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 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. Note that this algorithm does not preserve the information content of the password text, and so is not ideal for encryption based protocols.

```
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 research 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 servers) 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. Those recipients may themselves be groups. 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 foreign 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 (other than mailbox site lists, described later) 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 restarting or 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 about 14 days 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 testing 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-arrow" 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 whose value is initialized from the present value of the second name.