be all the messages in the folder, except as explained later in this document.

The server responds with three colon-separated fields per line for each member message, followed by a line containing a dot. The three fields are: message number; hexadecimal representation of key digest; and hexadecimal representation of header digest (see below).

Example:

C: ZHB2 3 6 1-67
S: +OK
S: 4:8692 08ce 0dcd 0479 8ae3 30da 0199 68dc:fa1c e3da bacc 9ccd a6d2 6fc0 a58d 4f6e
S: 31:36d4 7c2d 01c9 3503 433d 4ef3 b161 11a7:5a1f 82c3 1c1c 185a e920 a532 6f1a 84f8
S: .

Here the client is asking for the members of $\text{PMD}_{3,6}$. The server responds with the key and header digests of messages 4 and 31, which in this example are the members of $\text{PMD}_{3,6}$. There is no way for the client to know beforehand how many messages will be in a particular meta-digest partition. On average, $\text{PMD}_{b,k}$ will contain $n/2^b$ messages (where n is the number of messages in the folder), but may contain as few as 0 and as many as n.

2.2 Refining differences

Once the sets $\mathcal{C}$ and $\mathcal{S}$ are known, Z-POP performs another round of differencing called *refining*. The object of this round is to determine the set $\mathcal{D}$ of messages which appear in both the client and server spools, but which differ in certain ways. A message in one spool may be identical to a message in another spool and still differ from it, if we define "identical" to mean they have the same key

digest. Particularly we're interested in differences in non-key headers between two copies of an "identical" message. For example, if a user downloads a message from a host, then composes a reply to that message, the letter '`r`' will be added to the `Status` header of the client copy of the message (in certain e-mail clients, including Z-Mail). When differencing, we would like to detect this change in a message otherwise identical to its server-side counterpart.

In fact, once we know $\mathcal{C}$ and $\mathcal{S}$, what we'd really like to do is another folder-differencing round in which each message's digest is based on its non-key headers (rather than on key headers plus the message body). Meta-digests are then defined in terms of these non-key digests, and only messages in the subset $\mathcal{C} \cap \mathcal{S}$ get to participate. This should help us find identical messages with differing headers as quickly as the first folder differencing round narrowed the search for $\mathcal{C}$ and $\mathcal{S}$.

2.2.1 The header digest

To simplify implementation, rather than define a digest based on non-key headers, we define a *header digest* based on *all* headers except for "X-Key-Digest." (This header name is used by Z-POP to speed computation of a message's key digest. Its presence or absence should not affect the value of the message's header digest.)

The header digest is the result of running MD5 over a canonicalized version of the message's headers. Canonicalization in this case proceeds as follows:

1. Each newline-whitespace sequence (that is, each newline followed by one or more spaces or tabs) in a header body is replaced with a single space.

2. Each header body is terminated with a single CRLF.

Note that these are identical to two of the canonicalization steps from the key digest algorithm. Other canonicalizations from that algorithm are specifically not included in order that we may detect differences in the uncanonicalized data.

2.2.2 The header-based partitioned meta-digest

The header digest analog of the partitioned meta-digest is the *header-based partitioned meta-digest* (HPMD for short). It's the same as the key-based partitioned meta-digest except for two things: it's based on header digests, not key digests; and only a subset of the messages in a folder contributes its header digests—specifically, the messages in $\mathcal{C} \cap \mathcal{S}$. We use $\mathrm{HPMD}_{b,k}^{\mathcal{M}}$ to denote the kth partition of the HPMD constructed from messages in set $\mathcal{M}$ and partitioned on b bits.

Note that while each message in $\mathcal{C} \cap \mathcal{S}$ appears in both client and server spools, in general its position within each spool—i.e., its message number—will differ. Implementors must therefore be able to map each member of $\mathcal{C} \cap \mathcal{S}$ (whose

representation is not specified) into the corresponding message in each of the client spool and the server spool.

Important note: while the value of $\mathrm{HPMD}_{b,k}^{\mathcal{M}}$ is computed from the header digests of its constituent messages, the constituency of $\mathrm{HPMD}_{b,k}^{\mathcal{M}}$ is still based on key digests; that is, the members of $\mathrm{HPMD}_{b,k}^{\mathcal{M}}$ are those messages in $\mathcal{M}$ each of whose key digests matches k in the first b bits. Another way of saying this is that

$$\mathrm{Members}(\mathrm{HPMD}_{b,k}^{\mathcal{M}}) \equiv \mathcal{M} \cap \mathrm{Members}(\mathrm{PMD}_{b,k})$$

This is to ensure that identical messages end up in the same meta-digest partition on both the client and the server, even though their header digests may differ (which of course is the interesting case).

2.2.3 The refined differencing algorithm

The refined differencing algorithm is precisely the previous (key-based) differencing algorithm with header digests replacing key digests and HPMDs replacing PMDs.

$\mathcal{P} \leftarrow \{0\}$
FOR $0 \leq b < b_{\max}$
  $\mathcal{P}' \leftarrow \mathcal{P}$
  $\mathcal{P} \leftarrow \emptyset$
  FOREACH $p \in \mathcal{P}'$
    IF server's $\mathrm{HPMD}_{b,p}^{\mathcal{C}\cap\mathcal{S}} \neq$ client's $\mathrm{HPMD}_{b,p}^{\mathcal{C}\cap\mathcal{S}}$
      $\mathcal{P} \leftarrow \mathcal{P} \cup \{2p, 2p+1\}$

followed by

FOREACH $p \in \mathcal{P}$
  IF server's $\mathrm{HPMD}_{b_{\max},p}^{\mathcal{C}\cap\mathcal{S}} \neq$ client's $\mathrm{HPMD}_{b_{\max},p}^{\mathcal{C}\cap\mathcal{S}}$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \mathrm{DifferingMembers}(\mathcal{C} \cap \mathcal{S}, b_{\max}, p)$

"DifferingMembers" is a function which, given $\mathcal{M}$, b, and p, returns those members of $\mathrm{HPMD}_{b,p}^{\mathcal{M}}$ for which the header hash differs between the client copy and the server copy. Note that each member of $\mathcal{D}$ must be represented in such a way that it can be mapped to the correct individual message in both the client and the server spools.

2.2.4 Server commands

`ZPSH` *b p* `0` *s*. Requests HPMDs from the server partitioned on *b* bits. Requested partitions are given by *p*, which is a set of numbers denoted by comma-separated ranges; each range is either a number or a hyphen-separated range of numbers. Example: `2,4-7`. The "0" must appear

literally; this distinguishes the HPMD use of `ZPSH` from the PMD use of `ZPSH` (where that argument is "1"). The final argument *s* denotes the subset of messages to participate in the meta-digest and is a comma-separated list of ranges like *p*. It should be the set $\mathcal{C} \cap \mathcal{S}$ represented as server-side message numbers.

The server responds with a hexadecimal representation for each HPMD, each on a separate line, followed by a line containing a dot.

`ZHB2` *b p s*. Requests the members of the server spool's $\mathrm{HPMD}^s_{b,p}$. The argument *s* is a comma-separated list of message-number ranges. In the present implementation this actually yields the members of $\mathrm{PMD}_{b,p}$, which is a superset of $\mathrm{HPMD}^s_{b,p}$, and it is the client's responsibility to extract the members of *s* from the output; but `ZHB2` is ultimately intended to perform this computation itself.

The server responds with three colon-separated fields per line for each member message, followed by a line containing a dot. The three fields are: message number; hexadecimal representation of key digest; and hexadecimal representation of header digest.

2.3 Resolving differences

After folder differencing and refined differencing, there are three sets of messages: $\mathcal{S}$, the set of messages present in the server spool but not the client spool; $\mathcal{C}$, the set of messages present in the client spool but not the server spool; and $\mathcal{D}$, the set of messages common to both folders but differing in identity-preserving ways.

For each message in $\mathcal{S}$ we could resolve the difference between client and server by either (a) copying the message from the server to the client, or (b) deleting the message from the server.

This document does not specify how to decide between deleting and copying; but to illustrate, Z-Mail employs concepts called *ghosts* and *tombfiles*. When in the course of folder management (on the client) the user deletes a message, its "ghost" is "buried" in the user's "tombfile." During synchronization, the tombfile is consulted[4] for each message in $\mathcal{S}$. If the corresponding ghost is found, then we know the message once existed in the client spool and was deleted; so the correct action is likely to be deletion from the server. On the other hand, if no ghost is found, then this may be a message never seen on the client, and so should be downloaded. Ghosts are only kept in the tombfile for a limited time.[5]

Similar considerations apply to messages in $\mathcal{C}$.

For each message in $\mathcal{D}$, resolving the difference means replacing the server copy's headers with those of the client copy, or vice versa. (One can imagine

[4] We call this a "seance."

[5] Called the "afterlife." We got a little giddy when naming this stuff.

an elaborate `diff3`-type merging solution, even though that sort of approach usually requires the availability of a "common ancestor" of the two copies. In practice, such an approach is hardly if ever necessary.) To simplify the implementation, however, we took the view that the only header that's likely to change in an interesting way is the `Status` header, so for now the server only needs to be able to report and replace `Status` headers.[6]

2.3.1 Server commands

`RETR` *m*. This is the conventional POP `RETR` command, which retrieves message *m*. This is used in synchronization for downloading messages to the client.

`ZRTR` *m*. This is an alternative to `RETR` which downloads message *m* without flagging it as "read." This is important for users who invoke synchronization frequently—for instance, those who use it as an alternative to the conventional POP mechanism for retrieving newly-arrived mail. If `RETR` is used, then a header mismatch would exist between the server copy of a message (flagged as "read") and the just-downloaded client copy (not flagged). This difference would then be detected in the "refining" phase of the next folder synchronization, defeating the intent of synchronization which is to make the two folders difference-free.

`DELE` *m*. This is the conventional POP `DELE` command, which deletes message *m*. This is used in synchronization for deleting messages from the server spool.

`ZMSG`. Uploads a message from the client to the server. The position of the new message within the server spool is unspecified. After the server responds with '`+OK`', the client must supply the RFC-976 "envelope `From_` line" followed by the RFC-822 contents of the message, followed by a line containing a dot. Lines beginning with dot in the message body have the leading dot doubled.

Example:

C: ZMSG
S: +OK Send message.
C: From dreez Wed Jul 26 19:56:37 1995
C: Received: by zapato for bobg
C: X-Mailer: Z-Mail Lite (3.3dev.726 26jul95)
C: From: Andrea Dougherty <dreez@zerex>

[6] This is highly nonoptimal in certain cases, and can lead to unresolvable header differences between client and server spools. Resolving only the `Status` header, furthermore, means the whole "refined differencing" step is overkill. Nevertheless, it is our intention to eventually implement fully general header-difference resolution using refined differencing. When that occurs, we will necessarily introduce more new server commands for uploading and downloading message headers only.

C: Message-Id: <950726182247.ZM3918@zerex>

C: Date: Wed, 26 Jul 1995 18:22:42 -0700

C: To: Bob Glickstein <bobg@z-code.com>

C: Subject: Synchronicity

C:

C: Hi Bob,

C: I never dreamed synchronization could be

C: so cool!

C: .

S: +OK New message is 63 (355 octets)

The server's final response includes the newly uploaded message's message number. The client can parse this out of the server response by looking for the first digit string. The new message must become immediately accessible for the remainder of the server session.

`ZFRL` *m*. Reports the RFC-976 "envelope `From_` line" of server message *m*. This may contain information not found in the ordinary RFC-822 headers of the message, such as the date and time of message receipt, and is in fact used by Z-Mail when downloading new messages during synchronization, in the name of preserving as much information as possible.

Example:

C: ZFRL 11

S: +OK From schaefer Fri Aug 25 15:02:44 1995

`ZSTS` *m*. Reports the "status" of server message *m*. Status is reported as an integer in ASCII decimal representation. Its value is a bit vector constructed from the following codes:

1	The message is "new."
2	The message has been "saved."
4	The message has been "replied to."
8	The message has been "resent."
16	The message has been "printed."
32	The message has been "deleted." (That is, marked for deletion but not yet removed.)
64	The message has been "preserved." (This flag is unused.)
128	The message is "unread."

See appendix B for a discussion of the meanings of these values.

Example:

C: ZSTS 59

S: +OK 129

`ZST2` *msgs*. Reports the statuses of messages in the set *msgs*, which is a comma-separated list of message number ranges.

The server responds with a pair of numbers per line, separated by whitespace, for each message: the message number, and its status, encoded as described above. The last line is followed by a line containing a dot.

Example:

C: ZST2 3-5,20-22
S: +OK 6 messages
S: 3 129
S: 4 4
S: 5 0
S: 20 24
S: 21 129
S: 22 0
S: .

`ZSST` *msg mask val*. Sets the status of server message *msg* according to *mask* and *val*. For each nonzero bit in *mask*, the value of the corresponding bit in *val* is used. Other bits in *val* are ignored.

Example:

C: ZSST 36 133 4
S: +OK

This example changes the values of the "new," "replied to," and "unread" flags ($1 + 4 + 128 = 133$.) "New" and "unread" are turned off since their bits are zero in *val*, while "replied to" is turned on.

2.4 Presentation

To assist in presenting the synchronization process to the user, the server should be able to report three additional pieces of information about each message: its human-readable summary (which should include a message number); its size; and its date. The summary can be used in a "preview dialog" (which appears after differencing and before resolving) to help the user visualize which messages will be affected in a proposed synchronization. The size can help the user cull large messages which would be time-consuming to download or upload. The date can help the user cull old messages which are not of interest.

2.4.1 Server commands

`ZSMY` *m*. Yields a human-readable summary of server message *m*. The summary must begin with the message number followed by whitespace. Beyond that the format is unimportant, but the summary is confined to a single line. The goal is to produce a brief description of a message such that a human

reader can have an idea of which message is being described. Z-Mail uses this information in a "synchronization preview" dialog which appears after folder differencing. The user sees the summaries of the messages in $\mathcal{S}$, $\mathcal{C}$, and $\mathcal{D}$ and may select or deselect individual messages for participation in resolving folder differences.

Example:

C: ZSMY 29
S: +OK 29 Nathaniel Borenstein Sep 8 4:20pm (16) Congratulations!

`ZSIZ` *m*. Yields the size of message *m* in bytes, represented as an ASCII decimal string.

Example:

C: ZSIZ 47
S: +OK 729

`ZDAT` *m*. Yields the date of message *m* in Unix seconds (per `time`(2)), represented as an ASCII decimal string.

Example:

C: ZDAT 3
S: +OK 601318800

3 Real name service

For some purposes an e-mail client needs to know the user's full name—e.g., for placement in the `From` header of outgoing mail. That information is not normally available to PC operating systems, though it is stored as a matter of course on network hosts, such as those used to run Z-POP. To allow the client access to this information we've introduced the simple command `ZWHO`.

`ZWHO`. Requests the user's full name from the server.

Example:

C: ZWHO
S: +OK John Jacob Jingleheimer-Schmidt

4 Configuration downloading

There are at least two reasons for wanting to store configuration data for the e-mail client on the server. First, it remains accessible to users who switch among different client machines. Second, it provides a central location where a system administrator may control the configuration for all of a site's users.

Consequently, Z-POP incorporates the idea of a central master tree of configuration files, organized to contain one or more configuration files for each of one or more triples $< C, P, V >$ where C is the name of a configurable e-mail client (e.g., "zmail"), P is the name of a platform on which the client runs (e.g., "windows"), and V is a version string (e.g., "4.0"). The client informs the server of these parameters via the `ZMOI` command. Next the client may inform the server of which configuration files it already has using `ZHAV`. Finally, the client may request the latest configuration files using `ZGET`.

Each configuration file has associated with it an "attribute line" in the following format:

`-%-` *name*:*value*; *name*:*value*... `-%-`

where each *name* is the name of an attribute and each *value* is its value. Whitespace outside of names and values is ignored. The recognized attributes are:

`File-Id`. The name of the file.

`Client-Rev`. The version of the e-mail client for which this is a configuration file.

`Seq-Num`. A sequence number for the file itself which should be incremented each time the file is modified.

Typically, each file's attribute line is stored in the beginning of the file itself. Naturally, the syntax of the client's configuration files may not permit the appearance of such a line, so the line will usually be commented out using the client's comment syntax. The server will ignore everything not between the `-%-` delimiters.

The server compares the client's `Client-Rev` and `Seq-Num` with the server's `Client-Rev` and `Seq-Num` for each file in the master download tree. If, by this comparison, the client is deemed to have an up-to-date copy of a file, that file will not be downloaded by a `ZGET`.

4.1 Server commands

`ZMOI`. Permits the client to inform the server of parameters that are important for configuration downloading. The client supplies whitespace-separated attribute-value pairs, one per line, followed by a line containing a dot.

Example:

C: ZMOI
S: +OK Tell me about vous.
C: PRODUCT zmail
C: PLATFORM windows
C: VERSION 4.0

C: .
S: +OK

`ZHAV`. Permits the client to inform the server of the configuration files it already has. It does so by sending the attribute line of each configuration file. The server responds with the number of client files which are out of date with respect to downloadable configuration files on the server—i.e., the number of files that would be downloaded in a subsequent `ZGET`.

The mechanism for determining whether a file is out of date is a bit baroque. The server first parses the `Client-Rev` parameter which consists of a major number, a minor number, a release level, and a patch level. The syntax is *major.minor*[*release*][*.patch*]. Square brackets indicate optional parts of the string. Each of *major*, *minor*, and *patch* is an integer. If *patch* is omitted, it defaults to 0. On the other hand, *release* is one of the strings `dev`, `a`, `b`, and `B`. For sorting purposes these strings correspond to the values 1, 2, 3, and 4, respectively. If *release* is omitted, its default value is 5.

Examples of `Client-Rev` strings, in ascending order: `2.1dev`; `2.1dev.17`; `2.1`; `3.0b`; `3.0B`; `3.2a.6`.

Once the `Client-Rev` is parsed, a client file's up-to-date status is determined as follows:

1. If the client copy's major number is lower than the server copy's major number, it is out of date; otherwise
2. If the client copy's minor number is lower than the server copy's minor number, it is out of date; otherwise
3. If the client copy's release level is lower than the server copy's release level, it is out of date; otherwise
4. If the client copy's patch level is lower than the server copy's patch level, it is out of date; otherwise
5. If the client copy's sequence number (the `Seq-Num` parameter) is lower than the server copy's sequence number, it is out of date; otherwise it is up to date.

This command must be preceded by a `ZMOI` command in which at least the `PRODUCT`, `PLATFORM`, and `VERSION` parameters were specified.

Since the attribute line normally appears as part of the beginning of its associated file, the client may send the first few lines of the file rather than just the attribute line. It is the server's responsibility to parse out the attributes portion of all data transmitted to it.

Example:

C: ZHAV
S: +OK Send file stamps
C: # -%- File-Id:attach.typ; Client-Rev:4.0.13; Seq-Num: 0 -%-
C: # -%- File-Id:system.rc; Client-Rev:4.0.0; Seq-Num:0 -%-
C: .
S: +OK 2 files (11375 octets) are out of date.

`ZGET`. Requests that the server transmit downloadable configuration files which are absent or out-of-date on the client. The server responds with the number of files to follow, then sends each one preceded by a line containing its name and followed by a line containing a dot. After the last file and its terminating dot are sent, *another* line containing a dot is sent. This final dot is transmitted even if zero files are downloaded.

This command must be preceded by a `ZHAV` in order for the server to know which files to download.

Example:

C: ZGET
S: +OK 2 files (11375 octets)
S: attach.typ
S: # -%- File-Id:attach.typ; Client-Rev:4.0.13; Seq-Num: 3 -%-
S: . . . *contents of attach.typ*. . .
S: .
S: system.rc
S: # -%- File-Id:system.rc; Client-Rev:4.0.0; Seq-Num:1 -%-
S: . . . *contents of system.rc*. . .
S: .
S: .

5 User preferences

Centrally storing an individual user's preferences is similar to centrally storing client configuration information, with three main differences:

- While every user downloads the same configuration files, preferences files are different for each user.
- While configuration files are maintained by a site administrator, preferences files are under the control of individual users (though, being centrally stored, they are accessible to sysadmin tweaking).
- While different sets of configuration files may be selected based on the `PRODUCT`, `PLATFORM`, and `VERSION` parameters, for a given user there is only one preferences file.

Of course, the main benefit of centrally storing user preferences is the same as for centrally storing client configuration: accessibility to users who switch among different client machines.

5.1 Server commands

`GPRF`. Requests the user's preferences file. The server responds with the contents of the file followed by a line containing a dot.

Unfortunately, due to a lack of foresight, the client is required to recognize the absence of a user preferences file by distinguishing this server response:

```
+OK Preferences for bobg follow.
```

from this server response:

```
+OK No preferences for bobg.
```

In the latter case, no file contents or terminating dot are sent.

Example:

C: GPRF
S: +OK Preferences for bobg follow.
S: *. . . contents of bobg's preferences file. . .*
S: .

`SPRF`. Permits the client to upload a new version of the user's preferences file (for retrieval by later `GPRF`s). The client sends the file contents followed by a line containing a dot.

Example:

C: SPRF
S: +OK Send preferences.
C: *. . . contents of preferences file. . .*
C: .
S: +OK Wrote pref file for bobg.

A Meta-digest partition membership

Here's a sample implementation in C of a test for partition membership. Given a digest, a number of bits b and a partition number p in the range $[0, 2^b - 1]$, `DigestInPartition` tells whether the digest is a member of $\mathrm{PMD}_{b,p}$.

```
#define DigestInPartition(digest, bits, partition) \
    (DigestPartition((digest), (bits)) == (partition))
```

```
int
DigestPartition(digest, bits)
    digest_t digest;
    int bits;
{
    int i, result = 0;

    for (i = 0; i < bits; ++i) {
        if (DigestBit(digest, i))
            result += (1 << ((bits - 1) - i));
    }
    return (result);
}
```

`DigestBit` is a function for testing the `i`th bit of a digest. Its implementation naturally depends on the representation of a digest, which is not specified by this document. However, presuming digests are implemented in the obvious way—i.e., as an array of characters each holding `CHAR_BIT` bits—here's how one could define `DigestBit`.

```
typedef struct {
    unsigned char chars[128 / CHAR_BIT];
} *digest_t;

int
DigestBit(digest, bit)
    digest_t digest;
    int bit;
{
    return (digest->chars[bit / CHAR_BIT]
            & (1 << (bit % CHAR_BIT)));
}
```

B Message status

There is no standard for interpreting or setting the contents of a message's `Status` header. Z-POP uses some *de facto* interpretive rules for mapping from `Status` header contents to status flag values (as described under `ZSTS` above) and vice versa.

B.1 Deriving flag values

All messages start as "new" (1) and "unread" (128) for a default initial flags value of 129. This value is modified based on characters encountered in the

`Status` header, as follows.

- `D` The message is flagged for deletion. Turn off "new" and "unread" and turn on "deleted" (32).
- `O` (Capital letter O.) The message is old. Turn off "new."
- `R` The message has been read. Turn off "new" and "unread."
- `N` The message is new. Turn on "new" and "unread."
- `P` The message is "preserved." Turn on "unread."
- `S` The message has been saved. Turn on "saved" (2) and turn off "new."
- `r` The message has been replied to. Turn on "replied" (4) and turn off "new."
- `f` The message has been resent (forwarded). Turn on "resent" (8).
- `p` The message has been printed. Turn on "printed" (16).

B.2 Deriving `Status` header contents

All messages start with no `Status` header. If a message is "new" (status flag 1), no `Status` header is created. Otherwise, a status header is created and populated according to the following rules:

1. `O` (capital letter O) is added.
2. If the message is not unread, `R` is added.
3. If the message is saved, `S` is added.
4. If the message is replied to, `r` is added.
5. If the message has been resent, `f` is added.
6. If the message has been printed, `p` is added.