        PCMAIL: A Distributed Mail System for Personal Computers

1. Status of this Document

   This document is a preliminary discussion of the design of a
   personal-computer-based distributed mail system. It is published for
   discussion and comment, and does not constitute a standard. As the
   proposal may change, implementation of this document is not advised.
   Distribution of this memo is unlimited.

2. Introduction

   Pcmail is a distributed mail system that provides mail service to an
   arbitrary number of users, each of which owns one or more personal
   computers (PCs). The system is divided into two halves. The first
   consists of a single entity called the "repository". The repository
   is a storage center for incoming mail. Mail for a Pcmail user can
   arrive externally from the Internet or internally from other
   repository users. The repository also maintains a stable copy of
   each user's mail state (this will hereafter be referred to as the
   user's "global mail state"). The repository is therefore typically a
   computer with a large amount of disk storage.

   The second half of Pcmail consists of one or more "clients". Each
   Pcmail user may have an arbitrary number of clients, which are
   typically PCs. The clients provide a user with a friendly means of
   accessing the user's global mail state over a network. In order to
   make the interaction between the repository and a user's clients more
   efficient, each client maintains a local copy of its user's global
   mail state, called the "local mail state". Since clients are PCs,
   they may not always have access to a network (and therefore to the
   global mail state in the repository). This means that the local and
   global mail states may not be identical all the time, making
   synchronization between local and global mail states necessary.

   Clients communicate with the repository via the Distributed Mail
   System Protocol (DMSP); the specification for this protocol appears
   in appendix A. The repository is therefore a DMSP server in addition
   to a mail end-site and storage facility. DMSP provides a complete
   set of mail manipulation operations ("send a message", "delete a
   message", "print a message", etc.). DMSP also provides special
   operations to allow easy synchronization between a user's global mail
   state and his clients' local mail states. Particular attention has
   been paid to the way in which DMSP operations act on a user's mail
   state. All DMSP operations are atomic (that is, they are guaranteed

either to succeed completely, or fail completely). A client can be
abruptly disconnected from the repository without leaving
inconsistent or damaged mail states.

Pcmail is a mail system for PCs. Its design has therefore been
heavily influenced by several characteristics unique to PCs. First,
PCs are relatively inexpensive. This means that people may own more
than one PC, perhaps putting one in an office and one at home.
Second, PCs are portable. Most PCs can be packed up and moved in the
back seat of an automobile, and a few are truly portable--about the
size of a briefcase--and battery-powered. Finally, PCs are
resource-poor. A typical PC has a small amount (typically less than
one megabyte) of main memory and little in the way of mass storage
(floppy-disk drives that can access perhaps 360 kilobytes of data).

Because PCs are relatively inexpensive and people may own more than
one, Pcmail has been designed to allow users multiple access points
to their mail state. Each Pcmail user can have several client PCs,
each of which can access the user's mail by communicating with the
repository over a network. The client PCs all maintain local copies
of the user's global mail state, and synchronize the local and global
states using DMSP.

It is possible, even likely, that many PCs will only infrequently be
connected to a network (and thus be able to communicate with the
repository). The Pcmail design therefore allows two modes of
communication between repository and client. "Interactive mode" is
used when the client PC is always connected to the network. Any
changes to the client's local mail state are immediately also made to
the repository's global mail state, and any incoming mail is
immediately transmitted from repository to client. "Batch mode" is
used by clients that have infrequent access to the repository. Users
manipulate the client's local mail state, queueing the changes as
"actions". When next connected to the repository, the actions are
transmitted, and the client's local mail state is synchronized with
the repository's global mail state.

Finally, the Pcmail design minimizes the effect of using a
resource-poor PC as a client. Mail messages are split into two
parts: a "descriptor" and a "body". The descriptor is a capsule
message summary whose length (typically about 100 bytes) is
independent of the actual message length. The body is the actual
message text, including an RFC-822 standard message header. While the
client may not have enough storage to hold a complete set of
messages, it can always hold a complete set of descriptors, thus

providing the user with at least a summary of his mail state.
Message bodies can be pulled over from the repository as client
storage becomes available.

The remainder of this document is broken up into the following
sections: first, there is a detailed description of the repository
architecture. This is followed by a description of DMSP, its
operations, and motivation for its design. A third section describes
client architecture. Another section describes a typical DMSP
session between the repository and a client. The final section
discusses the current Pcmail implementation.

3. Repository Architecture

A machine running repository code is typically a medium-to-large size
computer with a large amount of disk storage. It must also be a
permanent network site, since client PCs communicate with the
repository over a network, and rely on the repository's being
available at any time.

The repository must perform several tasks. First, and most
importantly, the repository must efficiently manage a potentially
large number of users and their mail states. Mail must be reliably
stored in a manner that makes it easy for multiple clients to access
the global mail state and synchronize their local mail states with
the global state. Second, the repository must be able to communicate
efficiently with its clients. The protocol used to communicate
between repository and client must be reliable and must provide
operations that (1) allow typical mail manipulation, and (2) support
Pcmail's distributed nature by allowing efficient synchronization
between local and global mail states. Third, the repository must be
able to process mail from sources outside the repository's own user
community (a primary outside source is the Internet). Internet mail
will arrive with a NIC RFC-822 standard message header; the recipient
names in the message must be properly translated from the RFC-822
namespace into the repository's namespace.

3.1. Management of user mail state

Pcmail divides the world into a community of users. Each user is
referred to by a user object. A user object consists of a unique
name, a password (which the user's clients use to authenticate
themselves to the repository before manipulating a global mail
state), a list of "client objects" describing those clients
belonging to the user, and a list of "mailbox objects".

A client object consists of a unique name and a status. A user

has one client object for every client he owns; a client cannot
communicate with the repository unless it has a corresponding
client object in a user's client list. Client objects therefore
serve as a means of identifying valid clients to the repository.
Client objects also allow the repository to manage local and
global mail state synchronization; the repository associates with
every global state change a list of client objects corresponding
to those clients which have not recorded the global change
locally.

A client's status is either "active" or "inactive". The
repository defines inactive clients as those clients which have
not connected to the repository within a set time period (one week
in the current Pcmail implementation). When an inactive client
does connect to the repository, the repository notifies the client
that it has been "reset". The repository resets a client by
marking all messages in the user's mail state as having changed
since the client last logged in. When the client next
synchronizes with the repository, it will receive a complete copy
of the repository's global mail state. A forced reset is
performed on the assumption that enough global state changes occur
in a week that the client would spend too much time performing an
ordinary local state-global state synchronization.

Messages are stored in mailboxes. Users can have an arbitrary
number of mailboxes, which serve both to store and to categorize
messages. Since there can be any number of mailboxes, messages
can be categorized to an arbitrarily fine degree. A mailbox
object both names a mailbox and describes its contents. Mailboxes
are identified by a unique name; their contents are described by
three numeric values. The first is the total number of messages
in the mailbox, the second is the total number of unseen messages
(messages that have never been seen by the user via any client) in
the mailbox, and the third is the next available message unique
identifier (UID). This information is stored in the mailbox
object to allow clients to get a summary of a mailbox's contents
without having to read all the messages within the mailbox.

Associated with each mailbox are an arbitrary number of message
objects. Each message is broken into two parts--a "descriptor",
which contains a summary of useful information about the message,
and a "body", which is the message text itself, including NIC
RFC-822 message header. Each message is assigned a monotonically
increasing UID based on the owning mailbox's next available UID.
Each mailbox has its own set of UIDs which, together with the
mailbox name and user name, uniquely identify the message within
the repository.

A descriptor holds the following information: the message UID, the
message size in bytes and lines, four "useful" message header
fields (the "date:", "to:", "from:", and "subject:" fields), and
two groups of eight flags each. The first group of flags is
system defined. These flags mark whether the message has never
been seen, whether it has been deleted, whether it is a forwarded
message, and whether the message has been expunged. The remaining
four flags are reserved for future use. The second group of flags
is user defined. The repository never examines these flags
internally; instead they can be used by application programs
running on the clients. Descriptors serve as an efficient means
for clients to get message information without having to waste
time retrieving the message from the repository.

3.2. Repository-to-RFC-822 name translation

"Address objects" provide the repository with a means for
translating the RFC-822-style mail addresses in Internet messages
into repository names. The repository provides its own namespace
for message identification. Any message is uniquely identified by
the triple (user-name, mailbox-name, message-UID). Any mailbox is
uniquely identified by the pair (user-name, mailbox-name). Thus
to send a message between two repository users, a user would
address the message to (user-name, mailbox-name). The repository
would deliver the message to the named user and mailbox, and
assign it a UID based on the requested mailbox's next available
UID.

In order to translate between RFC-822-style mail addresses and
repository names, the repository maintains a list of address
objects. Each address object is an association between an
RFC-822-style address and a (user-name, mailbox-name) pair. When
mail arrives from the Internet, the repository can use the address
object list to translate the recipients into (user-name,
mailbox-name) pairs and route the message correctly.

4. Communication Between Repository and Client: DMSP

The Distributed Mail System Protocol (DMSP) is a block-stream
protocol that defines and manipulates the objects mentioned in the
previous section. It has been designed to work with Pcmail's
single-repository/multiple-client model of the world. In addition to
providing typical mail manipulation functions, DMSP provides
functions that allow easy synchronization of global and local mail
states.

DMSP is implemented on top of the Unified Stream Protocol (USP),

specified in MIT-LCS Technical Memo 255. USP provides a reliable
virtual circuit block-stream connection between two machines. USP
defines a basic set of data types ("strings", "integers", "booleans",
etc.). Instances of these data types are grouped in an
application-defined order to form USP blocks. Each USP block is
defined by a numeric "block type"; a USP application can thus
interpret a block's contents based on knowledge of the block's type.
DMSP consists of a set of operations, each of which is comprised of
one or more different USP blocks that are sent between repository and
client.

A DMSP session proceeds as follows: a client begins the session with
the repository by opening a USP connection to the repository's
machine. The client then authenticates both itself and its user to
the repository with a "login" operation. If the authentication is
successful, the user performs an arbitrary number of DMSP operations
before ending the session with a "logout" operation (at which time
the connection is closed by the repository).

Because DMSP can manipulate a pair of mail states (local and global)
at once, it is extremely important that all DMSP operations are
atomic. Failure of any DMSP operation must leave both states in a
consistent, known state. For this reason, a DMSP operation is
defined to have failed unless an explicit acknowledgement is received
by the operation initiator. This acknowledgement can take one of two
basic forms, based on two broad categories that all DMSP operations
fall into. First, an operation can be a request to perform some mail
state modification, in which case the repository will acknowledge the
request with either an "ok" or a "failure" (in which case the reason
for the failure is also returned). Second, an operation can be a
request for information, in which case the request is acknowledged by
the repository's providing the information to the client. Operations
such as "delete a message" fall into the first category; operations
like "send a list of mailboxes" fall into the second category.

Following are a list of DMSP operations by object type, their block
types and arguments, and their expected acknowledgement block types.
Each DMSP block has a different number; the first digit of each block
type defines the object being manipulated: Operations numbered 5xx
are general, operations numbered 6xx are user operations, operations
numbered 7xx are client operations, operations numbered 8xx are
mailbox and address operations, and operations numbered 11xx are
message operations.

Blocks marked "=>" flow from client to repository; blocks marked "<="
flow from repository to client. If more than one block can be sent,
the choices are delimited by "or" ("|") characters.

For clarity, each block type is put in a human-
understandable form. The block number is followed by an operation
name; this name is never transmitted as part of a USP block. Block
arguments are identified by name and type, and enclosed in square
brackets. "Record" data types are described by a list of
"field-name:field-type" pairs contained in square brackets. "Choice"
data types are described by a list of "tag:tag-name" pairs contained
in square brackets. USP data types are abbreviated as follows:

Primitive data types:

   - string: str

   - cardinal: card

   - long-cardinal: Lcard

   - integer: int

   - long-integer: Lint

   - boolean: bool

Compound data types:

   - sequence: SEQ

   - array: AR

   - record: REC

   - choice: CH

4.1. General operations

   The first group of DMSP operations perform general functions that
   operate on no one particular class of object. DMSP has six
   general operations, which provide the following services:

   If either a client or the repository thinks the other is
   malfunctioning, they can send an "abort-request". An
   abort-request is never acknowledged; after the request is sent,
   the sender immediately closes the USP connection and returns
   control to its application.

      => 503 (abort-request) [why:str]

DMSP provides a limited remote debugging facility via the
"start-debug" and "end-debug" operations. When a client sends a
"start-debug" request, the repository enables its idea of
remote-debugging. The exact definition of remote debugging is
implementation dependent; the current repository implementation
simply writes debugging information to a special file. The
"end-debug" request disables remote debugging.

```
=> 504 (start-debug) []

<= 500 (ok) [] |
   501 (failure) [why:str]

or

=> 505 (end-debug) []

<= 500 (ok) []
```

In order to prevent protocol version skew between clients and the
repository, DMSP provides a "send-version" operation. The client
supplies its DMSP version number as an argument; the operation
succeeds if the supplied version number matches the repository's
DMSP version number. It fails if the two version numbers do not
match.

```
=> 506 (send-version) [version-number:card]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

DMSP also provides clients with the ability to send an arbitrary
text message to the repository. The "log-message" operation takes
as an argument a string of arbitrary length; the repository
accepts the string; what is done with the string is
implementation-dependent.

```
=> 507 log-message[message:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

Finally, users can send mail to other users via the "send-message"
operation. The message must have an Internet-style header as
defined by NIC RFC-822. The repository takes the message and
distributes it to the mailboxes specified on the "to:", "cc:", and
"bcc:" fields of the message header. If one or more of the

mailboxes exists outside the repository's user community, the
repository is responsible for handing the message to a local SMTP
server.

An OK block is sent from the repository only if the entire message
was successfully transmitted. If the message was destined for the
Internet, the send-message operation is successful if the message
was successfully transmitted to the local SMTP server.

```
=> 508 (send-message) [message:SEQ[str]]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

4.2. User operations

The next series of DMSP operations manipulates user objects. The
most common of these operations are "login" and "logout". A
client must perform a login operation before being able to access
a user's mail state. A DMSP login block contains five items: (1)
the user's name, (2) the user's password, (3) the name of the
client performing the login, (4) a flag telling the repository to
create a client object for the client if one does not exist, and
(5) a flag set to TRUE if the client wishes to operate in "batch
mode" and FALSE if the client wishes to operate in "interactive"
mode. The flag value allows the repository to tune internal
parameters for either mode of operation.

The repository can return either an OK block (indicating
successful authentication), a FAILURE block (indicating failed
authentication), or a FORCE-RESET block. This last is sent if the
client logging in has been marked as "inactive" by the repository
(clients are marked inactive if they have not connected to the
repository in over a week). The FORCE-RESET block indicates that
the client should erase its local mail state and pull over a
complete version of the repository's mail state. This is done on
the assumption that so many mail state changes have been made in a
week that it would be inefficient to perform a normal
synchronization.

```
=> 600 (login) [user:str, password:str, client:str,
                create-client-object?:bool,
                batch-mode-flag:bool]

<= 500 (ok) [] |
   501 (failure) [why:str] |
   705 (force-client-reset) []
```

When a client is finished interacting with the repository, it
performs a logout operation. This allows the repository to
perform any necessary cleanup before closing the USP connection.

```
=> 601 (logout) []

<= 500 (ok) []
```

DMSP also provides "add-user" and "remove-user" operations, which
allow system administrators to remotely add new users to, and
remove users from, the repository. These operations are
privileged; the repository authenticates the user requesting the
operation before performing an add-user or remove-user operation.
Both operations require the name of the user to be added or
removed; the add-user operation also requires a default password
to assign the new user.

```
=> 602 (add-user) [user:str, password:str]

<= 500 (ok) [] |
   501 (failure) [why:str]

=> 603 (remove-user) [user:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

A user can change his password via the "set-password" operation.
The operation works much the same as the UNIX change-password
operation, taking as arguments the user's current password and a
desired new password. If the current password given matches the
user's current password, the user's current password is changed to
the new password given.

```
=> 604 (set-password) [old-password:str,
                        new-password:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

4.3. Client operations

   DMSP provides four operations to manipulate client objects. The
   first, "list-clients", tells the repository to send the user's
   client list to the requesting client. The list takes the form of
   a series of (name, status pairs).

      => 700 (list-clients) []

      <= 701 (client-list) [client-list:SEQ[
                               REC[name:str, status:card]]]

   The "add-client" operation allows a user to add a client object to
   his list of client objects. Although the login operation
   duplicates this functionality via the "create-this-client?" flag,
   the add-client operation is a useful means of creating a number of
   new client objects while logged into the repository via an
   existing client. The add-client operation requires the name of
   the client to add.

      => 702 (add-client) [client:str]

      <= 500 (ok) [] |
         501 (failure) [why:str]

   The most common failure mode for this operation is an attempt to
   add a client that already exists.

   The "remove-client" operation removes an existing client object
   from a user's client list. The client being removed can be the
   client requesting the operation. The remove-client operation
   requires the name of the client to remove.

      => 703 (remove-client) [client:str]

      <= 500 (ok) [] |
         501 (failure) [why:str]

   The most common failure mode here is an attempt to remove a
   non-existent client. This is a typical failure mode for any DMSP
   operation which operates on a named object.

   The last client operation, "reset-client", causes the repository
   to mark all messages in the user's mail state as having changed
   since the client last logged in. When a client next synchronizes
   with the repository, it will end up receiving a complete copy of
   the repository's global mail state. This is useful for two

reasons. First, a client's local mail state could easily become
lost or damaged, especially if it is stored on a floppy disk.
Second, if a client has been marked as inactive by the repository,
the reset-client operation provides a fast way of resynchronizing
with the repository, assuming that so many differences exist
between the local and global mail states that a normal
synchronization would take far too much time.

```
=> 704 (reset-client) [client:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

4.4. Mailbox operations

DMSP supports five operations that manipulate mailbox objects.
First, "list-mailboxes" has the repository send to the requesting
client information on each mailbox. This information consists of
the mailbox name, total message count, unseen message count, and
"next available UID". This operation is useful in synchronizing
local and global mail states, since it allows a client to compare
the user's global mailbox list with a client's local mailbox list.
The list of mailboxes also provides a quick summary of each
mailbox's contents without having the contents present.

```
=> 800 (list-mailboxes) []

<= 801 (mailbox-list) [mailbox-list:SEQ[
                       REC[mailbox:str,
                           next-UID:Lcard,
                           num-msgs:card,
                           num-unseen-msgs:card]]]
```

The "add-mailbox" has the repository create a new mailbox and
attach it to the user's list of mailboxes. An address object
binding the (user-name, mailbox-name) pair to an RFC-822-style
address is automatically created and placed in the repository's
list of address objects. This allows mail coming from the
Internet to be correctly routed to the new mailbox.

```
=> 802 (add-mailbox) [mailbox:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

"Remove-mailbox" removes a mailbox from the user's list of
mailboxes. All messages within the mailbox are also deleted and

permanently removed from the system. Any address objects binding
the mailbox name to RFC-822-style mailbox addresses are also
removed from the system.

```
=> 803 (remove-mailbox) [mailbox:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

DMSP also has an "expunge-mailbox" operation. Any message can be
deleted and "undeleted" at will. Deletions are made permanent by
performing an expunge-mailbox operation. The expunge operation
causes the repository to look through a named mailbox, removing
from the system any messages marked "deleted".

```
=> 808 expunge-mailbox[mailbox:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

Finally, "reset-mailbox" causes the repository to mark all the
messages in a named mailbox as having changed since the current
client last logged in. When the client next synchronizes with the
repository, it will receive a complete copy of the named mailbox's
mail state. This operation is merely a more specific version of
the reset-client operation (which allows the client to pull over a
complete copy of the user's global mail state). Its primary use
is for mailboxes whose contents have accidentally been destroyed
locally.

```
=> 809 (reset-mailbox) [mailbox:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

4.5. Address operations

DMSP provides three operations that allow users to manipulate
address objects. First, the "list-address" operation returns a
list of address objects associated with a particular (user-name,
mailbox-name) pair.

```
=> 804 (list-addresses) [mailbox:str]

<= 501 (failure) [why:str] |
   805 (address-list) [address-list:SEQ[str]]
```

        The "add-address" operation adds a new address object that
        associates a (user-name, mailbox-name) pair with a given
        RFC-822-style mailbox address.

            => 806 (add-address) [mailbox:str,
                                   RFC-822-mail-address:str]

            <= 500 (ok) [] |
               501 (failure) [why:str]

        Finally, the "remove-address" operation destroys the address
        object binding the given RFC-822-style mail address and the given
        (user-name, mailbox-name) pair.

            => 807 (remove-address) [mailbox:str,
                                     RFC-822-mail-address:str]

            <= 500 (ok) [] |
               501 (failure) [why:str]

    4.6. Message operations

        The most commonly-manipulated Pcmail objects are messages; DMSP
        therefore provides special message operations to allow efficient
        synchronization, as well as a set of operations to perform
        standard message-manipulation functions. In the following
        paragraphs, the terms "message" and "descriptor" will be used
        interchangeably.

        A client can request a particular message's flag values with the
        "get-descriptor-flags" operation. The repository sends over an
        array of boolean values, eight of which are system defined, and
        eight of which are user defined and ignored by the repository.

            => 1100 (get-descriptor-flags) [mailbox:str,
                                            uid:Lcard]

            <= 1101 (descriptor-flags) [flags:SEQ[bool]] |
               501 (failure) [why:str]

        A user may request a series of descriptors with the
        "get-descriptors" operation. The series is identified by a pair
        of message UIDs, representing the lower and upper bounds of the
        list. Since UIDs are defined to be monotonically increasing
        numbers, a pair of UIDs is sufficient to completely identify the
        series of descriptors. The repository returns a sequence of
        "choices". Elements of the sequence can either be descriptors, in

which case the choice is tagged as a descriptor, or they can be
notification that the requested message has been expunged
subsequent to the client's last connection to the repository.

```
    => 1102 (get-descriptors) [mailbox:str,
                               low-UID:Lcard,
                               high-UID:Lcard]


    <= 501 (failure) [why:str] |
       1103 (descriptor-list) [descriptor-list:SEQ[ CH[
                               expunged[uid:Lcard]
                               descriptor[REC[UID:Lcard,
                                               flags:SEQ[bool],
                                               from-field:str,
                                               to-field:str,
                                               date-field:str,
                                               subject-field:str,
                                               num-bytes:Lcard,
                                               num-lines:Lcard]
                                          ]]]]
```

The "get-changed-descriptors" operation is intended for use during
state synchronization. Whenever a descriptor changes state (is
deleted, for example), the repository notes those clients which
have not yet recorded the change locally. Get-changed-descriptors
has the repository send to the client a given number of
descriptors which have changed since the client's last
synchronization. The list sent begins with the earliest-changed
descriptor.

```
    => 1105 (get-changed-descriptors) [mailbox:str,
                                       max-to-send:card]

    <= 501 (failure) why:str] |
       1103 (descriptor-list) [descriptor-list:SEQ[
               CH[
                  expunged[uid:Lcard]
                  descriptor[REC[UID:Lcard,
                                 flags:SEQ[bool],
                                 from-field:str,
                                 to-field:str,
                                 date-field:str,
                                 subject-field:str,
                                 num-bytes:Lcard,
                                 num-lines:Lcard]
                            ]]]]
```

Once the changed descriptors have been looked at, a user will want
to inform the repository that the current client has recorded the
change locally. The "reset-changed-descriptors" causes the
repository to mark as "seen by current client" a given number of
changed descriptors, starting with the changed descriptor with
lowest UID.

```
=> 1106 (reset-changed-descriptors) [
        mailbox:str,
        number-to-reset:card]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

Message bodies are transmitted from repository to user with the
"get-message-text" operation. The separation of "get-descriptors"
and "get-message-text" operations allows clients with small
amounts of disk storage to obtain a small message summary (via
"get-descriptors" or "get-changed-descriptors") without having to
pull over the entire message.

```
=> 1107 (get-message-text)[mailbox:str,
                           uid:Lcard]

<= 501 (failure) [why:str] |
   1110 (message) [message:SEQ[str]]
```

Frequently, a message may be too large for some clients to store
locally. Users can still look at the message contents via the
"print-message" operation. This operation has the repository send
a copy of the message to a named printer. The printer name need
only have meaning to the particular repository implementation;
DMSP transmits the name only as a means of identification.

```
=> 1108 (print-message) [mailbox:str,
                         uid:Lcard,
                         printer-name:str]

<= 500 (ok) [] |
   501 (failure) [why:str]
```

The user can set and clear any of the 16 descriptor flags with the
"set-flag" operation. The desired flag is set or cleared
according to the operation arguments.

```
    => 1109 (set-flag) [mailbox:str,
                        uid:Lcard,
                        flag-number:card,
                        flag-setting:bool]

    <= 500 (ok) [] |
       501 (failure) [why:str]
```

Copying of one message into another mailbox is accomplished via
the "copy-message" operation.

```
    => 1111 (copy-message) [source-mailbox:str,
                            target-mailbox:str,
                            source-uid:Lcard]

    <= 500 (ok) [] |
       501 (failure) [why:str]
```

5. Client Architecture

   Clients are typically PCs; Pcmail's architecture must therefore take
   into account several characteristics common to PCs. First, PCs are
   cheap, therefore a user may well have more than one. Second, they
   are portable, therefore they are not expected to be constantly tied
   into a network. Finally, they are resource-poor, so they are not
   expected to be able to store a significant amount of state
   information locally. The following subsections describe the
   particular parts of Pcmail's client architecture that address these
   three characteristics.

   5.1. Multiple clients

      The fact that Pcmail users may own more than one PC forms the
      rationalization for the multiple client model that Pcmail uses. A
      Pcmail user may have a PC client at home, a PC at an office, and
      maybe even a third portable PC. Each client maintains a separate
      copy of the user's mail state, hence Pcmail's distributed nature.
      The notion of separate clients allows Pcmail users to access mail
      state from several different locations.

   5.2. Synchronization

      Since PCs are fairly portable, the likelihood of a PC's being
      always connected to a network is relatively small. This is
      another reason for each client's maintaining a local copy of a
      user's mail state. The user can then manipulate the local mail
      state while not connected to the network (and the repository).
      This immediately brings up the problem of synchronization between
      local and global mail states. The repository is continually in a
      position to receive global mail state updates, either in the form
      of incoming mail, or in the form of changes from other clients. A
      client that is not always connected to the net cannot immediately
      receive the global changes. In addition, the client's user can
      make his own changes on the local mail state.

      Pcmail's architecture permits efficient synchronization between
      client local mail states and the repository's global mail state.
      Each client is identified in the repository by a client object
      attached to the user. This object forms the basis for
      synchronization between local and global mail states. Some of the
      less common state changes include the adding and deleting of user
      mailboxes and the adding and deleting of address objects.
      Synchronization of these changes is performed via DMSP list
      operations, which allow clients to compare their local versions of
      mailbox and address object lists with the repository's global
      version and make any appropriate changes. The majority of
      possible changes to a user's mail state are in the form of changed
      descriptors. Since most users will have a large number of
      messages, and message states will change relatively often, special
      attention needs to be paid to message synchronization.

      An existing descriptor can be changed in one of two ways: first,
      one of its sixteen flags values can be changed (this encompasses
      reading an unseen message, deleting a message, and expunging a
      message). The second way to change a descriptor is via the
      arrival of incoming mail or the copying of a message from one
      mailbox to another. Both result in a new message being added to a
      mailbox.

      In both the above cases, synchronization is required between the
      repository and every client that has not previously noted a
      change. To keep track of which clients have noticed a global mail
      state change and changed their local states accordingly, each
      descriptor has associated with it a (potentially empty) "update
      list" of client objects. The list identifies those clients which
      have not yet recorded a change to that descriptor's state.

When a client connects to the repository, it executes a DMSP
"get-changed-descriptors" operation. This causes the repository
to return a list of all descriptor objects that have the
requesting client on their update list. As the client receives
the changed descriptors, it can store them locally, thus updating
the local mail state. After a changed descriptor has been
recorded, the client uses the DMSP "reset-descriptors" operation
to remove itself from the descriptor's update list. That
descriptor will now not be sent to the client unless (1) it is
explicitly requested, or (2) it changes again.

In this manner, a client can run through its user's mailboxes,
getting all changed descriptors, incorporating them into the local
mail state, and marking the change as recorded.

5.3. Batch operation versus interactive operation

Because of the portable nature of most PCs, they may not always be
connected to the repository. Since each client maintains a local
mail state, Pcmail users can manipulate the local state while not
connected to the repository. This is known as "batch" operation,
since all changes are recorded by the client and made to the
repository's global state in a batch, when the client next
connects to the repository. Interactive operation occurs when a
client is always connected to the repository. In interactive
mode, changes made to the local mail state are immediately
propagated to the global state via DMSP operations.

In batch mode, interaction between client and repository takes the
following form: the client connects to the repository and sends
over all the changes made by the user to the local mail state.
The repository changes its global mail state accordingly. When all
changes have been processed, the client begins synchronization, to
incorporate newly-arrived mail, as well as mail state changes by
other clients, into the local state.

In interactive mode, since local changes are immediately
propagated to the repository, the first part of batch-type
operation is eliminated. The synchronization process also
changes; interactive clients can periodically poll the repository
for a list of changes, synchronizing a small amount at a time.

5.4. Message summaries

   Since PCs are assumed to have little in the way of disk storage, a
   given client may never have enough room for a complete local copy
   of a user's global mail state. This means that Pcmail's client
   architecture must allow user's to obtain a clear picture of their
   mail state without having all their messages present.

   Descriptors provide message information without taking up large
   amounts of storage. Each descriptor contains a summary of
   information on a message. This information includes the message
   UID, its length in bytes and lines, its status (encoded in the
   eight system-defined and eight user-defined flags), and portions
   of its RFC-822 header (the "to:", "from:", "subject:" and "date:"
   fields). All of this information can be encoded in a small
   (around 100 bytes) data structure whose length is independent of
   the size of the message it describes.

   Any client should be able to store a complete list of message
   descriptors with little problem. This allows a user to get a
   complete picture of his mail state without having all his messages
   present locally. Short messages can reside on the client, along
   with the descriptors, and long messages can either be printed via
   the DMSP print-message operation, or specially pulled over via the
   fetch-message-text operation.

6. Typical Client-Repository Interaction

   The following example describes a typical communication session
   between the repository and a client. The client is one of three
   belonging to user "Fred". Its name is "office-client", and since
   Fred uses the client regularly to access his mail, the client is
   marked as "active". Fred has two mailboxes: "main" is where all of
   his current mail is stored; "archive" is where messages of lasting
   importance are kept. The example will run through a simple
   synchronization operation followed by a series of typical mail state
   manipulations. Typically, the synchronization will be performed by
   an application program that connects to the repository, logs in,
   synchronizes, and logs out.

   For the example, all DMSP operations will be shown in a user-readable
   format. In reality, the operations would be sent as a stream of USP
   blocks consisting of a block-type number followed by a stream of
   bytes representing the block's arguments. Both the block name and its
   number are included for convenience.

In order to access his global mail state, the client software must
authenticate Fred to the repository; this is done via the DMSP login
operation:

```
600 (login) ["fred", "ajyr63ywg", "office-client",
             FALSE, FALSE]
```

This tells the repository that Fred is logging in via
"office-client", and that "office-client" is identified by an
existing client object attached to Fred's user object. The second
login block argument in an encrypted version of Fred's password. The
final argument tells the repository that Fred's client is not
operating in batch mode but rather in interactive mode.

Fred's authentication checks out, so the repository logs him in,
acknowledging the login request with an OK block.

Now that Fred is logged in, he wants to bring
"office-client"'s local mail state up to date. To do this, the
client program asks for an up-to-date list of mailboxes:

```
800 (list-mailboxes) []
```

The repository replies with:

```
801 (mailbox-list) [["main", 10, 1, 253],
                    ["archive", 100, 0, 101]]
```

This tells the client that there are two mailboxes, "main" and
"archive". "Main" has 10 messages, one of which is unseen. The next
incoming message will be assigned a UID of 253. "Archive", on the
other hand, has 100 message, none of which are unseen. The next
message sent to "archive" will be assigned the UID 101. There are no
new mailboxes in the list (if there were, the client program would
create them. On the other hand, if some mailboxes in the client's
local list were not in the repository's list, the program would
assume them deleted by another client and delete them locally as
well).

To synchronize the client need only look at each mailbox's contents
to see if (1) any new mail has arrived, or (2) if Fred changed any
messages on one of his other two clients subsequent to
"office-client"'s last connection to the repository.

The client asks for any changed descriptors via the
"get-changed-descriptors" operation. It requests at most ten changed
descriptors since storage is very tight on "office-client".

```
    1105 (get-changed-descriptors) ["main", 10]
```

The repository responds with:

```
    1103 (descriptor-list) [[descriptor[
                            6,
                            [T T F F F F F F F F F F F F F F],
                            "Fred@borax",
                            "Joe@fab",
                            "Wed, 23 Jan 86 11:11 EST",
                            "tomorrow's meeting",
                            621,
                            10]]
                           [descriptor[
                            10,
                            [F T F F F F F F F F F F F F F F],
                            "Fred",
                            "Freds-secretary",
                            "Fri, 25 Jan 86 11:11 EST",
                            "Monthly progress report",
                            13211,
                            350]]
                          ]
```

The first descriptor in the list is one which Fred deleted on another
client yesterday. "Office-client" marks the local version of the
message as deleted. The second descriptor in the list is a new one.
"Office-client" adds the descriptor to its local list. Since both
changes have now been recorded locally, the descriptors can be reset:

```
    1106 (reset-descriptors) ["main", 2]
```

The repository clears each descriptor's update vector bit
corresponding to "office-client"'s client object. "Main" has now
been synchronized. The client now turns to Fred's "archive" mailbox
and asks for the first ten changed descriptors.

```
    1105 (get-changed-descriptors) ["archive", 10]
```

The repository responds with

```
    1103 (descriptor-list) []
```

The zero-length list tells "office-client" that no descriptors have
been changed in "archive" since its last synchronization. No new
synchronization needs to be performed.

Fred's client is now ready to pull over the new message so Fred can
read it. The message is 320 lines long; there might not be
sufficient storage on "office-client" to hold the new message. The
client tries anyway:

```
1107 (fetch-message-text) ["main", 10]
```

The repository begins transmitting the message:

```
1110 (message) ["From: Fred's-secretary",
                "To: Fred",
                "Subject: Monthly progress report",
                "Date: Fri, 25 Jan 86 11:11 EST",
                "",
                "Dear Fred,",
                "Here is this month's progress report",
                ...
                ]
```

Halfway through the message transmission, "office-client" runs out of
disk space. Because all DMSP operations are defined to be atomic,
the portion of the message already transmitted is destroyed locally
and the operation fails. "Office-client" informs Fred that the
message cannot be pulled over because of a lack of disk space. The
synchronization process is now finished and Fred's client logs out.

```
601 (logout) []
```

The repository does any housecleaning it needs to do, acknowledges
the logout request, and closes the USP connection.

7. A Current Pcmail Implementation

   The following section briefly describes a current implementation of
   Pcmail that services a small community of users. The Pcmail
   repository runs under UNIX on a DEC VAX-750 connected to the
   Internet. The clients are IBM PCs, XTs, and ATs. The network
   software that communicates with the repository allows only
   "batch-mode" operation. Users make local state changes, which are
   queued until the client connects to the repository. At that time,
   the changes are performed and the local and global states
   synchronized. The client then disconnects from the repository.

   Users access and modify their local mail state via a user interface
   program. The program uses windows and a full-screen mode of
   operation. Users are given a rich variety of commands to operate on

individual messages as well as mailboxes. The interface allows use
of any text editor to compose messages, and adds features of its own
to make RFC-822-style header composition easier.

Synchronization and the processing of queued changes is performed by
a separate program, which the user runs whenever he wishes. The
program takes any actions queued while operating the user interface,
and converts them into DMSP operations. All queued changes are made
before any synchronization is performed.

The limitation of client operation to batch mode was made for the
following reasons: first, the implementation is slanted toward use of
portable computers as clients. These computers are rarely connected
to the network, making interactive mode unnecessary. Those clients
that are constantly connected to the network run slightly less
efficiently than they could (since users must make changes locally
and then run the action-processing/synchronization program, rather
than simply making changes interactively).

Another important reason for limiting operation to batch mode is that
it allows a very simple locking scheme to prevent problems raised by
concurrent state updates. A user may have several clients; it is
therefore likely that the repository could get into a variety of
inconsistent states as different clients try to change the
repository's global mail state at the same time. To prevent these
inconsistencies, a user's mail state is locked as soon as a client
connects to the repository. The lock is released when the client
disconnects from the repository. This locking scheme is simple to
implement, but makes interactive-mode operation very cumbersome: if a
user remains constantly connected to the network (i.e. in interactive
mode), the repository would be unavailable to any of the user's other
clients for an unacceptable length of time.

8. Conclusions

Pcmail is now used by a small community of people at the MIT
Laboratory for Computer Science. The repository design works well,
providing a fairly efficient means of storing and maintaining mail
state for several users. Members of another research group at LCS
are currently working on a replicated, scaleable version of the
repository designed to support a very large community of users with
high availability. This repository also uses DMSP and has
successfully communicated with clients that use the current
repository implementation. DMSP therefore seems to be useable over
several flavors of repository design. The clients, being PCs, are
unfortunately very limited in the way of resources, making local mail
state manipulation difficult at times. Synchronization is also

relatively time consuming due to the low performance of the PCs. The
"batch-mode" of client operation is very useful for portable
computers that spend a large percentage of their time unplugged and
away from a network. It is somewhat less useful for the majority of
the clients, which are always connected to the network and could make
good use of an "interactive-mode" state manipulation.

I. DMSP Protocol Specification

   Following is a list of DMSP block types and DMSP operations by object
   type. Again, "=>" marks blocks flowing from client to repository;
   "<=" marks blocks flowing from repository to client.

      General operations:

          => or <= 503 (abort-request) [why:str]
          (no acknowledgement)

          => 504 (start-debug) []
          <= 500 (ok) [] |
             501 (failure) [why:str]

          => 505 (end-debug) []
          <= 500 (ok) []

          => 506 (send-version) [version:card]
          <= 500 (ok) [] |
             501 (failure) [why:str]

          => 507 (log-message) [message:str]
          <= 500 (ok) [] |
             501 (failure) [why:str]

          => 508 (send-message) [message:seq[str]]
          <= 500 (ok) [] |
             501 (failure) [why:str]

      User operations:

          => 600 (login) [name:str, password:str,
                          client:str, create-client-object?:bool
                          batch-mode-flag:bool]
          <= 500 (ok) [] |
             501 (failure) [why:str] |
             705 (force-client-reset) []

          => 601 (logout) []
          <= 500 (ok) []

          => 602 (add-user) [name:str, password:str]
          <= 500 (ok) [] |
             501 (failure) [why:str]

```
        => 603 (remove-user) [user:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

        => 604 (set-password) [old:str, new:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

    Client operations:

        => 700 (list-clients) []
        <= 701 (client-list) [client-list:seq[
                              rec[name:str], status:card]]

        => 702 (add-client) [client:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

        => 703 (remove-client) [client:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

        => 704 (reset-client) [client:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

    Mailbox operations:

        => 800 (list-mailboxes) []
        <= 801 (mailbox-list) [mailbox-list:seq[
                              rec[mailbox:str,
                                  next-uid:lcard,
                                  num-msgs:card,
                                  num-unseen-msgs:card]]]

        => 802 (add-mailbox) [mailbox:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

        => 803 (remove-mailbox) [mailbox:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]

        => 808 (expunge-mailbox) [mailbox:str]
        <= 500 (ok) [] |
           501 (failure) [why:str]
```

```
       => 809 (reset-mailbox) [mailbox:str]
       <= 500 (ok) [] |
          501 (failure) [why:str]

   Address operations:

       => 804 (list-addresses) [mailbox:str]
       <= 501 (failure) [why:str] |
          805 (address-list) [address-list:seq[str]]

       => 806 (add-address) [mailbox:str, address:str]
       <= 500 (ok) [] |
          501 (failure) [why:str]

       => 807 (remove-address) [mailbox:str, address:str]
       <= 500 (ok) [] |
          501 (failure) [why:str]

   Message operations:

       => 1100 (get-descriptor-flags) [mailbox:str, uid:lcard]
       <= 1101 (descriptor-flags) [flags:seq[bool]] |
          501 (failure) [why:str]

       => 1102 (get-descriptors) [mailbox:str,
                                  low-uid:lcard,
                                  high-uid:lcard]
       <= 501 (failure) [why:str] |
          1103 (descriptor-list) [descriptor-list:seq[
              ch[
                expunged[uid:lcard],
                descriptor[rec[uid:lcard,
                               flags:seq[bool],
                               from-field:str,
                               to-field:str,
                               date-field:str,
                               subject-field:str,
                               nun-bytes:lcard,
                               num-lines:lcard]
                    ]]]]
```

```
=> 1105 (get-changed-descriptors) [mailbox:str,
                                    max-to-send:card]
<= 501 (failure) [why:str] |
   1103 (descriptor-list) [descriptor-list:seq[
         ch[
            expunged[uid:lcard],
            descriptor[rec[uid:lcard,
                           flags:seq[bool],
                           from-field:str,
                           to-field:str,
                           date-field:str,
                           subject-field:str,
                           num-bytes:lcard,
                           num-lines:lcard]
              ]]]]

=> 1106 (reset-changed-descriptors) [
               mailbox:str,
               start-uid:lcard,
               end-uid:lcard]
<= 500 (ok) [] |
   501 (failure) [why:str]

=> 1107 (get-message-text) [mailbox:str,
                            uid:lcard]
<= 501 (failure) [why:str] |
   1110 (message) [message:seq[str]]

=> 1108 (print-message) [mailbox:str,
                         uid:lcard,
                         printer-name:str]
<= 500 (ok) [] |
   501 (failure) [why:str]

=> 1109 (set-flag) [mailbox:str,
                    uid:lcard,
                    flag-number:card,
                    flag-setting:bool]
<= 500 (ok) [] |
   501 (failure) [why:str]

=> 1111 copy-message[source-mailbox:str,
                     target-mailbox:str,
                     source-uid:lcard]
<= 500 (ok) [] |
   501 (failure) [why:str]
```

DMSP block types by number

General block types

```
ok                     500
failure                501
abort-request          503
start-debug            504
end-debug              505
send-version           506
log-message            507
send-message           508
```

User operation block types

```
login                  600
logout                 601
add-user               602
remove-user            603
set-password           604
```

Client operation block types

```
list-clients           700
client-list            701
add-clien              702
remove-client          703
reset-client           704
force-client-reset     705
```

Mailbox operation block types

```
list-mailboxes         800
mailbox-list           801
add-mailbox            802
remove-mailbox         803
expunge-mailbox        808
reset-mailbox          809
```

Address operation block types

        list-addresses   804
        address-list   805
        add-address   806
        remove-address   807

Message operation block types

        get-descriptor-flags   1100
        descriptor-flags   1101
        get-descriptors   1102
        descriptor-list   1103
        get-changed-descriptors 1105
        reset-changed-descriptors 1106
        get-message-text   1107
        print-message   1108
        set-flag   1109
        message   1110
        copy-message   1111