Redes de Computadores

1st Semester 2017/2018
LEIC Alameda

Programming using the *Sockets* interface

"RC Distributed Processing"

1. Introduction

The goal of this project is to develop a simple networking application that allows users to submit a text file to a central server (CS) and request some predefined, well-known operations, to be performed on the file.
The CS checks the available working servers (WSs) and distributes the task among them, each WS processing a part of the file for completing the requested task. WSs provide their results to the CS, which compiles the partial results and, upon completion of the requested task, sends the desired reply to the user.

For the project, the students will develop: (i) the *user application* (user), and two server applications: (ii) a *Central Server* (CS); and (iii) a *Working Server* (WS), which implements one or more file processing tasks. The user application and the various servers are intended to operate on different machines connected to the Internet.

For the operation, the user application first contacts the CS, which has a well-known URL, asking for a list of the available file processing tasks, each identified by a three letter code – the *processing task code* (PTC). For each file processing task in the list, there is at least one WS server instance implementing it, which has previously registered with the CS. To issue a file processing request, the user application sends a request message to the CS indicating the selected PTC code and including the text file to be processed.
The CS will store the received file, naming it with a sequential 5 digit request number (*nnnnn.txt*), on a local subdirectory named "*input_files*", decides how to split the request among a selection of the WSs implementing the task with code PTC, and send each selected WS the corresponding request and respective file fragment.
Each WS receives a file fragment from the CS, named with the 5 digit number plus a three digit fragment number (*nnnnnfff.txt*), stores it in a local subdirectory named "*input_files*", and performs the PTC task on it. There are two possible *response types* (RT): a report (*R*) or a processed file (*F*), depending on the PTC. When the WS replies to the CS it indicates the RT code and a reply file which contains a report of the partial operation result (R), or the processed file portion (F). This reply is stored by the CS on a subdirectory named "*output_files*", with the original name *nnnnnfff.txt*. Once all partial replies are received, the user reply is prepared by performing any requested operations, which can include concatenating the partial response files, and the reply is saved in a file named *nnnnn.txt*, on the "*output_files*" subdirectory of the CS, and finally the reply is sent by the CS server to the user.

The well-known file processing tasks (with a maximum of 99 tasks) include the following ones that should be implemented by the students:
- word count (*PTC = WCT; RT = R*);
- find longest word (*PTC = FLW; RT = R*);
- convert text to upper case (*PTC = UPP; RT = F*);
- convert text to lower case (*PTC = LOW; RT = F*);

The CS server will contain a list (e.g., stored in memory or in a file, such as "*file_processing_tasks.txt*"), where it will update the information related to the available file processing tasks supported by the available WSs, notably including a line per each task supported by each WS, including: the PTC, the IP and port number where the WS server accepts processing requests. This information is automatically provided to the CS when each WS is started.

The project tests will include one CS server and at least two WS servers, capable of operating on different machines. Several instances of the user application will also be executed simultaneously. Each WS server should implement at least two test file processing tasks. To simplify the implementation the maximum number of WSs running simultaneously can be limited to 10.

For the implementation, the application layer protocols operate according to the client-server paradigm, using the transport layer services made available by the socket interface, using the TCP and UDP protocols.
The CS accepts user requests and passes them to the WS servers using TCP. The registration of WS servers with the CS is done using UDP.

2. Project Specification

2.1 User Application

The program implementing the user application should be invoked using the command:

`./user [-n CSname] [-p CSport]`,

where:

CSname is the name of the machine where the central server (CS) runs. This is an optional argument. If this argument is omitted, the CS should be running on the same machine.

CSport is the well-known port where the CS server accepts user requests, in TCP. This is an optional argument. If omitted, it assumes the value 58000+GN, where GN is the group number.

Once the user program is running, it waits for the user to indicate the action to take, notably:

- *list* – following this instruction the user application should contact the CS, asking for the list of available file processing tasks. In its reply the CS provides the list of available file processing tasks (*PTC*$_1$ *PTC*$_2$ … *PTC*$_{nL}$), possibly by checking the "*file_processing_tasks.txt*" file. The list of processing tasks will be displayed to the user as a numbered list of the corresponding textual descriptions.
- *request PTC*$_n$ *filename* – following this instruction the user application sends a message to the CS server, providing the desired file processing task code (*PTC*$_n$), and sending the contents of the file named *filename*
  The CS will reply with the result of the processed file.
- *exit* – the user application terminates.

2.2 *Central Server* (CS)

The program implementing the *Central Server* should be invoked using the command:

`./CS [-p CSport]`,

where:

CSport is the well-known port where the CS server accepts requests, in TCP. This is an optional argument. If omitted, it assumes the value 58000+GN, where GN is the number of the group.

The central server (CS) makes available a server with well-known port *CSport*, supported in TCP, to answer user requests for the file processing tasks.
The central server (CS) makes available another server, also with well-known port *CSport*, supported in UDP, to manage the registration and deregistration requests sent by WS servers, storing information about the file processing tasks available in which WSs. This information is stored in a list (or in a file, e.g. "*file_processing_tasks.txt*").
The CS server outputs to the screen the received requests and the IP and port originating those requests.
Each received request should start being processed once it is received.

2.3 Working Server (WS)

The program implementing the *Working Server (WS)* should be invoked using the command:

`./WS PTC1 … PTCn [-p WSport] [-n CSname] [-e CSport]`,

where:

PTC_1 … PTC_n is the list of available file processing task codes implemented by this WS, with *PTC = WCT* standing for "word count", returning a file with a number (*RT = R*), *PTC = FLW* standing for "find longest word", returning a file with string (*RT = R*), *PTC = UPP* standing for "convert text to upper case", returning a file with the processed text (*RT = F*) and *PTC = LOW* standing for "convert text to lower case", returning a file with the processed text (*RT = F*). Other file processing tasks (maximum of 99), can be included by the students, each with a different 3 letter code.

WSport is the well-known port where the WS server accepts requests from users. This is an optional argument. If omitted, it assumes the value 59000.

CSname is the name of the machine where the central server (CS) runs. This is an optional argument. If this argument is omitted, the CS should be running on the same machine.

CSport is the well-known port where the CS server accepts requests. This is an optional argument. If omitted, it assumes the value 58000+GN, where GN is the group number.

The WS server provides services to the CS using the TCP protocol with port *WSport*.
The WS accepts user requests for file processing tasks.
The WS outputs to the screen the received requests.

3. Communication Protocols Specification

3.1 *User–CS* Protocol (in TCP)

The user program, following the `list` instruction interacts with the CS server in TCP according to the following request and reply protocol messages:

a) `LST`
Following the `list` instruction, the user application sends a user list request message to the CS server, asking the list of available file processing tasks.

b) `FPT` n_L *PTC1 PTC2 … PTC*n_L
In reply to a `LST` request the CS server replies in TCP indicating the list of available file processing tasks (*PTC1 PTC2 … PTC*n_L), where *PTCi* is a string of characters specifying the code of the file processing task number `i`. The list of processing tasks will be displayed to the user as a numbered list of the corresponding textual descriptions.
If the `LST` request cannot be answered (e.g., no WS servers available) the reply will be "`FPT EOF`". If the `LST` request is not correctly formulated the reply is "`FPT ERR`".

c) `REQ` *PTC*$_n$ *size data*
Following the `request` instruction, the user sends a request to the CS asking to perform the processing task *PTC*$_n$ on the selected text file. The file *size* in Bytes is sent, followed by the corresponding *data*.

d) `REP` *RT size data*
The CS reply to a `REQ` request, includes the output of the file processing task performed. There are two possible response types (*RT*) depending on the task; in both cases the reply contains a text file with size *size* in Bytes, and the data bytes are available in *data*. If *RT=R* the *data* is the report of the performed task (e.g., a number or a string containing a word); if *RT=F* the reply is the processed text file.
If the `REQ` request cannot be answered (e.g., invalid PTC) the reply will be"`REP EOF`". If the `REQ` request is not correctly formulated the reply is "`REP ERR`".

In the above messages the separation between any two items consists of a single space. Each message of request or reply ends with the character "`\n`".

3.2 *CS–WS* Protocol (in TCP)

After the CS receiving a file for processing (following the user `request` instruction and `REQ` protocol message) it stores the received file, naming it with a sequential 5 digit request number (*nnnnn.txt*), on a local subdirectory named "*input_files*". Then the CS decides on how to split the file among the WS servers that implement the requested file processing task, sending each one a file fragment named with the 5 digit number plus a three digit fragment number (*nnnnnfff.txt*).

The communication protocol includes the following requests and replies:

a) `WRQ PTC` *`filename size data`*
The CS requests the WS to perform the task with code `PTC` on a file with name *`filename`*. The file *`size`* in Bytes is sent, followed by the corresponding *`data`*.

b) `REP` *`RT size data`*
The WS reply to the CS server includes the output of the file processing task performed. There are two possible response types (*`RT`*) depending on the task. If *`RT=R`* the *`data`* contains the reply report in a file, and if *`RT=F`* the reply is a processed text file. The transmitted file will have size *`size`* in Bytes, and the data bytes are available in *`data`*.
If the `WRQ` request cannot be answered (e.g., invalid PTC) the reply will be "`WRP EOF`". If the `WRQ` request is not correctly formulated the reply is "`WRP ERR`".

Separation between two items is a single space. Messages end with the character "`\n`".

3.3 *WS – CS* Protocol (in UDP)

When the WS starts/ends it needs to register/deregister itself with the CS server.
Communication uses the UDP protocol and includes the following requests and replies:

a) `REG` PTC_1 … PTC_n *`IPWS portWS`*
The WS informs the CS that it implements the file processing tasks with codes PTC_1 … PTC_n, and that it accepts file processing requests on IP address *`IPWS`* and TCP port number *`portWS`*.

b) `RAK` *`status`*
The CS confirms (*`status = OK`*) or declines (*`status = NOK`*) the `RAK` message. If there is a protocol (syntax) error the answer will be "`RAK ERR`".

c) `UNR` *`IPWS portWS`*
The WS informs the CS that it stopped operating and therefore *`IPWS`* and *`portWS`* should be removed from list of available WS servers.

d) `UAK` *`status`*
The CS confirms (*`status = OK`*) or declines (*`status = NOK`*) the `UNR` message. If there is a protocol (syntax) error the answer will be "`UAK ERR`".

Separation between two items is a single space. Messages end with the character "`\n`".

4. Development

4.1 Development and test environment

Make sure your code compiles and executes correctly in the development environment available in lab LT5.

4.2 Programming

The operation of your program should be based on the following set of system calls:

- Computer name: `gethostname()`.
- Remote computer IP address from its name: `gethostbyname()`.
- UDP server management: `socket()`, `bind()`, `close()`.
- UDP client management: `socket()`, `close()`.
- UDP communication: `sendto()`, `recvfrom()`.
- TCP server management: `socket()`, `bind()`, `listen()`, `accept()`, `fork()`, `close()`.
- TCP client management: `socket()`, `connect()`, `close()`.
- TCP communication: `write()`, `read()`.

4.3 Implementation notes

Developed code should be adequately structured and commented.
The `read()` and `write()` system calls may read and write, respectively, a smaller number of bytes than solicited – you need to ensure that your implementation still works correctly.
Both the client and server processes should terminate gracefully at least in the following failure situations:

- wrong protocol messages received from the corresponding peer entity;
- error conditions from the system calls.

5 Bibliography

- W. Richard Stevens, Unix Network Programming: Networking APIs: Sockets and XTI (Volume 1), 2nd edition, Prentice-Hall PTR, 1998, ISBN 0-13-490012-X, chap. 5.
- D. E. Comer, Computer Networks and Internets, 2nd edition, Prentice Hall, Inc, 1999, ISBN 0-13-084222-2, chap. 24.
- Michael J. Donahoo, Kenneth L. Calvert, TCP/IP Sockets in C: Practical Guide for Programmers, Morgan Kaufmann, ISBN 1558608265, 2000
- On-line manual, `man` command
- Code Complete - http://www.cc2e.com/
- http://developerweb.net/viewforum.php?id=70

6 Project Submission

6.1 Code

The project submission should include the source code of the programs implementing the *user*, the *CS server* and the *WS server*, as well as the corresponding *Makefile*.
The makefile should compile the code and place the executables in the current directory.

6.2 Auxiliary Files

Together with the project submission you should also include any auxiliary files needed for the project operation together with a *readme.txt* file.

6.3 Submission

The project submission is done by e-mail to the lab teacher, **no later than October 13, 2017, at 23:59 PM**.
You should create a single `zip` archive containing all the source code, makefile and all auxiliary files required for executing the project. The archive should be prepared to be opened to the current directory and compiled with the command `make`.
The name of the archive should follow the format: **`proj_"group number".zip`**

7 Questions

You are encouraged to ask your questions to the teachers in the scheduled foreseen for that effect.

8 Open Issues

You are encouraged to think about how to extend this protocol in order to make it more generic. For instance, how could new file processing tasks be offered, or what algorithms to use for managing the distributed processing?