CVS Client/Server

This document describes the client/server protocol used by CVS. It does not describe how to use or administer client/server CVS; see the regular CVS manual for that. This is version 1.11.1p1 of the protocol specification—See Chapter 1 [Introduction], page 2, for more on what this version number means.

1 Introduction

CVS is a version control system (with some additional configuration management functionality). It maintains a central *repository* which stores files (often source code), including past versions, information about who modified them and when, and so on. People who wish to look at or modify those files, known as *developers*, use CVS to *check out* a *working directory* from the repository, to *check in* new versions of files to the repository, and other operations such as viewing the modification history of a file. If developers are connected to the repository by a network, particularly a slow or flaky one, the most efficient way to use the network is with the CVS-specific protocol described in this document.

Developers, using the machine on which they store their working directory, run the CVS *client* program. To perform operations which cannot be done locally, it connects to the CVS *server* program, which maintains the repository. For more information on how to connect see Chapter 3 [Connection and Authentication], page 4.

This document describes the CVS protocol. Unfortunately, it does not yet completely document one aspect of the protocol—the detailed operation of each CVS command and option—and one must look at the CVS user documentation, '`cvs.texinfo`', for that information. The protocol is non-proprietary (anyone who wants to is encouraged to implement it) and an implementation, known as CVS, is available under the GNU Public License. The CVS distribution, containing this implementation, '`cvs.texinfo`', and a copy (possibly more or less up to date than what you are reading now) of this document, '`cvsclient.texi`', can be found at the usual GNU FTP sites, with a filename such as '`cvs-`*version*`.tar.gz`'.

This is version 1.11.1p1 of the protocol specification. This version number is intended only to aid in distinguishing different versions of this specification. Although the specification is currently maintained in conjunction with the CVS implementation, and carries the same version number, it also intends to document what is involved with interoperating with other implementations (such as other versions of CVS); see Section 5.14 [Requirements], page 31. This version number should not be used by clients or servers to determine what variant of the protocol to speak; they should instead use the `valid-requests` and `Valid-responses` mechanism (see Chapter 5 [Protocol], page 7), which is more flexible.

2 Goals

- Do not assume any access to the repository other than via this protocol. It does not depend on NFS, rdist, etc.
- Providing a reliable transport is outside this protocol. The protocol expects a reliable transport that is transparent (that is, there is no translation of characters, including characters such as such as linefeeds or carriage returns), and can transmit all 256 octets (for example for proper handling of binary files, compression, and encryption). The encoding of characters specified by the protocol (the names of requests and so on) is the invariant ISO 646 character set (a subset of most popular character sets including ASCII and others). For more details on running the protocol over the TCP reliable transport, see Chapter 3 [Connection and Authentication], page 4.
- Security and authentication are handled outside this protocol (but see below about '`cvs kserver`' and '`cvs pserver`').
- The protocol makes it possible for updates to be atomic with respect to checkins; that is if someone commits changes to several files in one cvs command, then an update by someone else would either get all the changes, or none of them. The current CVS server can't do this, but that isn't the protocol's fault.
- The protocol is, with a few exceptions, transaction-based. That is, the client sends all its requests (without waiting for server responses), and then waits for the server to send back all responses (without waiting for further client requests). This has the advantage of minimizing network turnarounds and the disadvantage of sometimes transferring more data than would be necessary if there were a richer interaction. Another, more subtle, advantage is that there is no need for the protocol to provide locking for features such as making checkins atomic with respect to updates. Any such locking can be handled entirely by the server. A good server implementation (such as the current CVS server) will make sure that it does not have any such locks in place whenever it is waiting for communication with the client; this prevents one client on a slow or flaky network from interfering with the work of others.
- It is a general design goal to provide only one way to do a given operation (where possible). For example, implementations have no choice about whether to terminate lines with linefeeds or some other character(s), and request and response names are case-sensitive. This is to enhance interoperability. If a protocol allows more than one way to do something, it is all too easy for some implementations to support only some of them (perhaps accidentally).

3 How to Connect to and Authenticate Oneself to the CVS server

Connection and authentication occurs before the CVS protocol itself is started. There are several ways to connect.

server
: If the client has a way to execute commands on the server, and provide input to the commands and output from them, then it can connect that way. This could be the usual rsh (port 514) protocol, Kerberos rsh, SSH, or any similar mechanism. The client may allow the user to specify the name of the server program; the default is `cvs`. It is invoked with one argument, `server`. Once it invokes the server, the client proceeds to start the cvs protocol.

kserver
: The kerberized server listens on a port (in the current implementation, by having inetd call "cvs kserver") which defaults to 1999. The client connects, sends the usual kerberos authentication information, and then starts the cvs protocol. Note: port 1999 is officially registered for another use, and in any event one cannot register more than one port for CVS, so GSS-API (see below) is recommended instead of kserver as a way to support kerberos.

pserver
: The name *pserver* is somewhat confusing. It refers to both a generic framework which allows the CVS protocol to support several authentication mechanisms, and a name for a specific mechanism which transfers a username and a clear-text password. Servers need not support all mechanisms, and in fact servers will typically want to support only those mechanisms which meet the relevant security needs.

  The pserver server listens on a port (in the current implementation, by having inetd call "cvs pserver") which defaults to 2401 (this port is officially registered). The client connects, and sends the following:

  - the string '`BEGIN AUTH REQUEST`', a linefeed,
  - the cvs root, a linefeed,
  - the username, a linefeed,
  - the password trivially encoded (see Chapter 4 [Password scrambling], page 6), a linefeed,
  - the string '`END AUTH REQUEST`', and a linefeed.

  The client must send the identical string for cvs root both here and later in the `Root` request of the cvs protocol itself. Servers are encouraged to enforce this restriction. The possible server responses (each of which is followed by a linefeed) are the following. Note that although there is a small similarity between this authentication protocol and the cvs protocol, they are separate.

  `I LOVE YOU`
  : The authentication is successful. The client proceeds with the cvs protocol itself.

  `I HATE YOU`
  : The authentication fails. After sending this response, the server may close the connection. It is up to the server to decide whether to give this response, which is generic, or a more specific response using '`E`' and/or '`error`'.

`E` *text* Provide a message for the user. After this reponse, the authentication protocol continues with another response. Typically the server will provide a series of '`E`' responses followed by '`error`'. Compatibility note: CVS 1.9.10 and older clients will print `unrecognized auth response` and *text*, and then exit, upon receiving this response.

`error` *code text*
The authentication fails. After sending this response, the server may close the connection. The *code* is a code describing why it failed, intended for computer consumption. The only code currently defined is '`0`' which is nonspecific, but clients must silently treat any unrecognized codes as nonspecific. The *text* should be supplied to the user. Compatibility note: CVS 1.9.10 and older clients will print `unrecognized auth response` and *text*, and then exit, upon receiving this response. Note that *text* for this response, or the *text* in an `E` response, is not designed for machine parsing. More vigorous use of *code*, or future extensions, will be needed to prove a cleaner machine-parseable indication of what the error was.

If the client wishes to merely authenticate without starting the cvs protocol, the procedure is the same, except BEGIN AUTH REQUEST is replaced with BEGIN VERIFICATION REQUEST, END AUTH REQUEST is replaced with END VERIFICATION REQUEST, and upon receipt of I LOVE YOU the connection is closed rather than continuing.

Another mechanism is GSSAPI authentication. GSSAPI is a generic interface to security services such as kerberos. GSSAPI is specified in RFC2078 (GSSAPI version 2) and RFC1508 (GSSAPI version 1); we are not aware of differences between the two which affect the protocol in incompatible ways, so we make no attempt to specify one version or the other. The procedure here is to start with '`BEGIN GSSAPI REQUEST`'. GSSAPI authentication information is then exchanged between the client and the server. Each packet of information consists of a two byte big endian length, followed by that many bytes of data. After the GSSAPI authentication is complete, the server continues with the responses described above ('`I LOVE YOU`', etc.).

future possibilities
There are a nearly unlimited number of ways to connect and authenticate. One might want to allow access based on IP address (similar to the usual rsh protocol but with different/no restrictions on ports `<` 1024), to adopt mechanisms such as Pluggable Authentication Modules (PAM), to allow users to run their own servers under their own usernames without root access, or any number of other possibilities. The way to add future mechanisms, for the most part, should be to continue to use port 2401, but to use different strings in place of '`BEGIN AUTH REQUEST`'.

4 Password scrambling algorithm

The pserver authentication protocol, as described in Chapter 3 [Connection and Authentication], page 4, trivially encodes the passwords. This is only to prevent inadvertent compromise; it provides no protection against even a relatively unsophisticated attacker. For comparison, HTTP Basic Authentication (as described in RFC2068) uses BASE64 for a similar purpose. CVS uses its own algorithm, described here.

The scrambled password starts with '`A`', which serves to identify the scrambling algorithm in use. After that follows a single octet for each character in the password, according to a fixed encoding. The values are shown here, with the encoded values in decimal. Control characters, space, and characters outside the invariant ISO 646 character set are not shown; such characters are not recommended for use in passwords. There is a long discussion of character set issues in Chapter 6 [Protocol Notes], page 32.

		0	111			P	125			p	58
!	120	1	52	A	57	Q	55	a	121	q	113
"	53	2	75	B	83	R	54	b	117	r	32
		3	119	C	43	S	66	c	104	s	90
		4	49	D	46	T	124	d	101	t	44
%	109	5	34	E	102	U	126	e	100	u	98
&	72	6	82	F	40	V	59	f	69	v	60
'	108	7	81	G	89	W	47	g	73	w	51
(	70	8	95	H	38	X	92	h	99	x	33
)	64	9	65	I	103	Y	71	i	63	y	97
*	76	:	112	J	45	Z	115	j	94	z	62
+	67	;	86	K	50			k	93		
,	116	<	118	L	42			l	39		
-	74	=	110	M	123			m	37		
.	68	>	122	N	91			n	61		
/	87	?	105	O	35	_	56	o	48		

5 The CVS client/server protocol

In the following, '`\n`' refers to a linefeed and '`\t`' refers to a horizontal tab; *requests* are what the client sends and *responses* are what the server sends. In general, the connection is governed by the client—the server does not send responses without first receiving requests to do so; see Section 5.9 [Response intro], page 22 for more details of this convention.

It is typical, early in the connection, for the client to transmit a `Valid-responses` request, containing all the responses it supports, followed by a `valid-requests` request, which elicits from the server a `Valid-requests` response containing all the requests it understands. In this way, the client and server each find out what the other supports before exchanging large amounts of data (such as file contents).

5.1 Entries Lines

Entries lines are transmitted as:

/ *name* / *version* / *conflict* / *options* / *tag_or_date*

tag_or_date is either '`T`' *tag* or '`D`' *date* or empty. If it is followed by a slash, anything after the slash shall be silently ignored.

version can be empty, or start with '`0`' or '`-`', for no user file, new user file, or user file to be removed, respectively.

conflict, if it starts with '`+`', indicates that the file had conflicts in it. The rest of *conflict* is '`=`' if the timestamp matches the file, or anything else if it doesn't. If *conflict* does not start with a '`+`', it is silently ignored.

options signifies the keyword expansion options (for example '`-ko`'). In an `Entry` request, this indicates the options that were specified with the file from the previous file updating response (see Section 5.9 [Response intro], page 22, for a list of file updating responses); if the client is specifying the '`-k`' or '`-A`' option to `update`, then it is the server which figures out what overrides what.

5.2 File Modes

A mode is any number of repetitions of

mode-type = *data*

separated by '`,`'.

mode-type is an identifier composed of alphanumeric characters. Currently specified: '`u`' for user, '`g`' for group, '`o`' for other (see below for discussion of whether these have their POSIX meaning or are more loose). Unrecognized values of *mode-type* are silently ignored.

data consists of any data not containing '`,`', '`\0`' or '`\n`'. For '`u`', '`g`', and '`o`' mode types, data consists of alphanumeric characters, where '`r`' means read, '`w`' means write, '`x`' means execute, and unrecognized letters are silently ignored.

The two most obvious ways in which the mode matters are: (1) is it writeable? This is used by the developer communication features, and is implemented even on OS/2 (and could be implemented on DOS), whose notion of mode is limited to a readonly bit. (2) is it executable? Unix CVS users need CVS to store this setting (for shell scripts and the like).

The current CVS implementation on unix does a little bit more than just maintain these two settings, but it doesn't really have a nice general facility to store or version control the mode, even on unix, much less across operating systems with diverse protection features. So all the ins and outs of what the mode means across operating systems haven't really been worked out (e.g. should the VMS port use ACLs to get POSIX semantics for groups?).

5.3 Conventions regarding transmission of file names

In most contexts, '`/`' is used to separate directory and file names in filenames, and any use of other conventions (for example, that the user might type on the command line) is converted to that form. The only exceptions might be a few cases in which the server provides a magic cookie which the client then repeats verbatim, but as the server has not yet been ported beyond unix, the two rules provide the same answer (and what to do if future server ports are operating on a repository like e:/foo or CVS_ROOT:[FOO.BAR] has not been carefully thought out).

Characters outside the invariant ISO 646 character set should be avoided in filenames. This restriction may need to be relaxed to allow for characters such as '`[`' and '`]`' (see above about non-unix servers); this has not been carefully considered (and currently implementations probably use whatever character sets that the operating systems they are running on allow, and/or that users specify). Of course the most portable practice is to restrict oneself further, to the POSIX portable filename character set as specified in POSIX.1.

5.4 File transmissions

File contents (noted below as *file transmission*) can be sent in one of two forms. The simpler form is a number of bytes, followed by a linefeed, followed by the specified number of bytes of file contents. These are the entire contents of the specified file. Second, if both client and server support '`gzip-file-contents`', a '`z`' may precede the length, and the 'file contents' sent are actually compressed with '`gzip`' (RFC1952/1951) compression. The length specified is that of the compressed version of the file.

In neither case are the file content followed by any additional data. The transmission of a file will end with a linefeed iff that file (or its compressed form) ends with a linefeed.

The encoding of file contents depends on the value for the '`-k`' option. If the file is binary (as specified by the '`-kb`' option in the appropriate place), then it is just a certain number of octets, and the protocol contributes nothing towards determining the encoding (using the file name is one widespread, if not universally popular, mechanism). If the file is text (not binary), then the file is sent as a series of lines, separated by linefeeds. If the keyword expansion is set to something other than '`-ko`', then it is expected that the file conform to the RCS expectations regarding keyword expansion—in particular, that it is in a character set such as ASCII in which 0x24 is a dollar sign ('`$`').

5.5 Strings

In various contexts, for example the `Argument` request and the `M` response, one transmits what is essentially an arbitrary string. Often this will have been supplied by the user (for example, the '`-m`' option to the `ci` request). The protocol has no mechanism to specify the character set of such strings; it would be fairly safe to stick to the invariant ISO 646 character set but the existing practice is probably to just transmit whatever the user specifies, and hope that everyone involved agrees which character set is in use, or sticks to a common subset.

5.6 Dates

The protocol contains times and dates in various places.

For the '`-D`' option to the `annotate`, `co`, `diff`, `export`, `history`, `rannotate`, `rdiff`, `rtag`, `tag`, and `update` requests, the server should support two formats:

```
26 May 1997 13:01:40 -0000  ; RFC 822 as modified by RFC 1123
5/26/1997 13:01:40 GMT      ; traditional
```

The former format is preferred; the latter however is sent by the CVS command line client (versions 1.5 through at least 1.9).

For the '`-d`' option to the `log` and `rlog` requests, servers should at least support RFC 822/1123 format. Clients are encouraged to use this format too (the command line CVS client, version 1.10 and older, just passed along the date format specified by the user, however).

The `Mod-time` response and `Checkin-time` request use RFC 822/1123 format (see the descriptions of that response and request for details).

For `Notify`, see the description of that request.

5.7 Request intro

By convention, requests which begin with a capital letter do not elicit a response from the server, while all others do – save one. The exception is '`gzip-file-contents`'. Unrecognized requests will always elicit a response from the server, even if that request begins with a capital letter.

The term *command* means a request which expects a response (except `valid-requests`). The general model is that the client transmits a great number of requests, but nothing happens until the very end when the client transmits a command. Although the intention is that transmitting several commands in one connection should be legal, existing servers probably have some bugs with some combinations of more than one command, and so clients may find it necessary to make several connections in some cases. This should be thought of as a workaround rather than a desired attribute of the protocol.

5.8 Requests

Here are the requests:

`Root` *pathname* `\n`

Response expected: no. Tell the server which `CVSROOT` to use. Note that *pathname* is a local directory and *not* a fully qualified `CVSROOT` variable. *pathname* must already exist; if creating a new root, use the `init` request, not `Root`. *pathname* does not include the hostname of the server, how to access the server, etc.; by the time the CVS protocol is in use, connection, authentication, etc., are already taken care of.

The `Root` request must be sent only once, and it must be sent before any requests other than `Valid-responses`, `valid-requests`, `UseUnchanged`, `Set`, `Global_option`, `init`, `noop`, or `version`.

`Valid-responses` *request-list* `\n`

Response expected: no. Tell the server what responses the client will accept. request-list is a space separated list of tokens. The `Root` request need not have been previously sent.

`valid-requests \n`

Response expected: yes. Ask the server to send back a `Valid-requests` response. The `Root` request need not have been previously sent.

`Directory` *local-directory* `\n`

Additional data: *repository* `\n`. Response expected: no. Tell the server what directory to use. The *repository* should be a directory name from a previous server response. Note that this both gives a default for `Entry` and `Modified` and also for `ci` and the other commands; normal usage is to send `Directory` for each directory in which there will be an `Entry` or `Modified`, and then a final `Directory` for the original directory, then the command. The *local-directory* is relative to the top level at which the command is occurring (i.e. the last `Directory` which is sent before the command); to indicate that top level, '`.`' should be sent for *local-directory*.

Here is an example of where a client gets *repository* and *local-directory*. Suppose that there is a module defined by

```
moddir 1dir
```

That is, one can check out `moddir` and it will take `1dir` in the repository and check it out to `moddir` in the working directory. Then an initial check out could proceed like this:

```
C: Root /home/kingdon/zwork/cvsroot
. . .
C: Argument moddir
C: Directory .
C: /home/kingdon/zwork/cvsroot
C: co
S: Clear-sticky moddir/
S: /home/kingdon/zwork/cvsroot/1dir/
. . .
S: ok
```

In this example the response shown is `Clear-sticky`, but it could be another response instead. Note that it returns two pathnames. The first one, '`moddir/`', indicates the working directory to check out into. The second one, ending in '`1dir/`', indicates the directory to pass back to the server in a subsequent `Directory` request. For example, a subsequent `update` request might look like:

```
C: Directory moddir
C: /home/kingdon/zwork/cvsroot/1dir
```

```
. . .
C: update
```

For a given *local-directory*, the repository will be the same for each of the responses, so one can use the repository from whichever response is most convenient. Typically a client will store the repository along with the sources for each *local-directory*, use that same setting whenever operating on that *local-directory*, and not update the setting as long as the *local-directory* exists.

A client is free to rename a *local-directory* at any time (for example, in response to an explicit user request). While it is true that the server supplies a *local-directory* to the client, as noted above, this is only the default place to put the directory. Of course, the various `Directory` requests for a single command (for example, `update` or `ci` request) should name a particular directory with the same *local-directory*.

Each `Directory` request specifies a brand-new *local-directory* and *repository*; that is, *local-directory* and *repository* are never relative to paths specified in any previous `Directory` request.

Here's a more complex example, in which we request an update of a working directory which has been checked out from multiple places in the repository.

```
C: Argument dir1
C: Directory dir1
C: /home/foo/repos/mod1
. . .
C: Argument dir2
C: Directory dir2
C: /home/foo/repos/mod2
. . .
C: Argument dir3
C: Directory dir3/subdir3
C: /home/foo/repos/mod3
. . .
C: update
```

While directories `dir1` and `dir2` will be handled in similar fashion to the other examples given above, `dir3` is slightly different from the server's standpoint. Notice that module `mod3` is actually checked out into `dir3/subdir3`, meaning that directory `dir3` is either empty or does not contain data checked out from this repository.

The above example will work correctly in CVS 1.10.1 and later. The server will descend the tree starting from all directories mentioned in `Argument` requests and update those directories specifically mentioned in `Directory` requests.

Previous versions of CVS (1.10 and earlier) do not behave the same way. While the descent of the tree begins at all directories mentioned in `Argument` requests, descent into subdirectories only occurs if a directory has been mentioned in a `Directory` request. Therefore, the above example would succeed in updating `dir1` and `dir2`, but would skip `dir3` because that directory was not specifically mentioned in a `Directory` request. A functional version of the above that would run on a 1.10 or earlier server is as follows:

```
C: Argument dir1
C: Directory dir1
C: /home/foo/repos/mod1
. . .
C: Argument dir2
C: Directory dir2
C: /home/foo/repos/mod2
. . .
C: Argument dir3
C: Directory dir3
C: /home/foo/repos/.
. . .
C: Directory dir3/subdir3
C: /home/foo/repos/mod3
. . .
C: update
```

Note the extra `Directory dir3` request. It might be better to use `Emptydir` as the repository for the `dir3` directory, but the above will certainly work.

One more peculiarity of the 1.10 and earlier protocol is the ordering of `Directory` arguments. In order for a subdirectory to be registered correctly for descent by the recursion processor, its parent must be sent first. For example, the following would not work to update `dir3/subdir3`:

```
. . .
C: Argument dir3
C: Directory dir3/subdir3
C: /home/foo/repos/mod3
. . .
C: Directory dir3
C: /home/foo/repos/.
. . .
C: update
```

The implementation of the server in 1.10 and earlier writes the administration files for a given directory at the time of the `Directory` request. It also tries to register the directory with its parent to mark it for recursion. In the above example, at the time `dir3/subdir3` is created, the physical directory for `dir3` will be created on disk, but the administration files will not have been created. Therefore, when the server tries to register `dir3/subdir3` for recursion, the operation will silently fail because the administration files do not yet exist for `dir3`.

`Max-dotdot` *level* `\n`

Response expected: no. Tell the server that *level* levels of directories above the directory which `Directory` requests are relative to will be needed. For example, if the client is planning to use a `Directory` request for '`../../foo`', it must send a `Max-dotdot` request with a *level* of at least 2. `Max-dotdot` must be sent before the first `Directory` request.

`Static-directory \n`

Response expected: no. Tell the server that the directory most recently specified with `Directory` should not have additional files checked out unless explicitly requested. The client sends this if the `Entries.Static` flag is set, which is controlled by the `Set-static-directory` and `Clear-static-directory` responses.

`Sticky` *tagspec* `\n`

Response expected: no. Tell the server that the directory most recently specified with `Directory` has a sticky tag or date *tagspec*. The first character of *tagspec* is '`T`' for a tag, '`D`' for a date, or some other character supplied by a Set-sticky response from a previous request to the server. The remainder of *tagspec* contains the actual tag or date, again as supplied by Set-sticky.

The server should remember `Static-directory` and `Sticky` requests for a particular directory; the client need not resend them each time it sends a `Directory` request for a given directory. However, the server is not obliged to remember them beyond the context of a single command.

`Checkin-prog` *program* `\n`

Response expected: no. Tell the server that the directory most recently specified with `Directory` has a checkin program *program*. Such a program would have been previously set with the `Set-checkin-prog` response.

`Update-prog` *program* `\n`

Response expected: no. Tell the server that the directory most recently specified with `Directory` has an update program *program*. Such a program would have been previously set with the `Set-update-prog` response.

`Entry` *entry-line* `\n`

Response expected: no. Tell the server what version of a file is on the local machine. The name in *entry-line* is a name relative to the directory most recently specified with `Directory`. If the user is operating on only some files in a directory, `Entry` requests for only those files need be included. If an `Entry` request is sent without `Modified`, `Is-modified`, or `Unchanged`, it means the file is lost (does not exist in the working directory). If both `Entry` and one of `Modified`, `Is-modified`, or `Unchanged` are sent for the same file, `Entry` must be sent first. For a given file, one can send `Modified`, `Is-modified`, or `Unchanged`, but not more than one of these three.

`Kopt` *option* `\n`

This indicates to the server which keyword expansion options to use for the file specified by the next `Modified` or `Is-modified` request (for example '`-kb`' for a binary file). This is similar to `Entry`, but is used for a file for which there is no entries line. Typically this will be a file being added via an `add` or `import` request. The client may not send both `Kopt` and `Entry` for the same file.

`Checkin-time` *time* `\n`

For the file specified by the next `Modified` request, use *time* as the time of the checkin. The *time* is in the format specified by RFC822 as modified by RFC1123. The client may specify any timezone it chooses; servers will want to convert that to their own timezone as appropriate. An example of this format is:

```
26 May 1997 13:01:40 -0400
```

There is no requirement that the client and server clocks be synchronized. The client just sends its recommendation for a timestamp (based on file timestamps or whatever), and the server should just believe it (this means that the time might be in the future, for example).

Note that this is not a general-purpose way to tell the server about the timestamp of a file; that would be a separate request (if there are servers which can maintain timestamp and time of checkin separately).

This request should affect the `import` request, and may optionally affect the `ci` request or other relevant requests if any.

`Modified` *filename* `\n`

Response expected: no. Additional data: mode, \n, file transmission. Send the server a copy of one locally modified file. *filename* is a file within the most recent directory sent with `Directory`; it must not contain '`/`'. If the user is operating on only some files in a directory, only those files need to be included. This can also be sent without `Entry`, if there is no entry for the file.

`Is-modified` *filename* `\n`

Response expected: no. Additional data: none. Like `Modified`, but used if the server only needs to know whether the file is modified, not the contents.

The commands which can take `Is-modified` instead of `Modified` with no known change in behavior are: `admin`, `diff` (if and only if two '`-r`' or '`-D`' options are specified), `watch-on`, `watch-off`, `watch-add`, `watch-remove`, `watchers`, `editors`, `log`, and `annotate`.

For the `status` command, one can send `Is-modified` but if the client is using imperfect mechanisms such as timestamps to determine whether to consider a file modified, then the behavior will be different. That is, if one sends `Modified`, then the server will actually compare the contents of the file sent and the one it derives from to determine whether the file is genuinely modified. But if one sends `Is-modified`, then the server takes the client's word for it. A similar situation exists for `tag`, if the '`-c`' option is specified.

Commands for which `Modified` is necessary are `co`, `ci`, `update`, and `import`.

Commands which do not need to inform the server about a working directory, and thus should not be sending either `Modified` or `Is-modified`: `rdiff`, `rtag`, `history`, `init`, and `release`.

Commands for which further investigation is warranted are: `remove`, `add`, and `export`. Pending such investigation, the more conservative course of action is to stick to `Modified`.

`Unchanged` *filename* `\n`

Response expected: no. Tell the server that *filename* has not been modified in the checked out directory. The *filename* is a file within the most recent directory sent with `Directory`; it must not contain '`/`'.

`UseUnchanged \n`

Response expected: no. To specify the version of the protocol described in this document, servers must support this request (although it need not do anything) and clients must issue it. The `Root` request need not have been previously sent.

`Notify` *filename* `\n`

Response expected: no. Tell the server that an `edit` or `unedit` command has taken place. The server needs to send a `Notified` response, but such response is deferred until the next time that the server is sending responses. The *filename* is a file within the most recent directory sent with `Directory`; it must not contain '`/`'. Additional data:

notification-type `\t` *time* `\t` *clienthost* `\t`
working-dir `\t` *watches* `\n`

where *notification-type* is '`E`' for edit, '`U`' for unedit, undefined behavior if '`C`', and all other letters should be silently ignored for future expansion. *time* is the time at which the edit or unedit took place, in a user-readable format of the client's choice (the server should treat the time as an opaque string rather than interpreting it). *clienthost* is the name of the host on which the edit or unedit took place, and *working-dir* is the pathname of the working directory where the edit or unedit took place. *watches* are the temporary watches, zero or more of the following characters in the following order: '`E`' for edit, '`U`' for unedit, '`C`' for commit, and all other letters should be silently ignored for future expansion. If *notification-type* is '`E`' the temporary watches are set; if it is '`U`' they are cleared. If *watches* is followed by `\t` then the `\t` and the rest of the line should be ignored, for future expansion.

The *time*, *clienthost*, and *working-dir* fields may not contain the characters '`+`', '`,`', '`>`', '`;`', or '`=`'.

Note that a client may be capable of performing an `edit` or `unedit` operation without connecting to the server at that time, and instead connecting to the server when it is convenient (for example, when a laptop is on the net again) to send the `Notify` requests. Even if a client is capable of deferring notifications, it should attempt to send them immediately (one can send `Notify` requests together with a `noop` request, for example), unless perhaps if it can know that a connection would be impossible.

`Questionable` *filename* `\n`

Response expected: no. Additional data: no. Tell the server to check whether *filename* should be ignored, and if not, next time the server sends responses, send (in a `M` response) '`?`' followed by the directory and filename. *filename* must not contain '`/`'; it needs to be a file in the directory named by the most recent `Directory` request.

`Case \n`

Response expected: no. Tell the server that filenames should be matched in a case-insensitive fashion. Note that this is not the primary mechanism for achieving case-insensitivity; for the most part the client keeps track of the case which the server wants to use and takes care to always use that case regardless of what the user specifies. For example the filenames given in `Entry` and `Modified` requests for the same file must match in case regardless of whether the `Case` request is sent. The latter mechanism is more general (it could also be used for 8.3 filenames, VMS filenames with more than one '`.`', and any other situation in which there is a predictable mapping between filenames in the working directory and filenames in the protocol), but there are some situations it cannot handle (ignore patterns, or situations where the user specifies a filename and the client does not know about that file).

`Argument` *text* `\n`
: Response expected: no. Save argument for use in a subsequent command. Arguments accumulate until an argument-using command is given, at which point they are forgotten.

`Argumentx` *text* `\n`
: Response expected: no. Append `\n` followed by text to the current argument being saved.

`Global_option` *option* `\n`
: Response expected: no. Transmit one of the global options '`-q`', '`-Q`', '`-l`', '`-t`', '`-r`', or '`-n`'. *option* must be one of those strings, no variations (such as combining of options) are allowed. For graceful handling of `valid-requests`, it is probably better to make new global options separate requests, rather than trying to add them to this request. The `Root` request need not have been previously sent.

`Gzip-stream` *level* `\n`
: Response expected: no. Use zlib (RFC 1950/1951) compression to compress all further communication between the client and the server. After this request is sent, all further communication must be compressed. All further data received from the server will also be compressed. The *level* argument suggests to the server the level of compression that it should apply; it should be an integer between 1 and 9, inclusive, where a higher number indicates more compression.

`Kerberos-encrypt \n`
: Response expected: no. Use Kerberos encryption to encrypt all further communication between the client and the server. This will only work if the connection was made over Kerberos in the first place. If both the `Gzip-stream` and the `Kerberos-encrypt` requests are used, the `Kerberos-encrypt` request should be used first. This will make the client and server encrypt the compressed data, as opposed to compressing the encrypted data. Encrypted data is generally incompressible.

  Note that this request does not fully prevent an attacker from hijacking the connection, in the sense that it does not prevent hijacking the connection between the initial authentication and the `Kerberos-encrypt` request.

`Gssapi-encrypt \n`
: Response expected: no. Use GSSAPI encryption to encrypt all further communication between the client and the server. This will only work if the connection was made over GSSAPI in the first place. See `Kerberos-encrypt`, above, for the relation between `Gssapi-encrypt` and `Gzip-stream`.

Note that this request does not fully prevent an attacker from hijacking the connection, in the sense that it does not prevent hijacking the connection between the initial authentication and the `Gssapi-encrypt` request.

`Gssapi-authenticate \n`

Response expected: no. Use GSSAPI authentication to authenticate all further communication between the client and the server. This will only work if the connection was made over GSSAPI in the first place. Encrypted data is automatically authenticated, so using both `Gssapi-authenticate` and `Gssapi-encrypt` has no effect beyond that of `Gssapi-encrypt`. Unlike encrypted data, it is reasonable to compress authenticated data.

Note that this request does not fully prevent an attacker from hijacking the connection, in the sense that it does not prevent hijacking the connection between the initial authentication and the `Gssapi-authenticate` request.

`Set` *variable*=*value* `\n`

Response expected: no. Set a user variable *variable* to *value*. The `Root` request need not have been previously sent.

`expand-modules \n`

Response expected: yes. Expand the modules which are specified in the arguments. Returns the data in `Module-expansion` responses. Note that the server can assume that this is checkout or export, not rtag or rdiff; the latter do not access the working directory and thus have no need to expand modules on the client side.

Expand may not be the best word for what this request does. It does not necessarily tell you all the files contained in a module, for example. Basically it is a way of telling you which working directories the server needs to know about in order to handle a checkout of the specified modules.

For example, suppose that the server has a module defined by

```
aliasmodule -a 1dir
```

That is, one can check out `aliasmodule` and it will take `1dir` in the repository and check it out to `1dir` in the working directory. Now suppose the client already has this module checked out and is planning on using the `co` request to update it. Without using `expand-modules`, the client would have two bad choices: it could either send information about *all* working directories under the current directory, which could be unnecessarily slow, or it could be ignorant of the fact that `aliasmodule` stands for `1dir`, and neglect to send information for `1dir`, which would lead to incorrect operation.

With `expand-modules`, the client would first ask for the module to be expanded:

```
C: Root /home/kingdon/zwork/cvsroot
. . .
C: Argument aliasmodule
C: Directory .
C: /home/kingdon/zwork/cvsroot
C: expand-modules
S: Module-expansion 1dir
S: ok
```

and then it knows to check the '1dir' directory and send requests such as `Entry` and `Modified` for the files in that directory.

`ci \n`
`diff \n`
`tag \n`
`status \n`
`admin \n`
`history \n`
`watchers \n`
`editors \n`
`annotate \n`

Response expected: yes. Actually do a cvs command. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation. No provision is made for any input from the user. This means that `ci` must use a `-m` argument if it wants to specify a log message.

`log \n` Response expected: yes. Show information for past revisions. This uses any previous `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation. Also uses previous `Argument`'s of which the canonical forms are the following (CVS 1.10 and older clients sent what the user specified, but clients are encouraged to use the canonical forms and other forms are deprecated):

`-b, -h, -l, -N, -R, -t`
These options go by themselves, one option per `Argument` request.

`-d` *date1*`<`*date2*
Select revisions between *date1* and *date2*. Either date may be omitted in which case there is no date limit at that end of the range (clients may specify dates such as 1 Jan 1970 or 1 Jan 2038 for similar purposes but this is problematic as it makes assumptions about what dates the server supports). Dates are in RFC822/1123 format. The '`-d`' is one `Argument` request and the date range is a second one.

`-d` *date1*`<=`*date2*
Likewise but compare dates for equality.

`-d` *singledate*
Select the single, latest revision dated *singledate* or earlier.

To include several date ranges and/or singledates, repeat the '`-d`' option as many times as necessary.

`-r`*rev1*`:`*rev2*
`-r`*branch*
`-r`*branch*`.`
`-r` Specify revisions (note that *rev1* or *rev2* can be omitted, or can refer to branches). Send both the '`-r`' and the revision information

in a single `Argument` request. To include several revision selections, repeat the '`-r`' option.

`-s` *state*
`-w`
`-w`*login*
: Select on states or users. To include more than one state or user, repeat the option. Send the '`-s`' option as a separate argument from the state being selected. Send the '`-w`' option as part of the same argument as the user being selected.

`co \n`
: Response expected: yes. Get files from the repository. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. Arguments to this command are module names; the client cannot know what directories they correspond to except by (1) just sending the `co` request, and then seeing what directory names the server sends back in its responses, and (2) the `expand-modules` request.

`export \n`
: Response expected: yes. Get files from the repository. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. Arguments to this command are module names, as described for the `co` request. The intention behind this command is that a client can get sources from a server without storing CVS information about those sources. That is, a client probably should not count on being able to take the entries line returned in the `Created` response from an `export` request and send it in a future `Entry` request. Note that the entries line in the `Created` response must indicate whether the file is binary or text, so the client can create it correctly.

`rannotate \n`
`rdiff \n`
`rlog \n`
`rtag \n`
: Response expected: yes. Actually do a cvs command. This uses any previous `Argument` requests, if they have been sent. The client should not send `Directory`, `Entry`, or `Modified` requests for these commands; they are not used. Arguments to these commands are module names, as described for `co`.

`init` *root-name* `\n`
: Response expected: yes. If it doesn't already exist, create a CVS repository *root-name*. Note that *root-name* is a local directory and *not* a fully qualified `CVSROOT` variable. The `Root` request need not have been previously sent.

`update \n`
: Response expected: yes. Actually do a `cvs update` command. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation. The `-I` option is not used–files which the client can decide whether to ignore are not mentioned and the client sends the `Questionable` request for others.

`import \n`
: Response expected: yes. Actually do a `cvs import` command. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation - unlike most commands, the repository field of each `Directory`

request is ignored (it merely must point somewhere within the root). The files to be imported are sent in `Modified` requests (files which the client knows should be ignored are not sent; the server must still process the CVSROOT/cvsignore file unless -I ! is sent). A log message must have been specified with a `-m` argument.

`add \n` Response expected: yes. Add a file or directory. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation.

To add a directory, send the directory to be added using `Directory` and `Argument` requests. For example:

```
C: Root /u/cvsroot
. . .
C: Argument nsdir
C: Directory nsdir
C: /u/cvsroot/1dir/nsdir
C: Directory .
C: /u/cvsroot/1dir
C: add
S: M Directory /u/cvsroot/1dir/nsdir added to the repository
S: ok
```

You will notice that the server does not signal to the client in any particular way that the directory has been successfully added. The client is supposed to just assume that the directory has been added and update its records accordingly. Note also that adding a directory is immediate; it does not wait until a `ci` request as files do.

To add a file, send the file to be added using a `Modified` request. For example:

```
C: Argument nfile
C: Directory .
C: /u/cvsroot/1dir
C: Modified nfile
C: u=rw,g=r,o=r
C: 6
C: hello
C: add
S: E cvs server: scheduling file ‘nfile’ for addition
S: Mode u=rw,g=r,o=r
S: Checked-in ./
S: /u/cvsroot/1dir/nfile
S: /nfile/0///
S: E cvs server: use 'cvs commit' to add this file permanently
S: ok
```

Note that the file has not been added to the repository; the only effect of a successful `add` request, for a file, is to supply the client with a new entries line containing ‘`0`’ to indicate an added file. In fact, the client probably could perform this operation without contacting the server, although using `add` does cause the server to perform a few more checks.

The client sends a subsequent `ci` to actually add the file to the repository.

Another quirk of the `add` request is that with CVS 1.9 and older, a pathname specified in an `Argument` request cannot contain '`/`'. There is no good reason for this restriction, and in fact more recent CVS servers don't have it. But the way to interoperate with the older servers is to ensure that all `Directory` requests for `add` (except those used to add directories, as described above), use '`.`' for *local-directory*. Specifying another string for *local-directory* may not get an error, but it will get you strange `Checked-in` responses from the buggy servers.

`remove \n` Response expected: yes. Remove a file. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation.

Note that this request does not actually do anything to the repository; the only effect of a successful `remove` request is to supply the client with a new entries line containing '`-`' to indicate a removed file. In fact, the client probably could perform this operation without contacting the server, although using `remove` may cause the server to perform a few more checks.

The client sends a subsequent `ci` request to actually record the removal in the repository.

`watch-on \n`
`watch-off \n`
`watch-add \n`
`watch-remove \n`

Response expected: yes. Actually do the `cvs watch on`, `cvs watch off`, `cvs watch add`, and `cvs watch remove` commands, respectively. This uses any previous `Argument`, `Directory`, `Entry`, or `Modified` requests, if they have been sent. The last `Directory` sent specifies the working directory at the time of the operation.

`release \n`

Response expected: yes. Note that a `cvs release` command has taken place and update the history file accordingly.

`noop \n` Response expected: yes. This request is a null command in the sense that it doesn't do anything, but merely (as with any other requests expecting a response) sends back any responses pertaining to pending errors, pending `Notified` responses, etc. The `Root` request need not have been previously sent.

`update-patches \n`

Response expected: yes. This request does not actually do anything. It is used as a signal that the server is able to generate patches when given an `update` request. The client must issue the `-u` argument to `update` in order to receive patches.

`gzip-file-contents` *level* `\n`

Response expected: no. Note that this request does not follow the response convention stated above. `Gzip-stream` is suggested instead of `gzip-file-`

`contents` as it gives better compression; the only reason to implement the latter is to provide compression with CVS 1.8 and earlier. The `gzip-file-contents` request asks the server to compress files it sends to the client using `gzip` (RFC1952/1951) compression, using the specified level of compression. If this request is not made, the server must not compress files.

This is only a hint to the server. It may still decide (for example, in the case of very small files, or files that already appear to be compressed) not to do the compression. Compression is indicated by a '`z`' preceding the file length.

Availability of this request in the server indicates to the client that it may compress files sent to the server, regardless of whether the client actually uses this request.

`wrapper-sendme-rcsOptions \n`

Response expected: yes. Request that the server transmit mappings from filenames to keyword expansion modes in `Wrapper-rcsOption` responses.

`version \n`

Response expected: yes. Request that the server transmit its version message. The `Root` request need not have been previously sent.

other-request text `\n`

Response expected: yes. Any unrecognized request expects a response, and does not contain any additional data. The response will normally be something like '`error unrecognized request`', but it could be a different error if a previous request which doesn't expect a response produced an error.

When the client is done, it drops the connection.

5.9 Introduction to Responses

After a command which expects a response, the server sends however many of the following responses are appropriate. The server should not send data at other times (the current implementation may violate this principle in a few minor places, where the server is printing an error message and exiting—this should be investigated further).

Any set of responses always ends with '`error`' or '`ok`'. This indicates that the response is over.

The responses `Checked-in`, `New-entry`, `Updated`, `Created`, `Update-existing`, `Merged`, and `Patched` are refered to as *file updating* responses, because they change the status of a file in the working directory in some way. The responses `Mode`, `Mod-time`, and `Checksum` are referred to as *file update modifying* responses because they modify the next file updating response. In no case shall a file update modifying response apply to a file updating response other than the next one. Nor can the same file update modifying response occur twice for a given file updating response (if servers diagnose this problem, it may aid in detecting the case where clients send an update modifying response without following it by a file updating response).

5.10 The "pathname" in responses

Many of the responses contain something called *pathname*. The name is somewhat misleading; it actually indicates a pair of pathnames. First, a local directory name relative to the directory in which the command was given (i.e. the last `Directory` before the command). Then a linefeed and a repository name. Then a slash and the filename (without a '`,v`' ending). For example, for a file '`i386.mh`' which is in the local directory '`gas.clean/config`' and for which the repository is '`/rel/cvsfiles/devo/gas/config`':

```
gas.clean/config/
/rel/cvsfiles/devo/gas/config/i386.mh
```

If the server wants to tell the client to create a directory, then it merely uses the directory in any response, as described above, and the client should create the directory if it does not exist. Note that this should only be done one directory at a time, in order to permit the client to correctly store the repository for each directory. Servers can use requests such as `Clear-sticky`, `Clear-static-directory`, or any other requests, to create directories.

Some server implementations may poorly distinguish between a directory which should not exist and a directory which contains no files; in order to refrain from creating empty directories a client should both send the '`-P`' option to `update` or `co`, and should also detect the case in which the server asks to create a directory but not any files within it (in that case the client should remove the directory or refrain from creating it in the first place). Note that servers could clean this up greatly by only telling the client to create directories if the directory in question should exist, but until servers do this, clients will need to offer the '`-P`' behavior described above.

5.11 Responses

Here are the responses:

`Valid-requests` *request-list* `\n`

Indicate what requests the server will accept. *request-list* is a space separated list of tokens. If the server supports sending patches, it will include '`update-patches`' in this list. The '`update-patches`' request does not actually do anything.

`Checked-in` *pathname* `\n`

Additional data: New Entries line, \n. This means a file *pathname* has been successfully operated on (checked in, added, etc.). name in the Entries line is the same as the last component of *pathname*.

`New-entry` *pathname* `\n`

Additional data: New Entries line, \n. Like `Checked-in`, but the file is not up to date.

`Updated` *pathname* `\n`

Additional data: New Entries line, \n, mode, \n, file transmission. A new copy of the file is enclosed. This is used for a new revision of an existing file, or for a new file, or for any other case in which the local (client-side) copy of the file needs to be updated, and after being updated it will be up to date. If any directory in pathname does not exist, create it. This response is not used if `Created` and `Update-existing` are supported.

`Created` *pathname* `\n`
: This is just like `Updated` and takes the same additional data, but is used only if no `Entry`, `Modified`, or `Unchanged` request has been sent for the file in question. The distinction between `Created` and `Update-existing` is so that the client can give an error message in several cases: (1) there is a file in the working directory, but not one for which `Entry`, `Modified`, or `Unchanged` was sent (for example, a file which was ignored, or a file for which `Questionable` was sent), (2) there is a file in the working directory whose name differs from the one mentioned in `Created` in ways that the client is unable to use to distinguish files. For example, the client is case-insensitive and the names differ only in case.

`Update-existing` *pathname* `\n`
: This is just like `Updated` and takes the same additional data, but is used only if a `Entry`, `Modified`, or `Unchanged` request has been sent for the file in question.

  This response, or `Merged`, indicates that the server has determined that it is OK to overwrite the previous contents of the file specified by *pathname*. Provided that the client has correctly sent `Modified` or `Is-modified` requests for a modified file, and the file was not modified while CVS was running, the server can ensure that a user's modifications are not lost.

`Merged` *pathname* `\n`
: This is just like `Updated` and takes the same additional data, with the one difference that after the new copy of the file is enclosed, it will still not be up to date. Used for the results of a merge, with or without conflicts.

  It is useful to preserve an copy of what the file looked like before the merge. This is basically handled by the server; before sending `Merged` it will send a `Copy-file` response. For example, if the file is '`aa`' and it derives from revision 1.3, the `Copy-file` response will tell the client to copy '`aa`' to '`.#aa.1.3`'. It is up to the client to decide how long to keep this file around; traditionally clients have left it around forever, thus letting the user clean it up as desired. But another answer, such as until the next commit, might be preferable.

`Rcs-diff` *pathname* `\n`
: This is just like `Updated` and takes the same additional data, with the one difference that instead of sending a new copy of the file, the server sends an RCS change text. This change text is produced by '`diff -n`' (the GNU diff '`-a`' option may also be used). The client must apply this change text to the existing file. This will only be used when the client has an exact copy of an earlier revision of a file. This response is only used if the `update` command is given the '`-u`' argument.

`Patched` *pathname* `\n`
: This is just like `Rcs-diff` and takes the same additional data, except that it sends a standard patch rather than an RCS change text. The patch is produced by '`diff -c`' for CVS 1.6 and later (see POSIX.2 for a description of this format),

or '`diff -u`' for previous versions of CVS; clients are encouraged to accept either format. Like `Rcs-diff`, this response is only used if the `update` command is given the '`-u`' argument.

The `Patched` response is deprecated in favor of the `Rcs-diff` response. However, older clients (CVS 1.9 and earlier) only support `Patched`.

`Mode` *mode* `\n`
: This *mode* applies to the next file mentioned in `Checked-in`. `Mode` is a file update modifying response as described in Section 5.9 [Response intro], page 22.

`Mod-time` *time* `\n`
: Set the modification time of the next file sent to *time*. `Mod-time` is a file update modifying response as described in Section 5.9 [Response intro], page 22. The *time* is in the format specified by RFC822 as modified by RFC1123. The server may specify any timezone it chooses; clients will want to convert that to their own timezone as appropriate. An example of this format is:

```
26 May 1997 13:01:40 -0400
```

There is no requirement that the client and server clocks be synchronized. The server just sends its recommendation for a timestamp (based on its own clock, presumably), and the client should just believe it (this means that the time might be in the future, for example).

If the server does not send `Mod-time` for a given file, the client should pick a modification time in the usual way (usually, just let the operating system set the modification time to the time that the CVS command is running).

`Checksum` *checksum*`\n`
: The *checksum* applies to the next file sent (that is, `Checksum` is a file update modifying response as described in Section 5.9 [Response intro], page 22). In the case of `Patched`, the checksum applies to the file after being patched, not to the patch itself. The client should compute the checksum itself, after receiving the file or patch, and signal an error if the checksums do not match. The checksum is the 128 bit MD5 checksum represented as 32 hex digits (MD5 is described in RFC1321). This response is optional, and is only used if the client supports it (as judged by the `Valid-responses` request).

`Copy-file` *pathname* `\n`
: Additional data: *newname* \n. Copy file *pathname* to *newname* in the same directory where it already is. This does not affect `CVS/Entries`.

This can optionally be implemented as a rename instead of a copy. The only use for it which currently has been identified is prior to a `Merged` response as described under `Merged`. Clients can probably assume that is how it is being used, if they want to worry about things like how long to keep the *newname* file around.

`Removed` *pathname* `\n`
: The file has been removed from the repository (this is the case where cvs prints '`file foobar.c is no longer pertinent`').

`Remove-entry` *pathname* `\n`
: The file needs its entry removed from `CVS/Entries`, but the file itself is already gone (this happens in response to a `ci` request which involves committing the removal of a file).

`Set-static-directory` *pathname* `\n`
: This instructs the client to set the `Entries.Static` flag, which it should then send back to the server in a `Static-directory` request whenever the directory is operated on. *pathname* ends in a slash; its purpose is to specify a directory, not a file within a directory.

`Clear-static-directory` *pathname* `\n`
: Like `Set-static-directory`, but clear, not set, the flag.

`Set-sticky` *pathname* `\n`
: Additional data: *tagspec* `\n`. Tell the client to set a sticky tag or date, which should be supplied with the `Sticky` request for future operations. *pathname* ends in a slash; its purpose is to specify a directory, not a file within a directory. The client should store *tagspec* and pass it back to the server as-is, to allow for future expansion. The first character of *tagspec* is '`T`' for a tag, '`D`' for a date, or something else for future expansion. The remainder of *tagspec* contains the actual tag or date.

`Clear-sticky` *pathname* `\n`
: Clear any sticky tag or date set by `Set-sticky`.

`Template` *pathname* `\n`
: Additional data: file transmission (note: compressed file transmissions are not supported). *pathname* ends in a slash; its purpose is to specify a directory, not a file within a directory. Tell the client to store the file transmission as the template log message, and then use that template in the future when prompting the user for a log message.

`Set-checkin-prog` *dir* `\n`
: Additional data: *prog* `\n`. Tell the client to set a checkin program, which should be supplied with the `Checkin-prog` request for future operations.

`Set-update-prog` *dir* `\n`
: Additional data: *prog* `\n`. Tell the client to set an update program, which should be supplied with the `Update-prog` request for future operations.

`Notified` *pathname* `\n`
: Indicate to the client that the notification for *pathname* has been done. There should be one such response for every `Notify` request; if there are several `Notify` requests for a single file, the requests should be processed in order; the first `Notified` response pertains to the first `Notify` request, etc.

`Module-expansion` *pathname* `\n`
: Return a file or directory which is included in a particular module. *pathname* is relative to cvsroot, unlike most pathnames in responses. *pathname* should be used to look and see whether some or all of the module exists on the client side; it is not necessarily suitable for passing as an argument to a `co` request

(for example, if the modules file contains the '-d' option, it will be the directory specified with '-d', not the name of the module).

`Wrapper-rcsOption` *pattern* `-k '`*option*`' \n`
: Transmit to the client a filename pattern which implies a certain keyword expansion mode. The *pattern* is a wildcard pattern (for example, '`*.exe`'. The *option* is '`b`' for binary, and so on. Note that although the syntax happens to resemble the syntax in certain CVS configuration files, it is more constrained; there must be exactly one space between *pattern* and '`-k`' and exactly one space between '`-k`' and '`'`', and no string is permitted in place of '`-k`' (extensions should be done with new responses, not by extending this one, for graceful handling of `Valid-responses`).

`M` *text* `\n`
: A one-line message for the user. Note that the format of *text* is not designed for machine parsing. Although sometimes scripts and clients will have little choice, the exact text which is output is subject to vary at the discretion of the server and the example output given in this document is just that, example output. Servers are encouraged to use the '`MT`' response, and future versions of this document will hopefully standardize more of the '`MT`' tags; see Section 5.12 [Text tags], page 29.

`Mbinary \n`
: Additional data: file transmission (note: compressed file transmissions are not supported). This is like '`M`', except the contents of the file transmission are binary and should be copied to standard output without translation to local text file conventions. To transmit a text file to standard output, servers should use a series of '`M`' requests.

`E` *text* `\n`
: Same as `M` but send to stderr not stdout.

`F \n`
: Flush stderr. That is, make it possible for the user to see what has been written to stderr (it is up to the implementation to decide exactly how far it should go to ensure this).

`MT` *tagname data* `\n`
: This response provides for tagged text. It is similar to SGML/HTML/XML in that the data is structured and a naive application can also make some sense of it without understanding the structure. The syntax is not SGML-like, however, in order to fit into the CVS protocol better and (more importantly) to make it easier to parse, especially in a language like perl or awk.

  The *tagname* can have several forms. If it starts with '`a`' to '`z`' or '`A`' to '`Z`', then it represents tagged text. If the implementation recognizes *tagname*, then it may interpret *data* in some particular fashion. If the implementation does not recognize *tagname*, then it should simply treat *data* as text to be sent to the user (similar to an '`M`' response). There are two tags which are general purpose. The '`text`' tag is similar to an unrecognized tag in that it provides text which will ordinarily be sent to the user. The '`newline`' tag is used without *data* and indicates that a newline will ordinarily be sent to the user (there is no provision for embedding newlines in the *data* of other tagged text responses).

If *tagname* starts with '`+`' it indicates a start tag and if it starts with '`-`' it indicates an end tag. The remainder of *tagname* should be the same for matching start and end tags, and tags should be nested (for example one could have tags in the following order `+bold +italic text -italic -bold` but not `+bold +italic text -bold -italic`). A particular start and end tag may be documented to constrain the tagged text responses which are valid between them.

Note that if *data* is present there will always be exactly one space between *tagname* and *data*; if there is more than one space, then the spaces beyond the first are part of *data*.

Here is an example of some tagged text responses. Note that there is a trailing space after '`Checking in`' and '`initial revision:`' and there are two trailing spaces after '`<--`'. Such trailing spaces are, of course, part of *data*.

```
MT +checking-in
MT text Checking in 
MT fname gz.tst
MT text ;
MT newline
MT rcsfile /home/kingdon/zwork/cvsroot/foo/gz.tst,v
MT text   <--  
MT fname gz.tst
MT newline
MT text initial revision: 
MT init-rev 1.1
MT newline
MT text done
MT newline
MT -checking-in
```

If the client does not support the '`MT`' response, the same responses might be sent as:

```
M Checking in gz.tst;
M /home/kingdon/zwork/cvsroot/foo/gz.tst,v  <--  gz.tst
M initial revision: 1.1
M done
```

For a list of specific tags, see Section 5.12 [Text tags], page 29.

`error` *errno-code* '` `' *text* `\n`
: The command completed with an error. *errno-code* is a symbolic error code (e.g. `ENOENT`); if the server doesn't support this feature, or if it's not appropriate for this particular message, it just omits the errno-code (in that case there are two spaces after '`error`'). Text is an error message such as that provided by strerror(), or any other message the server wants to use. The *text* is like the `M` response, in the sense that it is not particularly intended to be machine-parsed; servers may wish to print an error message with `MT` responses, and then issue a `error` response without *text* (although it should be noted that `MT` currently has no way of flagging the output as intended for standard error, the way that the `E` response does).

`ok \n`
: The command completed successfully.

5.12 Tags for the MT tagged text response

The `MT` response, as described in Section 5.11 [Responses], page 23, offers a way for the server to send tagged text to the client. This section describes specific tags. The intention is to update this section as servers add new tags.

In the following descriptions, `text` and `newline` tags are omitted. Such tags contain information which is intended for users (or to be discarded), and are subject to change at the whim of the server. To avoid being vulnerable to such whim, clients should look for the tags listed here, not `text`, `newline`, or other tags.

The following tag means to indicate to the user that a file has been updated. It is more or less redundant with the `Created` and `Update-existing` responses, but we don't try to specify here whether it occurs in exactly the same circumstances as `Created` and `Update-existing`. The *name* is the pathname of the file being updated relative to the directory in which the command is occurring (that is, the last `Directory` request which is sent before the command).

```
MT +updated
MT fname name
MT -updated
```

The `importmergecmd` tag is used when doing an import which has conflicts. The client can use it to report how to merge in the newly imported changes. The *count* is the number of conflicts. The newly imported changes can be merged by running the following command:

```
cvs checkout -j tag1 -j tag2 repository
```

```
MT +importmergecmd
MT conflicts count
MT mergetag1 tag1
MT mergetag2 tag2
MT repository repository
MT -importmergecmd
```

5.13 Example

Here is an example; lines are prefixed by '`C: `' to indicate the client sends them or '`S: `' to indicate the server sends them.

The client starts by connecting, sending the root, and completing the protocol negotiation. In actual practice the lists of valid responses and requests would be longer.

```
C: Root /u/cvsroot
C: Valid-responses ok error Checked-in M E
C: valid-requests
S: Valid-requests Root Directory Entry Modified Argument Argumentx ci co
S: ok
C: UseUnchanged
```

The client wants to check out the `supermunger` module into a fresh working directory. Therefore it first expands the `supermunger` module; this step would be omitted if the client was operating on a directory rather than a module.

```
C: Argument supermunger
C: Directory .
```

```
C: /u/cvsroot
C: expand-modules
```

The server replies that the `supermunger` module expands to the directory `supermunger` (the simplest case):

```
S: Module-expansion supermunger
S: ok
```

The client then proceeds to check out the directory. The fact that it sends only a single `Directory` request which specifies '`.`' for the working directory means that there is not already a `supermunger` directory on the client.

```
C: Argument -N
C: Argument supermunger
C: Directory .
C: /u/cvsroot
C: co
```

The server replies with the requested files. In this example, there is only one file, '`mungeall.c`'. The `Clear-sticky` and `Clear-static-directory` requests are sent by the current implementation but they have no effect because the default is for those settings to be clear when a directory is newly created.

```
S: Clear-sticky supermunger/
S: /u/cvsroot/supermunger/
S: Clear-static-directory supermunger/
S: /u/cvsroot/supermunger/
S: E cvs server: Updating supermunger
S: M U supermunger/mungeall.c
S: Created supermunger/
S: /u/cvsroot/supermunger/mungeall.c
S: /mungeall.c/1.1///
S: u=rw,g=r,o=r
S: 26
S: int mein () { abort (); }
S: ok
```

The current client implementation would break the connection here and make a new connection for the next command. However, the protocol allows it to keep the connection open and continue, which is what we show here.

After the user modifies the file and instructs the client to check it back in. The client sends arguments to specify the log message and file to check in:

```
C: Argument -m
C: Argument Well, you see, it took me hours and hours to find
C: Argumentx this typo and I searched and searched and eventually
C: Argumentx had to ask John for help.
C: Argument mungeall.c
```

It also sends information about the contents of the working directory, including the new contents of the modified file. Note that the user has changed into the '`supermunger`' directory before executing this command; the top level directory is a user-visible concept because the server should print filenames in `M` and `E` responses relative to that directory.

```
C: Directory .
C: /u/cvsroot/supermunger
C: Entry /mungeall.c/1.1///
C: Modified mungeall.c
C: u=rw,g=r,o=r
C: 26
C: int main () { abort (); }
```

And finally, the client issues the checkin command (which makes use of the data just sent):

```
C: ci
```

And the server tells the client that the checkin succeeded:

```
S: M Checking in mungeall.c;
S: E /u/cvsroot/supermunger/mungeall.c,v  <--  mungeall.c
S: E new revision: 1.2; previous revision: 1.1
S: E done
S: Mode u=rw,g=r,o=r
S: Checked-in ./
S: /u/cvsroot/supermunger/mungeall.c
S: /mungeall.c/1.2///
S: ok
```

5.14 Required versus optional parts of the protocol

The following are part of every known implementation of the CVS protocol (except obsolete, pre-1.5, versions of CVS) and it is considered reasonable behavior to completely fail to work if you are connected with an implementation which attempts to not support them. Requests: `Root`, `Valid-responses`, `valid-requests`, `Directory`, `Entry`, `Modified`, `Unchanged`, `Argument`, `Argumentx`, `ci`, `co`, `update`. Responses: `ok`, `error`, `Valid-requests`, `Checked-in`, `Updated`, `Merged`, `Removed`, `M`, `E`.

A server need not implement `Repository`, but in order to interoperate with CVS 1.5 through 1.9 it must claim to implement it (in `Valid-requests`). The client will not actually send the request.

5.15 Obsolete protocol elements

This section briefly describes protocol elements which are obsolete. There is no attempt to document them in full detail.

There was a `Repository` request which was like `Directory` except it only provided *repository*, and the local directory was assumed to be similarly named.

If the `UseUnchanged` request was not sent, there was a `Lost` request which was sent to indicate that a file did not exist in the working directory, and the meaning of sending `Entries` without `Lost` or `Modified` was different. All current clients (CVS 1.5 and later) will send `UseUnchanged` if it is supported.

6 Notes on the Protocol

A number of enhancements are possible. Also see the file TODO in the CVS source distribution, which has further ideas concerning various aspects of CVS, some of which impact the protocol. Similarly, the `http://www.cvshome.org` site, in particular the *Development* pages.

- The `Modified` request could be speeded up by sending diffs rather than entire files. The client would need some way to keep the version of the file which was originally checked out; probably requiring the use of "cvs edit" in this case is the most sensible course (the "cvs edit" could be handled by a package like VC for emacs). This would also allow local operation of `cvs diff` without arguments.
- The fact that `pserver` requires an extra network turnaround in order to perform authentication would be nice to avoid. This relates to the issue of reporting errors; probably the clean solution is to defer the error until the client has issued a request which expects a response. To some extent this might relate to the next item (in terms of how easy it is to skip a whole bunch of requests until we get to one that expects a response). I know that the kerberos code doesn't wait in this fashion, but that probably can cause network deadlocks and perhaps future problems running over a transport which is more transaction oriented than TCP. On the other hand I'm not sure it is wise to make the client conduct a lengthy upload only to find there is an authentication failure.
- The protocol uses an extra network turnaround for protocol negotiation (`valid-requests`). It might be nice to avoid this by having the client be able to send requests and tell the server to ignore them if they are unrecognized (different requests could produce a fatal error if unrecognized). To do this there should be a standard syntax for requests. For example, perhaps all future requests should be a single line, with mechanisms analogous to `Argumentx`, or several requests working together, to provide greater amounts of information. Or there might be a standard mechanism for counted data (analogous to that used by `Modified`) or continuation lines (like a generalized `Argumentx`). It would be useful to compare what HTTP is planning in this area; last I looked they were contemplating something called Protocol Extension Protocol but I haven't looked at the relevant IETF documents in any detail. Obviously, we want something as simple as possible (but no simpler).
- The scrambling algorithm in the CVS client and server actually support more characters than those documented in Chapter 4 [Password scrambling], page 6. Someday we are going to either have to document them all (but this is not as easy as it may look, see below), or (gradually and with adequate process) phase out the support for other characters in the CVS implementation. This business of having the feature partly undocumented isn't a desirable state long-term.

  The problem with documenting other characters is that unless we know what character set is in use, there is no way to make a password portable from one system to another. For example, a with a circle on top might have different encodings in different character sets.

  It *almost* works to say that the client picks an arbitrary, unknown character set (indeed, having the CVS client know what character set the user has in mind is a hard problem otherwise), and scrambles according to a certain octet<->octet mapping. There are two problems with this. One is that the protocol has no way to transmit character 10 decimal (linefeed), and the current server and clients have no way to handle 0 decimal (NUL). This may cause problems with certain multibyte character sets, in which octets 10 and 0 will appear in the middle of other characters. The other problem, which is more minor and possibly not worth worrying about, is that someone can type a password on one system and then go to another system which uses a different encoding for the same characters, and have their password not work.

The restriction to the ISO646 invariant subset is the best approach for strings which are not particularly significant to users. Passwords are visible enough that this is somewhat doubtful as applied here. ISO646 does, however, have the virtue (!?) of offending everyone. It is easy to say "But the $ is right on people's keyboards! Surely we can't forbid that". From a human factors point of view, that makes quite a bit of sense. The contrary argument, of course, is that a with a circle on top, or some of the characters poorly handled by Unicode, are on *someone*'s keyboard.