Contents of /mc/vfs/README.fish

Revision **1.6** - (**show annotations**) (**download**)
Mon Jun 2 19:47:30 2003 UTC (18 years, 2 months ago) by *proskin*
Branch: **MAIN**
CVS Tags: **HEAD-merge-rillig, MC_4_6_2_pre1, MC_4_6_1_pre4, MC_4_6_1_pre1, MC_4_6_1_pre3, MC_4_6_1_pre2, rillig-merge-HEAD, MC_4_6_1_release, HEAD**
Branch point for: **MC_4_6_1, MC_4_6_1_PRE, rillig-experimental**
Changes since **1.5: +1 -1 lines**

```
Typos.
```

```

                FIles transferred over SHell protocol (V 0.0.2)
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

This protocol was designed for transferring files over a remote shell
connection (rsh and compatibles). It can be as well used for transfers over
rsh, and there may be other uses.

Client sends requests of following form:

#FISH_COMMAND
equivalent shell commands,
which may be multiline

Only fish commands are defined here, shell equivalents are for your
information only and will probably vary from implementation to
implementation. Fish commands always have priority: server is
expected to execute fish command if it understands it. If it does not,
however, it can try the luck and execute shell command.

Server's reply is multiline, but always ends with

### 000<optional text>

line. ### is prefix to mark this line, 000 is return code. Return
codes are superset to those used in ftp.

There are few new exit codes defined:

000 don't know; if there were no previous lines, this marks COMPLETE
success, if they were, it marks failure.

001 don't know; if there were no previous lines, this marks
PRELIMinary success, if they were, it marks failure

                                Connecting
                                ~~~~~~~~~~
Client uses "echo FISH:;/bin/sh" as command executed on remote
machine. This should make it possible for server to distinguish FISH
connections from normal rsh/ssh.

                                Commands
                                ~~~~~~~~
#FISH
echo; start_fish_server; echo '### 200'

This command is sent at the beginning. It marks that client wishes to
talk via FISH protocol. #VER command must follow. If server
understands FISH protocol, it has option to put FISH server somewhere
on system path and name it start_fish_server.

#VER 0.0.2 <feature1> <feature2> <...>
echo '### 000'

This command is the second one. It sends client version and extensions
to the server. Server should reply with protocol version to be used,
and list of extensions accepted.
```

```

VER 0.0.0 <feature2>
### 200

#PWD
pwd; echo '### 200'

Server should reply with current directory (in form /abc/def/ghi)
followed by line indicating success.

#LIST /directory
ls -lLa $1 | grep '^[^cbt]' | ( while read p x u g s m d y n; do echo "P$p $u.$g
S$s
d$m $d $y
:$n
"; done )
ls -lLa $1 | grep '^[cb]' | ( while read p x u g a i m d y n; do echo "P$p $u.$g
E$a$i
dD$m $d $y
:$n
"; done )
echo '### 200'

This allows client to list directory or get status information about
single file. Output is in following form (any line except :<filename>
may be omitted):

P<unix permissions> <owner>.<group>
S<size>
d<3-letters month name> <day> <year or HH:MM>
D<year> <month> <day> <hour> <minute> <second>[.1234]
E<major-of-device>,<minor>
:<filename>
L<filename symlink points to>
<blank line to separate items>

Unix permissions are of form X--------- where X is type of
file. Currently, '-' means regular file, 'd' means directory, 'c', 'b'
means character and block device, 'l' means symbolic link, 'p' means
FIFO and 's' means socket.

'd' has three fields: month (one of strings Jan Feb Mar Apr May Jun
Jul Aug Sep Oct Nov Dec), day of month, and third is either single
number indicating year, or HH:MM field (assume current year in such
case). As you've probably noticed, this is pretty broken; it is for
compatibility with ls listing.

#RETR /some/name
ls -l /some/name | ( read a b c d x e; echo $x ); echo '### 100'; cat /some/name; echo '### 200'

Server sends line with filesize on it, followed by line with ### 100
indicating partial success, then it sends binary data (exactly
filesize bytes) and follows them with (with no preceding newline) ###
200.

Note that there's no way to abort running RETR command - except
closing the connection.

#STOR <size> /file/name
> /file/name; echo '### 001'; ( dd bs=4096 count=<size/4096>; dd bs=<size%4096> count=1 ) 2>/dev/null | ( cat > %s; cat > /dev/null

This command is for storing /file/name, which is exactly size bytes
big. You probably think I went crazy. Well, I did not: that strange
cat > /dev/null has purpose to discard any extra data which was not
written to disk (due to for example out of space condition).

[Why? Imagine uploading file with "rm -rf /" line in it.]

#CWD /somewhere
cd /somewhere; echo '### 000'

It is specified here, but I'm not sure how wise idea is to use this
one: it breaks stateless-ness of the protocol.

```

```
Following commands should be rather self-explanatory:

#CHMOD 1234 file
chmod 1234 file; echo '### 000'

#DELE /some/path
rm -f /some/path; echo '### 000'

#MKD /some/path
mkdir /some/path; echo '### 000'

#RMD /some/path
rmdir /some/path; echo '### 000'

#RENAME /path/a /path/b
mv /path/a /path/b; echo '### 000'

#LINK /path/a /path/b
ln /path/a /path/b; echo '### 000'

#SYMLINK /path/a /path/b
ln -s /path/a /path/b; echo '### 000'

#CHOWN user /file/name
chown user /file/name; echo '### 000'

#CHGRP group /file/name
chgrp group /file/name; echo '### 000'

#READ <offset> <size> /path/and/filename
cat /path/and/filename | ( dd bs=4096 count=<offset/4096> > /dev/null;
dd bs=<offset%4096> count=1 > /dev/null;
dd bs=4096 count=<offset/4096>;
dd bs=<offset%4096> count=1; )

Returns ### 200 on successful exit, ### 291 on successful exit when
reading ended at eof, ### 292 on successfull exit when reading did not
end at eof.

#WRITE <offset> <size> /path/and/filename

Hmm, shall we define these ones if we know our client is not going to
use them?


That's all, folks!
                                                  pavel@ucw.cz
```