COMP30023 Project 2
Remote Procedure Call

Out date: 28 April 2023
Due date: No later than 5pm Monday 22 May, 2023 AEST

Weight: 15% of the final mark

1 Project Overview

Remote Procedure Call (RPC) is a crucial technology in distributed computing that enables software applications to communicate with each other seamlessly over a network. It provides a way for a client to call a function on a remote server as if it were a local function call. This abstraction allows developers to build distributed systems and applications that span multiple machines and platforms.

In this project, you will be building a custom RPC system that allows computations to be split seamlessly between multiple computers. This system may differ from standard RPC systems, but the underlying principles of RPC will still apply.

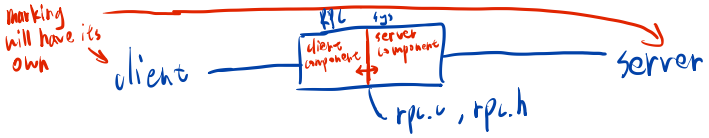
Your RPC system must be written in C. **Submissions that do not compile and run on a Linux cloud VM, like the one you have been provided with, may receive zero marks.** You must write your own RPC code, **without** using existing RPC libraries.

2 RPC System Architecture

Your task is to design and code a simple Remote Procedure Call (RPC) system using a client-server architecture. The RPC system will be implemented in two files, called `rpc.c` and `rpc.h`. The resulting system can be linked to either a client or a server. For marking, we will write our own clients and servers, and so you must stick to the proposed API carefully.

For testing purposes, you may run server and client programs on the same machine (e.g., your VM).

3 Project Details



Your task is to design and code the RPC system described above. You will design the application layer protocol to use. A skeleton is provided which uses a simple application programming interface (API). When we assess your submission, we will link our own testing code using the same RPC system API; what you will be assessed on is `rpc.c` (and any other supporting files compiled in by your `Makefile`).

> Note that implementing the API will require you to use sockets. This uses material covered in the lectures after the project is released. There is plenty to do on the project before you need to use sockets, so please do not say that you cannot start because you have not yet learned about sockets.

3.1 API

The basic API implemented by `rpc.c` consists of the following data structures and functions.

3.1.1 Data structures

The API will send and receive data structures of the form:

payload

```
typedef struct {
    int    data1;
    size_t data2_len;
    void   *data2;
} rpc_data;
```

where `data1` is an integer to be passed to the other side, and `data2` is a block of bytes (of length `data2_len`) to be sent. The purpose of `data1` is to allow simple functions that only pass an integer to avoid memory management issues, by setting `data2_len=0` and `data2=NULL`. Your protocol can limit `data1` to being no more than 64 bits.

Note that `size_t` depends on the architecture, and the sender and receiver can have different architectures. Think how this will affect your protocol.

The handler that implements the actual remote procedure will have the signature:

```
rpc_data *procedure (rpc_data *d);
```

That is, it takes a pointer to an `rpc_data` object and returns a pointer to another `rpc_data` object. This function will dynamically allocate memory with `malloc` for both the `rpc_data` structure and its `data2` field. It is the responsibility of the RPC system to `free` those after use.

The state of the client and server will be in data structures that you define. These are declared in `rpc.h` as:

```
typedef struct rpc_client rpc_client;
typedef struct rpc_server rpc_server;
```

and you should provide the actual `struct` definitions in `rpc.c`. These are returned by initialization functions (`rpc_init_client` and `rpc_init_server`, and passed to all other functions.

3.1.2 Server-side API

```
rpc_server *rpc_init_server (int port)
```

Called before `rpc_register`. Use this for whatever you need. It should return a pointer to a struct (that you define) containing server state information on success and `NULL` on failure.

```
int rpc_register (rpc_server *srv, const char *name, rpc_data* (*handler)(rpc_data*))
```

At the server, let the subsystem know what function to call when an incoming request is received.

It should return a non-negative number on success (possibly an ID for this handler, but a constant is fine), and `-1` on failure. If any of the arguments is `NULL` then `-1` should be returned.

Think:

1. What are the valid characters in `name`? (We will only test printable ASCII characters between 32 and 126.)
2. How does the server know the length of `name`? (We will not test names longer than 1000 bytes.)
3. Should there be a minimum length for `name`? (We will not test for empty names.)

If there is already a function registered with name `name`, then the old function should be forgotten and the new one should take its place.

To get full marks, you should be able to register at least 10 functions. You can still get most of the marks as long as you can register one function, so implement that first.

```
void rpc_serve_all (rpc_server *srv)
```

This function will wait for incoming requests for any of the registered functions, or `rpc_find`, on the port specified in `rpc_init_server` of any interface. If it is a function call request, it will call the requested function, send a reply to the caller, and resume waiting for new requests. If it is `rpc_find`, it will reply to the caller saying whether the name was found or not, or possibly an error code.

This function will not usually return.
It should only return if `srv` is `NULL` or you're handling `SIGINT` (not a requirement).

3.1.3 Client-side API

`rpc_client *rpc_init_client (const char *addr, int port)`

Called before `rpc_find` or `rpc_call`. Use this for whatever you need. The string `addr` and integer `port` are the text-based IP address and numeric port number passed in on the command line.

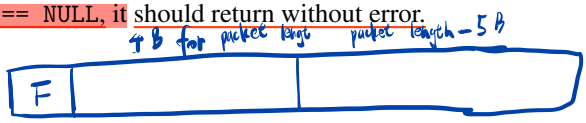
The function should return a non-`NULL` pointer to a struct (that you define) containing client state information on success and `NULL` on failure.

`void rpc_close_client (rpc_client *cl)`

Called after the final `rpc_call` or `rpc_find` (i.e., any use of the RPC system by the client).

Use this for whatever you need; it should at least `free(cl)`.

If it is (mistakenly) called on a client that has already been closed, or `cl == NULL`, it should return without error. (Think: How can you tell if it has already been closed?)



`rpc_handle *rpc_find (rpc_client *cl, const char *name)`

At the client, tell the subsystem what details are required to place a call. The return value is a handle (not handler) for the remote procedure, which is passed to the following function.

If `name` is not registered, it should return `NULL`. If any of the arguments are `NULL` then `NULL` should be returned. If the find operation fails, it returns `NULL`.

`rpc_data *rpc_call (rpc_client *cl, rpc_handle *h, const rpc_data *data)`

This function causes the subsystem to run the remote procedure, and returns the value.

If the call fails, it returns `NULL`. `NULL` should be returned if any of the arguments are `NULL`. If this returns a non-`NULL` value, then it should dynamically allocate (by `malloc`) both the `rpc_data` structure and its `data2` field. The client will free these by `rpc_data_free` (defined below).

The skeleton gives an example of how these functions can be used.

3.1.4 Shared API

`void *rpc_data_free (rpc_data* data)`

Frees the memory allocated for a dynamically allocated `rpc_data` struct.
Note that there is a reference implementation of this function in the skeleton code.

3.2 Planning task

When you are designing the protocol, ask yourself the following questions. Think of the answer for this particular project, and separately for the case of a "real world" RPC server.

Put the answers in a *plain text* file `answers.txt`, beginning with your name, login ID and student ID. These, together with protocol description below, are worth **1 mark**.

1. Should the server accept calls from everyone, or just a subset of users?
2. Should authentication etc. be provided by the RPC framework, or by the functions that use the RPC framework?
3. What transport layer protocol should be used? What are the trade-offs?
4. In which function(s) should the socket(s) be created?
5. Should `rpc_client` and `rpc_server` be allocated dynamically or statically? What are the implications for the client and server code?
6. What happens if one host uses big-endian byte order and the other uses little-endian? How does that relate to "network byte order"?

3.3 Protocol

You should design and document a simple application layer protocol for this RPC system. (It can be very simple.) Describe it in the file `answers.txt`, in enough detail that someone else could implement the protocol. Together with the questions above, this is worth **1 mark**.

Note that `size_t` is system-dependent. You will need to choose a system-independent way to encode the size of the data block in the packet. You can use a universal encoding, like Elias gamma coding (see Wikipedia) which can specify arbitrarily long strings, or you can use a fixed sized field, which is simpler to decode but limits the size of the string. *Explain your choice in your protocol description.*

In all test cases, `len < 100 000`. size_t data2_len

If `data2_len` is too large to be encoded in your packet format, the relevant function in `rpc.c` should print "`Overlength error`" to `stderr` and return an error.

The protocol should specify error responses for routine failures, such as a request for a procedure that does not exist.

The protocol should work correctly even if there are IP layer packet loss and duplication.

The protocol should handle the fact that IP packets have a maximum allowed size.

Decide what transport layer protocol to use. (You will almost certainly choose TCP, but briefly mention the pros and cons of alternatives.)

The transport layer protocol should run on top of IPv6.

3.4 Test harness

Code isn't complete until it has been thoroughly tested.

If your `Makefile` produces executables `rpc-server` and `rpc-client`, then they will be executed by the CI with the command line arguments shown below, with the results of `stdout` and `stderr` included in the test transcript.

No marks are allocated to `rpc-server` and `rpc-client`, either in execution or in code quality.

To run your server program on your VM prompt, type:

```
./rpc-server -p <port> &
./rpc-client -i <ip-address> -p <port>
```

where:

- The `&` tells the operating system to run the server in the background.
- `ip-address` is the IPv6 address of the VM on which the server is running.
- `port` is the TCP (or other transport layer) port number of the server.

The server is expected to listen for incoming connections on the port passed via command line arguments, on any of the hosts IPv6 network addresses.

4 Stretch goal: non-blocking performance

Many RPC operations, such as database look-ups, take a substantial time to complete. Instead of making each request wait for all those before it to complete, it is possible to continue to accept and start processing new requests while previous requests are executing, with each call returning as soon as it is finished. The simplest way to do this is with multiple threads: each time a request (or connection) is received, a new thread is spawned to execute the procedure, and is destroyed once the result has been sent back to the caller. Alternatives include creating new processes with `fork(2)`, or using `select(2)`.

If you implement non-blocking, include the line "`#define NONBLOCKING`" in your code. Otherwise, this functionality will not be tested/marked.

5 Marking Criteria

The marks are broken down as follows.

The column "In CI" specifies how many of the marks allocated for this are tested by the continuous integration system available before submission. If you pass these in CI, you can be confident of getting those marks. There are also tests that will only be run on your final submission, which give the remaining marks.

Task # and description	Marks	In CI
1. Client correctly finds module on server	2	1
2. Remote procedure is called correctly	2	1
3. Results are correctly returned to client	2	1
4. Supports multiple procedures	2	1
5. Portability and safety	2	1
6. Build quality	1	1
7. Quality of software practices	2	0
8. Planning task and protocol description, in `answers.txt`	1	0
9. Stretch goal. Non-blocking operation works	1	0.5

The Continuous Integration (CI) tests will cover at least half of the marks for code execution (tasks 1–6 and the stretch goal). If you pass all CI tests, and pass tasks 7–8 then you will pass.

Code that does not compile and run on your *cloud* VM will usually be awarded zero marks for parts 1–6. Use the Git continuous integration (CI) infrastructure to ensure your submission is valid. *This is very important. Nearly every year, someone gets code working on their own computer but it fails on the cloud. Please push your code to Git regularly, and check the CI output.*

Your submission will be tested and marked with the following criteria:

Task 1. Client correctly finds module on server Your protocol causes the client to ask the server for information on a remote function. If the function exists on the server, the server replies, and results in a valid data structure at the client being created, ready for Task 2. If an unregistered function is requested, `NULL` should be returned.

Task 2. Remote procedure is called correctly The remote procedure is called. These marks are awarded even if an error occurs causing the result not to be received by the client.

Note that a single remote procedure may be called multiple times.

Task 3. Results are correctly returned to the client The client receives the correct result and continues execution. This should not result in any memory overflow, even if a large block of data is returned.

Task 4. Supports multiple procedures Allows multiple calls to `rpc_register` with different function names. The protocol must indicate which function is being called by `rpc_call`, and `NULL` if an unregistered function is passed to `rpc_find` or an invalid handle is passed to `rpc_call`.

Task 5. Portability and safety The server component of the RPC system return errors on failure and does not crash after `listen(2)`, e.g., due to malformed input, unexpected termination of connection etc. handler loading, but client disconnect

The client component of the RPC system return errors on failure, e.g. cannot connect to server, if the server shuts down in the middle of an operation etc. check input

If you implement timeouts (not a requirement), the timeout duration must be longer than 30 seconds.

Data is sent in network byte order, regardless of whether the client and server are big-endian or little-endian.

(Note that the testing system will check this by snooping on packets. Encryption or compression will break this. If you use either of those, please comment it clearly in your code so that it can be tested by hand.)

server input: port, srv, handler name, handler

client input: addr, port, cl, name, handle, payload

Task 6. Build quality Running `make clean && make -B` should produce an object file `rpc.o` or static library `rpc.a`, which contains everything needed for the RPC system, and will be linked to our test client and server code. Include in your `Makefile` (in 1 line) a `LDFLAGS` variable describing any linker flag(s) that your submission requires.

If this fails for any reason, you will be told the reason, and be allowed to resubmit (with the usual late penalty). If it still fails, you will get 0 for Tasks 1–5 and the stretch goal. Test this by committing regularly, and checking the CI feedback. (If you need help, ask on the forum.)

A 0.5 mark penalty will be applied if compiling using "`-Wall`" yields a warning or if your final commit contains any executable, `.o`, or `.a` files.

Task 7. Quality of software practices Factors considered include **quality of code**, based on the choice of variable names, comments, formatting (e.g. consistent indentation and spacing), structure (e.g. abstraction, modularity), use of global variables, proper management of memory including not leaking memory, and **proper use of version control**, based on the regularity of commit and push events, their content and associated commit messages. Profanity or abuse in the commit messages will also result in mark deductions; everyone gets frustrated, but commits must remain professional.

Task 8. Answers to questions The answers in `answers.txt` should be short (perhaps one or two lines each) and clear. The justifications given are more important than the choices made.

Stretch goal. Non-blocking operation works The server starts remote procedures as soon as a request is received, and return in the order in which they complete, which may be different from the order in which requests arrive.

Since RPC calls are blocking at the client, this will only occur in the case of multiple clients or a multithreaded client. You may assume that the client is single threaded, or at least has locks around the RPC interface, so that within a transport layer connection – responses are in the order of the requests.

Your RPC system should be able to support at least 10 concurrent clients.

6 Submission

All code must be written in C (e.g., it should not be a C wrapper over non C-code) and cannot use any external libraries, except standard libraries as noted below.

You can reuse the code that *you wrote* for your other *individual* projects if you clearly specify when and for what purpose you have written it (e.g., the code and name of the subject, project description and date, that can be verified if needed). You may use code which we have provided for practicals. You may use `libc` and `POSIX` functions (e.g., to print, create sockets, manipulate threads etc.). Your code must compile and run on the provided VMs.

The repository must contain a `Makefile` which produces an object file `rpc.o` or static library `rpc.a`. The repository must contain all source files required for compilation.

Place the Makefile at the root of your repository, and ensure that running `make` places the requested files there too. *If the GitLab CI does not find these files, the marking system will not either.*

Ensure that your RPC system does not write to `stdout`. However, the client and server can.

The server component of the RPC system should not shut down by itself. `SIGINT` (like CTRL-C) will be used to terminate the server between test cases. You may notice that a port and interface which has been bound to a socket sometimes cannot be reused until after a timeout. To make your testing and our marking easier, please override this behaviour by placing the following lines before the `bind()` call:

```
int enable = 1;
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0) {
    perror("setsockopt");
    exit(EXIT_FAILURE);
}
```

Make sure that all source code is committed and pushed to git. **Executable files** (that is, all files with the executable bit which are in your repository) **will be removed** before marking. Hence, ensure that none of your source files have the executable flag set. (You can verify this by cloning your repo onto your VM, and using `ls -l`; only directories should have "x" flags.)

For your own protection, it is advisable to commit your code to git at least once per day. Be sure to `push` after you `commit`. The git history may be considered for matters such as special consideration, extensions and potential plagiarism. Your commit messages should be a short-hand chronicle of your implementation progress and will be used for evaluation in the Quality of Software Practices criterion.

You must **submit the full 40-digit SHA1 hash** of your chosen commit to the **Project 2 Assignment** on LMS. You must **also push your submission** to the repository named `comp30023-2023-project-2` in the subgroup with your username of the group `comp30023-2023-projects` on `gitlab.eng.unimelb.edu.au`.

You will be allowed to update your chosen commit. However, only the last commit hash submitted to LMS before the deadline will be marked without late penalty.

You should ensure that the commit which you submitted is accessible from a fresh clone of your repository. For example (below, the `...` are added for clarity to break the line):

```
git clone https://gitlab.eng.unimelb.edu.au/comp30023-2023-projects...
          /<username>/comp30023-2023-project-2
cd comp30023-2023-project-2
git checkout <commit-hash-submitted-to-lms>
```

Late submissions will incur a deduction of 2 mark per day (or part thereof).

Extension policy: If you believe you have a valid reason to require an extension, please fill in the form accessible on Project 2 Assignment on LMS. Extensions **will not be** considered otherwise. Requests for extensions are not automatic and are considered on a case by case basis.

7 Testing

The skeleton available from `https://gitlab.eng.unimelb.edu.au/comp30023-2023-projects/project2` gives a simple client and server. You can modify them to test your system more thoroughly.

Continuous Integration Testing: To provide you with feedback on your progress before the deadline, we will set up a Continuous Integration (CI) pipeline on GitLab.

Though you are strongly encouraged to use this service, the usage of CI is not assessed, i.e., we do not require CI tasks to complete for a submission to be considered for marking.

Note that test cases which are available on `GitLab` are not exhaustive. Hence, you are encouraged to write unit and integration tests to further test your own implementation.

The requisite `.gitlab-ci.yml` file will be provided and placed in your repository, but is also available from the `project2` repository linked above.

Please, please use this CI feature. Almost all failed projects come from not fixing bugs that are reported by CI.

8 Getting help

Please see Project 2 Module on LMS.

9 Collaboration and Plagiarism

You may discuss this project abstractly with your classmates but what gets typed into your program must be individual work, not copied from anyone else. Do **not** share your code and do **not** ask others to give you their programs. The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program,

point out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information.

Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. You should not post your code to any public location (e.g., `GitHub`) until final subject marks are released.

If you use a **small** amount of code not written by you, you must attribute that code to the source you got it from (e.g., a book or Stack Exchange).

Do **not** post your code on the subject's discussion board Ed.

Plagiarism policy: You are reminded that all submitted project work in this subject is to be your own individual work. Automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student concerned.

Using git properly is an important step in the verification of authorship. We should see the stages of your code being written, not just the finished product.

AI software such as ChatGPT can generate code, but it will not earn you marks. You are allowed to use AI tools, but if you do then you must strictly adhere to the following rules.

1. Have a file called `AI.txt`.
2. That file must state the query you gave to the AI, and the response it gave.
3. You will only be marked on the *differences* between your final submission and the AI output.

   If the AI has built you something that gains you points for Task 1, then you will not get points for Task 1; the AI will get all those points.

   If the AI has built you something that gains no marks by itself, but you only need to modify five lines to get something that works, then you will get credit for identifying and modifying those five lines.
4. If you ask a generic question like "How do I convert an integer to network byte order?" or "What does the error '`implicit declaration of function rpc_close_server`' mean?" then you will not lose any marks for using its answer, but please report it in your `AI.txt` file.

If these rules seem too strict, then do not use the AI tools.

These issues are new, and this may not be the best policy, but it is this year's policy. If you have suggestions for better rules for *future* years, please mention them on the forum.

Good luck!