CS-3411 Program V : Remote Procedure Call

Fall 2023

In this program, we develop a mini *Remote Procedure Call* (RPC) system that consists of a *server* and a *client*. The client implements procedures that tells the server to execute certain system calls. The client and server must support the following RPC system call variants: `rp_open()`, `rp_close()`, `rp_read()`, `rp_write()`, and `rp_lseek()`.

A *user* program can include the client to gain access to these remote system calls. The client facilitates communication with the server such that, from the perspective of the user, the remote system calls appear as if they are executing like normal system calls. However, the actions of the system calls are being performed on the server rather than locally.

Consider the following example. The user includes the client in its program. With these remote system calls now available, the user program chooses to call the procedure `rp_open("/user/my_file.txt", O_RDONLY)`. As this function is implemented in the client, the user is agnostic to all but the return value of `rp_open()`. In the client's implementation of `rp_open()`, the request type (`OPEN_CALL`) and the arguments (`"/user/my_file.txt"` and `O_RDONLY`) are packaged and sent over the network to the server. Receiving this request, the server unpackages the data and performs the specified system call with the provided arguments. The server then takes the return value of the system call, packages it, and sends it to the client. Now the client can unpackage the data, and provide a return value to the user through the `rp_open()` procedure call.

Remote Procedure Prototypes

You must make the following remote procedures available to the user:

- `int rp_connect(const char *hostname, unsigned short port)`: This procedure initiate the connection to the server specified by `hostname` and `port`. The user should only need to call this procedure once for the duration of the interactions with that server, but the other remote procedures will not work until this procedure is called.
- `int rp_open(const char *pathname, int flags, ...)`: The remote variant of the `open()` system call. This procedure sends its arguments to the server which calls the `open()` system call. The result of the call is sent back to this procedure and returned to the user. Just as with the standard library local variant, the `mode_t mode` can also be specified (think in conjunction with the `O_CREAT` flag). The "`...`" specifies a variable number of arguments, allowing `mode` to be an optional argument. Functions of this type are called *variadic* functions.
- These procedures transmit their arguments and call code to the server, which invokes the standard library local variant of the system call. The resulting value from this call is subsequently dispatched to our procedure, and serves as its own return value to the user.
  - `int rp_close(int fd)`
  - `ssize_t rp_read(int fd, void *buf, size_t count)`
  - `ssize_t rp_write(int fd, const void *buf, size_t count)`
  - `off_t rp_lseek(int fd, off_t offset, int whence)`
- `short rp_checksum(int fd)`: This procedure returns the checksum (an `unsigned char`) of the file corresponding to the file descriptor `fd` argument. Starting at the beginning, read the file in blocks of size 4096 bytes. Calculate a single, running checksum value using XOR. On success, the checksum value is returned; on error, -1 is returned.

User

The user program includes the client header file (i.e. `rp_client.h`) to access the remote system calls. There are two user programs to develop:

1. The first is a copy program which opens a file on the remote server and a file locally. The remote file is then read and written to the local file, thereby creating a copy from the server to the user. Implement this in the provided `rp_user1.c` file.
2. The second user program is a seek and copy. Similarly, open a file on the remote server, but seek forward 10 bytes from the beginning of the file. Then, open a local file and again initiate a copy as before. Implement this in the provided `rp_user2.c` file.

The `rp_connect()` procedure must be called before any other remote procedure. The procedure must be supplied the `hostname` and the associated `port`. You may, for example, pass the `hostname` and `port` through the arguments list `./rp_user1 <hostname> <port>`.

Once the connection is established, the remote procedures can be called. For example, call `open()` for the local variant and `rp_open()` for the remote variant.

Client

The client consists of `rp_client.h` and `rp_client.c` which contain the remote procedure function prototypes and their definitions respectively. There is no `main()` function in these files, that comes from the user program which includes `rp_client.h`. The development of the remote procedures in `rp_client.c` is broken up into two parts:

Part 1: For the first part, implement the remote procedures in `rp_client.c` by using the local system call variants. In this way, the `rp_*()` procedures will simply be a wrapper for the standard system call. With `rp_open()`, for example, simply call `open()` and return the result to the user. Following error conventions, set the `errno` variable from the `<errno.h>` library appropriately. As per the standard system calls, also return a value on error. Taking `open()` again as an example, -1 is returned on error and `errno` is set appropriately. See the RETURN VALUE section of the man page.

Part 2: After validating the first part, replace the local system calls with server interactions. First, the arguments of the remote procedure along with the calling code are sent to the server for processing. The server sends back the result/`errno`/data to the procedure which should be returned to the user as is appropriate.

Server

The server consists of the `rp_server.c` file which contains the remote procedure function handlers. As the server is an independent process, it has a `main()` function for handling user connections and processing user requests.

Part 1: After completing parts 1 and 2 of the client, start implementing the server to be able to handle user connections and requests sequentially:

1. Create the socket using IPv4, TCP/IP, and 0 for the protocol.
2. Create the socket address `struct`. Use IPv4 as the domain, `INADDR_ANY` or 0 for the address, and 0 for the port.
3. Bind the socket to the address `struct`. The 0 address binds to all local interfaces, and 0 for the port requests a random but available port.
4. Listen on that socket for incoming requests and specify an appropriate `backlog`.

5. Accept an incoming connection. Backlog the remaining connections until the current one is closed.
6. Process a connected user's request. Read the calling code and call the appropriate request handler.
7. Within the handlers, sequentially read the arguments from the connection socket. Pass these arguments into the appropriate system call and send the result/`errno`/data back to the client.
8. Close the current connection upon receiving an EOF from the connection. If you're using the provided `read_from_client()` function, `NULL` is returned as EOF.

Part 2: Now, modify the server such that multiple user connections and requests can be handled simultaneously:

1. Call `fork()` after accepting a connection such that each connection has a dedicated child. The parent process simply accepts the connections as they come in and fork in an infinite loop.
2. Each child closes any unnecessary file descriptors leftover from the fork call.
3. Each child handles the requests that come from its dedicated connection.
4. The child terminates after receiving EOF.

Template Files

The following template files are provided for you on Canvas. You do not need to use the provided template files, but using them should make the program significantly easier.

- `rp_calls.h`: This header is included by `rp_client.c` and `rp_server.c` and defines the calling code constants for the remote operations: `OPEN_CALL`, `CLOSE_CALL`, `READ_CALL`, `WRITE_CALL`, `LSEEK_CALL`, and `CHECKSUM_CALL`. You do not need to modify this file, but you want to use these constants.
- `rp_procs.h`: This header is included by `rp_client.h` and defines the function prototypes for the remote procedures that you need to implement in `client.c`. You do not need to modify this file.
- `rp_client.h`: This header is included by `rp_user1.c` and `rp_user2.c` to give these files access to the remote procedure functions defined in `rp_client.c`.
- `rp_client.c`: The remote procedures are implemented here. If you want to use the provided abstractions `void* read_from_server()` and `int send_to_server(void *data, size_t size)`, ensure you use the `static int sock_fd` global variable.
- `rp_server.c`: This is the independent server program. Only `rp_calls.h` is included from the template files. Note the function required by the autograder.
- `rp_user1.c` and `rp_user2.c`: Implement the first and second user programs here. Only `rp_client.h` is included from the template files.
- `Makefile`: The provided `Makefile` includes the required functionality to make all of the provided template files. Simply call `make` or `make all` as normal. There is also `make submission` and `make clean`. The `-g` flag is enabled by default. There is other functionality implemented for testing purposes.

Testing

For testing purposes, the `Makefile` has additional functionality. Run `make mixed`, `make remote`, or `make local` to configure the compilation process used for `rp_user1.c` and `rp_user2.c`. Any `make` call will compile all of the template files. Calling just `make` will call the default option which is `mixed`.

- `make mixed`: Using `mixed` allows for both the local standard library system call variants and the `rp_*()` remote variants to exist simultaneously.
- `make remote`: Using `remote` will configure all standard library system calls to use their remote procedure variants. The `rp_connect()` function is still required.
- `make local`: Using `local` leaves the standard library system calls alone. The `rp_connect()` and `rp_checksum()` functions are essentially ignored and simply return 0. Any other `rp_*()` functions give an error.

Consider this example where `rp_user1.c` calls the functions `rp_connect()` and `open()` within its `main()` function. Compiling with `make remote` interprets the `open()` call as an `rp_open()` call. Compiling with `make local` interprets the `open()` call as an `open()` call and ignores the `rp_connect()` function. Compiling with `make mixed` interprets both the standard library system calls and the remote procedure functions as they are such that, for example, `open()` and `rp_open()` can exist in the same file.

The ability to switch between remote and local modes is useful for testing purposes as you can quickly see the different between running a test program locally vs running it remotely. Therefore, `make remote` and `make local` are useful for initial testing, and `make mixed` is used for the final implementation of the `rp_user*.c` programs.

Requirements

Here is a list of additional requirements:

1. You must write the program in C.
2. You do not need to use the provided templates, but the server must write its port to a `.server_port` file for the autograder. This functionality is already implemented for you in the `rp_server.c` file.
3. If any error is detected, the program should print an appropriate error message and exit.
4. Do not have any memory leaks in your program. For example, the provided `void* read_from_server()` function returns malloc'ed data that you must `free()`. Run your executables through `valgrind` to check for memory leaks.
5. Do not have any GCC warnings when compiling with `-Wall -Wextra -pedantic`.
6. Calling either `make` or `make all` in the submission directory should correctly compile the `rp_user*.c` files and the `rp_server.c` file. Assuming you follow the provided template files, this functionality already exists.

Submission

Run `make submission` using the provided `Makefile` and all of the template files will be included in `prog5.zip`. Assuming you do not create any additional source files, you can turn this zip file into Canvas as is.