The University of Queensland
School of Information Technology and Electrical Engineering

CSSE2310 - Semester 2 2020
Assignment 3 (v1.1)

Marks: 46
Weighting: 33% of your overall assignment mark
Due: 23:59 30 October, 2020

Student conduct

This is an individual assignment. You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on Piazza. In general, questions like "How should the program behave if ⟨this happens⟩?" would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students with the actual design and coding of your assignment solution. It is **cheating to look at another student's assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you, and those you cheated with. That's right, if you share your code with a friend, **both of you are in trouble**.
You may use code provided to you by the CSSE2310 teaching staff **in this semester**.

Uploading or otherwise providing the assignment specification to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of these sites and they cooperate with us in misconduct investigations.

The course coordinator reserves the right to conduct interviews with students about their submissions, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process.

In short - **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. You should read and understand the statements on student misconduct in the course profile and on the school web-site:

`http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html`

Aim

Your task is to write two programs (`rpsserver` and `rpsclient`) that allow a rock/paper/scissors tournament to be played between an arbitrary number of game clients.

The `rpsserver` acts as a game matchmaker, and listens for connections from clients - `rpsclient`. The server introduces clients to each other, these clients then play a series of games (a 'match') against each other. After each match, the players report the result of that match back to the server (**SPEC UPDATE: This sentence clarified to refer to matches, not games**).

Your programs will use network sockets for communication, and the `pthreads` library for multi-threading. No additional processes may be created, and pipes are not to be used for inter-process communication. Your programs are not to create or read any files from the filesystem.

`rpsserver`

The `rpsserver` program will take no arguments.

```
./rpsserver
```

Upon starting, it will listen on an ephemeral port (its command port), and print that port number to `stdout`. Note: You should `fflush(stdout)` to ensure immediate output of the port number.

The server listens for messages on the command port (see Message Table below for details). Possible messages include

- Match request (`MR`) - a connecting client is requesting to be matched with an opponent.
- Match result (`RESULT`) - players report the result of their match back to the server

Match Request

When a client connects to the server, it should send a match request in order to be matched up with an opponent. If a client connects and does not send a match request as its first message (or sends an invalid request), the server should close that connection.

Upon receiving a match request, the server chooses an opponent from those waiting and sends a `MATCH` message to each of them. The `MATCH` message identifies the match number, opponent name, and opponent port number. Match identifiers issued by the server should start from 1 and increment for each new match it coordinates. Note that match identifiers can potentially be negative.

Players should be paired up for matches in the order that they connect to a server. The server will only send a match message once enough players are available to play a match.

If `rpsserver` receives a match request from a client that has the same name as a client already playing, it should reply with a `BADNAME` message and close that connection. A client is considered to be playing from the time they first send a (succesful) match request, up until they send a result message at the end of the match they requested.

An example of this interaction is as follows:

1. `Barney` sends a match request (and will be kept waiting since there aren't enough players yet)
2. `Barney` (another one) sends a match request (and will be rejected with a `BADNAME` message since another client with that name is already playing)
3. `Wilma` sends a match request
4. `rpsserver` sends `MATCH` messages to `Barney` and `Wilma`
5. `Pebbles` sends a match request (and will be kept waiting since there aren't enough players yet)
6. `Dino` sends a match request
7. `rpsserver` sends `MATCH` messages to `Pebbles` and `Dino`
8. ...
9. `Barney` and `Wilma` finish their match and send `RESULT` messages
10. `Barney` sends a match request (and will be kept waiting since there aren't enough players yet)
11. ...

Match Result

Once clients have finished their match, they should each send a `RESULT` message back to the server. This message should be sent using the same connection over which the initial `MATCH` message was received.

Once both player's messages are received, the server should close the connections to the players and cross-check the received messages for correctness. A result message pair is valid if:

- both messages are `RESULT` messages
- their IDs match the one given to the players in the initial match request the server sent
- the match results are the same

If the messages are invalid, the server should ignore the result of that match. Otherwise, updates its stored set of results. The connections to both players are then closed.

Handling `SIGHUP`

Upon receiving `SIGUP`, the server should output (to `stdout`), the match outcomes (win, lose, tie) of all clients who have played a match. This output should be sorted in lexicographic order (i.e. the order used by `strcmp()`) by client name. The output is terminated by a new line containing three `-` characters terminated with a newline character. An example output is as follows:

```
Barney 2 5 1
Betty 7 5 0
Fred 10 2 0
Wilma 7 3 1
---
```

`SIGHUP` must not terminate the server. It should keep handling match requests until terminated with `SIGINT` or `SIGKILL`.

`rpsserver` Errors

The following error messages should be sent to `stderr`.

Exit code	Condition	Message
1	Incorrect number of arguments	`Usage: rpsserver`

rpsclient

The `rpsclient` program takes 3 arguments:

```
./rpsclient client_name num_matches serverport
```

Where

- `client_name` is a string comprised strictly of alphanumeric characters (`[a-z][A-Z][0-9]`). No spaces or other special characters are permitted in client names. Further, the following strings (without quotes) are reserved and may not be used as player names:
  - "`TIE`"
  - "`ERROR`"

  If `rpsclient` is started with an invalid name, then it should exit with the appropriate error message and code (see below).
- `num_matches` is a positive non-zero integer indicating how many matches the client wishes to play
- `serverport` identifies the command port of the server (running on `localhost` - use IP address `127.0.0.1` or `getaddrinfo("localhost", ...)`).

Gameplay

Upon starting, the `rpsclient` should create and listen on an ephemeral port. It should then proceed to complete the number of matches specified on its command line. For each match, the client should:

- Connect to the server command port with a match request message, and wait for a response from the server
- Once a match message is received, connect to the opponent identified in that message, and play the match. A match is the best of 5 games, or (after 5 games and still having a tie) until one player has the lead (up to 20 games). For example, if the first five games are tied, then the winner of the next game wins the match. If 20 games are played without breaking the tie, the result is a tie.
- At the end of a match, each player sends its result back to the server's command port via a `RESULT` message. Note that the client should not create a new server connection to send the result. It should use the one it created at the start of the match to receive its opponent's information.
- Wait for 50 milliseconds before attempting to play a new match (hint, the `usleep()` function is helpful here).

If the client is unable to create a connection to the server or gets disconnected from the server at any point in time... If it receives invalid input from its opponent or the server, that match is considered to be an error

SPEC CLARIFICATION: Bold phrases in the following paragraph to clarify behaviour on disconnect or bad server messages.

If the client is unable to create a connection to the server **or gets disconnected from the server** at any point in time, it should exit with an "Invalid port" error (see the "Errors" section below). If the client receives invalid input from its opponent **or the server**, that match is considered to be an error.

SPEC CLARIFICATION: Agents should write their moves over the connection they initiated to their opponent, and read opponent moves from the socket on which they listened for incoming

connections. To avoid any doubt, ensure that your agent is able to play against the `demo_rpsclient` implementation on `moss`.

Once the client has completed the number of matches specified on its command line, it should output a list of the results of all of its matches, in the order they were played. Each line of output has the following form:

```
match_id opponent result
```

where `result` is from this player's point of view. For example:

```
1 Barney LOST
4 Wilma TIE
6 Barney ERROR
7 Betty WIN
```

means that this player

- lost against `Barney` in match 1,
- tied with `Wilma` in match 4,
- had match 6 against `Barney` end in an error (e.g. `Barney` quit unexpectedly or sent invalid messages),
- won against `Betty` in match 7

Example match results

The following examples aim to clarify the logic around match results, the 5/20 game min/max limit, and tied matches.

- Player 1 wins 5 games - Player 1 wins the match
- Player 1 wins 2 games, Player 2 wins 3 games - Player 2 wins the match
- Player 1 wins 2 games, Player 2 wins 2 games, game 5 is a tie, Player 1 wins game 6 - Player 1 wins the match
- Player 1 wins 2 games, Player 2 wins 2 games, games 5-20 are ties - Match result is a tie

`rpsclient` Errors

The following error messages should be sent to `stderr`.

Exit code	Condition	Message
0	Player completed matches succesfully	
1	Incorrect number of arguments	`Usage: rpsclient name matches port`
2	Invalid name	`Invalid name`
3	Invalid match count (e.g. non-integer argument)	`Invalid match count`
4	Invalid port number (e.g. non-integer argument, or unable to connect)	`Invalid port number`

Move Generation

Player agents issue pseudo-random moves. Upon *agent startup* (not the beginning of each match), agents are to initialise their random number seed using `srand()` with the following algorithm

```
char* agent_name;
...
int seed=0;
for(int i=0; i<strlen(agent_name); i++) {
   seed+=agent_name[i];
}
srand(seed);
```

Moves by each agent are then generated randomly as follows.

```
int move_val = rand() % 3;
```

As with Assignment 2, **it is critical that you initialise and generate random numbers in exactly this manner so that your results can be auto-marked**.

Rock-Paper-Scissors Rules

Move values are interpreted as follows:

Move value	Move
0	ROCK
1	PAPER
2	SCISSORS

In game play, the following defines the winner of each game

- ROCK beats SCISSORS
- SCISSORS beats PAPER
- PAPER beats ROCK

If two players make the same move, the game result is a tie.

Messages

Direction	Format	Detail
player → server	`MR:<name>:<port>`	Agent `<name>` is requesting a match, their opponent should connect on port `<port>` e.g. `MR:fred:57854` ("Hi, I'm `fred`. I'd like to play. I'm listening on port `57584`")
server → player	`BADNAME`	Your name is already in use. Please go away and come back with a new name.
server → player	`MATCH:<match_id>:<opponent>:<port>`	Server telling agent it has been paired with opponent `<name>`, who is listening on port `<port>`. `<match_id>` is a unique decimal integer identifier for the match, which must sent back with the match result message. e.g. `MATCH:37:Wilma:43765` ("Your opponent in match `37` is `Wilma`, listening on port `43765`")

player → player	`MOVE:<move>`	A player's move
		`<move>` = `ROCK|PAPER|SCISSORS`
		e.g.
		`MOVE:SCISSORS`

player → server	`RESULT:<match_id>:<result>`	Inform the server of match result. `<match_id>`=match ID (decimal) `<result>`=winner name, or `<result>=TIE` (for a tied match) `<result>=ERROR` (e.g. if the opponent disconnected unexpectedly) e.g. `RESULT:34:Barney` (`Barney` won match `34`) e.g. `RESULT:14:ERROR` (Match `14` ended with some error) e.g. `RESULT:76:TIE` (Match `76` was a tie)

Forbidden functions

You are not to use `poll()`, `select()` or any other non-blocking IO functions in your submission.

Style

You must follow version 2.0.4 of the CSSE2310/COMP7306 C programming style guide available on the course BlackBoard site.

Submission

Your submission must include all source and any other required files (in particular you must submit a Makefile). Do not submit compiled files (eg .o, compiled programs) or rules or map files. You may not create any subdirectories in your submission. Such subdirectories will be removed prior to marking.

Your program must compile with the `make` command. Your program must be compiled under `gcc` with at least the following switches: `-pedantic -Wall -pthread -std=gnu99`. You are not permitted to disable warnings or use pragmas to hide them.

If any errors result from the `make` command (i.e. an executable cannot be created), then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality). Your program must not invoke other programs or use non-standard headers/libraries.

The marking tests will run `make` in a clean SVN checkout of your repository, and will expect to find the following two programs

- `rpsserver`
- `rpsclient`

Your assignment submission must be commited to your subversion repository under

`https://source.eait.uq.edu.au/svn/csse2310-sXXXXXXX/trunk/ass3`

Note: `sXXXXXXX` is your moss/UQ login ID.

Your submission will be assessed by the contents of your SVN repository at the due date/time. Any subversion commits after the due date will be ignored. You must ensure that all files needed to compile and use your assignment (including a makefile) are committed and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes. The late submission policy in the CSSE2310 Electronic Course Profile applies. Be familiar with it!

Marks

Marks will be awarded for functionality and style.

Functionality (42 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program can detect hits correctly. The markers will make no alterations to your code (other than to remove code without academic merit).

Marks will be assigned in the following categories. Your server and player programs will be tested independently, so even if your server doesn't work you can still get marks for the player.

`rpsserver` (22 marks)

1. Program correctly handles invalid command lines [1 mark]
2. Coordinate a single match between two clients [4 total]
   - Correctly handle `SIGHUP` and required match result output (2 marks)
   - Correctly detect duplicate player name and send `BADNAME` message (2 marks)
   - Sub category marks to be advised

3. Coordinate multiple matches between two clients [5 total]
   - Correctly handle `SIGHUP` and required match result output (2 marks)
   - sub-category marks to be advised
4. Coordinate multiple matches between multiple clients [12 total]
   - Correctly handle `SIGHUP` and required match result output (4 marks)
   - sub-category marks to be advised

`rpsclient` (20 marks)

- Program correctly handles invalid command line arguments (1 marks)
- Program correctly handles early termination by opponent (2 marks)
- Program correctly handles `BADNAME` message from server (1 marks)
- Program correctly plays a single match against one other opponent (4 marks)
- Program correctly plays a multiple matches against one other opponent (4 marks)
- Program correctly plays a multiple matches against multiple other opponents (4 marks)
- Program correctly outputs result of all played matches upon exit (4 marks)
- sub-category marks to be advised

Style (4 marks)

For this assignment, style will be auto-marked only.

Style marks will be calculated as follows: Let

- A be the number of style violations detected by `style.sh` plus the number of compilation warnings.

Your style mark is simply:

- an automatic mark ($M_A = 4 \times 0.8^A$), as reported by the `style.sh` tool on `moss`

The number of compilation warnings will be the total number of distinct warning lines reported during the compilation process described above. The number of style guide violations refers to the number of violations of version 2.0.4 of the CSSE2310 C Programming Style Guide.

A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name).

To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark - this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the `style.sh` tool installed on `moss` to style check your code before submission.

Total mark

Let

- F be the functionality mark for your assignment.
- S be the style mark for your assignment.

Your total mark for the assignment will be:

$$M = F + \min\{F, S\}$$

In other words, you can't get more marks for style than you do for functionality. Pretty code that doesn't work will not be rewarded!

Late Penalties

Late penalties will apply as outlined in the course profile.

Hints

- Use appropriate modularisation techniques to share common code between the server and the agent programs
- Study and experiment with the example code provided in Weeks 9 and 10 on sockets and `pthreads`. Don't start writing the server until you understand these examples!
- Draw yourself a diagram that shows the relationships between the server threads
- You can use `netcat` to talk to the server, or another agent...

Demonstration / reference implementation

A reference solution is available on `moss` in the following location:

```
/local/courses/csse2310/resources/assignment3
```

In this directory you will find:

- `demo_rpsserver` - an implementation of rpsserver
- `demo_rpsclient` - an implementation of rpsclient
- `demo_rpsclient_{cheat|lie}_{win|lose|tie}` - test clients that will either cheat (to win, lose or tie), or lie about the match result (win/lose/tie). This allows you to deterministically test the behaviour of your server in the case of inconsistent match results reports for example. Don't try playing two cheating agents against each other, they will deadlock each waiting for their opponent's move before sending their own.

Reverse engineering and misconduct

These binaries are execute-only (non-readable) for students. You can run them (have to specify full path to the binary), but cannot copy them, debug them and so on. In principle it is possible, although difficult, for you to get access to the binary, and run it through a reverse engineering tool to try and recover the source code, or something approximating it.

First of all - if you think this will be an easier path to a solution than just doing the coding, then you are wrong! Please don't waste your time.

Secondly - if you do manage to do it, then you deserve the marks for the assignment. However, **under no circumstances are you to share with others your method, or code recovered through this method** - that will be considered misconduct.

Finally, if you do manage to crack these binaries please disclose it to me privately (no repercussions for **responsible** disclosure), and talk to us about joining the UQ Cyber Squad because we want your mad skillz.

Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on Piazza or emailed to `csse2310@helpdesk.eait.uq.edu.au`.

v1.1

- Add reference to demo implementations available on `moss`
- Clarify that agents write moves to their outbound connection, and read opponent moves from their inbound (listening) socket
- Clarify agent behaviour in case of disconnectionor bad messages from server
- Add forbidden functions list
- Fix misleading space in `Usage:` messages
- Clarify "game" vs "match" in paragraph 2 of Aims section
- Fix example '`RESULT`' messages in message table (was '`MR`')
- Add Hints section