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 1.8.1 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 5 [Connection and Authentication], page 6.

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 1.8.1 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 See Section 6.7 [Requirements], page 17. 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 6 [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. It is expected that it runs over TCP, UUCP, etc.
- Security and authentication are handled outside this protocol (but see below about '`cvs kserver`').
- This might be a first step towards adding transactions to CVS (i.e. a set of operations is either executed atomically or none of them is executed), improving the locking, or other features. The current server implementation is a long way from being able to do any of these things. The protocol, however, is not known to contain any defects which would preclude them.
- The server never has to have any CVS locks in place while it is waiting for communication with the client. This makes things robust in the face of flaky networks.
- Data is transferred in large chunks, which is necessary for good performance. In fact, currently the client uploads all the data (without waiting for server responses), and then waits for one server response (which consists of a massive download of all the data). There may be cases in which it is better to have a richer interraction, but the need for the server to release all locks whenever it waits for the client makes it complicated.

3 Notes on the Current Implementation

The client is built in to the normal `cvs` program, triggered by a `CVSROOT` variable containing a colon, for example `cygnus.com:/rel/cvsfiles`.

The client stores what is stored in checked-out directories (including '`CVS`'). The way these are stored is totally compatible with standard CVS. The server requires no storage other than the repository, which also is totally compatible with standard CVS.

The server is started by `cvs server`. There is no particularly compelling reason for this rather than making it a separate program which shares a lot of sources with cvs.

The server can also be started by `cvs kserver`, in which case it does an initial Kerberos authentication on stdin. If the authentication succeeds, it subsequently runs identically to `cvs server`.

The current server implementation can use up huge amounts of memory when transmitting a lot of data over a slow link (i.e. the network is slower than the server can generate the data). There is some experimental code (see `SERVER_FLOWCONTROL` in options.h) which should help significantly.

4 Notes on the Protocol

A number of enhancements are possible:

- 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, which would double client disk space requirements or require coordination with editors (e.g. maybe it could use emacs numbered backups). This would also allow local operation of `cvs diff` without arguments.
- Have the client keep a copy of some part of the repository. This allows all of `cvs diff` and large parts of `cvs update` and `cvs ci` to be local. The local copy could be made consistent with the master copy at night (but if the master copy has been updated since the latest nightly re-sync, then it would read what it needs to from the master).
- Provide encryption using kerberos.
- The current procedure for `cvs update` is highly sub-optimal if there are many modified files. One possible alternative would be to have the client send a first request without the contents of every modified file, then have the server tell it what files it needs. Note the server needs to do the what-needs-to-be-updated check twice (or more, if changes in the repository mean it has to ask the client for more files), because it can't keep locks open while waiting for the network. Perhaps this whole thing is irrelevant if client-side repositories are implemented, and the rcsmerge is done by the client.

5 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.

rsh
: 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 the kerberized client and server should be changed to use port 2401 (see below), and send a different string in place of '`BEGIN AUTH REQUEST`' to identify the authentication method in use. However, noone has yet gotten around to implementing this.

pserver
: The password authenticated 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, sends the string '`BEGIN AUTH REQUEST`', a linefeed, the cvs root, a linefeed, the username, a linefeed, the password trivially encoded (see scramble.c in the cvs sources), a linefeed, the string '`END AUTH REQUEST`', and a linefeed. The server responds with '`I LOVE YOU`' and a linefeed if the authentication is successful or '`I HATE YOU`' and a linefeed if the authentication fails. After receiving '`I LOVE YOU`', the client proceeds with the cvs protocol. 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.

6 The CVS client/server protocol

In the following, '`\n`' refers to a linefeed and '`\t`' refers to a horizontal tab.

6.1 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.

6.2 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?).

6.3 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).

6.4 Requests

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 newline, 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`'. 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 newline iff that file (or its compressed form) ends with a newline.

`Root` *pathname* `\n`

Response expected: no. Tell the server which `CVSROOT` to use. *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.

`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.

`valid-requests \n`
: Response expected: yes. Ask the server to send back a `Valid-requests` response.

`Repository` *repository* `\n`
: Response expected: no. Tell the server what repository to use. This 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 a `Repository` for each directory in which there will be an `Entry` or `Modified` , and then a final `Repository` for the original directory, then the command.

`Directory` *local-directory* `\n`
: Additional data: *repository* `\n`. This is like `Repository`, but the local name of the directory may differ from the repository name. If the client uses this request, it affects the way the server returns pathnames; see Section 6.5 [Responses], page 13. *local-directory* is relative to the top level at which the command is occurring (i.e. the last `Directory` or `Repository` which is sent before the command).

`Max-dotdot` *level* `\n`
: 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 `Repository` or `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 `Repository` has a sticky tag or date *tagspec*. The first character of *tagspec* is '`T`' for a tag, or '`D`' for a date. The remainder of *tagspec* contains the actual tag or date.

`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 `Repository`. 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`, `Unchanged`, or `Lost` for that file the meaning depends on whether `UseUnchanged` has been sent; if it has been it means the file is lost, if not it means the file is unchanged.

`Modified` *filename* `\n`
: Response expected: no. Additional data: mode, `\n`, file transmission. Send the server a copy of one locally modified file. *filename* is relative to the most recent repository sent with `Repository`. 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.

`Lost` *filename* `\n`
: Response expected: no. Tell the server that *filename* no longer exists. The name is relative to the most recent repository sent with `Repository`. This is used for any case in which `Entry` is being sent but the file no longer exists. If the client has issued the `UseUnchanged` request, then this request is not used.

`Unchanged` *filename* `\n`
: Response expected: no. Tell the server that *filename* has not been modified in the checked out directory. The name is relative to the most recent repository sent with `Repository`. This request can only be issued if `UseUnchanged` has been sent.

`UseUnchanged \n`
: Response expected: no. Tell the server that the client will be indicating unmodified files with `Unchanged`, and that files for which no information is sent are nonexistent on the client side, not unchanged. This is necessary for correct behavior since only the server knows what possible files may exist, and thus what files are nonexistent.

`Notify` *filename* `\n`
: Tell the server that a `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. Response expected: no. Additional data:

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

  where *notification-type* is '`E`' for edit or '`U`' for unedit, *time* is the time at which the edit or unedit took place, *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 to set; if it is followed by `\t` then the tab and the rest of the line are ignored.

`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.

`Case \n`
: Tell the server that filenames should be matched against ignore patterns in a case-insensitive fashion. Note that this does not apply to other comparisons—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.

`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`
: 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.

`Set` *variable*`=`*value* `\n`
: Set a user variable *variable* to *value*.

`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.

`co \n`
`ci \n`
`diff \n`
`tag \n`
`status \n`
`log \n`
`add \n`
`remove \n`

`rdiff \n`
`rtag \n`
`admin \n`
`export \n`
`history \n`
`watchers \n`
`editors \n`
`annotate \n`

: Response expected: yes. Actually do a cvs command. This uses any previous `Argument`, `Repository`, `Entry`, `Modified`, or `Lost` requests, if they have been sent. The last `Repository` 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.

`init` *root-name* `\n`

: Response expected: yes. If it doesn't already exist, create a CVS repository *root-name*. 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`, `Repository`, `Entry`, `Modified`, or `Lost` requests, if they have been sent. The last `Repository` 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`, `Repository`, `Entry`, `Modified`, or `Lost` requests, if they have been sent. The last `Repository` sent specifies the working directory at the time of the operation. 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.

`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`, `Repository`, `Entry`, `Modified`, or `Lost` requests, if they have been sent. The last `Repository` 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.

`update-patches \n`
: 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`
: This request asks the server to filter files it sends to the client through the '`gzip`' program, using the specified level of compression. If this request is not made, the server must not do any compression.

  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.

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 command which doesn't expect a response produced an error.

When the client is done, it drops the connection.

6.5 Responses

After a command which expects a response, the server sends however many of the following responses are appropriate. Pathnames are of the actual files operated on (i.e. they do not contain '`,v`' endings), and are suitable for use in a subsequent `Repository` request. However, if the client has used the `Directory` request, then it is instead a local directory name relative to the directory in which the command was given (i.e. the last `Directory` before the command). Then a newline and a repository name (the pathname which is sent if `Directory` is not used). Then the slash and the filename. 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
```

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

`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.

`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.

`Patched` *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 a patch produced by '`diff -u`'. This client must apply this patch, using the '`patch`' program, 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.

`Mode` *mode* `\n`
: This *mode* applies to the next file mentioned in `Checked-in`. It does not apply to any request which follows a `Checked-in`, `New-entry`, `Updated`, `Merged`, or `Patched` response.

`Checksum` *checksum*`\n`
: The *checksum* applies to the next file sent over via `Updated`, `Merged`, or `Patched`. 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. 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`.

`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 first character of *tagspec* is '`T`' for a tag, or '`D`' for a date. 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).

`M` *text* `\n`
: A one-line message for the user.

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

`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.

`ok \n`
: The command completed successfully.

6.6 Example

Lines beginning with '`c>`' are sent by the client; lines beginning with '`s>`' are sent by the server; lines beginning with '`#`' are not part of the actual exchange.

```
c> Root /rel/cvsfiles
# In actual practice the lists of valid responses and requests would
# be longer
c> Valid-responses Updated Checked-in M ok error
c> valid-requests
s> Valid-requests Root co Modified Entry Repository ci Argument Argumentx
s> ok
# cvs co devo/foo
c> Argument devo/foo
```

```
c> co
s> Updated /rel/cvsfiles/devo/foo/foo.c
s> /foo.c/1.4/Mon Apr 19 15:36:47 1993 Mon Apr 19 15:36:47 1993//
s> 26
s> int mein () { abort (); }
s> Updated /rel/cvsfiles/devo/foo/Makefile
s> /Makefile/1.2/Mon Apr 19 15:36:47 1993 Mon Apr 19 15:36:47 1993//
s> 28
s> foo: foo.c
s>         $(CC) -o foo $<
s> ok
# In actual practice the next part would be a separate connection.
# Here it is shown as part of the same one.
c> Repository /rel/cvsfiles/devo/foo
# foo.c relative to devo/foo just set as Repository.
c> Entry /foo.c/1.4/Mon Apr 19 15:36:47 1993 Mon Apr 19 15:36:47 1993//
c> Entry /Makefile/1.2/Mon Apr 19 15:36:47 1993 Mon Apr 19 15:36:47 1993//
c> Modified foo.c
c> 26
c> int main () { abort (); }
# cvs ci -m <log message> foo.c
c> Argument -m
c> Argument Well, you see, it took me hours and hours to find this typo and I
c> Argumentx searched and searched and eventually had to ask John for help.
c> Argument foo.c
c> ci
s> Checked-in /rel/cvsfiles/devo/foo/foo.c
s> /foo.c/1.5/ Mon Apr 19 15:54:22 CDT 1993//
s> M Checking in foo.c;
s> M /cygint/rel/cvsfiles/devo/foo/foo.c,v  <--  foo.c
s> M new revision: 1.5; previous revision: 1.4
s> M done
s> ok
```

6.7 Required versus optional parts of the protocol

The following are part of every known implementation of the CVS protocol 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, Repository, Entry, Modified, Argument, Argumentx, ci, co, update. Responses: ok, error, Valid-requests, Checked-in, Updated, Merged, Removed, M, E.

Failure to support the Directory, UseUnchanged, and Unchanged requests is deprecated. CVS 1.5 and later have supported these requests and in the future it will be considered reasonable behavior to completely fail to work with an implementation which attempts to not support them. Support for the Repository and Lost requests is deprecated; CVS clients 1.5 and later will not use them if communicating with a server which supports Directory and UseUnchanged.