The Grapevine Interface

by Andrew Birrell

Edition 2
January 1982

Abstract: Grapevine is a multi-computer system on the Xerox research internet. It provides facilities for the delivery of digital messages such as computer mail; for naming people, machines and services; for authenticating people and machines; and for locating services on the internet.

This document describes the semantics of the services provided by Grapevine, and the protocols that allow access to these services over the internet, in sufficient detail for a reader to program software that will use Grapevine.

Note: Grapevine is the outcome of a research project. The services and protocols described here are not part of a Xerox product, and are not related to the Xerox Network Systems protocols.

XEROX
PALO ALTO RESEARCH CENTER
COMPUTER SCIENCE LABORATORY
3333 Coyote Hill Road / Palo Alto / California 94304

1. Introduction

Grapevine is a distributed system spanning multiple computers, providing distributed and replicated services to *clients* on the Xerox research internet. (We use the term client to mean software making use of some facility, and *user* to mean human users of software.) The services provided include message delivery, resource location, authentication and access controls.

The Grapevine system was designed and implemented by Andrew Birrell, Roy Levin, Roger Needham and Michael Schroeder, with considerable help from several other members of the Computer Science Laboratory in the Xerox Palo Alto Research Center.

A general description of Grapevine is given in the paper "Grapevine: an Exercise in Distributed Computing", which was presented at the 8th Symposium on Operating System Principles in December 1981, and which is to be published in CACM (April 1981). Read that paper thoroughly before attempting to read this document. Before proceeding with this document, you should also have some familiarity with the PUP internet environment.

This document specifies the client interfaces to the Grapevine system: it treats the Grapevine system as a "black box", and defines the semantics that a client of the black box can observe and how a client may interact with the black box. The intention is to define the semantics and the relevant communication protocols in sufficient detail that a suitably skilled reader could proceed to program packages and systems that use Grapevine. There is no attempt here to explain how the inside of the black box is constructed or operates. The communication protocols allow a client to interact with particular Grapevine computers, but to take full advantage of Grapevine the client must also understand how the services provided by Grapevine are distributed and replicated. This document does not attempt to provide a global view of the Xerox internet message facilities, nor is it intended as a guide for those who administer Grapevine or other parts of the message system. The Grapevine system does not include the IFS mail servers, nor user interface programs such as Laurel.

We distinguish between a *service* and a *server* which provides an *instance* of that service. Grapevine provides several services; various computers within Grapevine provide instances of these services, or of part of these services. For example, *accepting mail for subsequent delivery* is a service; each Grapevine *Mail Server* provides an instance of that service. Accepting mail for delivery is thus a *replicated* service: it is provided by multiple computers, and any of these computers is adequate for providing the service. A client wishing to submit mail may submit it equally well to any of the Grapevine mail servers. A more complicated example is the Grapevine *Name Registration Service*, which is provided by the Grapevine *Registration Servers*. As will be seen, no one of the registration servers provides the entirety of this service, because each registration server knows about only some subset of the registered names. On the other hand, multiple registration servers know about each name. Thus the Grapevine name registration service is *distributed* as well as being replicated.

The services provided by Grapevine come in three major groups. Firstly, the Registration servers provide a naming database. This database is distributed and replicated. It is organized to allow for distributed administration of the names. The database provides its clients with a name-to-value mapping, and the ability to make changes to that mapping. This naming database is intended to be used for many purposes, including resource location and authentication in connection with the use of the Grapevine communication protocols. Secondly, the Mail Servers provide a message system. The facilities provided are the submission of a *message* together with a list of intended recipients. The mail servers will forward the message to a site convenient for each recipient, where the message is buffered until the recipient cares to retrieve the message. Thirdly, the software provided with the Grapevine servers provides various administrative facilities which are useful in connection with

running the physical computers on which the servers exist. These administrative facilities are in addition to the Registration Service facilities for updating the naming database: these administrative facilities are concerned with running individual computers. The administrative facilities are a log, a terminal interface to inspect and change the servers' state, and a terminal display on the physical computer.

This document proceeds in several parts. Section two describes the names and values contained in the naming database. Section three describes the message delivery facility provided by the Mail Servers. Section four specifies the network protocols for accessing individual servers. Section five discusses algorithms which allow you to provide transparency of replication, distribution and failure. Lastly, the administrative facilities are described.

2. The Naming Database

The Grapevine naming database is provides several facilities. These include: naming message recipients, handling distribution lists for messages, naming and locating services and servers, authenticating users and servers, providing public access control and authentication services, and configuring the naming database and the message delivery system. This section describes the naming database; subsequent sections describe how to use the database for these various purposes.

The naming database often is concerned with names or values which are *strings* of characters. In all cases, a string is restricted to be no more than 64 characters. The communication protocols consider an attempt to transmit a string of more than 64 characters to be a protocol violation.

A *Name* is a string of characters. The string usually contains the character ".". Only the last "." in the string is interesting. The part of the string after the last "." is known as the *registry*, and the part of the string before that "." is known as the *simple-name*. Thus a Grapevine name may be considered as

simple-name . registry

In the degenerate case where a name contains no ".", the entire name should be considered to be the registry. Names which differ only in the case of letters are considered to be equal.

Each name in the database is associated with a value. There are two *types* of name, each with its associated form of value; these types are *individual* and *group*. The type of a name is specified when the database entry is created, and may not be changed until the name is deleted.

The following information constitutes the value associated with a name whose type is individual. The precise bit patterns representing these values are of concern only when we discuss the actual communication protocols.

A *timestamp*. Timestamps are used extensively in Grapevine as unique identifiers. They are formed by concatenating a host identification with a clock value. The timestamp associated with a database entry has the property that if the value of the entry changes, then so does the timestamp. The timestamp may also change at other times. The timestamp is intended to assist a client in maintaining cached information that was derived from the database; it is also used internally by the registration servers.

A *password*. Universally within Grapevine, passwords are represented by 64-bit values. Passwords are used to allow authentication of individuals. They are also intended for use in encryption-based security protocols, although no such protocols are available at present. By convention, these values are derived from text strings by the following algorithm. Your users will probably be displeased if you use any different algorithm.

```
MakeKey: PROC[text: STRING]
         RETURNS[key: PACKED ARRAY [0..8) OF [0..256)] =
  BEGIN
  key _ ALL[0];
  FOR i: CARDINAL IN [0..text.length) DO
      j: [0..LENGTH[key]) = i MOD LENGTH[key];
      c: [0..128) = LOOPHOLE[LowerCase[text[i]]];
      key[j] _ BITXOR[ key[j], BITSHIFT[c, 1] ];
  ENDLOOP;
  END;
```

A *connect-site*. A connect-site is a string. Although the registration servers place no restriction on the contents of the string, it usually represents the address of a computer.

A *forwarding-list*. This is an ordered list of strings (usually names). The intended semantics of a forwarding-list are described in section three.

A *mailbox-list*. This is an ordered list of strings (usually names). The intended semantics of a mailbox-list are described in section three.

The following information constitutes the value associated with a name whose type is group. The precise bit patterns representing these values are of concern only when we discuss the actual communication protocols.

A *timestamp*. The value and semantics of this timestamp are as for the timestamp associated with an individual.

A *remark*. This is a string, which is intended as a human hint about the purpose or meaning of the group; Grapevine attaches no semantics to the remark.

A *member-list*. This is an ordered list of strings (usually names).

An *owners-list*. This is an ordered list of strings (usually names). The owners-list is used by Grapevine as an access control list for updates to the naming database, to describe the set of individuals who may make arbitrary modifications to group. Clients may possibly use this list for other purposes.

A *friends-list*. This is an ordered list of strings (usually names). The friends-list is used by Grapevine as an access control list for updates to the naming database, to describe the set of individuals who may add themselves to the group or remove themselves from the group. Clients may possibly use this list for other purposes.

In addition to the above values, the database entry for a name contains various values concerned with the propagation of database updates between servers. These additional values are not generally useful to clients. The client can observe these extra values only through the "ReadEntry" operation in the registration server enquiry protocol.

Certain *pseudo-names* are available, which are accepted by the registration servers when clients make certain database enquiries, although the names do not correspond to explicit entries in the naming database. These names may also be used in access control lists. The particular enquiry operations which accept these names are detailed with the registration server protocol specification. Each of these names behaves, for the purpose of those enquiry operations or access controls, as if it is a group. Only the timestamp and member-list of these pseudo-groups are accessible. For any registry *reg*, the names *Groups.reg* and *Groups^.reg* behave as if the name was a group with a member-list containing the name of each group in the registry *reg*. For any registry *reg*, the names *Individuals.reg* and *Individuals^.reg* behave as if the name was a group with a member-list containing the name of each individual in the registry *reg*. For any group of the form *x.reg*, the names *Owners-x.reg* and *Owner-x.reg* behave as if the name was a group whose member-list contains the owners-list of the group *x.reg* (but if that owners-list is empty, then the member-list contains the friends-list of the group *reg.gv*). If a string of the form **.reg* appears in an access control list, it is treated as matching any name of the form *x.reg*. Finally, the string "*" may be used in access control lists, to mean that unrestricted access is allowed. As will be seen, these names allow clients to send mail to the owners-list of a group, to determine the set of names which are individuals or groups in a particular registry, and to specify more flexible access controls.

Several names are guaranteed to be present in the naming database. Some of these are present to enable clients to locate appropriate registration servers when the client wishes to perform some enquiry or update on the naming database. Others are present as part of the message service, to enable clients to locate mail servers. Appropriate algorithms for using these names are described in section five. These names are as follows.

Every registration server and every mail server has a name which is registered in the naming database. Every registration server name is in the "gv" (for Grape vine) registry. These names have type "individual"; the connect-site of such a name is a string representation of the PUP network address of that server. Note that the network address of a Grapevine server may change from time to time.

The naming database also defines which strings are valid registries. For any string *reg*, the string is valid as a registry if and only if the group *reg.gv* exists in the naming database. Thus for a name such as "foo.baz" to be valid, the group "baz.gv" must exist. Each registration server knows only about the names in some set of registries; the registration server knows about all the names in those registries. Typically, any particular registry is known about by several registration servers. For any valid registry *reg*, the member-list of the group *reg.gv* contains precisely the names of the registration servers which know about names whose registry is *reg*. Thus, given a name such as "foo.baz", the member-list of the group "baz.gv" contains the names of the registration servers that are concerned with the database entry for "foo.baz". All registration servers know about names in the registry "gv".

Thus, in order to determine the network address of some registration server that is concerned with a name such as "foo.baz", the client may contact any registration server, then ask it for the member-list of "baz.gv", then for each name in that member-list determine its connect-site; the client may then choose from amongst that set of connect-sites. Section five describes how this algorithm may be made acceptably efficient.

The string "GrapevineRServer" is registered in the Xerox internet name lookup server database. This name in that database maps to the network addresses of several computers. Some (but not necessarily all) of those computers are registration servers. "GrapevineRServer" exists with the intent of assisting clients in contacting some initial registration server. Clients may also use other techniques, such as broadcast protocols, to contact an initial registration server. Once the client has contacted an initial registration server, the naming database should be used to contact other registration servers.

For example if "Cabernet.gv" and "Zinfandel.gv" are the names of two registration servers, then those names would be registered as individuals. The connect-site for Cabernet.gv would be a string such as "3#14#", and the connect-site for Zinfandel.gv would be a string such as "60#354#". If "pa" was a valid registry, then the group "pa.gv" would exist. If Cabernet.gv and Zinfandel.gv both know about names in the registry "pa", then "Cabernet.gv" and "Zinfandel.gv" would appear in the member-list for "pa.gv". Both servers necessarily know about names in the registry "gv". The name "GrapevineRServer" in the name lookup server database would be likely to contain the network addresses 3#14# and 60#354#.

The registry "ms" (for message sservers) is a valid registry. The name "MailDrop.ms" is registered in the naming database as a group; the member-list for that group contains precisely the names of those mail servers which may be willing to accept mail from clients for delivery. Thus, to determine the network address of *some* Grapevine mail server, a client should obtain the member-list of "MailDrop.ms", and obtain the connect-site of names appearing in the member-list of that

group. Use of "MailDrop.ms" is discussed further in section five.

For example, if the existing mail servers are "Cabernet.ms" and "Zinfandel.ms", then those names would be registered as individuals with appropriate connect-sites. The member-list for "MailDrop.ms" would contain those two names.

The name "DeadLetter.ms" is registered in the naming database; the use of this name is described in section three.

3. Messages

The Grapevine mail servers will transport messages on behalf of clients. It is the purpose of this section to describe the content of those messages and the semantics of message delivery.

A message should be thought of as a *property list* and a *body*. The property list of a message is constructed by a mail server when a client successfully submits a message to it for delivery, as described in the mail submission protocol. The property list contains the name of the *sender* as provided by the client, a *return-to* name for notifying non-delivery of the message, a *postmark* specifying the time at which the message was submitted, and a list of names which are the *recipients* to whom the client asked the mail server to deliver the message. The precise representation of these values is defined in the mail retrieval protocol. The body of a message consists of a sequence of *items*. Each item consists of a *type*, a *length*, and some data. The type is a number in the range $[0..2^{16})$. The length is a number in the range $[0..2^{32})$. The data consists of a sequence of bytes. The length specifies the number of bytes in the data.

The message system *guarantees* that when a client retrieves mail, the property list is as constructed when the message was submitted, and the message body is identical to that submitted. The message system is not concerned with the data content of clients' messages.

The "type" of message body items is a name space that must be managed by convention if the message system is to be of general utility. If a client wishes to attach a meaning to some type, he should register the value of that type with the Grapevine message system maintainers. Types in the range 0 through 777B are reserved for use in representing the property list of messages in the mail retrieval protocol. The following types are pre-defined, in that their values are used internally by Grapevine.

10B	Postmark in property lists
20B	Sender name in property lists
30B	Return-to name in property lists
40B	Recipient names list in property lists
1010B	Human-readable textual message
2000B	Registration server database update
2100B	Mail server "re-mail" hint
177777B	End-of-message item

When a client submits a message to a mail server, the mail server promises that it will *deliver* the message. We now define what this delivery means. The mail servers deliver the message (body plus property list) to each of the recipients. For each recipient, the meaning of delivery is as follows.

If the recipient name is registered in the naming database as an individual, and the forwarding-list of that individual is empty, and the mailbox-list of the individual is not empty, then we proceed as follows. The mailbox-list of an individual contains the names of several Grapevine mail servers. The names in an individual's mailbox-list are considered to be an ordered list of the names of the servers that may be suitable for buffering the individual's incoming messages. The mail server will attempt to cause the message to be buffered for the individual in one of those servers; the earlier names in the mailbox-list are preferred over the later names. Generally this will succeed, and generally the message will be buffered in the server whose name is first in the mailbox-list. If the attempt succeeds, the message is buffered for the client in the client's *in-box* in that Grapevine mail server. The property list and message body may subsequently be read from an in-box by a client using the Grapevine mail retrieval protocol. If the message cannot be delivered to any of the

servers in the recipient's mailbox-list within 2 days, or if the recipient name becomes invalid during the delivery process, then the message is considered to be undeliverable to this recipient and the mail server will attempt to send an *undeliverable* notification to the name given as the return-to name in the message's property list, as described below.

If the recipient name is registered in the naming database as an individual, and the individual's forwarding-list is not empty, then the individual is treated as if it were a group whose member-list contained the names found in the forwarding-list of the individual.

If the recipient name is registered in the naming database as a group, then the message is delivered to recipients whose names appear in the member-list of that group. If any of those recipients are invalid recipients, then the mail server will attempt to send an *undeliverable* notification to the name manufactured by concatenating "Owners-" with the group name, as described below.

If the recipient name is not registered in the naming database, or if the recipient name is registered in the naming database as an individual but the individual's forwarding-list and mailbox-list are both empty, then the name is considered to be an invalid recipient. In such a case, if the recipient name was one of those supplied by a client when the message was submitted, then the mail server will attempt to send an *undeliverable* notification to the name given as the return-to name in the message's property list, as described below; if the recipient name came from a group, then it is handled as described in connection with delivery to groups.

In order to send an *undeliverable* notification to some name (either that provided as return-to name by the originating client, or one representing the owners-list of a group), the mail server proceeds as follows. The mail server generates a new message whose body is a single item of type "text", containing a human-readable explanation of the failure that occurred; the mail server attempts to deliver this message to the appropriate name. If the appropriate name turns out to be an invalid recipient, then this message is sent to "DeadLetter.ms". The intention is that sending to "DeadLetter.ms" will cause the notification to be seen by some system administrator. Additionally, a summary of the undeliverable notification and a copy of the header part of any text item of the returned message are always sent in a text message to "DeadLetter.ms".

The above describes the delivery semantics as they would be if the Grapevine message system existed in isolation. However, this is not so and the semantics are actually modified as described below. The *foreign* mail systems with which Grapevine cooperates are: the IFS mail servers, the TENEX system running on MAXC, and the ARPAnet message system (accessed through MAXC). We use the phrase *foreign mail server* to indicate a host running one of these systems.

The mailbox-list of an individual may contain not only the names of Grapevine mail servers, but also names of foreign mail servers. These names are strings representing either the PUP network address of the host running the foreign mail server or a PUP Name Lookup Server name for that host. If during the delivery algorithm for an individual recipient the Grapevine mail server encounters a name from a mailbox-list specifying a foreign mail server, and if the Grapevine mail server decides that this is the best place for buffering the message, then the Grapevine mail server will forward the message to that foreign mail server using the MTP protocol. In doing so, the Grapevine mail server will forward only the first message body item of type text (if any); the property list and any other message body items are lost. If the chosen foreign mail server rejects the recipient name, or if the chosen foreign mail server appears not to exist, and if the postmark of the message is more than 24 hours old, then the recipient is considered to be an invalid recipient. (The 24 hour delay is to cover transients while the Grapevine database entry for the recipient is being modified, because such changes cannot be synchronized with changes to the foreing mail

server.) This facility allows individuals registered in the Grapevine naming database to have mailboxes on these foreign mail servers; typically, such an individual would have only one name in their mailbox-list.

If during the delivery algorithm a recipient name is encountered which is not registered in the naming database, but the recipient name is of the form *x.reg* and the name *reg.Foreign* is registered in the naming database as an individual, then the mail server will treat the recipient name as a *foreign recipient name*. If the recipient name does not contain the character "^", then the Grapevine mail server treats the recipient as if it was an individual whose mailbox-list was that of *reg.Foreign*. If the recipient name contains the character "^", then the mail server expects the recipient name to name a file on the MTP server *reg*; the mail server will retrieve this file through the FTP protocol to that server, connecting on socket 7; the mail server will attempt to parse this file as a distribution list and will deliver the message to the resulting names. This mechanism allows clients of Grapevine to address individuals and distribution lists which exist on foreign mail servers and are not registered in the Grapevine database. For example, if "xrcc.foreign" is registered as an individual with a mailbox-list containing "Aklak", then any name such as "foo.xrcc" is acceptable as an individual recipient and messages for "foo.xrcc" will be forwarded to the IFS mail server "Aklak". In particular, the name "ArpaGateway.foreign" is registered as an individual, whose mailbox-list causes forwarding to a foreign mail server which understands about ARPAnet mail recipients. To address a text message to an ARPAnet recipient such as "Saltzer@MIT-Multics", one would include "Saltzer@MIT-Multics.ArpaGateway" amongst the recipients of the message as presented to the Grapevine mail servers.

A mail server occasionally wishes to use a remote file server to store messages which are being buffered for clients. It does this to avoid over-filling its local disk with messages for people who do not read their mail often enough (or are on vacation, etc.). For a mail server whose name is *x.ms* the group *Archive-x.ms* should exist. The member-list of that group should contain path names indicating the desired remote server and directory. Each of these servers should have a LEAF protocol server enabled, and should have an account for the name *x*, with a password that matches that of *x.ms*. The mail server will try each of the file servers in that group (if necessary), in order of closeness on the Internet. For example, if "Cabernet.ms" is a mail server, and the group "Archive-Cabernet.ms" has members "[Ivy]<DMS>" and "[Ibis]<DMS>"; then the mail server would store on the server "Ivy" files with titles such as "<DMS>Cabernet.ms>Birrell.pa-19-Jan-81-23-20-59-PDT!1".

4. Protocols

This section describes the protocols which may be used to invoke the facilities provided by individual Grapevine servers. They will also be used to interrogate the Grapevine naming database in order to find suitable Grapevine servers at various times. In addition to the protocols defined here, the Grapevine servers implement the following PUP protocols: FTP, MTP, Telnet, Miscellaneous Services (for Authentication Request and Mail Check Laurel only), Echo.

Each of the following protocols uses some underlying transmission medium. This medium is either PUP packets, or PUP Byte Stream Protocol streams. In either case, the medium provides communication to a specified network address, and transports an ordered sequence of 8-bit bytes. PUP packets provide unreliable (but high probability) uni-directional transmission of small numbers of bytes with no notification of failure; the Byte Stream Protocol provides reliable bi-directional transmission of arbitrarily large numbers of bytes, with notification of failure. Values of several data types occur in the Grapevine protocols. The representation of these values is described here, in terms of the sequence of bytes provided by the transmission media.

The Grapevine servers are unforgiving of protocol violations. If a client violates the protocol, the server's typical response will be to terminate the byte stream or ignore the PUP, as appropriate.

A *character* is a single byte containing the ASCII value of the character.

A *boolean* is a single byte containing 1 for TRUE, 0 for FALSE.

An *ack* is a single byte of undefined value.

A *word* is a pair of bytes; the first contains the more significant 8 bits.

A *number* is a word containing the binary representation of an integer in the range $[0..2^{16})$.

A *long number* is a pair of words representing an integer in the range $[0..2^{32})$, the first word containing the less significant 16 bits (this is the Mesa representation).

A *timestamp* is three words. The first word is an uninterpreted bit-pattern (though strikingly similar to a PUP network address); the second and third constitute a long number. This long number is approximately the number of seconds since midnight, January 1st, 1901 that had passed at the time that the timestamp was created.

A *password* is 8 bytes representing in order the bits of a password value as defined above in connection with the naming database.

A *string* is a sequence of bytes as follows. The first two bytes form a number which specifies the number of characters in the string. The third and fourth bytes are ignored. This header is followed by the characters of the string. If the number of characters is odd, there follows one extra byte (with undefined value). This is derived from the Mesa representation of a string. Note that this representation occupies an even number of bytes. All strings are restricted to be no more than 64 characters. An attempt to transmit more than 64 characters (i.e. 34 words) in one of these values is a violation of these protocols.

A *name*, a *connect-site*, and a *remark* are each represented by a string (and are therefore restricted to 64 characters).

An *operation* is a number used to specify a *command*, whose values are specified in the individual protocols.

A *name-type* is a byte with values as specified by the following Mesa type constructor:

```
NameType: TYPE = MACHINE DEPENDENT{
    group(0), individual(1), notFound(2), dead(3), (255) };
```

A *code* is a byte with values as specified by the following Mesa type constructor:

```
Code: TYPE = MACHINE DEPENDENT{
    done(0), noChange(1), outOfDate(2), NotAllowed(3),
    BadOperation(4), BadProtocol(5), BadRName(6), BadPassword(7),
    WrongServer(8), AllDown(9), (255) };
```

A *return-code* is a code followed by a name-type. In the protocol descriptions, a return-code is denoted by [a, b] to indicate transmission of the code "a" followed by the name-type "b".

A *component* is a number, followed by a sequence of words. The number specifies how many words.

A *string-list* is a component whose words constitute a sequence of strings. The strings must be in alphabetic order. The ordering is obtained by converting all upper case letters to lower case, then sorting by ASCII value of the characters. Note that this implies that the servers will deliver the various lists of strings from database entries to clients in alphabetic order.

4.1 Registration Server Protocols

The registration server responds with PUP type "iAmEcho" (type 2) to PUP's of type "echoMe" (type 1) and length no more than 128 bytes sent to it on socket 52B.

The registration server is usually willing establish a Byte Stream Protocol connection in response to an RFC packet sent to socket 50B (any unwillingness usually indicates that this server is feeling overloaded). The protocol accepted on such byte streams is as follows. The protocol consists of a sequence of *commands*. For each command, the client sends a word representing an operation, probably followed by some *arguments*, then the server sends a return-code, possibly followed by some *results*. The arguments and results depend on the command. The commands each operate on the database entry for some name. Unless otherwise specified, that name may not be one of the pseudo-names ("Owners-foo.reg", *et al*). The commands have the following properties in common.

If the registry of the name in question is invalid, then the return-code is [BadRName, notFound] and there are no other results.

If the registry of the name is valid, but this registration server is not concerned with names in that registry, then the return-code is [WrongServer, notFound] and there are no other results; in this case the client should contact some other registration server. This facility allows a client to assume that a particular registration server knows about a name until the client is told otherwise.

If the registry of the name is valid and this registration server is concerned with that registry, but the name is not registered in the database, then (except for CreateIndividual, CreateGroup and NewName!) the return-code is [BadRName, notFound] or [BadRName, dead]. and there are no other results. The distinction between "notFound" and "dead" is not intended to be useful to clients, although it may occasionally be useful to a human administrator. The distinction arises when a name is deleted from the naming database. For some time after the deletion, the name may appear as "dead" instead of "notFound"; at any time a name may revert from "dead" to "notFound", at the whim of the registration servers. The distinction is important in the registration servers' database update propagation algorithm.

In the descriptions of the commands, the above possible outcomes are implicitly assumed.

The following commands allow clients to read the database.

Expand: operation=1, arguments = [name, timestamp]

> If the present timestamp of the database entry equals the given one, then the return-code is [noChange, group] or [noChange, individual] as appropriate, and there are no other results.
>
> Otherwise, the results are a timestamp and a string-list. The timestamp is the current timestamp of the database entry. If the name has type group then the return-code is [done, group] and the string-list contains the group's member-list; if the name has type individual and the individual's forwarding-list is non-empty, then the return-code is [done, group] and the string-list contains the individual's forwarding-list; if the name has type individual and the individual's forwarding-list is empty, then the return-code is [done, individual] and the string-list contains the strings (if any) from the individual's mailbox-list, in chronological order of when they were added to the mailbox-list (this is used as the order of priority by mail servers). The argument of this command may be a pseudo-name of the form *Owners-x.reg* or *Owner-x.reg*, but not one of the form *Groups.reg*, *Groups^.reg*, *Individuals.reg*,

Individuals^.reg, *.*reg* or "*".

ReadMembers: operation=2, arguments = [name, timestamp]

If the database entry has type individual, then the return-code is [BadRName, individual] and there are no other results.

If the present timestamp of the database entry equals the given one, then the return-code is [noChange, group], and there are no other results.

Otherwise, the return-code is [done, group] and the results are a timestamp and a string-list. The timestamp is the current timestamp of the database entry. The string-list contains the group's member-list. The argument of this command may be any pseudo-name other than *.*reg* or "*".

ReadOwners: operation=3, arguments = [name, timestamp]

This is the same as ReadMembers, except that the string-list returned contains the owners-list of the database entry, and the pseudo-names are not supported.

ReadFriends: operation=4, arguments = [name, timestamp]

This is the same as ReadOwners, except that the string-list returned contains the friends-list of the database entry.

ReadEntry: operation=5, arguments = [name, timestamp]

The result is a timestamp followed by a number, followed by that number of components. The timestamp is at present of undefined value. The components are the entire database entry for the name. If the caller has been authenticated by the IdentifyCaller command (as for database updates) and is in the "gv" registry, then the components for an individual include the individual's password, otherwise the password is represented by zeroes.

CheckStamp: operation=6, arguments = [name, timestamp]

If the present timestamp of the database entry equals the given one, then the return-code is [noChange, group] or [noChange, individual] as appropriate, and there are no other results.

Otherwise, the return-code is [done, group] or [done, individual] as appropriate and the result is a timestamp which is the current timestamp of the database entry. The argument of this command may be any pseudo-name other than *.*reg* or "*".

ReadConnect: operation = 7, argument = [name]

If the database entry has type group, then the return-code is [BadRName, group] and there are no other results.

Otherwise the return-code is [done, individual] and the result is the connect-site given in the database entry.

ReadRemark: operation = 8, argument = [name]

If the database entry has type individual, then the return-code is [BadRName, individual] and there are no other results.

Otherwise the return-code is [done, group] and the result is the remark given in the database entry.

Authenticate: operation = 9, argument = [name, password]

If the database entry has type group, then the return-code is [BadRName, group] and there are no other results.

If the given password does not equal the password given in the database entry then the return-code is [BadPassword, individual].

Otherwise the return-code is [done, individual]; there is no further result.

IdentifyCaller: operation = 33, argument = [name, password]

If this registration server does not know about the registry of the given name, then this server will consult other registration servers as necessary to perform the operation; it will not give the return-code [WrongServer, name-type]. If the necessary other servers are not contactable, then the return-code is [AllDown, notFound] and there is no other result.

If the database entry has type group, then the return-code is [BadRName, group] and there is no other result.

If the given password does not equal the password given in the database entry then the return-code is [BadPassword, individual].

Otherwise the return-code is [done, individual]; there is no further result.

Use of this command affects the state of the connection, as described with the database update commands.

The following commands allow clients to perform database enquiries in support of access controls. For each of them, if the database entry is of type individual, then the return-code is [BadRName, individual] and there is no other result. For the "closure" commands, the registration server may need to consult other registration servers; if for some reason it cannot do so, then the return-code will be [AllDown, group] and there will be no other result. Otherwise, the return-code is [done, group] and the result is a boolean, TRUE iff the second given name is a member of the appropriate list(s). The *IsInList* operation provides access to all the facilities of operations 40 through 45, and should be used instead of those operations. Operations 40 through 45 are historical.

IsMemberDirect: operation = 40, argument = [name, string]

The result is TRUE if the string is one of the strings in the member-list of the database entry. Each argument of this command may be any pseudo-name.

IsOwnerDirect: operation = 41, argument = [name, string]

The result is TRUE if the string is one of the strings in the owners-list of the database entry.

IsFriendDirect: operation = 42, argument = [name, string]

The result is TRUE if the string is one of the strings in the friends-list of the database entry.

IsMemberClosure: operation = 43, argument = [name, string]

The result is TRUE if the string is one of the strings in the member-list of the database entry, or if this operation would return TRUE when applied to any of the names in that list. Thus this operation traverses the graph implied by the names in that list, and may involve communication with other registration servers. Loop detection is applied as necessary. This operation may be quite expensive. Each argument of this command may be any pseudo-name.

IsOwnerClosure: operation = 44, argument = [name, string]

As for IsMemberClosure, but starting with the owners-list of the database entry. The name may not be one of the pseudo-names.

IsFriendClosure: operation = 45, argument = [name, string]

As for IsMemberClosure, but starting with the friends-list of the database entry.

IsInList: operation = 46, argument = [name, string, byte, byte, byte]

This provides a general access control testting operation, and supersedes operations 40 through 45. If the first byte is 1, then instead of testing membership in lists associated with the name, it tests in lists associated with the name's registry; that is, if the name is of the form *x.reg*, the operation will test membership in lists of the group *reg.gv*; otherwise the first byte should be 0. The second byte should be 0 to test membership of the members-list, 1 to test the owners-list, 2 to test the friends-list. The third byte should be 0 to test direct membership, 1 to test membership in the closure, and 2 to test membership in the "up-arrow closure". The "up-arow" closure is like normal closure, except that the closure pursues only contained names of the form *x*^.*reg*, such as "CSL^.pa" but not "Birrell.pa"; this is very much more efficient than full closure. If any of the bytes does not have one of the specified values, it is treated as a protocol error.

The algorithms used by the registration servers to propagate updates of the naming database are not *atomic*, in the following sense. When a client requests a registration server to perform a database update, if the registration server is able to perform the update, the client is given an acknowledgement when the registration server has made the required change to its own copy of the database. Subsequently, the registration server guarantees to propagate the update to other registration servers so that all copies of the database will be updated. In the interval after the first registration server has performed the update but before all other registration servers have performed the update, clients may observe two copies of the database, one in which the update has occurred and one where it has not yet occurred. Thus clients may get a transiently inconsistent view of the naming database, and clients should be prepared to deal with this fact. In all normal cases, the window for this inconsistency is short (in the region of one minute), but there is no upper bound to the length of this window.

The following commands allow clients to make updates to the naming database. In all circumstances, these commands return a return-code and no other result. In addition to the possible outcomes described above as being in common for all registration server commands, the update commands have the following properties in common.

The update commands ask the registration server to change the state of registration of some name in the naming database; this name is sent as the first argument. The registration server will consult various access control lists associated with this name to determine whether the requested update should be permitted. For any given name there are four access control lists that are of interest. We will call them "owners-acl", "friends-acl", "reg-owners-acl" and "reg-friends-acl". For any group *sn.reg* the owners-acl is the owners-list of that group and the friends-acl is the friends-list of that group. For any group *sn.reg*, the reg-owners-acl is the owners-list of the group *reg.gv* and the reg-friends-acl is the friends-list of *reg.gv* (the reg-owners-acl and reg-friends-acl are both empty lists if *reg.gv* doesn't exist). If during this connection the command IdentifyCaller has not been made, or if the last call of that command gave a return-code other than [done, individual], then any of the following commands will give the return-code [NotAllowed, notFound]. Otherwise, the "caller" is the name that was given as argument of the last call of IdentifyCaller during this connection. The registration server consults the access control list indicated for each particular command below, to determine whether the caller's name is in that list. This is done using the closure operations (IsMemberClosure, IsFriendClosure, IsOwnerClosure, as appropriate). If the caller is not in the indicated list, and the list was the friends-acl, then the registration server applies this algorithm recursively using the owners-acl. If the caller is not in the indicated list, and the list was the owners-acl, then the registration server applies this algorithm recursively using the reg-friends-acl. If the caller is not in the indicated list, and the list was the reg-friends-acl, then the registration server applies this algorithm recursively using the reg-owners-acl. If the caller is not in the indicated lists, then the return-code is [NotAllowed, notFound]. These access control checks are typically more efficient than this algorithm would indicate, but they are still potentially quite expensive. The lists controlling the various update commands are as follows.

CreateIndividual, DeleteIndividual, CreateGroup, DeleteGroup, NewName, AddMailbox, RemoveMailbox: the reg-owners-acl.

ChangePassword, ChangeConnect: if the name is the caller's, then the operation is allowed; otherwise it is controlled by the reg-friends-acl.

AddForwarding, RemoveForwarding: the reg-friends-acl.

AddMember, RemoveMember: if the string which is the second argument is equal to the caller's name, then the operation is treated as *AddSelf* or *RemoveSelf*; otherwise if the name is in the "gv" registry, then the reg-friends-acl; otherwise the owners-acl.

ChangeRemark, AddListOfMembers: if the name is in the "gv" registry, then the reg-friends-acl; otherwise the owners-acl.

AddSelf, RemoveSelf: if the name is not in the "gv" registry then the operation is controlled by the friends-acl; if the name is in the "gv" registry then if the caller is in the "gv" registry, then the operation is allowed otherwise it is controlled by the reg-friends-acl.

AddOwner, RemoveOwner, AddFriend, RemoveFriend: the owner-acl.

If the requested update is such that it would not change the database, then the return-code is [noChange, name-type]. If the registration server determines that there is contradictory information in the database that is newer than the requested update, the return-code is [outOfDate, name-type]; this situation is exceedingly unlikely, but is possible. Several of the update commands have restrictions on the type or existence of some name before the operation may be performed. If these restrictions are violated, the return-code is [BadRName, name-type], where "name-type" is individual or group if the name is registered with inappropriate type, and is notFound or dead if

the name is needed but is not registered.

Otherwise, the update is made and the return-code is [done, name-type].

CreateIndividual: operation = 12, argument = [name, password]

> The name must not presently be registered in the database. This registers the name with type individual, with the given password, with an empty string for connect-site and with empty forwarding-list and mailbox-list.

DeleteIndividual: operation = 13, argument = [name]

> The name must have type individual. This removes the name from the database.

CreateGroup: operation = 14, argument = [name]

> The name must not presently be registered in the database. This registers the name with type group, with an empty string for remark, with an empty member-list and friends-list, and with an empty owners-list.

DeleteGroup: operation = 15, argument = [name]

> The name must have type group. This removes the name from the database.

ChangePassword: operation = 16, argument = [name, password]

> The name must have type individual. This sets the individual's password to be that given.

ChangeConnect: operation = 17, argument = [name, connect-site]

> The name must have type individual. This sets the individual's connect-site to be that given.

ChangeRemark: operation = 18, argument = [name, remark]

> The name must have type group. This sets the group's remark to be that given.

AddMember: operation = 19, argument = [name, string]

> The name must have type group. This adds the string to the members-list of the group.

AddMailbox: operation = 20, argument = [name, string]

> The name must have type individual. This adds the string to the mailbox-list of the individual.

AddForward: operation = 21, argument = [name, string]

> The name must have type individual. This adds the string to the forwarding-list of the individual.

AddOwner: operation = 22, argument = [name, string]

> The name must have type group. This adds the string to the owners-list of the group.

AddFriend: operation = 23, argument = [name, string]

The name must have type group. This adds the string to the friends-list of the group.

RemoveMember: operation = 24, argument = [name, string]

The name must have type group. This removes the string from the members-list of the group.

RemoveMailbox: operation = 25, argument = [name, string]

The name must have type individual. This removes the string from the mailbox-list of the individual.

RemoveForward: operation = 26, argument = [name, string]

The name must have type individual. This removes the string from the forwarding-list of the individual.

RemoveOwner: operation = 27, argument = [name, string]

The name must have type group. This removes the string from the owners-list of the group.

RemoveFriend: operation = 28, argument = [name, string]

The name must have type group. This removes the string from the friends-list of the group.

AddSelf: operation = 29, argument = [name]

The name must have type group. This adds the caller to the members-list of the group. Note that this is equivalent to AddMember[name, caller].

RemoveSelf: operation = 30, argument = [name]

The name must have type group. This removes the caller from the members-list of the group. Note that this is equivalent to RemoveMember[name, caller].

AddListOfMembers: operation = 31, argument = [name, string-list]

The name must have type group. Note that the string-list must be in alphabetic order (see the specification of "string-list"); violation of this restriction is a violation of the protocol. This adds each string in the string-list to the members-list of the group.

NewName: operation = 32, argument = [name, second name]

The first name must not be registered in the database, and the second name must be registered. The second name must have the same registry as the first name; violation of this restriction is a violation of the protocol. This registers the first name as a database entry having the same value as the present value of the second name.

4.2 Mail Server Protocols

The mail server responds with PUP type "iAmEcho" (type 2) to PUP's of type "echoMe" (type 1) and length no more than 128 bytes sent to it on socket 54B.

Any Grapevine mail server will accept messages submitted to it by the following protocol, and will deliver them to the specified recipients, according to the delivery semantics defined in section three. This *mail submission protocol* allows a sequence of *commands* on a Byte Stream Protocol connection. The mail server is usually willing to establish such a connection in response to an RFC packet sent to the mail server on socket 56B. The commands each consist of the client sending a word representing an operation, probably followed by some *arguments*, then the server generally sends some *results*. Not all commands are allowed at any time. The restrictions on the order of the commands are described in terms of the *state* of the stream; violation of these restrictions is a protocol violation. The state may be *idle*, *started*, *noItem*, or *inItem*. The state is initially "idle".

StartSend: operation = 20, arguments = [sender name, password, return-to name, boolean]

> This command is allowed only if the state is "idle". The password should be that of the sender. At present, the server ignores the password, but checks that the sender name would be valid as a recipient name; we reserve the right to start checking the sender password some time in the future, without notice. The server checks that the return-to name would be valid as a recipient. The boolean should be TRUE iff the client wishes the server to validate the recipient names during the connection. The result is a byte, with the following values. 0 means everything is ok. 1 means the sender password is incorrect (not checked at present). 2 means the sender name is invalid. 3 means the return-to name is invalid. 4 means that the server could not validate some name because of communication problems. If this byte is 0, then the state of the connection becomes "started".

AddRecipient: operation = 21, arguments = [name]

> This command is allowed only if the state is "started". The name is added to the list of recipients for this message; there is no result.

CheckValidity: operation = 22, no arguments.

> This command is allowed only if the state is "started". If the boolean argument of the StartSend command was TRUE, then for each recipient which appears to be invalid the server sends a number specifying which recipient it was (counting the calls of AddRecipient, from 1) followed by the name of the invalid recipient, and these recipients are removed from the recipient list of the message. Then (regardless of the value of the boolean) the server sends the number 0 followed by a number indicating how many names remain in the message's recipient list. The state of the stream becomes "noItem".

StartItem: operation = 23, arguments = [word]

> If the state of the stream is "inItem", then the current message body item is terminated and its length calculated and the state of the stream becomes "noItem". The state of the stream must now be "noItem". The word specifies the type of a message body item, and the state of the stream becomes "inItem". The acceptable types of message body item have been described in section three, in connection with the message delivery semantics. There is no result.

AddToItem: operation = 24, arguments = [number, sequence of bytes]

> This command is allowed only if the state is "inItem". The number specifies how many bytes are in the sequence. The bytes are appended to the current message body item. There is no result.

Send: operation = 26, no arguments.

> This command is allowed only if the state is "inItem". The current message body item is terminated (as in StartItem), and all the data associated with the message is recorded in stable storage. The server commits to deliver the message. The result is an ack. The state of the stream becomes "idle".

Expand: operation = 27, argument = [name]

> This command is allowed at any time and does not affect the state of the stream. If the name would be treated by the mail servers as a distribution list during the delivery algorithm (including a foreign distribution list) then for each name in that list, the server sends a boolean TRUE followed by the name. Then a boolean FALSE is sent. Then a byte is sent. The value of this byte is: 0 if the name was treated as a distribution list, 1 if the name would be an invalid recipient, 2 if the name would be an individual recipient (including foreign mail system recipients), 3 if the mail server could not decide because of communication failures. The intent of this command is to allow a client to show a user the potential contents of a distribution list.

The following protocols allow a client to inspect and modify the state of an in-box on a mail server. Mail arrives in an in-box as the major effect of the mail server delivery algorithm. Note that an individual may have several in-boxes, defined by the individual's mailbox-list in the naming database, and any mail retrieval package should arrange to inspect all of the individual's in-boxes. The in-box mechanism is complicated by facilities (*TOC entries* and *deleted messages*) which are designed to assist a dumb terminal system to provide a user with temporary access to mail in the user's in-boxes. The intent is that a user should retrieve all messages from the in-box and flush the in-box when software with secondary storage is available to the user. Keeping substantial amounts of mail in an in-box for substantial periods of time will degrade the performance of the mail servers. The TOC mechanism allows a client to associate a *TOC entry* (represented by a remark, i.e. a string of up to 64 characters) with each message in an in-box. Provision is also made for a client to *delete* individual messages from an in-box; a deleted message still occupies a place in the in-box until the in-box is *flushed*, but deleting the message frees the resources used by the message body and property list. No TOC entry may be associated with a deleted message. If there is no in-box on this server for an individual, then the protocol provides the illusion that there is an in-box containing no messages.

The mail server accepts PUP's of type "mailCheckLaurel" (type 214B) and length no more than 128 bytes on socket 54B. If the PUP contains the characters of a name, and this mail server has a non-empty in-box containing messages for a recipient of that name (even if they are all deleted messages, see below), then the mail server replies with a PUP of type "mailIsNew" (type 211B); otherwise it replies with a PUP of type "mailNotNew" (type 212B). Note that the server does not respond to PUP's of type "mailCheck" (type 210B). Note that the server does not check whether the naming database presently indicates that this server is in the user's mailbox-list. This polling protocol is intended to allow a client to inspect the state of an individual's in-box without going to

the expense of establishing a Byte Stream Protocol connection.

An in-box may be accessed after establishing a Byte Stream Protocol connection in response to an RFC packet sent to a mail server on socket 57B. The state of this stream is described as one of *idle*, *open*, or *inMessage*, and there are restrictions on which commands may be used depending on the state. Violation of these restrictions is a violation of the protocol. The state is initially "idle". Each command consists of the client sending a word representing an operation, possibly followed by some arguments, then the server sending some results. The commands allow the client to open a mailbox, then sequentially inspect or modify the messages in it; the order of the messages is approximately that in which they were sent, and the order does not change between sessions; messages are not added to an in-box while a client has it open; only one client may have a particular in-box open at one time. A client, having opened an in-box, may flush it: this causes the in-box to be empty. If a client wishes to close an in-box without flushing it, the client must terminate the connection.

A mail server may occasionally use a remote file server for storing contents of in-boxes; if this has been done and the file server is unavailable, the mail server will close the client's mail retrieval connection arbitrarily (sorry!). The location of such a remote file server is of no concern to a client of the mail retrieval protocol, but has been described in section three.

OpenInBox: operation = 0, arguments = [name, password]

> The state must be "idle". The result is a byte with the following values: 1 if the name is for a group, 2 if the name is for an individual and the password is correct, 3 if the name is not registered in the naming database, 4 if the name could not be checked because of communication failures, 5 if the password is incorrect. If this byte has the value 2, then the state becomes "open". The byte is followed by a word, which should be ignored.

NextMessage: operation = 1, no arguments.

> If the state is "inMessage", it becomes "open". The state must now be "open". The result is a sequence of three booleans. The first is TRUE iff there is another message in the in-box, and this becomes the current message; if this boolean is FALSE the others are undefined. The second is TRUE iff the message is *archived*; this is a hint to the client that access to the message may involve access to a remote file server; we recommend that you indicate this fact to your user. The third is TRUE iff the message is deleted. If the message is not deleted, then the state becomes "inMessage".

ReadTOC: operation = 2, no arguments.

> The state must be "inMessage". If there is a TOC entry associated with this message, the result is a remark. Otherwise the result is an empty string.

ReadMessage: operation = 3, no arguments.

> The state must be "inMessage". The result is a sequence of message body items, representing the message's property list followed by the message body. Each item has the following format: a number, which is the item's type, followed by a long number, which is the item's length, followed by that number of bytes, followed by an extra byte if the length was odd (thus each item occupies an even number of bytes). The items sent are: an item of type "postmark" (whose bytes are the timestamp of the message's property list), an item of type "sender" (whose bytes are the sender name of the message's property list), an item of type "return-to" (whose bytes are the return-to name of the message's property list), an

item of type "recipients" (whose bytes are the recipient names of the message's property list), the items that constitute the message body in the order provided by the submitting client, and an item of type 177777B. The item of type 177777B is of length 0. The items are followed by a Byte Stream Protocol *mark* byte. The value and meaning of the property list item types is given earlier in the description of the message delivery semantics.

WriteTOC: operation = 4, argument = [remark]

The state must be "inMessage". The remark is associated with the message as a TOC entry, (replacing any earlier TOC entry for this message); if the remark is the empty string, no TOC entry is associated. The result is an ack.

DeleteMessage: operation = 5, no arguments.

The state must be "inMessage". The current message is permanently deleted from this in-box (as is any associated TOC entry). The result is an ack. The state becomes "open".

Flush: operation = 6, no arguments.

The state must not be "idle". The in-box is emptied. The result is an ack. The state is now "idle".

5. The Mesa Grapevine client interface

This section describes informally the interface provided to clients programming in Mesa by the Mesa *GrapevineUser* package. The purpose of this section is not primarily to document that package for its clients, but to indicate how the Grapevine naming database may be used to provide an interface that makes transparent such network effects as the replication and distribution of services, and the failure of particular instances of services. If the reader is intending only to be a client of the Mesa package, then this section should be read briefly; precise specification of the package is in the public definitions files for the package. If the reader intends to implement an equivalent package (in Mesa or another language), then I strongly recommend perusal of the source files of both the definitions and implementations of this package. All files concerned with this package are available on [Indigo]<Grapevine>User>*.mesa and *.bcd. The BCD's there are, at the time of writing, suitable for use with Alto/Mesa 6.0. They are source compatible with the Mokelumne and Rubicon releases of the Pilot operating system.

GrapevineUser provides several public interfaces. These are named *NameInfoDefs*, *SendDefs*, and *RetrieveDefs*. The NameInfoDefs interface provides access to most of the naming database enquiry operations; SendDefs provides for submission of messages; RetrieveDefs allows a client to read messages from in-boxes. At present the package makes no provision for updates to the naming database. Each of these interfaces uses the naming database to provide an interface that is independent of particular computers or network addresses. A client of GrapevineUser never is concerned with such things as host names.

In the following descriptions, the precise declarations found in the definitions files are not repeated: the reader should look in the sources of those definitions files.

The NameInfoDefs interface provides for database enquiries. The results of these enquiries are a return code, represented by some subset of the enumerated type NameType, and in some cases a list of names, represented by a value of type RListHandle. Each enquiry may give a return code "allDown", to indicate that because of communication failures the requested enquiry could not be performed; this indicates that after its best efforts to contact any of the multiple suitable registration servers, the package had failed. None of the enquiries raises a SIGNAL. If the enquiry is such as to deliver an RListHandle, and the enquiry succeeds, then the client may enumerate the names represented by the RListHandle (as often as desired), then should close the RListHandle. The RListHandle mechanism is implemented, in the default released package, by buffering the names in main memory but this is implemented by one very simple module which a client is welcome to replace if other buffering strategies are more appropriate to his application. (See the interface "RListDefs" and the module "FSPRList" if you are interested in this.) The enquiries each have an obvious mapping into the commands described in the registration server protocol.

The transparency provided by the NameInfoDefs interface is achieved as follows. Generally, the NameInfo implementation caches a stream to some registration server. When it is asked to perform an enquiry about some name, the implementation uses the cached stream to attempt the enquiry. If this stream fails (because that registration server is now unavailable), or if this enquiry gives a "WrongServer" return-code, then the implementation attempts to locate a suitable registration server. It does this by using a resource location interface (described below) to find a network address of some server which knows about the name in question. That is, if the name in question is of the form *x.reg*, the implementation tries to locate a server in the group named *reg.gv*. If this succeeds, such a server should be able to answer the enquiry; if this fails because *reg.gv*" is not a group, then the name is invalid; if this fails because none of those servers is available then the return code given is "allDown".

The SendDefs interface implementation proceeds in a similar manner. Each of its procedures has a straightforward mapping into a command specified in the mail submission protocol. The SendDefs implementation attempts to cache the network address of a suitable mail server, but if this fails (or initially) the implementation uses the resource location interface to locate a server in the group "MailDrop.ms". Only if this resource location fails is mail submission not possible.

The RetrieveDefs interface is more complicated, because it provides a client with access to all of the client's in-boxes. The implementation determines the sites of a client's in-boxes by using the Expand database enquiry. The interface also uses the single PUP polling protocol to obtain hints about the state of the in-boxes. When a client wishes to inspect mail, the implementation establishes connections only to those in-boxes which have indicated they are non-empty during the polling protocol; this is important to minimize the connection load on the mail servers, since a client's secondary in-boxes will almost always be empty. The RetrieveDefs interface will also provide clients with access to their mailbox on a foreign mail server (using the MTP protocol), and will function satisfactorily in an environment where there are no Grapevine servers, only IFS mail servers. To determine whether it is in a non-Grapevine environment, the RetrieveDefs implementation considers whether the client's registry is registered in the PUP Name Lookup Server database; if the registry is registered there and maps to precisely *one* network address, then the registry is assumed to be implemented on an IFS mail server (at that network address) with no aid from Grapevine; otherwise the Grapevine servers are used.

The GrapevineUser package uses the following algorithm to perform resource location. The resource location interface is provided with the name of a group in the naming database. It reads the members of this group, giving it a list of names of potential servers. For each of these, it reads their connect-site from the database. The implementation then sends a PUP of type "echoMe" to each of these servers. For each server which replies with an "iAmEcho" PUP, the implementation calls back to its caller, offering the network address of that server. In this manner, the caller of this interface is provided with network addresses of potential instances of the service named by the given group, and the caller can attempt to establish a connection with these, in turn, until a satisfactory one is found. The "echoMe" mechanism is intended for efficiency: the servers are tried in order of their responsiveness, and each one tried is responding to PUP's, so an attempt to connect will be resolved rapidly.

There are several optimizations used within the GrapevineUser implementation, and the potential implementor is strongly recommended to inspect the GrapevineUser implementations.

6. Administrative Facilities

The major administrative facility in Grapevine is the naming database. Most system administration can be done by performing updates to that database, and an interactive program, called *Maintain*, is available to perform these updates. Maintain is documented (partially, at the present time) elsewhere.

In addition, there are some facilities provided by Grapevine to monitor and affect the state of the computers which constitute Grapevine. These facilities change from release to release of the software, so this section is likely to be slightly incorreect at any time. These facilities are defined precisely only by their implementation. The facilities in question are: a local terminal interface, an FTP server, a Telnet server, and a log file.

The local display shows various statistics about the state of the servers. These statistics are the same as those available through the Telnet server "Display Statistics" command. There is also a short typescript, which is of use only to wizards. When the local keyboard and mouse have not been touched for a minute, the display reverts to plain white, with a cursor slowly traversing near the top of the screen; moving a key or the mouse causes the statistics to reappear.

The local keyboard is used only when a server is initialized or restarted. When a server is initialized for the first time, it will determine its own name, ask you to type its password, and arrange to fetch copies of the appropriate database entries. When a server is restarted, it will verify its password (asking for it to be re-typed only if this verification fails), and if necessary will alter its connect-site in the naming database. Note that registration servers and mail servers cohabiting a single machine must have distinct names (typically, the same simple-name, with registries "gv" and "ms"). The case of initializing the first server in the world is necessarily special - consult an expert for details.

Each Grapevine server provides an FTP server which allows access to files on the Grapevine server's local disk filing system. Clients of this FTP server must provide credentials corresponding to a valid name and password of an individual in the Grapevine database. The files which support the naming database and message buffering are not accessible. To be permitted to write files, the client must be a member of the group "Transport.ms". The FTP server supports enumeration of files, with the string "*" meaning all files.

The log file contains single-line entries recording various events as they occur in the server. The file is named "GV.log" and is 120 Alto pages long. The file is used as a circular buffer. Each cycle round this buffer may also be written to backing files on an IFS file server. For a server whose mail server has a name of the form *x.ms*, the individual *Log-x.ms* should have a connect-site whose value is a string of the form *[host]<path>*. This will cause the log files to be written cyclically to 40 files with names of the form *<path>x-00!1* through *<path>x-39!1* on the file server *host* using the FTP protocol. For example, if "Cabernet.ms" is a mail server and the individual "Log-Cabernet.ms" exists and has connect-site "[Ivy]<DMS>Log>", then files such as "<DMS>Log>Cabernet-09!1" will be stored on the file server "Ivy". If the required *Log-x.ms* individual does not exist, no attempt will be made to write the log files to a file server; the individual is examined only when the server is restarting, and is not revisited during normal running.

The Telnet server (known as the "Viticulturists' Entrance") provides various facilites to a remote terminal user. Some of these facilities are privileged: they are allowed only to an individual who has logged in, and who is a member of the group "Transport.ms" or whose name is "Wizard.gv" and they are only available after using the "Enable" command. Most commands can be stopped by typing DEL. The facilities change frequently, but at present are as follows.

Display Histograms:

Types histograms of various events. At present only mail retrieval delays.

Display Inboxes:

Types a summary of the state of clients' in-boxes on this server.

Display Other-Servers:

Types this server's view of the availability of other servers that it knows about.

Display Policy-Controls:

Types information about the various internal server controls on operations. This includes whether the operation is presently "allowed", how many instances of the operation are allowed simultaneously, the highest number that have occurred simultaneously, and the total number that have occurred.

Display Queues:

Types a summary of the queues of not-yet-delivered messages maintained by the mail server.

Display Statistics:

Types various statistics kept by the server, including server version and uptime.

Enable:

Allows use of the privileged commands if the logged in user is "Wizard.gv" or is a member of the group "Transport.ms".

Force Archive:

Forces transfer of the contents of an in-box to a backing file server. The choice of backing file server is described below.

Force Background-Process:

Forces immediate activation of one of the periodic processes in the server. The "Archiver" process scans inboxes looking for ones which should be written out to a remote file server (this normally hapens about 11 p.m. each the evening); the "ReadPending" process considers those messages which are on the pending queue because there was nowhere to deliver them (this normally happens every 15 minutes); the "RegPurger" process scans the naming database looking for information representing dead entries or removed members of lists which is more than 14 days old and can be removed from the disk (this normally hapens about 11 p.m. each the evening).

Force MSMail-Login:

Causes the process that reads mail server internal mail to login to GrapevineUser afresh, causing it to reconsider the location of its inboxes.

Force RSMail-Login:

Causes the process that reads registration server internal mail to login to GrapevineUser afresh, causing it to reconsider the location of its inboxes.

Force Purge:

Asks for an R-Name. If this R-Name corresponds to a dead entry in the naming database, removes that entry immediately (without waiting for the normal 14 day timeout). This allows immediat re-use of that name, but is incorrect unless all copies of the appropriate registry throughout Grapevine know that the entry was dead.

Login:

Asks for name and password, and attempts to authenticate the name by use of the naming database.

Maintain:

Enters the Maintain program, which allows maintenance of the naming database.

Quit:

Terminates this connection.

Restart:

Asks for a line to place in the local file "Rem.cm", asks for a comment to write in the log, asks for more confirmation, then stops the server.

Set Archive-Days:

Alters the timeout used by the in-box archiver process. This alteration applies only to the next run of the archiver, then the timeout reverts to its default 7 days.

Set Policy-Control:

Alters the state of various internal server controls from "allowed" to "not allowed", or *vice versa*. Setting the control to be "not allowed" prevents the operation in question from being started subsequently; it does not affect operations currently in progress. The controls form a hierarchy; all appropriate levels in the hierarchy must be "allowed" for an operation to be permitted. The controls in question are:

Work: must be allowed for any operation to be allowed
Connection: controls incoming connections other than the Viticulturists' Entrance:
ClientInput: controls mail submission connections
ServerInput: controls mail forwarding connections from other Grapevine servers
ReadMail: controls mail retrieval connections.
RegExpand: controls registration server enquiry connections.
Lily: controls Telnet server connections to any local Lily server.
MTP: controls MTP server connections from clients.
FTP: controls FTP server connections from clients.
Telnet: controls Viticulturists' Entrance connections.
MainLine: controls the following four operations:
ReadInput: controls processing messages queued for delivery.

ReadPending: controls processing messages from the "pending" queue.
ReadForward: controls forwarding message to other servers.
Remailing: controls remailing of messages from inboxes.

Background: controls the following operations:

RSReadMail: controls reading registration server database update messages
MSReadMail: controls reading mail server in-box re-mail requests.
Archiver: controls the transfer of in-boxes to remote file servers.
RegPurger: controls the nightly clean-up of the naming database.

Wait-until-idle:

Waits until the only activity in the server is this Telnet connection. This wait may also be terminated by typing DEL.

Appendix: The Mesa Grapevine Interfaces

This appendix contains those interfaces of the Mesa Grapevine client interface which were referred to earlier in this document.

```
-- Transport mechanism:  Client reading of R-Server database
-- [Juniper]<Grapevine>User>NameInfoDefs.mesa
-- Andrew Birrell  27-Oct-80 15:39:10

DIRECTORY

BodyDefs         USING[ Connect, oldestTime, Password, Remark, RName, Timestamp ];

NameInfoDefs: DEFINITIONS =

BEGIN

NameType:        TYPE = { noChange,
                          group, individual, notFound, allDown,
                          badPwd };
        -- represents the result states of enquiries --


-- "RLists" are sequences of R-Names returned from the server --

RListHandle:     TYPE[SIZE[POINTER]];

Enumerate:       PROC[list: RListHandle,
                      work: PROC[BodyDefs.RName]RETURNS[done: BOOLEAN] ];

Close:           PROC[list: RListHandle];


-- "Expand" returns mailbox site names for individuals, membership list
-- for groups. If the old stamp is still current, returns "noChange".  Will
-- not return "noChange" if the old stamp is defaulted. --

ExpandInfo:      TYPE = RECORD[ SELECT type: NameType[noChange..allDown] FROM
                         noChange => NULL,
                         group => [ members: RListHandle,
                                    stamp: BodyDefs.Timestamp ],
                         individual => [ sites: RListHandle,
                                    stamp: BodyDefs.Timestamp ],
                         notFound => NULL,
                         allDown => NULL,
                         ENDCASE ];

Expand:          PROC[name: BodyDefs.RName,
                      oldStamp: BodyDefs.Timestamp _ BodyDefs.oldestTime]
                 RETURNS[ ExpandInfo ];


-- "GetMembers" returns the membership list for a group. If the old stamp
-- is still current, returns "noChange".  Will not return "noChange" if
-- the old stamp is defaulted. --

MemberInfo:      TYPE = RECORD[ SELECT type: NameType[noChange..allDown] FROM
```

```
                        noChange => NULL,
                        group => [ members: RListHandle,
                                   stamp: BodyDefs.Timestamp ],
                        individual => NULL,
                        notFound => NULL,
                        allDown => NULL,
                        ENDCASE ];

GetMembers:     PROC[name: BodyDefs.RName,
                     oldStamp: BodyDefs.Timestamp _ BodyDefs.oldestTime]
                RETURNS[ MemberInfo ];

GetOwners:      PROC[name: BodyDefs.RName,
                     oldStamp: BodyDefs.Timestamp _ BodyDefs.oldestTime]
                RETURNS[ MemberInfo ];

GetFriends:     PROC[name: BodyDefs.RName,
                     oldStamp: BodyDefs.Timestamp _ BodyDefs.oldestTime]
                RETURNS[ MemberInfo ];


-- "CheckStamp" performs basic name validation, also telling the caller the
-- name type. If the old stamp is still current, returns "noChange".  Will
-- not return "noChange" if the old stamp is defaulted. --

StampInfo:      TYPE = NameType[noChange..allDown];

CheckStamp:     PROC[name: BodyDefs.RName,
                     oldStamp: BodyDefs.Timestamp _ BodyDefs.oldestTime]
                RETURNS[ StampInfo ];


-- "GetConnect" returns the connect-site for an individual.  "connect" is
-- undisturbed if the result is not "individual".  The connect-site is
-- either an NLS name or a net-address.  "connect.maxlength" should equal
-- "BodyDefs.maxConnectLength". --

ConnectInfo:    TYPE = NameType[group..allDown];

GetConnect:     PROC[name: BodyDefs.RName, connect: BodyDefs.Connect]
                RETURNS[ ConnectInfo ];


-- "GetRemark" returns the remark for a group.  "remark" is
-- undisturbed if the result is not "group".  The remark is a human readable
-- string.  "remark.maxlength" should equal "BodyDefs.maxRemarkLength". --

RemarkInfo:     TYPE = NameType[group..allDown];

GetRemark:      PROC[name: BodyDefs.RName, remark: BodyDefs.Remark]
                RETURNS[ RemarkInfo ];


-- "Authenticate" checks a user name and password. --

AuthenticateInfo: TYPE = NameType[group..badPwd];

Authenticate:   PROC[name: BodyDefs.RName, password: STRING]
                RETURNS[ AuthenticateInfo ];
```

```
AuthenticateKey:PROC[name: BodyDefs.RName, key: BodyDefs.Password]
                RETURNS[ AuthenticateInfo ];


-- Access control primitives --

Membership:      TYPE = { yes, no, notGroup, allDown };

IsMemberDirect:  PROC[name: BodyDefs.RName, member: BodyDefs.RName]
                 RETURNS[ Membership ];

IsOwnerDirect:   PROC[name: BodyDefs.RName, owner: BodyDefs.RName]
                 RETURNS[ Membership ];

IsFriendDirect:  PROC[name: BodyDefs.RName, friend: BodyDefs.RName]
                 RETURNS[ Membership ];

IsMemberClosure:PROC[name: BodyDefs.RName, member: BodyDefs.RName]
                 RETURNS[ Membership ];

IsOwnerClosure:  PROC[name: BodyDefs.RName, owner: BodyDefs.RName]
                 RETURNS[ Membership ];

IsFriendClosure:PROC[name: BodyDefs.RName, friend: BodyDefs.RName]
                 RETURNS[ Membership ];

END.
```

```
-- Transport Mechanism - DEFS for client sending mail --
-- [Juniper]<Grapevine>User>SendDefs.mesa
-- Andrew Birrell  23-Jan-81 11:00:14 --

DIRECTORY
BodyDefs         USING[ ItemType, Password, RName ];

SendDefs:        DEFINITIONS = BEGIN

-- These defs allow clients to inject messages into the mail system.  They
-- are designed so that they can be used by multiple processes creating
-- different messages;  the state of creation of a single message is
-- represented by a "Handle".  The interface is also designed so that it
-- may be implemented either by transmission over the network to a remote
-- mail server, or by calls on a local mail server. --

Handle:          TYPE[SIZE[POINTER]];

Create:          PROCEDURE RETURNS[handle: Handle];

Destroy:         PROCEDURE[ handle: Handle ];


-- For any one Handle, the following calls must be made in the order:
--       StartSend,
--       AddRecipient, AddRecipient, AddRecipient, . . .
--       CheckValidity {iff "validate=TRUE" when "StartSend" was called},
--       StartItem, (AddToItem, AddToItem, . . .), StartItem, (...), . . .
--       Send
-- Abort may be called at any point to abandon the sequence.


SendFailed:      ERROR[notDelivered: BOOLEAN];
   -- "StartSend" raises no signals that may be caught by the client.
   -- The ERROR "SendFailed" may be raised by any of AddRecipient,
   -- CheckValidity, StartItem, AddToItem, or Send if some communication
   -- failure occurs.  If it is raised, the client should generally catch
   -- it, go back to the start of the message, and re-call "StartSend".
   -- "StartSend" will then attempt to find a better mail server to talk
   -- to.  Only when "StartSend" returns "allDown" is it not possible to
   -- send the message.  The client may want to inform the user if this
   -- re-try mechanism has been invoked.


StartSendInfo:   TYPE = { ok, badPwd, badSender, badReturnTo, allDown };

StartSend:       PROC[ handle:    Handle,
                       senderPwd: STRING,
                       sender:    BodyDefs.RName,
                       returnTo:  BodyDefs.RName _ NIL,
                       validate:  BOOLEAN ]
                 RETURNS[ StartSendInfo ];
   -- Starts a message.  If "returnTo" is NIL, the sender name is used as
   -- return-to name.  "validate" says whether recipient names should be
   -- validated during the communication with the mail server. --


SendFromClient:  PROC[ handle:    Handle,
                       fromNet:   [0..256),
```

```
                        fromHost:  [0..256),
                        senderKey: BodyDefs.Password,
                        sender:    BodyDefs.RName,
                        returnTo:  BodyDefs.RName,
                        validate:  BOOLEAN ]
                  RETURNS[ StartSendInfo ];
   -- Note: this procedure is intended for use only by the remote server.
   -- Starts a message.  "fromNet" and "fromHost" are ignored if the mail
   -- server is remote.  "validate" says whether recipient names should be
   -- validated during the communication with the mail server. --


AddRecipient:    PROC[ handle: Handle, recipient: BodyDefs.RName ];
   -- Adds to the recipient list. --


CheckValidity:   PROC[ handle: Handle,
                        notify: PROCEDURE[CARDINAL,BodyDefs.RName] ]
                  RETURNS[ ok: CARDINAL ];
   -- Must be called after all the recipients have been given, iff the
   -- "validate" argument to "StartSend" was TRUE.  Calls "notify" for each
   -- bad recipient.  The arguments to "notify" are the recipient number
   -- (counting from 1) and name of an illegal recipient.  Returns the
   -- number of valid recipients.  If any recipients were invalid, delivery
   -- of the message is still allowed.


StartItem:       PROC[ handle: Handle, type: BodyDefs.ItemType ];
   -- Start a message body item.  The type must not be "Postmark",
   -- "Sender", "ReturnTo", or "Recipients". --


StartText:       PROC[ handle: Handle ] = INLINE{ StartItem[handle,Text] };


AddToItem:       PROC[ handle: Handle,
                        buffer: DESCRIPTOR FOR PACKED ARRAY OF CHARACTER ];
   -- Add the data to the current message body item. --


Send:            PROC[ handle: Handle ];
   -- Commit to sending the message; returns only when the mail server has
   -- commited to delivering the message. --


Abort:           PROC[ handle: Handle ];
   -- Abandon the message.  May be called at any time. --


ExpandInfo:      TYPE = { ok, notFound, individual, allDown};

ExpandFailed:    ERROR;

Expand:          PROC[name: BodyDefs.RName, work: PROC[BodyDefs.RName]]
                  RETURNS[ ExpandInfo ];
   -- If the name will be interpreted by the mail server as a distribution
   -- list, enumerates the names which are direct members of that list.
   -- This is intended for use only if the user wants to inspect the
   -- contents.  Note that the contents may change, or the name may become
```

```
   -- invalid, before delivery of any message.  "Expand" works even if the
   -- list has to be read from an MTP server.  May raise "ExpandFailed".
   -- If ExpandFailed is raised, some communication error has occurred;
   -- you should re-call Expand, which will try another server.  Note that
   -- failure of Expand may be caused by failure of some remote server;
   -- you may still be able to send a message successfully.
   -- "notFound" means the name is invalid; "individual" means the name
   -- specifies an individual; "allDown" means either all mail servers
   -- are inaccessible, or some other server (possibly MTP) needed for the
   -- expansion is inaccessible.

END.
```

```
-- Transport Mechanism - DEFS for retrieval of new mail from GV Servers --
-- [Juniper]<Grapevine>User>RetrieveDefs.mesa
-- M. D. Schroeder  February 20, 1980  5:05 PM --
-- Andrew Birrell  21-Jan-81 17:13:34 --

DIRECTORY
BodyDefs         USING[ ItemHeader, RName, Timestamp ];

RetrieveDefs:    DEFINITIONS = BEGIN

-- No procedures in this interface other than the "AccessProcs" returned by
-- "NextServer" ever raise a SIGNAL or ERROR.



Handle:          TYPE[SIZE[POINTER]];
   -- This interface is intended to be able to be used by multiple clients.
   -- They are distinquished by a "handle", created by "Create" and
   -- destroyed by "Destroy" --


Create:          PROC[pollingInterval: CARDINAL,
                      reportChanges: PROCEDURE[MBXState] _ NIL]
                 RETURNS[Handle];
   -- Must be called before any other entries in this interface.  Can be
   -- called many times.  "pollingInterval" is the interval in seconds to
   -- wait between successive inbox checks and "reportChanges" (if
   -- provided) is called  whenever the state of the user's authentication
   -- or mailboxes changes;  "reportChanges" will not be called if the
   -- state changes to "unknown" or "userOK".


Destroy:         PROC[Handle];
   -- Terminates use of this handle, releasing all resources used by it. --



-- AUTHENTICATION AND MAILBOX POLLING --

NewUser:         PROC[ handle: Handle, user: BodyDefs.RName,
                       password: STRING];
   -- Provides new user name and password, and starts authentication and
   -- mailbox checking.


MBXState:        TYPE = { unknown, badName, badPwd, cantAuth, userOK,
                          allDown, someEmpty, allEmpty, notEmpty };
   -- Records current state of the user's mailboxes. Initially "unknown".
   -- Set to "badName", "badPwd", "cantAuth" or "userOK" after
   -- authentication check.  Set to "allDown", "someEmpty", "allEmpty", or
   -- "notEmpty" after mail polling is complete.  "someEmpty" means not all
   -- servers replied and none had mail; "allEmpty" means all replied and
   -- none had mail; "notEmpty" means at least one has mail; "allDown"
   -- means none replied.

MailboxState:    PROC[ handle: Handle] RETURNS[ state: MBXState];
   -- Returns the current mailbox state.  Will not return "unknown" or
   -- "userOK" (These change to "cantAuth" or "allDown" after suitable
   -- timeouts if necessary.)
```

```
WaitForMail:     PROC[ handle: Handle ];
   -- returns only when there is likely to be mail for the user --
   -- Possible ERRORS: none


SetMTPRetrieveDefault: PROC[host, reg: STRING];
   -- records "host" and "reg", and subsequently if the user name is such
   -- that its registry is an MTP registry, and its registry equals "reg",
   -- then the retrieve host is forced to be "host".
   -- NB: This is a temporary facility for the benefit of Laurel.



-- ACCESS TO MAILBOXES --

-- The intended use is as follows.

-- The user has a number of mailboxes, each of which is on an MTP server or
-- on a Grapevine server.  To access all of a client's mail, call
-- "NextServer" repeatedly until it returns noMore=TRUE.  For each
-- successful call of "NextServer", use the AccessProcs to read the mail in
-- the mailbox.

-- For either type of server, call "nextMessage" until it returns
-- msgExists=FALSE.  The first call of "nextMessage" for each server will
-- attempt to create a stream to the server (signalling if it fails).
-- While accessing a mailbox, "Failed" may be signalled at any time if the
-- communication system fails (because of network or server error).  If
-- "Failed" is signalled, no further operations on this mailbox are allowed

--  If "nextMessage" returns deleted=TRUE it indicates that the message is
-- really just a placeholder and has been removed from the mailbox; you
-- should not attempt to access the message.  Returning archived=TRUE
-- indicates that the message has been spilled to some file server, and
-- accessing it is likely to be much slower.  For each message that exists
-- and is not deleted, the message may be manipulated by the other
-- procedures provided.

-- If the server type is GV, "readTOC" may be used to read any TOC entry
-- for the message (giving length=0 if there is no TOC entry), then
-- "startMessage" may be called to read the guaranteed properties of the
-- message; these are not available for MTP servers; these may not be
-- called after you have called "nextItem" for this message.

-- For either type of server, "nextItem" may be called to access in
-- sequence the items which are the contents of the message body.  Note
-- that the ItemHeader contains the item type and length in bytes.  For an
-- MTP server, the only item will be of type "text".  For a GV server, the
-- first item will be the guaranteed recipient list.  For all servers, the
-- message body is followed by an item of type "LastItem".  Within an item,
-- use "nextBlock" to access the data of the item.  Each call of
-- "nextBlock" within an item will fill its buffer if the data exists;  the
-- end of the item is indicated by "nextBlock" returning 0.

-- If the server is GV, you may call "writeTOC" to change or create a TOC
-- entry for the message, or you may call "deleteMessage" to remove this
-- single message from the mailbox;  "readTOC", "startMessage", "nextItem"
```

```
-- or "nextBlock" may not be called after calling "writeTOC" or
-- "deleteMessage" for this message.

-- At any time within an item, you may call "nextItem" to skip the
-- remainder of the item;  at any time within a message, you may call
-- "nextMessage" to skip the remainder of this message.

-- At any time within a mailbox, you may call "accept".  This
-- terminates reading the mailbox and deletes all messages from
-- the mailbox.  Calling "accept" will not delete any messages which you
-- haven't been given a chance to read.  No other operations on the mailbox
-- are allowed after calling "accept".  If you call "NextServer" without
-- having called "accept", the mailbox is closed (if necessary) without
-- deleting the messages (except those which were deleted by calling
-- "deleteMessage").


ServerType:      TYPE = { MTP, GV };

ServerState:     TYPE = { unknown, empty, notEmpty };
   -- "unknown" means the server didn't reply to mail check packets.

AccessProcs:     TYPE = RECORD[ -- procedures to access mailbox --
        nextMessage:     PROC[handle: Handle]
                         RETURNS[msgExists, archived, deleted: BOOLEAN],
        nextItem:        PROC[handle: Handle]
                         RETURNS[BodyDefs.ItemHeader],
        nextBlock:       PROC[handle: Handle,
                              buffer: DESCRIPTOR FOR PACKED ARRAY OF CHARACTER]
                         RETURNS[bytes: CARDINAL],
        accept:          PROC[handle: Handle],
        extra:  SELECT type: ServerType FROM
            MTP => NULL,
            GV => [
                 readTOC:         PROC[handle: Handle, text: STRING],
                 startMessage:    PROC[handle: Handle,
                            postmark: POINTER TO BodyDefs.Timestamp _ NIL,
                            sender: BodyDefs.RName _ NIL,
                            returnTo: BodyDefs.RName _ NIL],
                 writeTOC:        PROC[handle: Handle, text: STRING],
                 deleteMessage:   PROC[handle: Handle] ],
          ENDCASE
        ];


NextServer:      PROCEDURE[ handle: Handle ]
                   RETURNS[ noMore: BOOLEAN,
                            state: ServerState,
                            procs: AccessProcs ];
   -- Returns information about the next server in the mailbox site list of
   -- the user, and that server becomes the "current server".  If there is
   -- no such server, noMore=TRUE, in which case the next call to
   -- "NextServer" will start a new sequence of mail retrieval.  If the
   -- state is "unknown", attempting to access the mailbox is inadvisable,
   -- as the server is probably down.  If the state is "empty", there may
   -- in fact be mail, as the state is only a hint obtained by polling.


ServerName:      PROC[ handle: Handle,
```

```
                        serverName: BodyDefs.RName];
   -- Provides the name of the current server.  For MTP registries, this
   -- will be equivalent to the registry name.


FailureReason:   TYPE = { communicationFailure, -- server or network down --
                          noSuchServer,         -- server name incorrect --
                          connectionRejected,   -- server full, mbx busy, etc --
                          badCredentials,       -- name/pwd rejected --
                          unknownFailure        -- protocol violation
                                                 -- or unknown MTP error:
                                                 -- likely to be permanent --
                        };

Failed:          ERROR[why: FailureReason];
   -- May be signalled by any of the "AcceptProcs" returned by "NextServer"

END.
```

```
-- Transport Mechanism - DEFS for location of server by client --
-- [Juniper]<Grapevine>User>LocateDefs.mesa
-- Andrew Birrell  14-Aug-80 12:06:21 --

DIRECTORY
BodyDefs         USING[ RName ],
PupDefs          USING[ PupAddress ];

LocateDefs:      DEFINITIONS = BEGIN

FoundState:      TYPE = { allDown, notFound, found };

FoundServerInfo: TYPE = RECORD[ SELECT t: FoundState FROM
                         allDown =>       NULL,
                         notFound =>      NULL,
                         found =>         [where: PupDefs.PupAddress],
                         ENDCASE ];

FindNearestServer: PROCEDURE[list: BodyDefs.RName,
                    accept: PROCEDURE[PupDefs.PupAddress]RETURNS[BOOLEAN] ]
                   RETURNS[ FoundServerInfo ];

FindLocalServer: PROCEDURE[list, local: BodyDefs.RName ]
                 RETURNS[ FoundState ];

FindRegServer: PROCEDURE[ who: BodyDefs.RName,
                  accept: PROCEDURE[PupDefs.PupAddress]RETURNS[BOOLEAN] ]
               RETURNS[ FoundServerInfo ];

AcceptFirst:     PROC[PupDefs.PupAddress]RETURNS[BOOLEAN];
                 -- returns TRUE --

END.
```

```
-- Transport Mechanism: DEFS for message body layout --
-- [Juniper]<Grapevine>User>BodyDefs.mesa
-- Andrew Birrell  14-Aug-80 10:31:53 --

BodyDefs:   DEFINITIONS =

BEGIN

-- Note that incompatible changes to these definitions may require the
-- cooperation of all mail servers and their clients, and the flushing of
-- the mail server filestores. --


-- The following types are basic to the transport mechanism.
-- "Connect", "Password" and "Remark" don't really occur in message bodies,
-- but this is the most stable defs file for them. They are used by public
-- clients of the transport mechanism.


-- An R-Name (Recipient-name) is the basic name within the transport
-- mechanism.  It is of the form SN.Reg ( Simple-Name . Registry ).  The
-- representation is as a string of up to maxRNameLength characters.
-- In message bodies, R-Names occupy an integral number of words.

RName:          TYPE = STRING;

maxRNameLength:  CARDINAL = 64;

RNameSize:      PROC[name: RName] RETURNS[CARDINAL] = INLINE
   { RETURN[SIZE[StringBody[name.length]]] };


-- "Connect" is the representation of a connect-site for an individual
-- (typically a server).  It is a string which is either an NLS name
-- or a PUP address.

Connect:        TYPE = STRING;

maxConnectLength: CARDINAL = 64;


-- "Remark" is the representation of a remark associated with a group,
-- or of a TOC entry in a mailbox.  It is a human readable string.

Remark:         TYPE = STRING;

maxRemarkLength: CARDINAL = 64;


-- "Password" is the representation of an individual's encryption key.
-- It is intended to be used with the DES encryption algorithm.

Password:       TYPE = ARRAY[0..3] OF CARDINAL;


-- The following definitions are concerned with the layout of "message bodies".  A message
body is the internal representation of a message within and between mail servers.  It is
also sent to the client when he retrieves his mail.  A message body contains a number of
"items".  Items are used to represent such things as postmark, recipients, sender, as well
```

```
as the message text (if any), or other content of the message such as audio or
capabilities.  Some items are mandatory and always occur precisely once, others may occur
any number of times (including zero).  Each Item has a header, followed by the number of
bytes of data specified by the header, followed by an extra byte if its length is odd.
Thus items always start at a word boundary.  A complete message body consists of the
mandatory items followed by the optional ones. --


-- Time stamps --

Timestamp:       TYPE = MACHINE DEPENDENT RECORD[
                     net:  [0..256), -- the PUP net number --
                     host: [0..256), -- the PUP host number --
                     time: PackedTime];

PackedTime:      TYPE = LONG CARDINAL;
                     -- the number of seconds since midnight, January 1,
                     -- 1901 GMT --

oldestTime:      Timestamp = [net:0, host:0, time:0];



-- Layout of items --

ItemHeader:      TYPE = MACHINE DEPENDENT RECORD[
                     type:       ItemType,
                     length:     ItemLength ];
                     -- Each item consists of an ItemHeader followed by a
                     -- variable length array, containing the number of
                     -- bytes specified by the length.  The item is
                     -- followed by an extra byte if its length is odd. --

Item:            TYPE = POINTER TO ItemHeader;

ItemLength:      TYPE = LONG CARDINAL;
                     -- Number of data bytes in the item, excluding header--

ItemType:        TYPE = MACHINE DEPENDENT {

--- Mandatory items --

-- In each message body, each of these items occurs precisely once, and they occur in the
order given here --

PostMark(10B),   -- the item contains a timestamp giving the
                 -- originating host and approximate time at which the
                 -- message was given to the transport mechanism. --

Sender(20B),     -- the item contains precisely one R-Name, being that
                 -- of the sender of this message --

ReturnTo(30B),   -- the item contains precisely one R-Name, being that
                 -- of the client to whom non-delivery of the message
                 -- should be notified --

Recipients(40B), -- the item contains a sequence of R-Names, being the
                 -- intended recipients of this message, as provided by
                 -- the sender --
```

```
-- Items used solely by clients --

Text(1010B),     -- the item contains a sequence of characters forming
                 -- a textual message --

Capability(1020B), -- the item contains a capability --

Audio(1030B),    -- the item is an audio message --


-- Items used in registration server internal mail --

updateItem(2000B), -- the item contains a registration server entry --

reMail(2100B),   -- the item is internal mail to a mail server,
                 -- containing precisely one R-Name, indicating that
                 -- the corresponding mailbox should be re-mailed --


-- Mandatory last item --

LastItem(LAST[CARDINAL]) -- the item contains no data, and always occurs
                 -- as the last item in a message body --

        };

END.
```

```
-- Transport mechanism: DEFS for lists of R-Names
-- [Juniper]<Grapevine>User>RListDefs.mesa
-- Andrew Birrell   4-Aug-80 16:14:17

DIRECTORY
BodyDefs         USING[ RName ],
ProtocolDefs     USING[ Handle ];

RListDefs: DEFINITIONS =

BEGIN

-- These defs are intended to be implemented on the disk heap in the server
-- and in main memory in the GV-User package --

RListHandle:     TYPE[ SIZE[POINTER] ];

Receive:         PROC[ str: ProtocolDefs.Handle ]
                 RETURNS[ RListHandle ];
                 -- may raise ProtocolDefs.Failed --

Enumerate:       PROC[ list: RListHandle,
                       work: PROC[BodyDefs.RName]RETURNS[done:BOOLEAN] ];

Close:           PROC[list: RListHandle ];

END.
```