Ettcl 14

A Tcl interpreter reduced in size and enhanced in features,
the core for The EtLinux mini-distribution for embedded systems.
August 2004

by Alessandro Rubini and Davide Ciminaghi ({rubini,ciminaghi}@gnudd.com)

Ettcl-14

This file documents version 14 of the Ettcl version of the Tcl interpreter. Ettcl is based on Tcl-7.6.

1 Introduction

The Ettcl project was born in 1998 as the core of EtLinux, a small distribution that includes the Linux kernel and a minimal set of tools, meant to run on a 386 with 2MB of memory. Everything in EtLinux is written in Tcl, on order to save memory by aggressively exploiting memory sharing across processes.

The Tcl interpreter, therefore, has been extended with process-control primitives in order for *init* to be able to spawn processes and reap zombies. Moreover, the interpreter has been removed by trimming most of the features that were not going to be used, especially those that needed a non-negligible amount of memory. Moreover, floating point support has been completely removed, so we could run a system without either a hardware FPU or the software emulator. Trimming the FPU emulator from the kernel saved us 40kB of memory, a huge amount when you have 2MB to host the whole runtime.

1.1 Portability

While the EtLinux project was born on the *i386* platform and Ettcl needs a CPU with memory management, we have always been careful to keep cross-platform portability. Ettcl has been used on several platforms: Alpha, Sparc, Sparc64, PowerPc, MIPS, Arm. On the *i386* platform you can compile it to work with *libc5* or *libc6*, by selecting *CC* in your environment before compiling.

The interpreter is less than 250kB (525kB if statically linked with *libc*) when compiled against *libc6*, and less than 220kB (470kB if statically linked) when compiled against *libc5*.

1.2 Compilation

The package is compiled in the usual way:

```
./configure && make
```

The following option can be passed to *configure*. Only differences against the pristine Tcl-7.6 tree are documented; use "`./configure --help`" to get a complete list of options.

- –disable-shared

  Compilation of shared libraries is the default with Ettcl, but it can be disabled

- –enable-double

- –enable-math

  By default, all floating point has been removed, as well as all dependencies from *libm*. The interpreter can thus run in FPU-less computers (like the 80386) without the need for an FPU emulator in the kernel (but this requires changes to *libc* as well, see the *EtLinux* distribution about that.

- –enable-history

  History management in Tcl has been disabled to save space in the executable, it can be reenabled.

- –disable-strerror

  By default, Ettcl uses *strerror* and *strsignal* to map error and signal names to strings. With this directive you can restore the original Tcl behavior.

- –disable-prosaext

  This directive disables all Ettcl extensions (i.e., everything that is described in this manual).

- –disable-386

  By default, compilation is optimized for the 80386 CPU. This directive is valid only when compiling for the xi86 platforms, and disables 386-specific features. Those features are "-m386" in the compiler flags and not requiring alignment of functions and loops, in order to save space in the binary.

2   Features removed from Tcl-7.6

The following features have been removed from Tcl-7.6, or made optional and disabled by default:

- multi-platform support

  While Tcl-7.6 compiles on Unix, Windows and MacOS, we removed all non-Unix material to save space in the distribution.

- manual pages

  Since everyone has Tcl manual pages, we chose to remove the manual pages from this distribution. Please refer to the original Tcl-7.6 package for documentation about the language.

- floating point

  No floating point support is there by default.

- history

  No command history is there by default

- package support

  Package support has been removed to save space, as EtLinux does not use it.

3   Features added to Tcl-7.6

Development of *ettcl* added several commands to the interpreter. Some of them are system calls or other low-level utilities, some other are simple substitutes for system commands (like *ps* and *cat*). The former group is implemented in C, the latter group as procedures within *init.tcl*.

3.1 new commands

The following new commands have been added to the Tcl language. They are grouped by functionality rather than alphabetically: this sections describes basic multiprocess system calls, then other useful system calls, pseudo-tty management, file management, network management, and finally generic utilities.

`sys_fork`

Implements the *fork* system call. It returns 0 to the child, a positive PID to the parent and -1 in case of error.

`sys_wait`
`sys_wait <pid>`
`sys_wait nohang`

Implements the *wait* and *waitpid* system calls. Without arguments, the command waits until a child exits. With a numeric argument, it waits until that specific child exits. With a `nohang` argument it calls `waitpid(-1, &result, WNOHANG)`. The return value is a list of two elements, the return value of *wait* (i.e., the PID of the child), and the exit status of the child process. "`sys_wait nohang`" returns an error of "`No child process`" if the current process has no children, and an error of `EAGAIN` (i.e., "Try again" or "Resource temporarily unavailable") if no child is ready to be collected.

Examples:

```
% if ![sys_fork] {exit 34} else {puts [sys_wait]}
=> 25958 34
% sys_wait nohang
=> sys_wait: No child processes
```

`sys_exec <command> [<args> ...]`

Implements the *execvp* system call (i.e., it passes the *argv* vector unchanged and searches the system path). This usually makes only sense after *sys_fork*, in the child process.

Examples:

```
% if ![sys_fork] {sys_exec true} else {puts [sys_wait]}
=> 25978 0
% if ![sys_fork] {sys_exec false} else {puts [sys_wait]}
=>25979 1
%  if ![sys_fork] {sys_exec grep . /etc/issue} else \
        {puts [sys_wait]}
=>Debian GNU/\s 2.2 \n \l
=> 25980 0
% if ![sys_fork] {sys_exec grep . /dev/null} else \
        {puts [sys_wait]}
=> 25981 1
```

`sys_pipe [<varname> <varname>]`

Implements the *pipe* system call. The newly created file channels are either returned as a Tcl list or saved in the variables provided on the command line.

This example shows the minimal use of a pipe:

```
% sys_pipe a b
% if ![sys_fork] { puts $b pio; exit }
% close $b
% while {[gets $a s]>=0} { puts $s }
=> pio
% sys_wait
=> 27324 0
```

sys_dup <oldfile> <newfile>

Implements the *dup2* system call. The new file must be opened (so there is already the internal Tcl structure for it) and must be in the same mode as the original file (i.e., opened for reading, for writing, or both). This is used when creating pipes and attaching them to, for example, *stdin* and *stdout*.

Example:

```
sys_pipe in out
if ![sys_fork] {
    sys_dup $out stdout; close $in; close $out
    sys_exec cat /etc/issue
}
close $out
while {[gets $in string]>0} {
    puts "Read: $string"
}
close $in
sys_wait
```

sys_kill [-<signal>] <pid> [<pid> ...]

Implements the *kill* system call. If the first argument is prefixed with a dash, it is interpreted as a signal number or signal name (uppercase without leading SIG, e.g. -USR1); by default the signal being sent is SIGTERM. All remaining arguments are used as process identifiers or group identifiers (if negative). Note that currently it isn't possible to sent SIGTERM to a process group by only specifying the process group (as the negative argument will be parsed as a signal number.

sys_nice <value>

Implements the *nice* system call. The argument must be an integer number. Lower priority is specified by negative numbers ("dash-10" like it was an option). Higher priority is specified with a positive number (no dash), and is only available for the superuser (there are no checks in Ettcl, it's the *nice* system call that will complain).

```
scheduler
scheduler -priority [<prio>]
scheduler <type>
scheduler -priority <prio> <type>
scheduler -mlock <lock_status>
scheduler -priority <prio> <type> [-mlock <lock_status>] [<cmd>]
scheduler -yield
```
When invoked without parameters returns the current scheduling policy (RR or FIFO or OTHER).

-priority **<prio>** sets **<prio>** as the priority of the current process and returns the old priority. If **<prio>** is not given, then no priority change takes place and the old priority is returned.

<type> (RR, FIFO, OTHER) controls the scheduling policy with which **<cmd>** is executed. If no **<cmd>** is given, then the current process is affected.

-mlock **<lock_status>** controls the memory lock status during the execution of **<cmd>** or the memory lock status of the current process (if **<cmd>** is not given). If **<lock_status>** is non zero, then *mlockall()* is invoked before executing **<cmd>**. Otherwise *munlockall()* is invoked. After **<cmd>** has been executed, *!<status>* is set. Note that the original memory lock status is not restored.

-yield makes the current process voluntarily release the CPU.

Note that this option overrides the others.

If the scheduling policy is changed, the old policy is returned. If both policy and priority are changed, the old values of both are returned. If **<cmd>** is given, then its output is returned.

For example:

scheduler -priority 10 -mlock on "FIFO" { my_special_command par1 par2 }

sets the scheduling policy to FIFO and priority to 10, locks memory, executes *my_special_command* with parameters *par1* and *par2*, resets the original scheduling policy and priority and finally unlocks memory.

```
sys_sync
```
Implements the trivial *sync* system call.

```
sys_signal <signame> IGNORE
sys_signal <signame> DEFAULT
sys_signal <signame> <tcl-script>
```
The command is a frontend to *sigaction* or *signal*. A signal can be ignored or restored to default behavior, or handled by a Tcl script. The signal name is specified in uppercase and with no leading `SIG`, or as a decimal number. If a script is provided, it will be run at global scope as an asynchronous event. During execution of the script the variable `tcl_signal` is set to the name of the signal being handled (or the number if there is no mapping to the name). Ettcl currently can handle the following signals by name, other signals can still be specified by number: `HUP`, `INT`, `ILL`, `TERM`, `SEGV`, `USR1`, `USR2`, `CHLD`, `IO`, and `WINCH`.

The preferred wait to handle `SIGCHLD` loops on "`sys_wait nohang`" until the call fails, to account for the case of a lost signal – when multiple children die before the first is reaped. For example:

```
sys_signal CHLD {while {![catch {sys_wait nohang}]} {}}
```

sys_reboot reboot

sys_reboot halt

Reboots or halts the system, and is only available to the *init* process (i.e., if the PID of this process is not 1, the command is not executed). This is used within EtLinux.

sys_ttypair [<varname> <varname>]

Tries to open a master and a slave tty's. It is used in *telnetd* (see Section 4.5 [Telnetd and Telnetc], page 26) to open a controlling terminal for new inbound connections. The new Tcl channels are returned either as values of the two specified variables, or in a list of two elements as return value of the command. Note that you need superuser privileges to open a pair of terminals.

sys_opentty

Tries to elect the *stdin* channel for the process as the controlling tty for the process. It is used in *telnetd* (see Section 4.5 [Telnetd and Telnetc], page 26) to associate a controlling terminal to the child process.

sys_chmod <mode> <file> [<file> ...]

Implements the *chmod* system calls. The mode is accepted as a number (prefix with a 0 to get octal notation). For compatibility with earlier versions of *Ettcl*, it is possible to call "`sys_chmod file mode`".

sys_umask <value>

Implements the *umask* system call. Use a leading 0 to specify an octal number.

sys_setuid <uid>

Implements the *setuid* system call. **<uid>** must be an integer. Returns the old user id.

sys_seteuid <uid>

Implements the *seteuid* system call. **<uid>** must be an integer. Returns the old user id.

sys_setgid <uid>

Implements the *setgid* system call. **<gid>** must be an integer. Returns the old user id.

sys_setegid <uid>

Implements the *setegid* system call. **<gid>** must be an integer. Returns the old user id.

sys_mknod <type> <pathname> <mode> [<major> <minor>]

Implements the *mknod* system call. The *type* argument can be one of `S_IFREG` (regular file), `S_IFCHR` or `c` or `u` (char device), `S_IFBLK` or `b` (block device), `S_IFIFO` or `p` (fifo, or named pipe). The *pathname* is the new node to create, *mode* is the file permission modes for the new file (use a leading 0 to specify an octal number), and *major* and *minor* are used when creating device nodes.

```
ifconfig <interface>
ifconfig <interface> addr <addr>
ifconfig <interface> netmask <netmask>
ifconfig <interface> broadcast <broadcast>
ifconfig <interface> pointopoint <pointopoint>
```
> Permits basic interface configuration functionality. In EtLinux it is used to bring up networking at boot time. When called with a single argument, the command returns the current configuration for the interface. When called with one of the listed extra arguments, it configures the interface.

```
route
route add <addr> netmask <mask> dev <dev>
route add default gw <gw> dev <dev>
```
> Permits basic routing configuration. Note that route deletion is not implemented. If you need such functionalities at runtime you'll need to have a real-world *route* command in your system. Please note that adding a route for the network directly connected to the device is not needed any more with kernel 2.2. If you boot 2.2 or later kernels, therefore, you'll typically only need to add a default to your system, after the network card has been configured. With 2.0, on the other hand, the LAN route must be added explicitly.

```
udp open [-o <option>[,<option>...]] <port> [<remoteIP>:<port>]
udp open [-o <option>[,<option>...]] client [<remoteIP>:<port>]
udp close <sock>
udp sendto <sock> [<remoteIP>:<port>] <msg>
udp recvfrom <sock> <fromvar>
udp join <sock> <multicastIP>
udp setmonitor <sock>
```
> The *udp* command implements UDP networking. Due to limitations int Tcl-7.6, data being transmitted may only be text.
>
> The *open* subcommands opens a new Tcl channel, and returns its identifier string. Allowed options are `reuseaddr` and `broadcast`; refer to the *setsockopt* manual page for details on their meaning. If the *port* argument is the literal string `client`, then the local port is randomized in the interval from 60000 to 62000. The optional *remoteIP:port* argument specifies the default destination for packets sent through this socket.
>
> The *close* subcommand closes the socket and frees any allocated memory associated to that socket.
>
> The *sendto* subcommand is used to send an UDP frame. The destination of the frame (*remoteIP:port*), if missing, is taken from the default specified in *udp open*.
>
> The *recvfrom* subcommand waits for an UDP frame, and returns its contents as return value of the command. The *remoteIP:port* information for the sender is stored in the *fromvar* variable, specified by the caller.
>
> The *join* subcommand associates the socket to a specified multicast address. This allows to receive multicast packets.

The *setmonitor* subcommand elects the specified socket as source for monitor information. If a monitor is specified, any outgoing UDP frame is first transmitted through the monitor address. The elected socket must feature a default destination, specified at *udp open* time. Using the monitor feature, and by setting a multicast address as destination address, you can easily monitor all UDP communication happening between cooperating applications by means of an external process that associate itself to the multicast group (actually, by using multicast, you can have monitor information collected independently in several places of the network).

```
xtime [-milli|-micro]
xtime -format <fmt> [<time_t>]
xtime -http [<time_t>]
xtime -diff <time>
xtime -timeto <time>
xtime -wait <tvar> <step>
xtime -split <array> [<time_t>]
xtime -join <array>
```

The *xtime* command replaces and expands the *clock* Tcl command. All floating-point values (fractions of seconds: milliseconds or microseconds) are handled using integer arithmetic, so everything can run on processors that lack an FPU (like a 386 or 386SX, but this is not limited to the i386 architecture).

The basic *xtime* command returns the number of seconds from the epoch. This is the standard *time_t* value used in all Unix systems to report the current time. If -milli or -micro is specified, a fractional part is added to the number.

Examples:

```
% xtime
=> 1006574784
% xtime -milli
=> 1006574786.829
% xtime -micro
=> 1006574789.629305
```

The -format option specifies a format to report a *time_t* according to *strftime(3)* (see the man page for *strftime* for details). The time being printed is either the current time or the one specified on the command line.

The -http option requests a time representation confirming to the HTTP protocol. This can't be accomplished using -format because HTTP requires the time to be reported in the GMT time zone, while xtime -format uses the local time zone. The command is used in the *httpd* implementation (see Section 4.3 [Httpd and Phttpd], page 17).

Specifying -diff makes *xtime* return the difference between the specified time and the current time (thus, negative values means the specified time has already passed). The -timeto option, on the other hand, returns the number of milliseconds before the specified time is reached; -timeto is designed to return a value suitable for the Tcl *after* command, thus 0 is returned for time events in the past (instead of returning a negative number that would make *after* spit an error if the value isn't checked by the application).

Examples:

```
% set t1 [xtime -micro]
=> 1006575363.669081
% xtime -diff $t1
=> -6.405620
%  set t2 [xtime]
=> 1006575469
% incr t2 60
=> 1006575529
% xtime -diff $t2
=> 43
% xtime -timeto $t2
=> 37371
% after [xtime -timeto $t2]
% puts "$t2 [xtime]"
=> 1006575529 1006575533
```

The -wait options reads the *tvar* variable as a time value; adds *tstep* and saves the resulting value in the same *tvar* variable. It then waits until that time instant is reached and returns. The command is designed to allow execution iterations of a task with a sharp time interval from each loop to the next.

```
puts [set t [xtime].000000]
=> 1006581978.000000
for {set i 0} {$i<4} {incr i} {
    xtime -wait t 1.500000
    puts "$t [xtime -micro]"
    # possibly do other tasks...
}
=> 1006581979.500000 1006581979.500020
=> 1006581981.000000 1006581980.999996
=> 1006581982.500000 1006581982.500029
=> 1006581984.000000 1006581984.000003
```

If the time to be waited for has already elapsed, the command returns imme-diately. In any case, the return value is the number of events that has elapsed; this is normally 1, but can be greater if the command is invoked much later than expected (for example, due to high system load). For example, if you want to execute a task once per second but sometimes you have a delay if more than one second across invocations of the command, you'll get 2 or more as return value. In case of delay, if you need to execute the task several times in a row to get the same number of events each day (as opposed to simply loosing events), you can use the following code:

```
set t [xtime]
while 1 {
    for {set i 0} {$i<[xtime -wait t $interval]} {incr i} {
        # do stuff
    }
}
```

If, on the other hand, you'd better loose events, use:

```
set t [xtime]
while 1 {
    xtime -wait t $interval
    # do stuff
}
```

The -split and -join commands are used to extract human-readable items from a time_t value and to do the inverse operation. Splitting uses the current time or the one specified on the command line, and sets a Tcl array as result, similarly to the file stat command included in standard Tcl. The target array is made up of the following elements: sec, min, hour, mday (day of the month), mon, year, wday (day of the week), yday (day of the year) and isdst (whether or not daylight saving is in effect). For the actual meaning of the elements please refer to the manual page for the *localtime* C function. Joining is the inverse operation, and the resulting time_t value is returned. Not all elements must be set, and the rules of mktime apply. No error is returned when required elements of the array are undefined.

mount
mount /proc

The command implements a minimal version of *mount*, usable to handle a single-partition system. When called without arguments, *mount* remounts the root filesystem in read-write mode. When called with a single argument, it mounts the *proc* filesystem in the /proc directory. Even though the argument is not currently parsed, it's suggested to use /proc as argument when mounting *proc*, to be forward compatible with possible later enhancements.

Note that in order to remount the root filesystem, the command needs to know what device is used as root filesystem. Therefore, *mount* reads '/etc/fstab', and expects the first three words in the file to be the block device used as root filesystem, the / directory, and the type of filesystem. Leading comment lines in /etc/fstab are not currently ignored.

umount

The *umount* command, trivially un-mounts */proc* and /. Errors are not checked.

uuencode <file>

uuencode can be used to transfer files from an embedded system to a computer connected to the serial console. Sometimes the serial console is the only communication means you have with your computer, so *ettclsh* has been enhanced to be able to send and receive files.

uudecode

The command decodes a file received from standard input, reading it until the end line is found. This allows to upload files to a system running a serial console with *ettclsh*.

fcopy <infile> <outfile> [once]

The command copies a Tcl channel to another Tcl channel. For example, this is used to serve http pages in the Ettcl web server (see Section 4.3 [Httpd and

Phttpd], page 17). If the `once` flag is specified, the input file is read-from only once, and only that data is pushed to the output file; this is used when Ettcl manages a pseudo-tty, to copy binary data from across the master and the slave. If not `once` flag is there, the input file is read up to end-of-file.

newcmdname <name>

This command saves *name* in '`/var/run`', in a file associated to the PID of the current process. This is how *ps* (see Section 3.2 [Procedures defined in init.tcl], page 11) can tell several ettcl processes apart.

inp <port>
inw <port>
outp <port> <byte-value>
outw <port> <word-value>

The commands read and write 8-bit and 16-bit I/O ports. Arguments are generic integers, so you must use a `0x` prefix for hex numbers. The commands use *ioperm* on the i386 platform, and use '`/dev/port`' on other platforms. The return value of the input commands is an hex number, output commands return no value. No endian conversion is performed.

readb <addr>
readw <addr>
readl <addr>
writeb <addr> <byte-value>
writew <addr> <word-value>
writel <addr> <long-value>

The commands read and write 8-bit, 16-bit, and 32-bit values from/to main memory. They use '`/dev/mem`' to access system memory, and can be used to access I/O memory if needed. Arguments and return values are the same as for I/O port commands, described above, and no endian conversion is performed.

3.2 Procedures defined in init.tcl

The following commands are defined within '`init.tcl`'. They are various utilities and replacements for system commands. They are used mainly in the EtLinux system. Some of the commands are not defined as procedures if a real external command exists. For example, you can just use `ls` to use the real '`ls`' command if available; this is different from internal commands, that are always defined (thus, you need to type `exec ifconfig` to access the external '`ifconfig`' command, if any, as `ifconfig` always refers to the internal commands.

If you want to run *ettclsh* without installing the package, please remember to set `LD_LIBRARY_PATH` and `TCL_LIBRARY` in your environment; the latter must be the directory where '`init.tcl`' is found.

which <cmd>

The command returns the full pathname of *cmd*, if found in one of default directories for command search. If no such command is found, an error is issued. Note that currently *which* doesn't use `$env(PATH)`.

unknown <args>

> The *unknown* procedure by default uses *which* to see if an external command is available with the same name as the unknown command. If any exists, it is run.

bgerror <error>

> The command spits the error message to *stderr*, prefixed with the PID of the process raising the error.

default_opt <name> <value>

> If not already set, the command sets the item *name* in the global array *options* to *value*. If the item is already set, it is not changed. This command is used by the various sample daemons.

cat <file> [<file> ...]

> Reads all of its arguments and prints their content to stdout. Files are assumed to be text files. The command is not defined if a real *cat* command is installed.

free

> Reports the amount of free memory, by returning the first two lines of '/proc/meminfo'. The command is not defined if a real *free* command is installed.

grep <expr> <file> [<file> ...]

> Runs a minimal version of *grep*, using the *regexp* Tcl command. If doesn't accept any option and doesn't use *stdin* if no files are specified (use '/dev/stdin' if you need to grep from *stdin*). The command is not defined if a real *grep* command is installed.

ls <file> [<file> ...]

> *ls* implements a small replacement for **ls -l**. Note that if a file is a directory, the directory itself is listed instead of its contents (this is the behavior selected by the **-d** option to the standard *ls* command). The command is not defined if a real *ls* command is installed.

cp
mv
rm

> These three commands are simply wrappers to the subcommands of the Tcl *file* command (**file copy**, **file rename**, **file delete**). Each of them is only defined if no external command with the same name exists.

ps

> Implements a simple replacement for the *ps* command. For each process, it uses the information in '/var/run' (if any) to tell the command name. Such information is stored by *newcmdname*, and is used to differentiate between several *ettclsh* processes that perform different tasks.

seteuid <uid> <command>

> Changes the current process user id to *<uid>*, executes *<command>* and restores the old process user id. *<uid>* must be a number.

`seteugid <uid> <gid> <command>`

> Changes the current process user and group ids to `<uid>`, `<gid>`, executes `<command>` and restores the old process user and group ids. `<uid>` must be a number.

`halt`
`reboot`

> The commands are used by the *Init* process (see Section 4.1 [Init], page 14); they send *init* a `SIGUSR1` to halt the system or a `SIGUSR2` to reboot the system. Actual shutdown is handled by *init*.

`cat-f <file>`
`tail-f <file>`

> These two commands run a separate process that reads the specified file and prints its contents to stdout whenever it is readable. They are quick hacks to read '`/proc/kmsg`' and similar FIFO-like data channels; there is no support to follow a regular file by monitoring size changes. The *tail-f* command reads from a few hundred bytes before the current file end (using *seek*), while the *cat-f* commands reads from the beginning of the file. The child process terminates when `EOF` is reached.

`interact <file> <prompt>`
`interact_oneline <ID> <prompt> <input> <output> <error>`

> The *interact* procedure forks a child process and uses *sys_dup* to attach the file to *stdin stdout* and *stderr*. It then calls sets *interact_oneline* as fileevent handler for *stdin*. You can set *interact_oneline* as fileevent handler for your own file, specifying an identifier, which prompt string to use, and the files to use for input, output and errors. The identifier is used as array element of the global *interact* array, to build Tcl commands from several lines.
>
> This mechanism is used by the console interaction script included in *init* (see Section 4.1 [Init], page 14).

`getAllowedHosts <service> [<file>]`

> The command returns a list of hosts (IP numbers or glob expressions) allowed to use the service *service*. This is used in *httpd* (see Section 4.3 [Httpd and Phttpd], page 17), *telnetd* (see Section 4.5 [Telnetd and Telnetc], page 26), *cmdd* see Section 4.4 [Cmdd], page 23), *smtpd* (see ⟨undefined⟩ [Smtpd], page ⟨undefined⟩). The configuration file being used is '`/etc/hosts.allow`', by default, but the caller can specify a different configuration file. The return value is a Tcl list of glob expressions for IP numbers.

`isAllowedHost <ip> <list>`

> This command matches an IP address against a list, usually returned by *getAllowedHosts*, described above. It returns a boolean value. If the list of allowed hosts for a service is empty, 1 is returned. Otherwise, the procedure returns 1 or 0 according to whether there is a match for the IP in the list.

4 Applications

The subdirectory '`Applications`' includes a few sample applications, that are used in EtLinux.

4.1 Init

The *init* application is the code of EtLinux. It's designed to run as process 1 of an embedded system, and it reads the various configuration files to know how the system is set up.

4.1.1 Boot time

When run, */sbin/init* first reads '`/etc/init.d/options`'; the file sets elements of the *options* array, to configure system operation. It can also set the default command search path by setting `env(PATH)`, and invoke other Tcl commands. Please remember that evaluating this file is the first thing *init* does when it starts.

These options are used to select *init* behavior during system boot. They are listed in the same order as they are used.

`options(fsck.root)`

This boolean option chooses whether the root filesystem is checked at boot time, using either '`/sbin/fsck.minix`' or, if available, '`/sbin/fsck`'. The device hosting the root filesystem is assumed to be the first word in '`/etc/fstab`'

If your system runs with NFS root, or if you don't ship an fsck program, you should set *options(fsck.root)* to zero.

`options(remount)`

This boolean options states whether the root filesystem must be remounted in read-write mode. This option must be set to zero if you use NFS root, if you mount read-write at boot time (in that case you can't set *options(fsck.root)* either, or if you want to run with a read-only root filesystem (in that case you'll need to set up at least '`/tmp`' and '`/var/run`' as separate partitions.

Note that support for running with a read-only root filesystem is not complete, as *init* calls the Tcl *exec* command a few times before mounting all filesystems, and the *exec* command creates files in '`/tmp`'. You'll need to change *init* to support a read-only root filesystem.

`options(proc)`

This boolean option states whether the '`proc`' filesystem must be mounted under '`/proc`'. It's usually set to 1, and you'll need to change a lot of applications if you choose not to mount '`/proc`'.

`options(swap)`

This boolean option selects whether or not to run "`/sbin/swapon -a`". To set it, you need to have the *swapon* executable installed and at least one entry for swap space in '`/etc/fstab`'

`options(fsck)`

If this boolean option is set to 1, then *init* will execute "`fsck -R -A -V -a`". You'll need an executable called *fsck* in your command search path, and a valid '`/etc/fstab`'.

`options(mount)`

This option is, once again, boolean. It selects whether or not all local filesystems must be mounted. If set, you'll need to have external *mount* and *umount*

commands. *Init* will invoke "`mount -a -t nonfs`" after umounting '`/proc`'. If you need to mount NFS volumes, you can do that later using a script in '`/etc/init.d/scripts`'.

`options(lowmem)`

This option, if set, specifies the three watermarks used for memory management: three value that will be written to '`/proc/sys/vm/freepages`'. Please note that recent 2.4 kernels do not have that file any more and use a different mechanism for setting memory limits. Since hosts running 2.4 are typically not memory constrained, no memory control is offered for 2.4. If the option is not set, the *init* process won't do anything.

`options(rm)`

The boolean option controls whether or not *init* will try to remove all files in '`/var/run`' and '`/tmp`'. If it does, it will recreate '`/var/run/wtmp`', to login programs won't complain (if installed).

`options(modules)`

The option, if set, must be a list of module names, that will be passed to */sbin/insmod*, one at a time. To load modules you'll need a version of *insmod* that is compatible with the kernel you are running.

`options(logfile)`

The option holds the name of the system log file. Each application included in *Ettcl* defines its own file name, using this as default.

`options(logfmt)`

The option defines the format of timestamps in log messages. In the distribured file it defaults to "`%Y-%m-%d %H:%M:%S %Z`", that expands to a string like "`2002-03-05 17:38:36 MET`". Each application included in *Ettcl* defines its own string but uses this variable as default.

`options(hostsfile)`

The option defines the default file used for host access, but each application can define its own. It defauls, in the distributed '`options`' file, to '`/etc/hosts.allow`'.

`options(hostname)`

The option sets the host name. The distributed file sets it to `localhost`.

`options(domainname)`

The option sets the domain name. The distributed file uses `localdomain`.

After processing the options described, he *init* program also reads all files in the directory '`/etc/init.d/options.d`' so system applications can set their options without modifying the main file. Note that all files in the directory are source as Tcl scripts, so you can actually do more than setting options, if you need that. The files are read in alphanumeric order.

The next step in booting is running all scripts found in '`/etc/init.d/scripts`'. The name of each script must start with "S" followed by a two-digit number. These scripts are run once only. Typical tasks for these scripts are setting up the network, or mounting NFS filesystems.

Then, *init* sources all scripts in '/etc/init.d/respawn', restarting each of them as soon as it terminates. The name of each script must start with "S" followed by a two-digit number. These scripts are usually terminal login (or the shell running on the serial console).

Finally, any regular file found in /etc/applications is run as a separate process. These processes are not restarted when they terminate, so if your application may terminate as part of normal operation, you should either place it in '/etc/init.d/respawn' or arrange for restarting in the Tcl file.

Please note that everything that is started by *init* (i.e., all scripts in 'scripts', in 'respawn', and in 'applications') are Tcl scripts. They are read as Tcl commands using the *source* command, even though the process that reads the file is not *init* but a child process. If one of your applications is an executable file, you should write a short wrapper script like this one:

```
sys_exec /your/path/to/application
```

Please don't use "exec" in this case, to avoid running two processes where one will do. Any file in 'scripts' and 'respawn' whose name doesn't match "S*" will not be read. Any subdirectory of 'applications' is ignored, so you can use for example a directory called 'warehouse' as a repository of applications that you want to keep around but don't want to automatically run.

4.2 Network configuration scripts

The following network configuration scripts are included in the ettcl distribution:

/etc/init.d/scripts/S00loopback
> This script sets up the loopback interface by means of the built in `ifconfig` commmand.

/etc/init.d/scripts/S10network
> This script sets up the other network interfaces, always using the built-in ifconfig command and settings taken from the `options()` array.

Here's a list of options() entries used by S10network:

options(if:<if>)
> The option sets the configuration settings for network interface <if> (for instance <if>=eth0). It must have the following format: "addr netmask network [pointopoint]", where `addr` is the interface's IP address in decimal dotted notation, while `netmask` is the IP network mask for the interface. `pointopoint` is the address of the other end of the line in case of a point-to-point link (optional). Finally, `network` is used by S10network to add a route for the network to which the interface is connected. See also the `ifconfig` and `route` commands in See Section 3.1 [New commands], page 3. Example of a configuration string for eth0:
>
> > ```
> > set options(if:eth0) "192.168.16.10 255.255.255.0 192.168.16.0"
> > ```

options(defaultroute)
> The option sets the default route. It must have the following format: "gw dev".
> A default route is set by S10network with the following command:

```
route add default gw  [lindex $options(defaultroute) 0] \
              dev [lindex $options(defaultroute) 1]
```
See also the `route` command in See Section 3.1 [New commands], page 3.

`options()` is filled with network options when '/etc/options' is read at boot time. What happens is that if '/etc/network.options' exists, then it is read when the last line of '/etc/options' is executed.

4.2.1 Shutdown time

During system runtime *init* waits for pending signals. Whenever `SIGCHLD` is received, a process is reaped; if the process just died is to be respawned, init respawns it

If `SIGUSR1` or `SIGUSR2` are received, *init* halts or reboots the system (resp.); all processes are sent the `TERM` signal and after a while the `KILL` signal; filesystems are unmounted (first calling "`umount -a`", then using the internal *umount* command, so all cases are covered); and then the system is halted or rebooted. In order to halt or reboot the system from an interactive *ettclsh* session you can invoke the *halt* or *reboot* procedures, that simply send a signal to *init*. If you have no console access to see shutdown messages, you can count shutdown to be over in 5 seconds at most.

4.3 Httpd and Phttpd

The *httpd* program included in *ettcl* is compliant with version 1.0 of the HTTP protocol; even though version 1.1 of the protocol is widely used, version 1.1 requires compatibility with version 1.0, so this *httpd* works with all recent browsers (but it might not work with very old browsers, as it doesn't accept queries that are not HTTP-1.0 compliant).

phttpd is a parsing version of *httpd*. It can parse files using external filters or internal procedures.

4.3.1 Httpd Configuration

The server is configured by setting items in the *options* array. These items can be set in Tcl before the server is executed, and if they are not set the default is provided by *httpd* itself.

The distribution includes *httpd-run*, a script that sets configuration variables and then runs *httpd*. As an alternative, you can fill '/etc/httpd.cfg', as described in Section 4.3.2.1 [Httpd Configuration Files], page 19.

These are the configuration variables, and their defaults:

`options(httpd:cfg)`
> If the variable is set, then its value is used as a configuration script name. The script is executed using the *source* command. If it is not set, nothing happens.

`options(httpd:name)`
> The name of this service, used for host access control. If unset, it defaults to "`httpd`".

`options(httpd:port)`
> The TCP port where the server runs. If unset, it defaults to 80.

`options(httpd:docroot)`

> The document root, where HTML pages are looked for. If unset, it defaults to '/html'.

`options(httpd:logfile)`
`options(logfile)`

> The name of the log file. The former variable defaults to the latter, which in turn defauls to '/var/log/ettcl.log'. The log messages being written includes the application name, so different servers can share the same log file. All write operations are atomic (done with a single *write* system call).

`options(httpd:logfmt)`
`options(logfmt)`

> The format for timestamp strings; the timestamp string is used as a leading string for every line written in the log file. The default for the former variable is the value of the latter, which defaults to "%Y-%m-%d %H:%M:%S %Z".

`options(httpd:hostsfile)`
`options(hostsfile)`

> The name of the configuration file for host access control. If unset, the former variable default to the value of the latter. You can thus elect a system-wide default hostsfile, or one for this process alone. In an EtLinux system you might want to set `options(logfmt)` from within '/etc/init.d/options', so all servers will use the same format.

`options(http:type:.html)`

> This variable is set to "`text/html`". Whenever a file is served by the web server, it's extension is used to address an item `http:type:`*extension* in the `options` array. You might set the types from a standard 'mime.types':

```
if ![catch {set F [open /etc/mime.types]}] {
    while {[gets $F str]>=0} {
        if [regexp "^ *#" $str] continue
        if [catch {set type [lindex $str 0]}] continue
        for {set n 1} \$n<[llength $str] {incr n} {
            set options(httpd:type:.[lindex $str $n]) $type
        }
    }
    close $F
}
```

`options(httpd:403)`
`options(httpd:404)`
`options(httpd:501)`

> The server can report HTTP errors of type 403 (forbidden), 404 (not found), and 501 (not implemented). When the error is reported, if the associated variable is set, its value is opened as a file name that is returned to the client, after the standard error message. This allows customization of error reporting.

4.3.2 Httpd Features

The server supports up to two configuration files (to set array elements in the *options* array or run arbitrary Tcl commands), host access control based on IP address, GET and POST methods (using the CGI standard to talk with external applications), error reporting with user-provided error notification files, Mime types associated to file names.

4.3.2.1 Httpd Configuration Files

When the server starts up, it reads the configuration file '/etc/httpd.cfg', if it exists. It then reads the file *$options(httpd:cfg)* if the variable is set.

If you want to run several web servers on several TCP ports, you can avoid using '/etc/httpd.cfg' and use a per-process configuration file by setting the *httpd:cfg* item in *options* to a different value for each process.

Each configuration file, then, can set a different value for the TCP port number and for the service name (to differentiate host access control while using the default hostsfile), or change the name of the hostsfile, as well as other server features.

4.3.2.2 Httpd Host Access

The *httpd* process reads '/etc/hosts.allow' (or whatever hostsfile defined by *options(httpd:hostsfile)* by calling the procedure *getAllowedHosts* using *httpd* as a key (or whatever application name defined by *options(httpd:name)*). The configuration file specifies which hosts are allowed to access the various Ettcl services.

Empty lines and lines that begin with a hash mark are ignored, other lines are formatted like "*service*: *host*", where *host* is a glob expression for an IP address. If a given *service* doesn't appear in the file, then all hosts are allowed. If the service appears in the file at least once, then only IP addresses that match the glob expression are allowed. For example, the following lines allow access from *localhost*, a C class and a single host. Note that you can also place several glob expressions on the same lines, separated by space characters.

```
httpd: 127.*
        httpd: 192.168.1.*
        httpd: 151.38.134.203
```

To allow no hosts at all, use "httpd: none" or any other expression that doesn't match any IP address. To allow any host, use httpd: *, or no httpd line at all.

If you want to use the same access rules for all Ettcl services, you can define *options(httpd:name)* and all other service names to the same value (e.g., "any"), in order to use the same set of host glob expression for all services.

4.3.2.3 GET and POST Methods

The server accepts GET and POST methods. Any query including either of "..", "&", "?", "*", "(", ")", "$", ";", "#", "|" is refused with an error of *403 Forbidden*. See Section 4.3.2.5 [Httpd Error Reporting], page 21.

Any query that starts with cgi-bin, with or without a leading slash is considered a CGI request. See Section 4.3.2.4 [CGI Interface], page 20.

If the file being requested doesn't exist, and error of *404 Not found* is reported. See Section 4.3.2.5 [Httpd Error Reporting], page 21.

Note that this version of the server doesn't handle hex-encoded characters in file names (for example, "%7E" isn't converted to "~"). It allows them in CGI parameters, though.

4.3.2.4 CGI Interface

If the filename requested by either a `GET` or a `POST` query begins with `cgi-bin`, either with or without a leading slash, everything in the file name up-to and excluding the first question mark is considered an application name (the application is searched in the filesystem in the `cgi-bin` directory under the document root selected for the process).

If the application is written in *ettclsh*, then the internal *sys_fork* and *sys_dup* commands are used to attach to the application. Otherwise, the application is executed using the *exec* Tcl command.

When the application is executed by the `GET` method, it will find `QUERY_STRING` set in the environment. Moreover, if the CGI application is not a Tcl script, all arguments are set as environment variables, and the current date in http format is recorded as `DATE` in the environment. The example *sh* CGI applications use this information.

When the application is called via `POST`, it will find `CONTENT_LENGTH` set in the environment and will be able to read data from *stdin*. Note that you may need to remove trailing newlines from query data.

The stdout of the application is directly connected to the network socket, with no intervention from the *httpd* server.

Several CGI examples are provided in the source package, with a '`cgidemo.html`' html page that links to all of them.

Note that you can use the *etget* application (part of the *httpd* source dir) to time GET and POST requests. These are some timings I get with a 386 running Linux-2.2.18 and libc5 in 4MB of memory:

```
morgana% etget pico 80 '/cgi-bin/glob?FILE=%2Fvar%2Frun&GLOB='
HTTP/1.0 200 OK
0.502689
morgana% etget pico 80 '/cgi-bin/ls?FILE=%2Fvar%2Frun&GLOB='
HTTP/1.0 200 OK
0.362456
```

This is the same example run with 1MB less RAM:

```
morgana% etget pico 80 '/cgi-bin/ls?FILE=%2Fvar%2Frun&GLOB='
HTTP/1.0 200 OK
1.203962
morgana% etget pico 80 '/cgi-bin/glob?FILE=%2Fvar%2Frun&GLOB='
HTTP/1.0 200 OK
3.515961
```

To run POST queries in *etget*, specify the POST data as fourth argument.

4.3.2.5 Httpd Error Reporting

The *httpd* application reports errors according to HTTP/1.0. According to the setting of configuration variables, an html file can be returned to the network connection, so you can choose to return a custom error page instead of the plain error message. The custom error page can even include a form (for example, to make the user report the error).

The daemon reports three errors:

- 403 Forbidden Returned if the query includes .. or one of the forbidden characters. The html file, if any, is *$options(httpd:403)*. See Section 4.3.2.3 [GET and POST Methods], page 19. The same error is generated whenever a client is rejected by host access policy.
- 404 Not found Returned whenever the requested file or CGI application is not available. The html file, if any, is *$options(httpd:404)*.
- 501 Not Implemented Returned when a method other than GET or POST is received, For example, HEAD is not yet implemented. The html file, if any, is *$options(httpd:404)*.

4.3.2.6 Httpd Logs

The applications writes log information to the log file, opened in append mode. Each even being logged is written by issuing a single *write* system call, to prevent problems when several processes write to the same log file.

The following events are logged, each of them is timestamped according to *options(httpd:-logfmt)*:

- All errors Whenever an HTTP error is generated, the server writes one record to the log file, including the error number, error string and detail. This includes all refused connections because of host access policies.
- All queries As soon as a query is received, the server writes one log record including the IP address of the client and the whole query string.

4.3.3 Phttpd Specifics

The *phttpd* server is a superset to the *httpd* server, and might eventually replace it. You'll need to run it if your pages are written in *php* or similar tools, of if you want to run the "*server side include*" machinery.

In addition to everything described above, the parsing server tries to match the filename component of all queries with a list of parsers. If a match is found, then the query is managed as a CGI one, filtering the page with a Tcl procedure or an external process. Optionally, a query string can be accepted by the filter with the usual syntax for GET queries.

Configuration variables use the *httpd* name, not *phttpd*, since they work in the same way in the two servers. Only, *phttpd* accepts more variables to activate parsers.

4.3.3.1 How to Activate a Parser

The best way to learn how to configure *phttpd* is looking at *phttpd-run* and actually run it. The script is very similar to *httpd-run*, but uses the current directory as document root by default, to ease testing the three sample files.

A parser is specified by setting an array entry in the *options* array, whose name is "`options(httpd:parse:`*regexp*`)`", for an appropriate regular expression *regexp*.

The regexp is tried against all file names queried to the server, and a CGI is activated if a match is found (if more than one regexp matches, only one of them will apply).

When a match occurs, the value of the variable is expected to identify a parser and a flag. The parser is either an absolute pathname or the name of a Tcl procedure; the flag is 0 or 1, telling whether or not the server should print the "`HTTP/1.0 200 OK`" message before running the parser.

The regular expression must include one or two subexpressions: the former must identify the part of the pathname that will become the argument of the parser, and the latter, if present, the query string. For example:

```
set {options(httpd:parse:^(.*\.php)(\?.*)?$)}  {/usr/bin/php3 1}
set {options(httpd:parse:^(.*\.tcl)(\?.*)?$)}  {/usr/bin/tclsh 0}
set {options(httpd:parse:^(.*\.shtml)$)}  {shtml_parse 0}
```

The first two examples show how to call an external program, possibly passing extra arguments in the usual form. The last example shows how to call an internal Tcl procedure without parameters in the query

When running *phttpd-run* with the default arguments, you'll be able to try out '`test.php`' (a program that prints the current date), '`test.tcl`' (a program that prints its own environment) and '`test.shtml`' (a simple file that includes another).

This is how passing two parameters to '`test.tcl`' is accomplished:

```
morgana% lynx -dump 'http://localhost:8080/test.tcl?ARG1=me&ARG2=you' \
             | grep ARG
ARG1 = me
ARG2 = you
QUERY_STRING = ARG1=me&ARG2=you
```

4.3.3.2 The Mechanism in Serving Parsed Pages

When the filename component of an HTTP query matches one of the parser regular expressions, the first subexpression is extracted from the filename and used as command-line argument when calling the parser later on.

The second subexpression, if any, is used to set `QUERY_STRING` in the environment (and later individual environment variables, in the child process). If no second subexpression is there, or if it is empty, the `QUERY_STRING` environment variable will be set to the empty string.

The server then forks a subprocess as usual (whether the pathname was a `/cgi-bin/` one, or a parser matched the file name), and redirects standard input and output of such a process to the network socket (please note that the code does that in two different ways according to the type of coprocess, but the result is the same).

If the *flag* argument was set to true in the parser specification, the `HTTP/1.0 200 OK` string is printed. The string is never printed for `/cgi-bin/` pages.

If we are serving a `/cgi-bin/` file or if the parser is specified as an absolute pathname, the file is examined as to whether it's an EtTcl script or not. If not, the program is executed after splitting the query string and setting environment variables accordingly. If it is a

script, it is simply `source`-d at stack level 0, after setting *argv* and *argv0* in the usual Tcl way.

Otherwise, the parser is expected to be a Tcl procedure, and it is invoked at stack level 0 with the specified filename as argument.

4.4 Cmdd and Cmdc

The *cmdd* application is a non-forking daemon for executing Ettcl commands. It runs on both TCP and UDP sockets. To connect to the TCP port you can use *telnet*, to connect to the UDP port you must use *cmdc*, which requires *ettclsh* on the client host.

cmdd is designed to allow interactive access to small computers, that would suffer from running a real forking *telnetd*.

Warning: Please note that with *cmdd* you have unlimited access to the server system. The program is meant to be used only for debugging and administration. Please be very careful about host access configuration to *cmdd*. Also, please note that each client can change the *options* array and thus redirect log files and to other hairy stuff, possibly unwillingly.

4.4.1 Cmdd Configuration

The server is configured by setting items in the *options* array. These items can be set in Tcl before the server is executed, and if they are not set the default is provided by *cmdd* itself.

These are the configuration variables, and their defaults:

`options(cmdd:cfg)`

> If the variable is set, then its contents are used as a configuration file name. The file is executed using the *source* command. If it is not set, nothing happens.

`options(cmdd:port)`

> The TCP and UDP ports where the server runs. If unset, it defaults to 2300 (reminiscent of *telnet*, that runs on port 23).

`options(cmdd:name)`

> The name of this service, used for host access control. If unset, it defaults to `"cmdd"`.

`options(cmdd:tcp)`

> Whether or not the server should listen to the TCP port. The variable defaults to 1 and can be set to 0 to prevent the TCP socket to be opened.

`options(cmdd:udp)`

> Whether or not the server should listen to the UDP port. The variable defaults to 1 and can be set to 0 to prevent the UDP socket to be opened.

`options(cmdd:udptimeout)`

> The timeout value, in seconds, before an UDP connection is considered inactive. It defaults to 60. Please note that *cmdc* explicitly closes the connection on exit, so you can tell logout events from timeout events. The timeout mainly exists in order to release memory associated to inactive client connections.

```
options(cmdd:hostsfile)
options(hostsfile)
```
>   The name of the configuration file for host access control. If unset, the for-
>   mer variable defaults to the value of the latter, which in turn defaults to
>   '/etc/hosts.allow'. You can thus elect a system-wide default hostsfile, or
>   one for this process alone.

```
options(cmdd:logfile)
options(logfile)
```
>   The name of the log file. The former variable defaults to the value of the
>   latter, which in turn defaults to '/var/log/ettcl.log'. The log messages
>   being written includes the application name, so different servers can share the
>   same log file. All write operations are atomic (done with a single *write* system
>   call).

```
options(cmdd:logfmt)
options(logfmt)
```
>   The format for timestamp strings; the timestamp string is used as a lead-
>   ing string for every line written in the log file. The default for the format
>   string is $options(logfmt), which in turn defaults to "%Y-%m-%d %H:%M:%S
>   %Z". In an EtLinux system you might want to set options(logfmt) from
>   within '/etc/init.d/options', so all servers will use the same format.

4.4.2 Cmdd Features

The server supports up to two configuration files (to set array elements in the *options*
array or run arbitrary Tcl commands), host access control based on IP address, TCP and
UDP operation.

4.4.2.1 Cmdd Configuration Files

Whe the server starts up, it reads the configuration file '/etc/cmdd.cfg', if it exists. It
then reads the file *$options(cmdd:cfg)* if the variable is set.

If you might want to run several *cmdd* servers on several TCP/UDP ports, although it
isn't as interesting as it is with HTTP. To run several servers, please refer to Section 4.3.2.1
[Httpd Configuration Files], page 19.

4.4.2.2 Cmdd Host Access

The *cmdd* process reads '/etc/hosts.allow' (or whatever hostsfile defined by *op-
tions(cmdd:hostsfile)* by calling the procedure *getAllowedHosts* using *cmdd* as a key (or
whatever application name defined by *options(cmdd:name)*). The configuration file speci-
fies which hosts are allowed to access the various Ettcl services.

For detailed syntax and example use, please refer to Section 4.3.2.2 [Httpd Host Access],
page 19.

4.4.2.3 The TCP Server

The TCP server listens to a TCP socket and reads input line by line. When the concatenation of all pending lines build to a complete Tcl command, the server executes the command at global scope. Input data received from different client sockets is never mixed, so you can get multiplexing operation even if the server doesn't fork child servers for each active connection.

Please note that if the command takes time to execute, the server won't be able to server other client in the meanwhile. If you want to run such kind of tasks you need to run *telnetd*.

To connect to the TCP *cmdd* server, you need to use the standard *telnet* client, like this:

```
rudo% telnet morgana 2300
Trying 192.168.16.1...
Connected to morgana.systemy.it (192.168.16.1).
Escape character is '^]'.
% puts pio
pio
% set options(cmdd:port)
2300
% exit
Connection closed by foreign host.
```

The *exit* command, as shown, terminates the TCP connection. It doesn't, however, terminate the server process. Note that it is possible to terminate the server in some other way (by killing it, for example). It's worth repeating that *cmdd* is designed to be used only as a debugging and administration tool for small EtLinux computers (those too small to run a forking *telnetd*).

Please note that with *cmdd*, communication is line-oriented. Therefore you can't run a full-screen editor or anything like that in your interactive session. To that aim, please use *telnetd* (see Section 4.5.2 [Telnetd Features], page 28).

4.4.2.4 The UDP Server

The UDP server listens to an UDP socket. Input is expected to be received line-by-line and if the packets have a trailing dot it is removed. The trailing dot is added by *cmdc* in order to preserve the trailing newline, tha would otherwise be stripped by the implementation of the *udp* Tcl command (that is text-oriented).

When the concatenation of all pending lines build to a complete Tcl command, the server executes the command at global scope. Input data received from different client sockets is never mixed, so you can get multiplexing operation for different UDP clients, as well as multiplexing TCP and UDP clients.

Please note that if the command takes time to execute, the server won't be able to server other client in the meanwhile. If you want to run such kind of tasks you need to run *telnetd*.

To connect to the UDP *cmdd* server, you need to use *cmdc* or an equivalent program.

The *exit* command, if sent by an UDP client, terminates the UDP connection (i.e., it makes the server release memory associated to this client and record a *logout* event to the log file. If a client doesn't log out, a *timeout* event is marked after the timeout period elapsed.

The same caveat about security as for TCP (see above) applies. Again, *cmdd*, communication is line-oriented. Therefore you can't run a full-screen editor with it.

4.4.2.5 Cmdd Logs

The *cmdd* application records the following events to system logs:

- TCP Login When a new tcp connection is opened, the server logs the client's IP address and port.
- TCP Logout When a connection is closed, the server logs the IP address, the port, and the duration of the connection.
- UDP Login When an UDP packet is received by a new client, the IP address and port are logged.
- UDP Logout When a client sends an empty packet (i.e., only the trailing dot), a logout event is logged, with IP, port, and duration of the connection.
- UDP Timeout If a client doesn't send packets for the timeout time lapse (one minute by default), a timeout event is logged with IP, port, and duration.
- TCP and UDP Rejected clients If a client (TCP connection or UDP packet) doesn't pass the host access control, the event is logged as well.

4.4.3 Cmdc

The *cmdc* program is the UDP client for *cmdd*. It is used much like the normal *telnet* application but it send UDP packets for each input line (with a trailing dot added, as explained in Section 4.4.2.4 [The UDP Server], page 25).

At startup, the program sends an empty packet in order to mark a login event and get back a prompt. When it reads EOF from *stdin*, it sends an `exit\n` line to the server. When an empty packet is received the program terminates. If the empty packet is the first reply, it reports a message of "Connection refused by remote host".

If no prompt is shown at program startup, then the server did not reply or the reply was lost. Please remember that this simple UDP communication is only reliable on a non-saturated LAN.

```
rudo% cmdc morgana 2300
% puts pio
pio
% set options(cmdd:udp)
1
% exit
```

4.5 Telnetd and Telnetc

The *telnetd* application is a daemon that execute Ettcl commands running a subprocess per TCP connection. Such subprocess is run ina virtual terminal, so you can call *vi* or other full-screen programs from the shell. The server must be run by the superuser (otherwise, it wouldn't be able to open a tty pair), and you should use *telnetc* as a client. The daemon listens to port 230 by default.

Warning: Please note that with *telnetd* you have unlimited access to the server system. The program is meant to be used only for debugging and administration. Please be very careful about host access configuration to *telnetd*. However, unlike *cmdd*, the client can't change the *options* array because each shell runs in a different process.

4.5.1 Telnetd Configuration

The server is configured by setting items in the *options* array. These items can be set in Tcl before the server is executed, and if they are not set the default is provided by *telnetd* itself.

These are the configuration variables, and their defaults:

`options(telnetd:cfg)`

> If the variable is set, then its contents are used as a configuration file name. The file is executed using the *source* command. If it is not set, nothing happens.

`options(telnetd:port)`

> The TCP port where the server runs. If unset, it defaults to 230 (reminiscent of *telnet*, that runs on port 23).

`options(telnetd:name)`

> The name of this service, used for host access control. If unset, it defaults to `"telnetd"`.

`options(telnetd:hostsfile)`
`options(hostsfile)`

> The name of the configuration file for host access control. If unset, the former variable defaults to the value of the latter, which in turn defaults to '`/etc/hosts.allow`'". You can thus elect a system-wide default hostsfile, or one for this process alone.

`options(telnetd:logfile)`
`options(logfile)`

> The name of the log file. The default for the latter variable is '`/var/log/ettcl.log`'. The log messages being written includes the application name, so different servers can share the same log file. All write operations are atomic (done with a single *write* system call).

`options(telnetd:logfmt)`
`options(logfmt)`

> The format for timestamp strings; the timestamp string is used as a leading string for every line written in the log file. The default for the format string is `$options(logfmt)`, which in turn defaults to "`%Y-%m-%d %H:%M:%S %Z`". In an EtLinux system you might want to set `options(logfmt)` from within '`/etc/init.d/options`', so all servers will use the same format.

4.5.2 Telnetd Features

The server supports up to two configuration files (to set array elements in the *options* array or run arbitrary Tcl commands), host access control based on IP address, TCP and UDP operation.

4.5.2.1 Telnetd Configuration Files

Whe the server starts up, it reads the configuration file '`/etc/telnetd.cfg`', if it exists. It then reads the file *$options(telnetd:cfg)* if the variable is set.

Unlike *httpd* and *cmdd*, you would never need to run several servers at once, since each client gets its own process anyways.

4.5.2.2 Telnetd Host Access

The *telnetd* process reads '`/etc/hosts.allow`' (or whatever hostsfile defined by *options(telnetd:hostsfile)* by calling the procedure *getAllowedHosts* using *telnetd* as a key (or whatever application name defined by *options(telnetd:name)*). The configuration file specifies which hosts are allowed to access the various Ettcl services.

For detailed syntax and example use, please refer to Section 4.3.2.2 [Httpd Host Access], page 19.

4.5.2.3 The Telnet Server

The TCP server listens to a TCP socket. When a new connection is received, after host access control the daemon opens tty pair and forks a child. The child gets its standard files connected to the slave tty, and forks an interactive shell.

The parent process continues to monitor the master tty and the network socket, copying data from the pty to the network and vice versa as needed.

To connect to the *telnetd* server, you need to use the *telnetc* client, like this:

```
telnetc pico 230
accept TCP: sock5 192.168.16.1 3111
% free
        total:     used:     free:   shared: buffers:  cached:
Mem:    1695744  1540096    155648    176128     8192   126976
% ps
  User  PPid  Pid St Size    RSS  Name
 ------------------------------------
     0     0    1 S  1048    104 init
     0     1    2 S     0      0 kflushd
     0     1    3 S     0      0 kupdate
     0     1    4 S     0      0 kpiod
     0     1    5 S     0      0 kswapd
     0     1    6 S     0      0 rpciod
     0     1   19 S   808     56 inetd
     0     1   20 S  1048    104 console-interaction
     0     1   22 S  1076    148 cmdd
     0     1   23 S  1072    144 httpd
     0     1   24 R  1076    196 telnetd
     0    22   39 S   816     56 cat
     0    24   41 R  1064    140 -sh(192.168.16.1)
% exit
morgana%
```

The *exit* command terminates the child process, that is reaped by the parent.

4.5.2.4 Telnetd Logs

The *telnetd* application records the following events to system logs:

- Login When a new tcp connection is opened, the server logs the client's IP address and port.
- Logout When a connection is closed, the server logs the IP address, the port, and the duration of the connection.
- Rejected clients If a client doesn't pass the host access control, the event is logged as well.

4.5.3 Telnetc

The *telnetc* program is the client for *telnetd*. It is written in standard Tcl, so you don't need to have Ettcl installed on the client computer. It behaves like a normal *telnet* application, reading standard input in raw form and disabling character echo.

The program need to run the external *stty* command, to change tty modes on its own terminal.

4.6 Crond

The *crond* program is an implementation of the usual *cron* Unix facility. It awakes every minute and runs any pending command according to a single crontab file. If the crontab is modified, you'll need to send SIGHUP signal to *crond>* in order to install the new configuration.

4.6.1 Crond Configuration

The server is configured by setting items in the *options* array. These items can be set in Tcl before the server is executed, and if they are not set the default is provided by *crond* itself.

These are the configuration variables, and their defaults:

`options(crond:cfg)`
> If the variable is set, then its contents are used as a configuration file name. The file is executed using the *source* command. If it is not set, nothing happens.

`options(crond:crontab)`
> The crontab being used. It defaults to '`/etc/crontab`'.

`options(crond:recalc)`
> This boolean variable states whether or not *crond* will ask the current time again before going to sleep. If you are changing the system date and time by more than a few seconds after boot you'll need this, otherwise, *crond* will either sleep for a long time (if you move the date backwards) or will loop (if you move the date forward). If you don't change the date after starting the application, or you only fine-tune it to an external reference, then you can save cycles by setting the variable to 0.

```
options(crond:logfile)
options(logfile)
```
> The name of the log file. The former variable defaults to the latter, and the latter defaults to '`/var/log/ettcl.log`'. The log messages being written includes the application name, so different servers can share the same log file. All write operations are atomic (done with a single *write* system call).

```
options(crond:logfmt)
options(logfmt)
```
> The format for timestamp strings; the timestamp string is used as a leading string for every line written in the log file. The default for the format string is `$options(logfmt)`, which in turn defaults to "`%Y-%m-%d %H:%M:%S %Z`". In an EtLinux system you might want to set `options(logfmt)` from within '`/etc/init.d/options`', so all servers will use the same format.

4.6.2 Crond Features

The application reads a standard '`crontab`' file.

Empty lines and lines starting with a hash mark are ignored. Lines that look like variable assignment are used to set environment variables; the variable name must be made up of uppercase letters and the underscore character), spaces and tabs around the = character are ignored, and the rest of the line is used as value. Please note that trailing spaces in the line are not trimmed.

Other lines in '`crontab`' are parsed like the real *cron* does. But whereas the Unix *cron* application passes the command to an external shell, this *crond* evaluates the command using the Tcl *uplevel* command. Each command is evaluated in a child process.

4.7 Smtpd

The *smtpd* program is an implementation of the *Simple Mail Transfer Protocol* service. The program is RFC2821[1] compliant and implements a simple back-end to allow implementation of email-driven services.

4.7.1 Smtpd Configuration

The server can be configured by setting items in the *options* array. Such items can be set before the server is executed; in case they are not, *smtpd* uses the default values.

The distribution includes *smtpd-run*, a script that sets the configuration variables to values for which superuser privileges are not required, and then runs *smtpd*.

The following list describes the configuration variables and their defaults:

```
options(smtpd:cfg)
```
> If this variable is set, then its value is used as a configuration script name and is executed using the *source* command. If the variable is not set, nothing happens.

---

[1] Please refer to the '`README`' in '`Applications/smtpd`' for details

`options(smtpd:name)`
> The name of this service, used for host access control. If unset, it defaults to "`smtpd`".

`options(smtpd:port)`
> The TCP port the server listens to. If unset, it defaults to 25.

`options(smtpd:maildir)`
> The mailboxes default directory: the place where received data are stored and where configuration lives. If unset, it defaults to '`/var/spool/mail`'.

`options(smtpd:logfile)`
`options(logfile)`
> The name of the log file. The former variable defaults to the value of the latter, which defauls to '`/var/log/ettcl.log`'. Log messages being written include the application name. All write operations are atomic (done with a single *write* system call).

`options(smtpd:logfmt)`
`options(logfmt)`
> The format for timestamp strings; the timestamp string is used as a leading string for every line written in the log file. The default for the former variable is `$options(`logfmt`)`, which in turn defauls to "`%Y-%m-%d %H:%M:%S %Z`".

`options(smtpd:hostsfile)`
`options(hostsfile)`
> The name of the configuration file for host access control. If unset, the former variable defaults to the value of the latter, which defauls to '`/etc/hosts.allow`' You can thus elect a system-wide default hostsfile, or one for this process only.

`options(smtpd:hostname)`
`options(hostname)`
> The host name, not including the domain name. If unset, the former variable defaults to the value of the latter, which defauls to *localhost*.

`options(smtpd:domainname)`
`options(domainname)`
> The host domain name. If unset, the former variable defaults to the value of the latter, which defaults to *localdomain*.

`options(smtpd:alias:`*name*`)`
> Array items with a leading `smtpd:alias:` are used in alias expansion, allowing email for one user name to be delivered to another user name.
>
> For instance, if you want specify root as a postmaster's alias you should set `options(smtpd:alias:postmaster)` to `root`. The value of an alias must be a single user name. Delivery is then performed as specfied in Section 4.7.2.3 [Smtpd User Validation], page 32.

4.7.2 Smtpd Features

The server supports up to two configuration files (to set array elements in the *options* array or run arbitrary Tcl commands), host access control based on IP address, user valida-

tion control based on the contents of the directory `$options(smtpd:maildir)`, execution of scripts using the message body as an argument (again, configured by files in `maildir`.

4.7.2.1 Smtpd Configuration Files

When the server starts up, it reads the configuration file '`/etc/smtpd.cfg`', if it exists. It then reads the file *$options(smtpd:cfg)* if such variable is set.

4.7.2.2 Smtpd Host Access

The *smtpd* process reads '`/etc/hosts.allow`' (or whatever file defined by *options(smtpd:hostsfile)*) by calling the procedure *getAllowedHosts* using *smtpd* as a key (or whatever application name defined by *options(smtpd:name)*). The configuration file specifies which hosts are allowed to access the various Ettcl services.

For detailed syntax and example use, please refer to Section 4.3.2.2 [Httpd Host Access], page 19.

4.7.2.3 Smtpd User Validation

When the *smtpd* process receives an email message, it uses the local part of the recipient name to choose whether to accept or refuse the message. The domain is simply ignored.

Validation is performed by looking at the contents of `$options(smtpd:maildir)` ('`/var/spool/mail`' by default), after alias substitution (see Section 4.7.1 [Smtpd Configuration], page 30).

When a client tries to send email to a user *user*, *smtpd* subtitutes the alias (if any). Next, it looks for either *user*`.mbox` or *user*`.post` within '`maildir`'. If neither file exists, the receipient is refused unless it is *postmaster*. Mail for postmaster is always accepted, and delivered to '`/dev/null`' if no action is configured for that user; RFC2821 requires email to *postmaster* to always be accepted.

If the file *user*`.mbox` exists, then the message is appended to the '`user`' file in mailbox format. If *user*`.post` exists, the message header and body are split, and the *user*`.post` script is invoked.

As post-processing takes place, the message header is appended to the file *user*`..head`, the body is appended to *user*, and the `.post` script is executed as a different process. The script will find itself within the *maildir* directory, and will be able to access the global variables `RCPT` (after alias substitution) and `FROM`. The files *user* and *user*`.head` are removed after postprocessing is over.

4.8 Smtpc

The *smtpc* program is an RFC2821[2] compliant implementation of a *Simple Mail Transfer Protocol* client.

---

[2] Please refer to the '`README`' in '`Applications/smtpd`' for details

4.8.1 Smtpc Configuration

The client is configured by setting items in the *options* array. Please see Section 4.7.1 [Smtpd Configuration], page 30 for an explanation, but keep in mind that *smtpc* only uses `smtpc:options(hostname)` and `smtpc:options(domainname)` (note the smtpc prefix instead of smtpd).

4.8.2 Smtpc Features

smtpc needs at least two arguments: the first is the sender address and the second is the receiver address, both of the form `name@domain`. By default *smtpc* reads messages from *stdin* and doesn't add any header, so messages must be completely input via *stdin*.

You can also give two more parameters to *smtpc* in order to get it to work in a different way.

If a third argument is supplied, then it will be considered as the subject of the message. In such a case, *smtpc* will send its own headers to the server (*"From:"*, *"To:"* and *"Subject:"* fields), while the body will be read from *stdin*.

A fourth argument will be considered as the name of a file from which *smtpc* shall read the mail body (headers NOT included). If the third argument is set to a null string, then *smtpc* will read the whole mail message (headers INCLUDED) from the file the name of which is supplied as the fourth argument.

Concept Index

Command Index

Table of Contents