IJESRT
INTERNATIONAL JOURNAL OF ENGINEERING SCIENCES & RESEARCH TECHNOLOGY

CONFIGURATION SOFTWARE FOR WIRELESS COMMUNICATION DEVICES

Robert DEMETER[*]
[*] Electric Engineering and Computer. Science, University Transilvania of Brasov, Brasov, Romania

DOI: 10.5281/zenodo.221007

ABSTRACT

The author will describe an implementation of the web based configuration software concept for Wireless communication devices that uses a custom CGI, developed in C++ programming language, on an embedded Linux operating system.
The proposed solution has service oriented architecture, minimizing the memory usage and applying the settings on the fly. The web interface show the current settings, obtained from common Linux commands, writes the new configuration in service's configuration file and restart the service without reboot the device.
The old web based configuration interface can be easily updated or extended, by modifying the design of pages or adding new features, without being necessary to recompile or do further development in the code of the CGI module that fills the interface data, by the use of simple description in a self-designed file type that will be described in this paper.

KEYWORDS: Computer Networks, Wireless Communication, Embedded Systems

INTRODUCTION

Due to the fact that Linux is a modular system and compilers for a large variety of microprocessor families can be found (x86, ARM, MIPS, PowerPC), all the applications and drivers were developed under Linux. Among the most important ones can remind: *busybox, boa, inetd, telnetd, wireless-tools* and *dhcpd*. The Linux kernel was patched with *Wireless Extensions* [1]. For the configuration of the wireless cards, Wireless Tools were used, with the most important tools: *iwconfig, iwspy, iwlist* and *iwpriv*. The applications, drivers and libraries are all compressed in a gzip file and together with the kernel they make up the firmware of this product, which is stored in the flash memory and loaded by bootloader. The firmware update can be accomplished by tftp or web interface. To avoid damage of the AP, a special partition of flash memory was used for backup. If the bootloader detect checksum or version error, the backup kernel image will be started.
To obtain the desired set of functionalities an AP must first be configured. Because there are many parameters that can be set and because common user should not access to the services configuration files, most of the time the devices configuration is realized by the use of the WEB interfaces using CGI (Common Gateway Interface) technology.
The simplest means of building a CGI is using the *Libcgi* library developed for the C language, which offers a wide set of functions for constructing HTML elements, refreshing HTML pages and retrieving the data from HTML forms. But there are memory allocations, conversions in each CGI script, which can lead to run-time errors.
Another solution is the CGILua developed by TeCGraf [2], based on Lua, a configuration language with a special interpreter meant for the execution of code chunks written in this language and a programming interface that assures the communication with the host program. Lua complies with the following requests: clear and easy syntax, small size, important features for the description of the data and extensions. The implementation of this language is available on platforms which have compilers like gcc, Visual C++, CodeWarrior etc.
In [7] the author explores the topic of an efficient and lightweight embedded Web server for Web-based network element management, and presents the architecture of an embedded Web server that can provide a simple but powerful API. Web-based network element management gives an administrator the ability to configure and monitor network devices over the Internet using a Web browser. The most direct way to accomplish this is to embed a Web server into a network device, and use that server to provide a Web-based

management user interface constructed using HTML, graphics and other features common to Web browsers.
A preliminary study over other similar products developed by companies like CISCO, Linksys [3, 4] strengthens the idea that the configuration solution by web is the most appropriate in the case of such devices.

THE WIRELESS ACCESS POINT CONFIGURATION

The software solution developed for the AP configuration is based on a web interface. The web interfaces are very easy to build graphically, very accessible for a common user, friendly and easy to handle without the need of any special requirements of the communication device.
The technology adopted for the development of the web interface is the well-known CGI and was developed under Linux using the C++ programming language.
When choosing this solution several things were kept in mind that lead one by one to disregard well known solution and taking the decision of implementing a custom solution. The first and the most important factor was the necessity on developing on Linux operating system, because it is an open source system, due to which his embedding on the hardware platform would not have raised the production costs. Another reason would be that Linux, is a modular operating system and constrains of physical storing space and memory space can be efficiently accomplished in this manner. Due to the same constraints of storage space and low memory available at run-time solutions like PHP and Perl could not be considered. In this way the advantages offered by a custom solution became obvious:

- possibility of developing it on embedded Linux;
- low storage space and memory requirements, taking into consideration both what was necessary for the web server that has to run on this device and for the module;
- developing only features that were necessary, decreasing the amount of storage and memory used;
- modify the service's configuration files, so no additional configuration files are requires.

Also the solution of embedding the HTML code in the application was considered, but the solution did not prove to be a good one because any modification, due to several requests for customization from the client side, would have made it very heavy to deal with.
For the reasons presented earlier a custom solution was chosen, one that involves the creation of a Script Engine (SE) for interpreting code that can be generated by any HTML/text editor. Inside these scripts, keywords and Linux commands were introduced. These can be composed by a series of rules presented further in this paper.

The SE parses special files having *.ews* extensions and works as an intermediary between the Linux operating system and the user. A short design of the solution is presented in Fig.1.

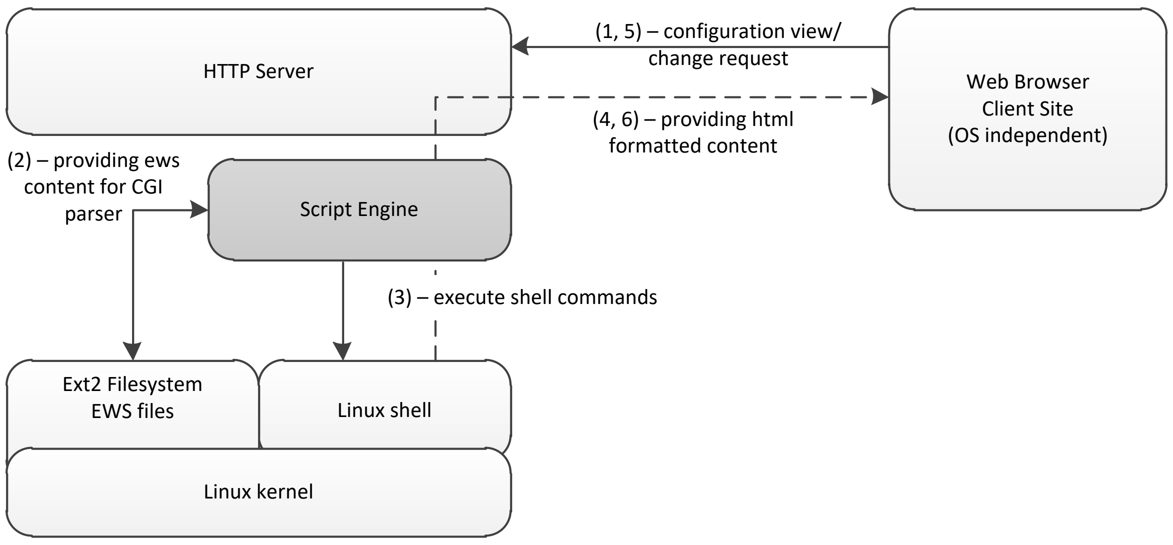
HTTP Server
(1, 5) – configuration view/ change request
Web Browser Client Site (OS independent)
(4, 6) – providing html formatted content
(2) – providing ews content for CGI parser
Script Engine
(3) – execute shell commands
Ext2 Filesystem EWS files
Linux shell
Linux kernel

Fig1. The software solution for the configuration management of AP

As described before, the script engine module acts as an intermediary. The user does not need to have knowledge about operating and configuring a Linux services. He can use this device through the help of a friendly graphical web interface. The operations flow as follows:

1. web browser sends request for the desired configuration page, which can configure an interface, application or service;
2. the HTTP server takes from the filesystem the specified *ews* file that then sends to the script engine module;
3. the script engine module analyzes the content of the ews file and executes the necessary Linux commands from the header section of *ews* file to supply the correct data for the user interface. In order to execute a command the SE module has to directly interact with the operating system;
4. the data obtained in this manner is placed in HTML format and communicated to the browser found at the client side;
5. the user makes the desired changes for the configuration and they are sent back to the web server that calls the CGI module for the execution of the necessary commands from the footer section of *ews* file;
6. after the SE module accomplishes all those changes through the interaction with the operating system it sends back to the user the new content with the modified data.

The UML architecture of the script engine is presented in Fig. 2 in a minimal form.

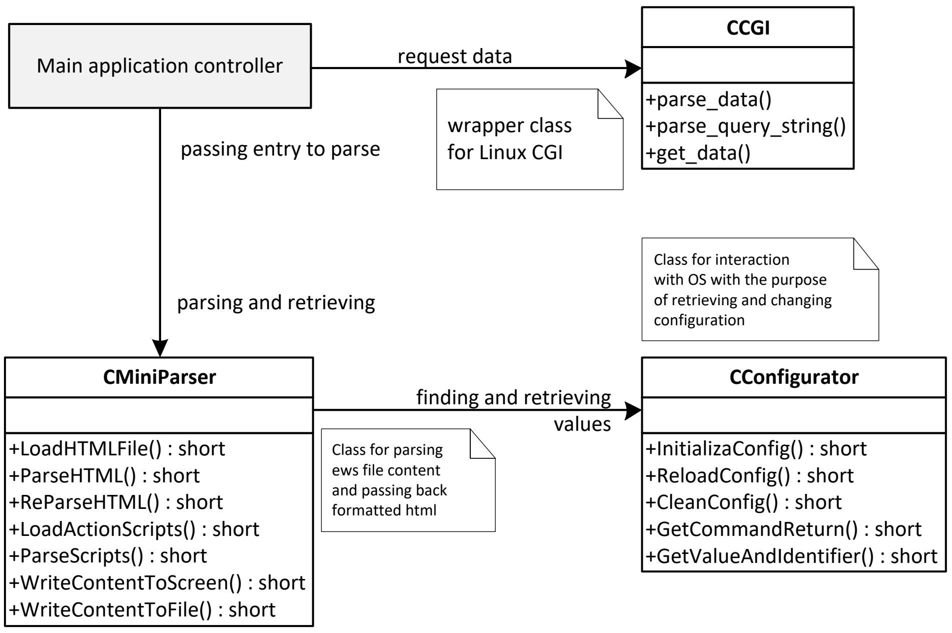
Main application controller
request data
CCGI
+parse_data()
+parse_query_string()
+get_data()
wrapper class for Linux CGI
passing entry to parse
Class for interaction with OS with the purpose of retrieving and changing configuration
parsing and retrieving
CMiniParser
+LoadHTMLFile() : short
+ParseHTML() : short
+ReParseHTML() : short
+LoadActionScripts() : short
+ParseScripts() : short
+WriteContentToScreen() : short
+WriteContentToFile() : short
finding and retrieving values
Class for parsing ews file content and passing back formatted html
CConfigurator
+InitializaConfig() : short
+ReloadConfig() : short
+CleanConfig() : short
+GetCommandReturn() : short
+GetValueAndIdentifier() : short

Fig. 2. UML architecture of the script engine

The *Main application* coordinates the entire activity of the script engine. It interacts initially with the CGI wrapper class to gather information about the environmental variables. This class exposes three main methods *parse_data(), parse_query_string()* and *get_data()*. The first two parse the data obtained from the web server considering the transmission type that was used. There are two known types for passing the data: directly through the method PUT, or by sending them inside the query string, method known as GET. Using the method *get_data()* the main application obtains the necessary information from this class, like the *ews* file name that must be parsed and exposed, the command that the user gave, or if the reload of the page was requested by such a command. The class *CMiniParser* handles the parsing of the EWS file, the most important methods are the ones stated in the UML structure. The main application makes a request to this class through his object in order for the file to be loaded (*LoadHTMLFile*) afterwards a call for the parsing function is made, simple parse or re-parse if the page is to be reloaded. In a similar way are treated the function that connect to the parsing of the action script found in the footer of the EWS files, details will be presented further on in the paper.

The class *CConfigurator* is the class that realizes the execution of the commands on the operating system. The commands are of two types, commands that interrogate the system about certain values, this commands are called to initial fill the HTML content with data, and the second type of commands are the ones used to change the system configuration, this commands are executed after the HTML content data has been changed to reflect the changes in the system.

THE STRUCTURE OF THE EWS FILES

The web interface is based on special files that will be described below, called ews files. The ews file has the extension .ews and in its essence is an HTML file which header and footer are formatted in a specific way.
The information in these two parts is organized in such manner to be used in the internal processing to obtain and alter the data from the HTML content.
The script engine parses the header of the EWS file before it is sent back to the web server and contains all the commands that have to be performed by the module to fill in with valid date the HTML content. The structure of the header is presented below:

```
#command=[command_name] [comand_type=[type_ specifier]]
#commparam=[parameters_name]=[default_value],
          [parameters_name1]=[default_value1],...

#parameters=[parameters_list_separated_by_comma]
#mparameters=[parameters_list_separated_by_comma]
```

where:
#command is a keyword that designates the beginning of a new command for the parser. The command starts with this keyword and ends at the point where another "#command" keyword is found or it reached the end of the header part - specified by the tag [HTML]
[command_name] – is a Linux shell command that will be executed (e.g. ifconfig eth0, iwconfig wlan0). The command name should contain the complete path from where the command can be executed.
[comand_type=[type_specifier]] – specifies the parsing methods. There are three command types at the moment with different parsing methods:
i) leave blank the field - designates a normal parsing - the command is executed and the parser looks for the parameters requested bellow in the returned output;
ii) **#list=[type_specifier]** - the parser expects the output of the command executed to be formatted as a list (the information is organized in lines – each line contains a given set of parameters - with specific value for the beginning of the line). (e.g. *iwspy wlan0* command);
iii) **#table=[type_specifier]-** the parser expects the output of the command executed to be formatted as a table (the information is organized in columns - each column represents a certain parameter).
[type_specifier]- the type specifier are only valid for the last two command types and represent in this two cases an information expressed by the form: nBeginLines:nEndLines. nBeginLines - represents the number of lines at the beginning of the command that do not contain valid information, and nEndLines – it is similar to the above specified just that it refers to the lines at the end of a command.

#commparam - marks the beginning of the line where the list of parameters can be set with their default value;

#parameter - marks the beginning of the line where the parameters can be set to extract from a given command and replace in the actual HTML body of the document. When the type of the command is #list or #table the parameters will be appended an index for each new line in which they appear in the command. If a parameter value cannot be extracted from the command it will be replaced with blank.

#mparameters - marks the beginning of the line where the multiple parameters can be set to extract from a given command and replace in the actual html body of the document. A multiple parameter is a parameter that can have multiple possible values that are part of an enumeration. It is used to set a selection or a check for its real value inside of an enumeration of possible values. The parser will look for the value on each line - if the value is the real one - the one extracted from the command will replace the multiple parameters "[$requested_value]" with the keyword "SELECTED". The rest will replace with blank.
As in the case of simple parameters - the multiple parameters for a command of type #list or #table will be appended an index for each new line in which they appear in within the command.

The parser expects to find the parameters listed by their name in the parameters list, in the html body or in any other place where they should be replaced (e.g. command syntax, footer body) with the following syntax:
`[%parmeter_name] [$mparameter_name]`

The footer of the EWS file contains a script that will be executed when one of the possible actions is triggered. The possible actions are: Save, Add, Delete or Replace. The footer has the following structure:

```
[SS]
#open=[file_name]
#save=[command] [#cond=[condition]]
#add=[command] [#cond=[condition]]
#delete=[command] [#cond=[condition]]
#replace=[replaced_string][=][replaced_value][#cond=[condition]]
[ENDSS]
```

where:
[SS], [ENDSS] – are tags - specify the beginning and the end of the footer section.

#open= designates the beginning of a new action script for the parser. A script starts with this keyword and ends at the point where another "#open" keyword is found or it reached the end of the script part - specified by the tag [ENDSS]

[file_name] - specifies the file for the scripting action; this file will be used as follows:
i) **#save** - if executed, a save command will be written in the file - all the commands of type #save belonging to the same script action are saved in that file. The first #save command of an action script will open the file with write only attributes - therefore any previous content of the file will be erased;
ii) **#add** - if executed, the add command will be appended to the file if the appended line will be unique - it does not already exists in the file;
iii) **#delete** - will delete a line from the file. The line will be identified by the delete command parameters;
iv) **#replace** - will perform a replace action inside the file.

[command] - consists of a line made up by constant string and parameters to be replaced. The parameters that are replaced here are the ones that were set in the header of the EWS file, if a parameter is needed just for the scripting and it is not needed for the parsing of the HTML body it still has to be set at the header otherwise it will be disregarded.
[#cond=[condition]] - it is used to restrict the command execution and manipulation of the file according to the condition. The [condition] is first parsed before it is evaluated. After it is parsed, the condition is evaluated with the help of the "expr" command from Linux - therefore the syntax of the evaluation has been written according to the syntax of this command.

#save - specifies the beginning of the line for a save command, if this command is executed - the submission action was "Save" and the #cond, if exists, is evaluated true, it will manipulate the file set by [file_name].

#add - specifies the beginning of the line for an add command, if this command is executed - the submission action was "Add" and the #cond, if exists, is evaluated true, it will manipulate the file set by [file_name].

#delete - specifies the beginning of the line for a delete command, if this command is executed - the submission action was "Delete" and the #cond, if exists, is evaluated true, it will manipulate the file set by [file_name].

#replace - specifies the beginning of the line for a replace command, if this command is executed - the submission action was "Replace" and the #cond, if exists, is evaluated true, it will manipulate the file set by [file_name].

Within a script can be found as many actions as possible and the order in which they are set it is not important. Very important features of *ews* file can be notice: the simplicity and the great number of possibilities used for the description of the *ews* file, the content and its formatting are totally independent of the command that it is executed when loading, or as a result of user action.

CONCLUSION

Among the important characteristics of the SE we can record the following:

- allows the fast design, programming and debugging of the configuration applications;
- allows the extension of these applications by adding lines to the EWS files. This way the user can create configuration scripts for the different structures of AP, bridge, router etc. without being necessary to modify the SE;
- to modify an *ews* file the user can use a simple text editor;
- it is optimized for configuring command line based services and applications;
- it is able to generate shell bash scripts for starting processes and initializing the board;
- uses a low amount of memory space allowing the development of embedded systems with hardware platforms of small dimensions.
- HTML is not embedded in CGI code which allow to customize the interface without modify the SE;
- the SE has no configuration file to save the settings. To backup user settings the SE will archive entire etc directory and for restore, the SE will overwrite the etc directory with files from the archive;
- SE was successfully tested on platforms with IDT RC32334 processor and AMD AU1500 processor with 4MB Flash and 16MB SDRAM.

The execution of the commands is totally independent by the HTML content. The graphic design of the WEB interface can be easily changed without requiring any intervention over its functionality. Also having the minimum knowledge as regards to the Linux commands, can be easily extend the interface by adding new functionalities without being necessary to interfere with the CGI code that supplies the data for the web interface, but by simple descriptions in the header and footer of the EWS file.

ACKNOWLEDGEMENTS

We hereby acknowledge the structural found project PRO-DD (POS-CCE, O.2.2.1., ID 123, SMIS 2637, ctr. No 11/2009) for providing the infrastructure used in this work.

REFERENCES

1. Tourrilhes, Jean. "Wireless extensions for linux." Wireless Tools for Linux (1997).
2. Ierusalimschy, Roberto, Luiz Henrique De Figueiredo, and Waldemar Celes. "Lua 5.2 reference manual." (2011).
3. Cisco, “Aironet Access Point Software Configuration Guide”, San Jose, C. A. USA, 2003. *
4. Linksys, “EtherFast Wireless Access Point, Cable/DSL Router with 4-Port Switch”. User Guide, USA, 2001
5. Raghavan, Pichai, Amol Lad, and Sriram Neelakandan. Embedded Linux system design and development. CRC Press, 2005.
6. Tian-huang, C. H. E. N., and H. U. A. N. G. Jia-xi. "Design and Realization of CGI in Embedded Dynamic Web Technology." Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on. IEEE, 2007.
7. Choi, Mi-Joung, et al. "An efficient embedded web server for web-based network element management." Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP. IEEE, 2000.