Redes de Computadores

1st Semestre, P2, 2022/2023
LEIC Alameda

Programming using the *Sockets* interface

"RC Word Game"

1. Introduction

The goal of this project is to develop a simple word game, inspired in the classical *Hangman Game*.

The development of the project requires implementing a *Game Server* (*GS*) and a *Player Application* (*Player*). The *GS* and multiple *Player* application instances are intended to operate simultaneously on different machines connected to the Internet.
The *GS* will be running on a machine with known IP address and ports.

The interface with the user of the *Player* application supports a set of commands, provided using the keyboard, to control the actions to take:

- Start new game. Each player is identified by a *player ID* `PLID`, a 6-digit IST student number. When the *GS* receives a request to start a new game it checks if this player already has any ongoing game (each player can only play one game at any given instant). If a new game can be started the *GS* informs the *Player* of the number of letters composing the word to be guessed, and the maximum number of errors that can be committed before losing the game (this value depends on the word length).
- Play. For an ongoing game the *Player* application sends the *GS* server a letter that the player believes to be part of the word to be guessed. If the letter belongs to the word, the *GS* replies indicating in which position(s) that letter appears in the word. If all the letters of the word were guessed the player wins the game. If the maximum number of errors (letter attempts that do not belong to the word) is reached the player loses the game. The number of trials is increased with every play.
- Guess word. The player can try to guess the full word. If the guess is correct the player wins the game. If not, the number of errors is increased by 1.
- View scoreboard. The player can request to see the scoreboard, to which the *GS* replies with a text file containing the top 10 scores (including: `PLID`, number of plays to win, word).
- Hint. The player asks for a hint to help guessing the word, to which the *GS* replies with an image related to the category of the word (e.g., an image of an animal).
- State. The player asks about the status of the ongoing game.
- Quit. The player can ask to terminate the game at any moment. If a game was under way, the *GS* is informed.
- Exit. The player can ask to exit the *Player* application. If a game was under way, the *GS* is informed.

The implementation uses the application layer protocols operating according to the client-server paradigm, using the transport layer services made available by the socket interface. The applications to develop and the supporting communication protocols are specified in the following.

2. Project Specification

2.1 Player Application (*Player*)

The program implementing the game playing application (*Player*) is invoked using:

`./player [-n GSIP] [-p GSport]`,

where:

GSIP is the IP address of the machine where the game server (*GS*) runs. This is an optional argument. If this argument is omitted, the *GS* should be running on the same machine.

GSport is the well-known port (TCP and UDP) where the *GS* accepts requests. This is an optional argument. If omitted, it assumes the value 58000+GN, where GN is the group number.

Once the *Player* application is running, it can start or end a game, as well as exchange messages related to game playing or scoreboard visualization. When starting a new game, the player sends its *player ID*, *PLID*, so that statistics can be collected by the *GS* and to include in the scoreboard.

The commands supported by the *Player* user interface (using the keyboard for input) are:

- ***start*** *PLID* or ***sg*** *PLID* – following this command the *Player* application sends a message to the *GS*, using the UDP protocol, asking to start a new game, and provides the player identification *PLID*.
  The *GS* randomly selects a word from its `word_file`. Words are in English and can have between 3 and 30 letters. For playing there is no distinction between the usage of lowercase or uppercase letters.
  The *GS* also sets the maximum number of errors that can be committed before losing the game, *max_errors*, according to the selected word length:
  - *max_errors* = 7, for word lengths up to 6 letters
  - *max_errors* = 8, for word lengths between 7 and 10 letters
  - *max_errors* = 9, for word lengths of 11 letters or more.

  The result of the request should be displayed, according to the response received from the *GS* (e.g., "`New game started. Guess` *nn* `letter word: _ _ … _`").
- ***play*** *letter* or ***pl*** *letter* – the *Player* application sends a message to the *GS*, using the UDP protocol, with a new play to check if *letter* is part of the word to be guessed. The *GS* replies informing if the guess was correct and the position(s) where the *letter* appears, and if the player won the game. This information should be displayed (e.g., "`Word: _ e _ _ o _ k`").
- ***guess*** *word* or ***gw*** *word* – the *Player* application sends a message to the *GS*, using the UDP protocol, with an attempt to guess the hidden word, and thus win the game. The *GS* will reply informing whether the player won the game or not. This information should be displayed.
- ***scoreboard*** or ***sb*** – following this command the *Player* establishes a TCP session with the *GS* and sends a message asking to receive an updated scoreboard. In reply, the *GS* sends a text file containing the top 10 scores (including for each: *PLID*, number of plays to win, word) – the file only contains scores for games where the user won the game and discovered the word. After receiving the reply from the *GS*, the scoreboard is displayed as a numbered list.

- ***hint*** or ***h*** – following this command the *Player* establishes a TCP session with the *GS* and sends a message asking to receive an image illustrating the class to which the word belongs (e.g., an animal, or the flag of a country). In reply, the *GS* sends an image representative of the class the word belongs to. After receiving the reply from the *GS*, the name and size of the file where the image was stored are displayed.
- ***state*** or ***st*** – following this command the *Player* establishes a TCP session with the *GS* and sends a message asking to receive the state of a game for which a termination confirmation was not yet received. If the *GS* server has an active game for this player, it responds with a file containing the summary of all the trials of that game so far. If the *GS* server has no active game for this player it responds with a file containing a summary of the most recently finished game for this player, and the *Player* can then consider the current game as terminated. If the *GS* server finds no games (active or finished) for this player, it responds accordingly. The *Player* displays the received information.
- ***quit*** – the player can ask to terminate the game at any moment. If a game was under way, the *GS* server should be informed, by sending a message using the UDP protocol.
- ***exit*** – the player asks to exit the *Player* application. If a game was under way, the *GS* server should be informed, by sending a message using the UDP protocol.

Only one messaging command can be issued at a given time.
The result of each interaction with the *GS* should be displayed to the user.

2.3 *Game Server* (*GS*)

The program implementing the Game Server (*GS*) is invoked with the command:

`./GS word_file [-p GSport] [-v],`

where:

`word_file` is the name of a file containing a set of words that the *GS* can select from when a new game is started. Each line of the file contains a *word* and a *class* to which the word belongs, separated by a single space. This file is located in the same directory as the *GS* executable.

`GSport` is the well-known port where the *GS* server accepts requests, both in UDP and TCP. This is an optional argument. If omitted, it assumes the value 58000+GN, where GN is the number of the group.

The *GS* makes available two server applications, both with well-known port `GSport`, one in UDP, used for playing the game, and the other in TCP, used to transfer the scoreboard text file and the game logo image file to the *Player* application.
If the `–v` option is set when invoking the program, it operates in *verbose* mode, meaning that the *GS* outputs to the screen a short description of the received requests (`PLID`, type of request) and the IP and port originating those requests.
Each received request should start being processed once it is received.

3. Communication Protocols Specification

The communication protocols to implement the game playing and scoreboard transfer functionalities are described in this section.
For the communication protocols *PLID* is always sent using 6 digits.

3.1 *Player–GS* Game Protocol (in UDP)

The interaction between the player application (*Player*) and the game server (*GS*) for playing the game is supported by the UDP protocol.
The request and reply protocol messages to consider are:

a) SNG *PLID*
Following the ***start*** command, the *Player* requests to start a new game and sends the *player ID*, *PLID*, to the *GS* server.

b) RSG *status* [*n_letters max_errors*]
In reply to a SNG request the *GS* server checks if player *PLID* has any ongoing game (with play moves), in which case the reply *status* is NOK. If the game can be started (or no play was yet received) the *status* is OK and the *GS* server selects a word from the specified word_file, and sends to the *Player* application the number of letters of the selected word, as well as the maximum number of errors that can be committed before losing the game *max_errors*.

c) PLG *PLID letter trial*
Following the ***play*** command, the *Player* sends the *GS* a request to check if *letter* is part of the word to be guessed, as well as the number of trials (letters + words) send so far, *trial*. Also the *player ID* *PLID* is sent.

d) RLG *status trial [n pos*]*
In reply to a PLG request the *GS* replies informing whether the letter guess was successful or not. *status* can take the following values:

- OK, if the *letter* guess was successful, in which case also the number of occurrences, *n*, and the position(s), *pos*, where the *letter* appears are sent;
- WIN, if the *letter* guess completes the word;
- DUP, if this *letter* was sent in a previous *trial*. When the *GS* responds with DUP, the number of trials in the *GS* is not increased;
- NOK, if the *letter* is not part of the word to be guessed and the number of attempts left is not zero;
- OVR, if the *letter* is not part of the word to be guessed and there are no more attempts available – the *max_errors* limit was reached;
- INV, if the *trial* number is not valid – it was not the number expected by the GS, or if the player is repeating the last PLG stored at the *GS* with a different letter;
- ERR, if the syntax of the PLG was incorrect, the *PLID* is invalid, or there is no ongoing game for the specified player *PLID*. When the status is ERR, the response is simply RLG *ERR*\n.

The *Player* application should display this information.

e) PWG *PLID word trial*

Following the ***guess*** command, the *Player* sends the *GS* a request to check if the word to guess is *word*, as well as the number of trials (letters + words) send so far, *trial*. Also the player ID *PLID* is sent.

f) RWG *status trials*

In reply to a PLG request the *GS* replies whether the word guess was successful or not, and informs about the number of trials received by the *GS*.

status can take the following values:

- WIN, if the *word* guess was successful;
- NOK, if the *word* guess is not correct and the number of attempts left is not zero;
- OVR, if the *word* guess is not correct and there are no more attempts available (the maximum number of errors or the maximum number of trials was reached);
- INV, if the *trial* number is not the one expected by the *GS*, or if the player is repeating the last PWG message received by the GS with a different word;
- ERR, if the syntax of the PWG was incorrect, the *PLID* is invalid, or there is no ongoing game for the specified player *PLID*. When the status is ERR, the response is simply RLG *ERR*\n.

The *Player* application should display this information.

g) QUT *PLID*

Following the **quit** or **exit** commands, and if there is an ongoing game, the *Player* application sends the *GS* a message with the player *PLID* requesting to terminate the game.

h) RQT *status*

In reply to a QUT request the *GS* server sends a status message. If player *PLID* had an ongoing game, the *GS* replies with *status* = OK. Otherwise the *status* is ERR.

i) REV *PLID*

Following the **rev** debug command, and if there is an ongoing game, the *Player* application sends the *GS* a message with the player *PLID* requesting to reveal the word to be guessed.

j) RRV *word/status*

During project development in reply to a REV request the *GS* server sends a back the *word* to be guessed.

For the final version of the project this command should result in game termination, with the *GS* server sending a status message. If player *PLID* had an ongoing game, the *GS* replies with *status* = OK. Otherwise the *status* is ERR.

Note:

If the *Player* doesn't receive the response `RLG(RWG)` to a given `PLG(PWG)`, it should not increase the number of trials. The next `PLG(PWG)` message to be sent should repeat the previous one: same letter(word) and number of trial. This can happen if:

a. The last `PLG(PWG)` was not received by the *GS;* or
b. The `RLG(RWG)` response to the last `PLG(PWG)`, sent by the *GS*, wasn't received by the *Player*.

When the *GS* receives a `PLG(PWG)` message being repeated by the *Player*, there are three possibilities:

a. The *GS* hasn't received the previous `PLG(PWG)`, treating it as a new message.
b. The *GS* has received the previous `PLG(PWG)` and this message repeats the same information (number of trial and letter) – in this case the *GS* repeats the previous `RLG(RWG)` response, assuming it had been lost in the network (the status `DUP` doesn't apply).
c. The *GS* has received the previous `PLG(PWG)` and this message repeats the number of trial but providing a different letter(word) – in this case the *GS* response *`status`* will be `INV`.

If an unexpected protocol message is received, the reply is `ERR`.
In the above messages the separation between any two items consists of a single space.
Each request or reply message ends with the character "`\n`".

3.2 *Player–GS* Messaging Protocol (in TCP)

The interaction between the player application (*Player*) and the game server (*GS*) for messaging related to the visualization of the scoreboard and the transmission of an hint image file is supported by the TCP protocol.
The request and reply protocol messages to consider are:

a) `GSB`
Following the ***`scoreboard`*** command, the *Player* application opens a TCP connection with the *GS* and asks to receive the top-10 scoreboard.

b) `RSB` *`status`* `[`*`Fname Fsize Fdata`*`]`
In reply to a `GSB` request the *GS* sends a status response. If the scoreboard is still empty (no game was yet won by any player) the *GS* replies with *`status`* = `EMPTY`, otherwise the *GS* replies with *`status`* = `OK`, and sends a text file containing the top-10 scores of the game. The information sent includes:

- the filename *`Fname`*;
- the file size *`Fsize`*, in bytes;
- the contents of the selected file (*`Fdata`*).

The text file contains one score per line consisting of the **score** value, in the range [001-100], with 100 being the top score, the player ID *`PLID`*, followed by the **word** guessed, the **number of successful plays**, and the **total number of plays** needed to win; these fields are separated by spaces. The file starts with the top score and can contain up to 10 lines, each line terminated with a '`\n`'. These scores are displayed by the *Player*.
A local copy of the score board is stored using the filename *`Fname`*.
After receiving the reply message, the *Player* closes the TCP connection with the *GS*.

c) `GHL` *`PLID`*
Following the ***`hint`*** command, the *Player* application opens a TCP connection with the *GS* and asks to receive an image illustrating the class to which the word belongs.

d) `RHL` *`status [Fname Fsize Fdata]`*
In reply to a `GHL` request the *GS* replies with *`status`* = `OK` and sends a file containing the image illustrative of the word class. The information sent includes:

- the filename *`Fname`*;
- the file size *`Fsize`*, in bytes;
- the contents of the selected file (*`Fdata`*).

The file is locally stored using the filename *`Fname`*.
The *Player* displays the name and size of the stored file.
If there is no file to be sent, or some other problem, the *GS* replies with *`status`* = `NOK`.
After receiving the reply message, the *Player* closes the TCP connection with the *GS*.

e) STA *PLID*

Following the ***state*** command, the *Player* application opens a TCP connection with the *GS* and asks about the state of the ongoing game at the *Player*.

f) RST *status [Fname Fsize Fdata]*

In reply to a STA request the *GS* replies depending on whether player *PLID* has an ongoing game or not.

If there is an ongoing game, then *status* = ACT and the *GS* server sends a text file containing the summary game so far. Upon receiving this summary, the player displays the information and continues with the game.

If there is no ongoing game for this player then *status* = FIN and the *GS* server responds with a file containing the summary of the most recently finished game for this player. Upon receiving this summary, the player displays the information and considers the current game as terminated.

If the *GS* server finds no games (active or finished) for this player, then *status* = NOK.

When replying with the game state information, a text file is sent. The reply message then includes:

- the filename *Fname*;
- the file size *Fsize*, in bytes;
- the contents of the selected file (*Fdata*).

The text file includes the game state and a local copy of the game state is stored using the filename *Fname*. The name and size of the stored file are displayed.

After receiving the reply message, the *Player* closes the TCP connection with the *GS*.

The filenames *Fname*, are limited to a total of 24 alphanumerical characters (plus '-', '_' and '.'), including the separating dot and the 3-letter extension: "*nnn…nnnn.xxx*". The file size *Fsize* is limited to 1GiB (1024^3B), being transmitted using a maximum of 10 digits.

If an unexpected protocol message is received, the reply will be ERR.
In the above messages the separation between any two items consists of a single space.
Each request or reply message ends 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 the labs *LT4* or *LT5*.

4.2 Programming

The operation of your program, developed in *C* or *C++*, may need to use the following set of system calls:

- Reading user information into the application: `fgets()`;
- Manipulation of strings: `sscanf()`, `sprintf()`;
- UDP client management: `socket()`, `close()`;
- UDP server management: `socket()`, `bind()`, `close()`;
- UDP communication: `sendto()`, `recvfrom()`;
- TCP client management: `socket()`, `connect()`, `close()`;
- TCP server management: `socket()`, `bind()`, `listen()`, `accept()`, `close()`;
- TCP communication: `write()`, `read()`;
- Multiple inputs multiplexing: `select()`.

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.

The client and the server processes should not terminate abruptly in failure situations, such as these:

- wrong protocol messages received by the server: an error message should be returned, as specified by the protocol;
- wrong protocol messages received by the client: the interaction with the server is not continued and the user is informed;
- error conditions from system calls: the programs should not terminate abruptly, avoiding cases of "segmentation fault" or "core dump".

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 *Player* and the *GS server*, as well as the corresponding *Makefile* and the *auto-avaliação* excel file.
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 December 22, 2022, at 23:59 PM**.
You should create a single `zip` archive containing all the source code, makefile, all auxiliary files required for executing the project and the *auto-avaliação* excel file. 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 Open Issues

You are encouraged to think about how to extend this protocol in order to make it more generic.