htt User's Guide

by Marcel Schoen, Switzerland

April 9, 2011

Contents

Chapter 1

htt Functionality Overview

- The htt provides a large variety of HTTP-related functionality, useful for implementing all kinds of HTTP-based tests:
- Advanced HTTP protocol handling, including fine-grained timeout handling, request and response validation
- Simulating clients and servers, including startup and shutdown of server daemons. This allows to create mock-ups of back-end systems in more complex test situations, for example when the tested application needs to interact with a 3rd-party back-end system which is not available in the testing environment.
- Execution of external commandline tools, using their output as request or response data, or for validation purposes.
- Copying stream data (e.g. from a response) and re-using it in variables.

Chapter 2

Quickstart

This short chapter shows some simple, basic htt examples without explaining everything. Some parts should be obvious for anyone who has a solid knowledge of HTTP, everything else will be explained in more detail in the other chapters.

NOTE: Depending on the version and platform you are using `httest` on, there may be a bug which requires that every script ends with an empty line, below the last statement or variable definition!

2.1 Step 1: GET www.wikipedia.org

This first example sends a simple GET request to "www.wikipedia.org" and checks that the page contains the word "Wikipedia", and the HTTP response code is 200.

Enter the following example code into a file named "wikipedia.htt":

```
CLIENT
_REQ www.wikipedia.org 80
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
END
```

Then execute the newly created script using the `httest` binary:

```
%> httest wikipedia.htt
```

And, given that the system you are running `httest` on has internet access, the following result should be displayed:

```
run wikipedia.htt
CLT0-0 start ...
_REQ www.wikipedia.org 80
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
>GET / HTTP/1.1
>Host: www.wikipedia.org
>
<HTTP/1.0 200 OK
<Date: Tue, 18 Jan 2011 12:38:51 GMT
...
...many more headers, and the HTML page...
...
<</body>
<</html> OK
```

Some explanations concerning the output:

- The first part (everything before the lines with the ">" prefix) shows the commands executed by the httest binary.
- All lines beginning with ">" show the data sent to the HTTP server.
- All lines beginning with "<" show the data received in the HTTP response.
- The "OK" at the end of the output signals a successful test completion.

2.2 Step 2: Externalize host and port values

Create a new file named "values.htb" in the same directory like the "wikipedia.htt" file. Insert these contents:

```
# Set host and port as variables
SET PORT=80
SET HOST=www.wikipedia.org
```

Then change the contents of the "wikipedia.htt" file like this:

```
INCLUDE values.htb

CLIENT
_REQ $HOST $PORT
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
END
```

Now you can use that include file to use the same host and port value in multiple htt scripts, while being able to maintain those values in only one file.

Chapter 3

htt Commandline Interface

3.1 Files

The htt tool is available for multiple platforms. On Unix platforms, the htt provides two binaries:

- `httest`: The actual HTTP testing tool.
- `htntlm`: A tool required to perform an NTLM authentication.

On Windows, a number of other files (.exe and .dll) are required to run the htt.

3.2 Syntax

The htt binary uses the following commandline syntax:

```
    %> httest [OPTIONS] scripts
```

Where *scripts* is just one or multiple ".htt"-files, for example:

```
    %> httest -Ts testone.htt testtwo.htt
```

File names must be specified either relative to the current working directory, or with an absolute path.

3.3 Options

A list of all options can always be retrieved by using the ”–help” argument:

```
%> httest -Ts testone.htt testtwo.htt

httest is a script based tool for testing and benchmarking web
applications, web servers, proxy servers and web browsers. httest
can emulate clients and servers in the same test script, very
useful for testing proxys.

Usage: httest [OPTIONS] scripts

Options:
  -V --version          Print version number and exit
  -h --help             Display usage information (this message)
  -n --suppress         do no print start and OK|FAILED
  -s --silent           silent mode
  -i --info             log level info
  -e --error            log level error
  -w --warn             log level warn
  -d --debug            log level debug
  -L --list-commands    List all available script commands
  -C --help-command     Print help for specific command
  -T --timestamp        Time stamp on every run
  -S --shell            Shell mode

Examples:
httest script.htt

httest -Ts script.htt

Report bugs to http://sourceforge.net/projects/htt
```

Chapter 4

htt Script Structure

A htt test usually consists of a ".htt" file (the main script), and optionally some include files (usually with the suffix ".htb").

The main script consists of *include statements*, global or local *commands*, *variables*, *bodies* and comments:

- *Command*: An instruction to perform an action, like sending a request, setting a header in that request, evaluating the response etc. There are global and local commands (explained in the chapter "Commands").
- *Include*: A global command used to include re-usable scripting code from other files.
- *Variables*: Holder for values created during the execution of the script.
- *Body*: A block of code containing a number of commands.
- *Comment*: A line of non-executable text

Commands, variables and body types are explained in more detail in the corresponding chapters.

4.0.1 Comments

Comment lines must begin with the "#" character.

4.1 Structure Example

A simple htt script code example for showing the structure:

```
# Include some macros
INCLUDE macros.htb

# The actual test client part
CLIENT
   ...
END

# Optional: a re-used code block
BLOCK DOIT
   ...
END

# Optional: A test monitoring daemon
DAEMON
   ...
END
```

In the example above, `INCLUDE` is a global statement used to include another script file with some re-usable code blocks. It is followed by the `CLIENT` block which starts the actual HTTP test client, a block of re-usable functionality named "DOIT" and a `DAEMON` body which contains some test monitoring functionality.

4.2 Control Flow

The following options are available to control the flow of a httest script:

- Conditions: The `_IF` command supports the execution of a code body depending on a condition.
- Loops: While the `_FOR` command allows to execute a code body for a given list of elements, the `_LOOP` command allows to do the same for a certain number of times.

Chapter 5

htt Variables

Within a htt script, variables can store values read from files or created during the execution of the script.

5.1 Variable Names

A variable has a name and a value. Variable names are NOT case-sensitive, so a variable named "VarA" can be used by refering to the name "VARA" as well.

```
CLIENT
# Set variable with value "First"
_SET VarA=First

# This will overwrite the value of the
# variable "VarA", since it is internally
# the same variable as "VARA".
_SET VARA=Second
END
```

5.2 Variable Scope

A variable is defined either as a global variable, or as a local within a body (like `CLIENT`, `SERVER` etc.). Note that unlike as in common programming languages, local variables created in a `BLOCK` are also available in the invoking bodies.

So, if a `CLIENT` body invokes a `BLOCK` where a local variable "A" is created, that variable will also become available in the invoking `CLIENT` body.

5.3 Setting A Variable

The syntax for setting a global variable is:

```
SET name=value
```

The syntax for setting a local variable in a body is:

```
_SET name=value
```

Examples:

```
SET MYVAR=SomeValue

CLIENT
_SET MYLOCAL=LocalValue
# Invoke block DOIT, makes variable "DoLocal"
# available in this CLIENT body as well.
_CALL DOIT
END

BLOCK DOIT
_SET DoLocal=anotherValue
END
```

5.4 Using Variables

In order to resolve a variable and use the value it is holding, its name must be prefixed with a "$" character, just like in a shell script. Example:

```
# Set global Variables for host and port
SET MyHost=www.acme.com
SET MyPort=8080

CLIENT
# Invoke named block DOIT which
# sets the variable for the URI path
_CALL DOIT

_REQ $MyHost $MyPort
__GET $UriPath HTTP/1.1
__Host: www.acme.com
__
_WAIT
END

BLOCK DOIT
# Set URI path variable which will also
# become available in the invoking body.
_SET UriPath=/webapp
END
```

Chapter 6

Body Types

The following global body types exist:

- `CLIENT` - The main HTTP test client logic.
- `SERVER` - Allows to start a HTTP server which will react and respond as defined in the given block. This allows to simulate HTTP back-end servers (like some application server) that may not be available in the testing environment.
- `BLOCK` - A block is a named section of the script with a defined begin and ending, not unlike functions or methods in common scripting or programming languages.
  - `BLOCK FINALLY` - A special block. The FINALLY block is always executed at the end of the test, and can be used for clean-up work.
  - `BLOCK ON_ERROR` - Another special block. A global error handler that will be executed when an error occurs in a test script. While some errors can be handled within a regular body (e.g. during the evaluation of a HTTP response), other errors (especially technical ones) sometimes can only be handled in this block.
- `DAEMON` - Other than a `SERVER`, a `DAEMON` does not implement any kind of listener. It is just a separate thread that can be used to monitor or control the execution of the test, e.g. implement timeout functionality to interrupt a hanging test etc.

The following local body types exist:

- `_BPS` - Limits the number of bytes sent during a period of time.
- `_ERROR` - Ensures that a certain expected error happened within the body.
- `_FOR` - Allows to define code with is executed for a given number of elements.
- `_IF` - Allows to define code with is executed only if a given condition is true.

- `_LOOP` - Allows to execute the given block of code a certain number of times.
- `_RPS` - Limits the number of requests sent during a period of time.
- `_SOCKET` - Spawns a socket reader.

6.1 Bodies And Threads

6.1.1 Parallel Execution

Note that all body blocks are executed in separate threads! So, if a htt script contains more than one `CLIENT` body, each one will be started in a separate thread, allowing to simulate two distinct, separate HTTP clients in one test case.

6.1.2 Execution Order

While multiple bodies are executed in parallel threads in general, `SERVER` and `DAEMON` bodies are always started before `CLIENT` bodies.

6.1.3 Lifespan

If a `CLIENT` body (or all of them, if there's more than one) ends, the test ends, and the htt process exits. The completion of `SERVER` and `DAEMON` bodies, on the other hand, will not automatically terminate the test.

6.1.4 Synchronization

There is no means of synchronization between those threads as of now.

6.2 CLIENT

A `CLIENT` body is the part of a htt script, where the actual HTTP requests are generated to perform some kind of test. Usually, there is just one single `CLIENT` body in a htt script, but there can be multiple ones; in that case, they are executed in parallel threads.

A htt test ends, when all `CLIENT` bodies complete their work.

A simple `CLIENT` example:

```
CLIENT
_REQ www.wikipedia.org 80
__GET / HTTP/1.1
__Host: www.wikipedia.org
__
_EXPECT . "200 OK"
_EXPECT . "Wikipedia"
_WAIT
END
```

6.3 SERVER

A SERVER body can start a HTTP listener and react to incoming HTTP requests with predefined responses. This allows to simulate application backend servers that may not be available in a test environment.

There can be multiple SERVER bodies; in that case, they are executed in parallel threads.

A htt test does not end when the SERVER bodies complete their work, only when the CLIENT bodies have finished.

A simple SERVER example:

```
# Simple HTTP listener which just responds with
# a HTTP 200 return code, and an empty body.
SERVER
_RES
_WAIT
__HTTP/1.1 200 OK
__Content-Length: AUTO
__Content-Type: text/html
--

END
```

6.4 DAEMON

A `DAEMON` body does not start a listener, and also not necessarily send any HTTP request. It's purpose is to perform monitoring tasks during a test. For instance, if a `CLIENT` body sends a HTTP request that may hang for a long time due to slow response time of the server, the `DAEMON` may enforce an exit of the htt test process after a certain time, essentially implementing a timeout function.

This is only one possibility. A `DAEMON` can also monitor log files written during the test, execute external binaries etc.

There can be multiple `DAEMON` bodies; in that case, they are executed in parallel threads.

A htt test does not end when the `DAEMON` bodies complete their work, only when the `CLIENT` bodies have finished.

A simple `DAEMON` example:

```
# Timeout daemon thread, which exits the
# test process with a failure return code,
# if it takes longer than 30 seconds.
DAEMON
_SLEEP 30000
_DEBUG Test duration too long
_EXIT FAILED
END
```

6.5 BLOCK

A BLOCK body is not executed by itself like CLIENT or SERVER bodies; it just serves as a holder for a fragment of re-usable script code, similar to a function or method in common programming languages.

Every BLOCK must have a name, in order to distinguish them from each other, and to be able to be invoked from another body.

Just like high-level programming language functions or methods, a BLOCK can be parameterized, in a way very similar to shell scripts. The name of the BLOCK, mentioned in the previous paragraph, is just the first parameter (and it's mandatory). Within the BLOCK body, all parameter values are available as variables with simple numeric names, like in a shell script:

- $0 - The name of the BLOCK
- $1 - The first parameter after the name
- $2 - The second parameter after the name
- $3 - The third parameter, and so on

A simple BLOCK example, without parameters:

```
BLOCK SENDREQ
_REQ www.acme.com 80
__GET / HTTP/1.1
__Host: www.acme.com
__
_WAIT
END
```

The following example shows a BLOCK with some parameters:

```
# Block takes 3 parameters:
# 1: hostname, like www.acme.com
# 2: port
# 3: requested URI path
BLOCK SENDREQ
_REQ $1 $2
__GET $3 HTTP/1.1
__Host: $1
__
_WAIT
END
```

Chapter 7

htt How To's

7.1 File Upload

Performing a file upload with httest can be a bit tricky. Here's an example of how to upload a file, where the actual data to be posted is read from a local file `./data.xml`:

```
_REQ $HOST $PORT
__POST /myapp HTTP/1.1
__Host: $HOST:$PORT
__Connection: keep-alive
__Content-Type: multipart/form-data; boundary=AaB03x
__Content-Length: AUTO
__
__--AaB03x
__Content-Disposition: form-data; name="xy"; filename="data.xml"
__Content-Type: text/xml
__
_PIPE
_EXEC cat ./data.xml
__
__--AaB03x
__Content-Disposition: form-data; name="file_upload"
__
__Upload
_--AaB03x--
_EXPECT headers "HTTP/1.1 200"
_WAIT
```

Note the additional empty line after the `_EXEC` call; this may be necessary depending on the content of the posted file data.

Also, the `Content-Type` header value must match the type of data being posted, of course.

Using a tool like the Firefox add-on "HTTP Live-Headers" can be very helpful to get the correct request data for posting a given file!

Chapter 8

htt Commands

Commands are the actual instructions that make up the test script, such as sending a request to a HTTP server, evaluating the response etc. There are two different types of commands:

- *Global* commands always begin with a capital letter (e.g. "INCLUDE") and can be placed in the "root" of the script, outside of any *body* (such as a block).
- *Local* commands always begin with an underscore (e.g. "_GET") and can only be placed within a *body*.

Some commands take arguments to perform their operation. Usually, the syntax is

```
    COMMAND [argument] [argument] [argument] ...
```

There is a special exception, the double-underscore local command (see section "Local Commands" for details) which does not take a blank character between the command and its argument.

8.1 Global Commands

Global commands are used to perform initialization work, clean-up steps, start and end bodies etc.

Global commands can only be used within the "root" part of the script, i.e. outside of a body. It is not possible to use a global command within a `CLIENT`, `SERVER`, `DAEMON` or `BLOCK` body.

Note that some commands can are available in two forms, both as global and local command.

The following global commands perform an operation by themselves:

- `AUTO_CLOSE` -
- `EXEC` -
- `FILE` -
- `GO` -
- `INCLUDE` - Includes content from a file.
- `PROCESS` - Runs the test in multiple processes.
- `SET` - Sets a global variable.
- `TIMEOUT` -

While the global commands listed below are used to start or end a body (a block of script code):

- `CLIENT` - Starts a `CLIENT` body.
- `SERVER` - Starts a `SERVER` body.
- `DAEMON` - Starts a `DAEMON` body.
- `END` - Ends the current body.

All these commands are explained in more detail in 9.

8.2 Local Commands

Local commands always begin with an underscore ("_") character. There are a large number of local commands, and they are explained in more detail in 10.

8.2.1 Double Underscore Command

Please note that the double-underscore is also a local command! But contrary to all other commands, with the double-underscore there's no blank character between the command and it's parameters.

The reason for this somewhat peculiar design is that this command is used to insert a line of data into the output stream of a HTTP request. And a line always ends with two invisible characters for the line-break and line-feed. Now, in some cases it is necessary to calculate the exact length of a line in order to know the correct length of the HTTP request body; in that case, the length of a line displayed by a text editor can be used directly as the length of the actual line of HTTP data for body length calculations.

Chapter 9

List Of Global Commands

This chapter contains a comprehensive list of all the global commands available in httest scripts.

9.1 AUTO_CLOSE

Syntax:

```
AUTO_CLOSE on|off
```

Purpose:

Closes the current connection automatically.

9.2 BLOCK

Syntax:

```
BLOCK <name>
```

Purpose:

Begins a body with scripting code that can be invoked from anywhere else, using the `CALL` or `_CALL` command.

The `BLOCK` must be closed using the `END` command.

Error Handler

A special, reserved name for a `BLOCK` is `ON_ERROR`. It designates the block to be invoked if an unexpected error occurs during a test. The block can then handle the error, output some information etc. Example:

```
    BLOCK ON_ERROR
    _DEBUG
    _DEBUG !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    _DEBUG An error occurred, please fix it and try again!
    _DEBUG !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    _DEBUG
    _EXIT FAILED
    END
```

9.3 CLIENT

Syntax:

```
CLIENT [<number of concurrent clients>]
```

Purpose:

Begin of a body which starts one or multiple threads running a test client for sending HTTP requests to a server, and evaluating the response.

The number of threads started depends on the value of the argument for the number of concurrent clients. If none is given, it defaults to 1 and only a single thread will be started. Otherwise, the same `CLIENT` body will be executed in the specified number of parallel threads.

The `CLIENT` must be closed using the `END` command.

9.4 DAEMON

Syntax:

```
DAEMON
```

Purpose:

Begins a body with scripting code that is executed in a separate thread, and can be used to perform tasks like monitoring the tests, enforcing timeouts etc.

The `DAEMON` must be closed using the `END` command.

9.5 END

Syntax:

```
END
```

Purpose:

Ends the current body (either a `CLIENT`, `SERVER`, `DAEMON` or `BLOCK` body).

9.6 EXEC

Syntax:

```
EXEC <shell command>
```

Purpose:

Executes the given command, e.g. an external shell script. Note: The execution will not join any of the CLIENT or SERVER threads. In other words, the command will execute and complete independently of any such threads.

9.7 FILE

Syntax:

```
FILE <name>
```

Purpose:

Creates a temporary file with the given name.

9.8 GO

Syntax:

```
GO
```

Purpose:

Starts all `CLIENT`, `SERVER` and `DAEMON` bodies that have been defined up to this point.

9.9 INCLUDE

Syntax:

```
INCLUDE <include file>
```

Purpose:

Includes the given file into the current script code. The file name parameter is processed relative to the working directory of the current process, if it is not an absolute path.

9.10 MODULE

Syntax:

```
MODULE <name>
```

Purpose:

A module represents a collection of `BLOCKs`. It adds a namespace element that can help organize re-usable blocks by adding the module name as a prefix for each block.

So, every block contained in a module must be invoked by adding the module name as a prefix.

```
# Begin a module "utility"
MODULE utility

BLOCK one
...
END
```

Invoke the block ”one” from this example then like that:

```
_CALL utility:one
```

9.11 PROCESS

Syntax:

```
PROCESS <n>
```

Purpose:

Runs the script in <n> parallel processes simultanously.

9.12 SERVER

Syntax:

```
SERVER [<SSL>:]<addr_port> [<number of concurrent servers>]
```

Purpose:

Begin of a body which starts one or multiple threads running a test server for responding to HTTP requests in predefined ways.

The number of threads started depends on the value of the argument for the number of concurrent servers. If none is given, it defaults to 1 and only a single thread will be started. Otherwise, the same `SERVER` body will be executed in the specified number of parallel threads.

The `SERVER` must be closed using the `END` command.

9.13 SET

Syntax:

```
SET <variable>=<value>
```

Purpose:

Sets a global variable. The variable can then be used in the script with a ”$”-sign prefix.

Example:

```
SET MyHost=www.acme.com
...
_REQ $MyHost 80
```

9.14 TIMEOUT

Syntax:

```
TIMEOUT <timeout in ms>
```

Purpose:

Sets the global TCP socket connection timeout.

Chapter 10

List Of Local Commands

This chapter contains a comprehensive list of all the local commands available in httest scripts.

10.1 __

Syntax:

```
__<string>
```

Purpose:

Sends the given string data to the current socket, with CRLF line termination characters.

NOTE: There MUST NOT BE a blank between the command and the following string data (or the blank itself will be part of the string data as well).

The reason for using this peculiar design lies in the history of the httest utility. In early versions, the value of the "content-length" HTTP header of a request had to be calculated and set manually. In that situation, it <n> was easier to calculate the actual number of bytes a line of data would take up, because the two underscores at the beginning of the line allowed to just use the number of columns value displayed by any text editor. The reason: Without these two characters, a text editor would display only the number of visible characters, but the CRLF characters had also to be taken into account for a correct "content-length" header value. So the double underscore was a replacement for the invisible line termination characters.

However, it's not necessary for this purpose anymore as httest allows to set the correct content length value automatically now.

10.2 _-

Syntax:

```
_-<string>
```

Purpose:

Sends the given string data to the current socket, without any CRLF line termination characters. This is mostly used for sending POST body data.

NOTE: There MUST NOT BE a blank between the command and the following string data (or the blank itself will be part of the string data as well).

10.3 _ADD_HEADER

Syntax:

```
_ADD_HEADER <header> <value>
```

Purpose:

Adds an additional header to received headers.

10.4 _AUTO_CLOSE

Syntax:

```
_AUTO_CLOSE on|off
```

Purpose:

"`_AUTO_CLOSE on`" lets `httest` automatically close a connection when an incoming "`Connection: close`" header is received.

10.5 _AUTO_COOKIE

Syntax:

```
_AUTO_COOKIE on|off
```

Purpose:

Automatically handles incoming cookies, so that they are sent with the next request on the same connection they were received on. Does not check cookie expiration or path.

10.6 _BPS

Syntax:

```
_BPS <number of bytes> <duration>
```

Purpose:

Defines a code body which limits the maximum number of bytes that may be sent during a certain period of time (specified in number of seconds).

The body must be closed with the `_END BPS` command.

The following example limits the number of bytes to be sent within the next 300 seconds to 50000:

```
    _BPS 50000 300
    _REQ localhost 8080
    __GET / HTTP/1.1
    __Host: localhost
    __
    _WAIT
    _END RPS
```

10.7 _B64DEC

Syntax:

```
_B64DEC "<string>" <variable>
```

Purpose:

Stores a Base64-decoded string into <variable>.

10.8 _B64ENC

Syntax:

```
_B64ENC "<string>" <variable>
```

Purpose:

Stores a Base64-encoded string into <variable>.

10.9 _BREAK

Syntax:

```
_BREAK
```

Purpose:

Breaks out of the current loop.

String example:

Break the loop if variable `$ABC` has been set to ”`Stop`”:

```
    _LOOP 500
    _CALL SomeBlock
    _IF $ABC MATCH "Stop"
    _BREAK
    _END IF
    _END LOOP
```

10.10 _CALL

Syntax:

```
_CALL <name of block>
```

Purpose:

Invokes the given `BLOCK`. See section 6.5 on page 31 for examples.

Namespaces:

Namespace prefixes can be added with the local command `_USE` (see section 10.67 on page 122 for details).

10.11 _CERT

Syntax:

```
_CERT <cert-file> <key-file> [<ca-cert-file>]
```

Purpose:

Configures the certificate to use for the current SSL connection. This is mainly necessary for server certificates (because an SSL server always must present a certificate). For clients, this is necessary only if mutual authentication is required.

The file formats are basically the same ones used to configure SSL in an Apache webserver (PEM certificate and key files).

10.12 _CHUNK

Syntax:

```
_CHUNK
```

Purpose:

This marks the end of a chunk data block. It counts the number of bytes sent since the last invocation of `_FLUSH` and automatically adds chunk info to the request.

10.13 _CLOSE

Syntax:

```
_CLOSE
```

Purpose:

Closes the current connection (and sets the connection state to `CLOSED`).

10.14 _DEBUG

Syntax:

```
_DEBUG <string>
```

Purpose:

Prints a text string to `stderr` for debugging purposes.

10.15 _DETACH

Syntax:

```
_DETACH
```

Purpose:

Daemonizes the current httest process, which can be useful if it is just a test server among others.

10.16 _ELSE

Syntax:

`_ELSE`

Purpose:

Defines a code body which is executed if the condition defined in the preceeding `_IF` command was not fulfilled. The body must then be closed by the `_END IF` command.

NOTE: This command can only be used together with, and following the `_IF` command. Example:

```
    _IF "foo" EQUAL "foo"
      _DO_SOMETHING
    _ELSE
      _DO_SOMETHING_OTHER
    _END IF
```

10.17 _ERROR

Syntax:

```
_ERROR <message>
```

Purpose:

Defines a body wherein a certain error is expected to happen. The body must be closed with the `_END ERROR` command.

In the following example, the error message ”Broken pipe” is expected to be received during one of the three requests:

```
    _ERROR "Broken pipe"
    _REQ localhost 8080
    __GET /foo HTTP/1.1
    __Host: localhost
    _WAIT

    _REQ localhost 8080
    __GET /bar HTTP/1.1
    __Host: localhost
    _WAIT

    _REQ localhost 8080
    __GET /and/another/one HTTP/1.1
    __Host: localhost
    _WAIT
    _END ERROR
```

10.18 _EXEC

Syntax:

```
_EXEC <shell command>
```

Purpose:

Executes a shell command. `_EXEC|` will pipe the current socket input stream into the invoked shell command.

Use `_PIPE` for streaming the output of the shell command into the current socket output stream (See section 10.33 on page 88).

10.19 _EXIT

Syntax:

```
_EXIT [OK|FAILED]
```

Purpose:

Exits with `OK` or `FAILED` (default is `FAILED`).

10.20 _EXPECT

Syntax:

_EXPECT `.|headers|body|error|exec|var() "|'[!]<regex>"|'`

Purpose:

Defines what data is expected to be received, either on the current connection, or the stdout of an executed program, or in a certain variable. A regular expression is used to define the expected data.

Use a preceeding exclamation mark ("!") before the regular expression to negate the check.

NOTE: This command does not perform any check yet. The actual check on the data is performed in the following `_WAIT` command.

Example 1: Expect a certain output from an executed binary:

```
    _EXPECT exec "myValue"
    _EXEC echo myValue
```

Example 2: Expect a certain value in variable $A:

```
    _SET A=myValue
    _EXPECT var(A) "myValue"
```

10.21 _FLUSH

Syntax:

```
_FLUSH
```

Purpose:

Flushes the cached lines of data that must be sent in a request.

NOTE: The calculation of the content length value for an `AUTO` content-length header is performed by this command!

10.22 _FOR

Syntax:

```
_FOR <variable-name> "|'<string>*"|'
```

Purpose:

Allows to define a body of script code (similar to a `BLOCK`) which is executed once for each of a given number of elements, similar to `for-each` constructs of some programming languages.

The elements are in a string, separated by blank characters (spaces). The `_FOR`-body must be closed with a corresponding `_END FOR` command.

```
    _FOR I "First Second Third"
      _DEBUG $I
    _END FOR
```

10.23 _GREP

Syntax:

```
_GREP (.|headers|body|error|exec|var()) "|'<regex>"|' <variable>
```

Purpose:

Defines a regular expression for a match which has two purposes:

1. If it matches, the matching data is stored in the <variable>.
2. If there is a match, the test fails (just the other way around than `_MATCH`).

10.24 _HEADER

Syntax:

```
_HEADER ALLOW|FILTER <header name>
```

Purpose:

Defines a filter for incoming request headers. The filter can either be a blacklist ("`_HEADER FILTER <headers>`") or a whitelist ("`_HEADER ALLOW <headers>`").

By default, there's no filtering; all headers are allowed.

10.25 _IF

Syntax:

`_IF ("<string>" [NOT] MATCH "regex")|"<string>" EQUAL "<string>"|("<number>" [NOT] EQ|LT|GT|LE|GT "<number>)"`

Purpose:

Allows to define a body of script code (similar to a `BLOCK`) which is executed only if the given condition is true. The condition is based on either a given string which must match a regular expression or another string, or on a numerical value to be equal, less than or greater than a certain number.

The `_IF`-body must be closed with a corresponding `_END IF` command.

String example:

Tests if the given <string> matches the regular expression (that string can be a variable, of course):

```
    _IF $MyVar MATCH ".*xyz.*"
      _CALL SomeBlock
    _END IF
```

Number example:

Tests if the given <string> containing a numerical value is larger than a given threshold value:

```
    _IF $MyNumber GT "1500"
    _CALL SomeBlock
    _END IF
```

10.26 _IGNORE_BODY

Syntax:

```
_IGNORE_BODY on|off
```

Purpose:

Recieves the request body data, but does not store it anywhere or inspect it. This is used for performance testing.

10.27 _LOG_LEVEL

Syntax:

```
_LOG_LEVEL <level>
```

Purpose:

Sets the log level. <level> must be a number from 0 - 4.

10.28 _LOOP

Syntax:

`_LOOP <n>`

Purpose:

Allows to define a body of script code (similar to a `BLOCK`) which is executed <n> times. The `_LOOP`-body must be closed with a corresponding `_END LOOP` command.

String example:

Calls the block "SomeBlock" 15 times:

```
    _LOOP 15
    _CALL SomeBlock
    _END LOOP
```

10.29 _MATCH

Syntax:

```
_MATCH (.|headers|body|error|exec|var()) "|'<regex>"|' <variable>
```

Purpose:

Defines a regular expression for a match which has two purposes:

1. If it matches, the matching data is stored in the <variable>.
2. If there is no match, the test fails (just the other way around than `_GREP`)..

10.30 _ONLY_PRINTABLE

Syntax:

```
_ONLY_PRINTABLE on|off
```

Purpose:

Replaces all characters of ASCII code lower than 32 and higher than 127 with a blank (space) character.

10.31 _OP

Syntax:

```
_OP <left> ADD|SUB|DIV|MUL <right> <variable>
```

Purpose:

Performs a calculation on the two values <left> and <right>, with the given operand:

- `ADD`: Addition
- `SUB`: Subtraction
- `DIV`: Division
- `MUL`: Multiplication

The result is stored in <variable>.

10.32 _PID

Syntax:

```
_PID <variable>
```

Purpose:

Stores the current PID into <variable>.

10.33 _PIPE

Syntax:

```
_PIPE [chunked [<chunk_size>]]
```

Purpose:

Starts a pipe for streaming the output of `_EXEC` into the socket output stream, with optional chunk support.

10.34 _PLAY

Syntax:

```
_PLAY SOCKET | VAR <variable>
```

Purpose:

Plays back recorded data, either on the current socket or into a variable.

10.35 _PRINT_HEX

Syntax:

```
_PRINT_HEX on|off
```

Purpose:

Output data is displayed as hex digits.

10.36 _RAND

Syntax:

```
_RAND <start> <end> <variable>
```

Purpose:

Generates a random number between <start> and <end> and stores the result in <variable>.

10.37 _RECORD

Syntax:

```
_RECORD RES [ALL] STATUS | HEADERS | BODY*
```

Purpose:

Records a response for later replay, or stores it.

10.38 _RECV

Syntax:

```
_RECV <bytes>|POLL|CHUNKED|CLOSE [DO_NOT_CHECK]
```

Purpose:

Receives an amount of bytes, either specified by the number of bytes to read, or until a socket timeout will occur in `POLL` mode.

Optional: `DO_NOT_CHECK` does not check the `_MATCH` and `_EXPECT` clauses.

With `_CHECK` this can be done later, over a couple of not yet checked `_RECVs`.

10.39 _RENEG

Syntax:

```
_RENEG [verify]
```

Purpose:

Starts an SSL renegotiation.

10.40 _REQ

Syntax:

```
_REQ <host> [<SSL>:]<port>[:<tag>] [<cert-file> <key-file> <ca-cert-file>]
```

Purpose:

Opens a new TCP connection to the given host and port. If the connection exists already, no new connection will be created.

Parameters:

`SSL`: Defines the type of SSL connection. Must be one of these values: `SSL`, `SSL2`, `SSL3` or `TLS1`.

`host`: The host name or IPv4 / IPv6 address of the target system. Note: An IPv6 address must be enclosed in square brackets!

`tag`: Setting a tag on a connection allows to open multiple connections to the same target. If a connection with the same tag already exists, that connection will be selected (for the following request).

`cert-file`, *`key-file`*, *`ca-cert-file`*: Optional; required for SSL connections.

10.41 _RES

Syntax:

```
_RES
```

Purpose:

Waits for a connection accept. This is used within a `SERVER` body to receive incoming requests from clients.

10.42 _RESWAIT

Syntax:

```
_RESWAIT
```

Purpose:

Combines the `_RES` and the `_WAIT` command, ignoring connections that are not sending any data at all.

10.43 _RPS

Syntax:

```
_RPS <number of requests> <duration>
```

Purpose:

Defines a code body which limits the maximum number of requests that may be sent during a certain period of time (specified in number of seconds). The requests are counted during the invocation of the `_WAIT` command.

The body must be closed with the `_END RPS` command.

The following example limits the number of requests to be sent within the next 300 seconds to 20:

```
    _RPS 20 300
    _REQ localhost 8080
    __GET / HTTP/1.1
    __Host: localhost
    __
    _WAIT
    _END RPS
```

10.44 _SEQUENCE

Syntax:

```
_SEQUENCE <variable-sequence>
```

Purpose:

Defines a sequence of `_MATCH` or `_GREP` variables which must appear in the given order in the response.

With `_MATCH` or `_GREP`, it is possible to define certain values that are expected in a response, one at a time. But the order of those values within the response is not determined by those commands. In the following example, the `_MATCH` commands make sure that the response contains three strings "one", "two" and "three". And the `_SEQUENCE` command then makes sure that they are in exact this order within the response:

```
CLIENT
_REQ localhost 8080
__GET /index.html HTTP/1.1
__Host: localhost
__
_MATCH body "(one)" first
_MATCH body "(two)" second
_MATCH body "(three)" third
_SEQUENCE first second third
_WAIT
END
```

10.45 _SET

Syntax:

```
_SET <variable>=<value>
```

Purpose:

Allows to set a variable. See chapter 5 on page 21 for usage examples.

10.46 _SH

Syntax:

```
_SH shell script line or END
```

Purpose:

Embeds a shell script within a tmp file, and executes it when `END` is found.

10.47 _SLEEP

Syntax:

```
_SLEEP <milliseconds>
```

Purpose:

Stops execution for the defined amount of time (number of milliseconds).

10.48 _SOCKET

Syntax:

```
_SOCKET
```

Purpose:

Spawns a socket reader over the next `_WAIT` and `_RECV` commands.

close body with `_END SOCKET`.

10.49 _SOCKSTATE

Syntax:

```
_SOCKSTATE <variable>
```

Purpose:

Stores connection state `CLOSED` or `CONNECTED` of the current connection in the variable.

10.50 _SSL_BUF_2_CERT

Syntax:

```
_SSL_BUF_2_CERT pem cert
```

Purpose:

Reads the given buf as a PEM certificate, usable with `_SSL_CERT_VAL`.

10.51 _SSL_CERT_VAL

Syntax:

```
_SSL_CERT_VAL <cert entry> <variable>
```

Purpose:

Extracts a <cert entry> from a certificate which was received with `_RENEG` or `_VERIFY_PEER`, and stores it in <variable>.

Valid values for <cert-entry> are:

- `M_VERSION`
- `M_SERIAL`
- `V_START`
- `V_END`
- `V_REMAIN`
- `S_DN`
- `S_DN <var>`
- `I_DN`
- `I_DN <var>`
- `A_SIG`
- `A_KEY`
- `CERT`

10.52 _SSL_CONNECT

Syntax:

```
_SSL_CONNECT SSL|SSL2|SSL3|TLS1 [<cert-file> <key-file>]
```

Purpose:

Performs an SSL connect on an already existing TCP socket.

10.53 _SSL_ENGINE

Syntax:

```
_SSL_ENGINE <engine>
```

Purpose:

Sets an OpenSSL engine, to run tests with hardware crypto devices.

10.54 _SSL_GET_SESSION

Syntax:

```
_SSL_GET_SESSION <variable>
```

Purpose:

Stores the SSL session of this connection in a variable as a Base64 encoded string.

10.55 _SSL_LEGACY

Syntax:

```
_SSL_LEGACY on|off
```

Purpose:

Turns SSL legacy behaviour on or off for renegotiation (for OpenSSL libraries 0.9.81 and above).

10.56 _SSL_SECURE_RENEG_SUPPORTED

Syntax:

```
_SSL_SECURE_RENEG_SUPPORTED
```

Purpose:

Tests if the remote peer supports secure SSL renegotiation.

10.57 _SSL_SESSION_ID

Syntax:

```
_SSL_SESSION_ID <variable>
```

Purpose:

Stores the session ID of the current SSL socket in <variable>, Base64 encoded.

10.58 _SSL_SET_SESSION

Syntax:

```
_SSL_SET_SESSION <variable>
```

Purpose:

Sets the SSL session for the current connection from <variable> which must contain a Base64 encoded SSL session (stored with `_SSL_GET_SESSION`).

10.59 _STRFTIME

Syntax:

```
_STRFTIME <time> "<format-string>" <variable> [Local|GMT]
```

Purpose:

Stores the given time (in milliseconds) as a string in <variable>. The string is formatted as defined in <format-string>.

10.60 _SYNC

Syntax:

```
_SYNC
```

Purpose:

Waits until the next full second is reached. This can be used to synchronize code that is executed in separate threads, such as multiple `CLIENT`s or a `CLIENT` and a `SERVER`.

10.61 _TIME

Syntax:

```
_TIME <variable>
```

Purpose:

Stores the current time in milliseconds in <variable>.

10.62 _TIMEOUT

Syntax:

```
_TIMEOUT <milliseconds>
```

Purpose:

Sets an idle timeout for the current socket in number of milliseconds.

10.63 _TIMER

Syntax:

```
_TIMER GET|RESET <variable>
```

Purpose:

Stores the time duration since the last reset, or since the start of the test.

Start measuring time:

Starts time measuring, using the variable `$HOLDER` to store the result:

```
    _TIMER RESET $HOLDER
```

Measure time:

This makes the time available as a text string in the variable `$HOLDER`:

```
    _TIMER GET $HOLDER
```

10.64 _TUNNEL

Syntax:

```
_TUNNEL <host> [<SSL>:]<port>[:<tag>] [<cert-file> <key-file> [<ca-cert-file>]]
```

Purpose:

Opens a tunnel to the defined host and port, with SSL support. If a connection to that host and port already exists, it will be re-used and no connect will be performed.

Parameters:

Valid parameter values are:

- `SSL`: `SSL`, `SSL2`, `SSL3`, `TLS1`
- `tag`: The tag of the connection, if multiple connections to the same target host and port are used.

<cert-file>, <key-file> and <ca-cert-file> are optional, for client/server authentication.

10.65 _URLDEC

Syntax:

```
_URLDEC "<string>" <variable>
```

Purpose:

Stores a URL-decoded string into <variable>.

10.66 _URLENC

Syntax:

```
_URLENC "<string>" <variable>
```

Purpose:

Stores a URL-encoded string into <variable>.

10.67 _USE

Syntax:

```
_USE <module>
```

Purpose:

Allows to invoke blocks from the given module without having to specify the module prefix. So, it's quite similar to a static import in a Java source file.

For example, assume a htt file that contains a module named "UTILITY" like this:

```
MODULE UTILITY

BLOCK ONE
...
END

BLOCK _TWO
...
END
```

Usually, to invoke the block "ONE", the code in the actual test script would look like this:

```
_CALL UTILITY:ONE
```

Instead, with `USE` the call gets simpler:

```
_USE UTILITY
...
_CALL ONE
```

Or, in case of the second block named "_TWO", it can be even simpler (because blocks whose name starts with an underscore don't need to be invoked with `CALL`):

```
    _USE UTILITY
    ...
    _TWO
```

So, in a sense, by using modules and naming the blocks with underscores, it is possible to really create custom htt local commands.

10.67.1 Overriding commands

Using this mechanism, it's possible to override the existing htt commands (by creating a custom `_WAIT` command in a module, for example). However, overriding isn't really a tested feature, so results can be unpredictable.

For this reason, it's not recommended to override existing, real htt commands!

10.68 _WAIT

Syntax:

```
_WAIT [<amount of bytes]
```

Purpose:

Waits for data on a connection and reads it. Without this command, the script would just continue after sending out data, without reading the response.

`_EXPECT` and `_MATCH` commands will be applied to the incoming data at this point. This is why they must always preceed the `_WAIT` command.

Optionally, it's also possible to just read a given amount of bytes.

10.69 _WHICH

Syntax:

```
_WHICH <variable>
```

Purpose:

Stores the concurrency number of the current thread in <variable>.