ECE 454/750T10, Spring 2014 — Assignment 1
Due Mon, June 2, 11:59:59 PM

(Follow the submission instructions at the end. If you are in 454, mention the names of both group members in the README.)

1.(100 points) Objective: Realize a simplified RPC mechanism.

You are to exploit C programming language features to realize a simplified RPC mechanism. Specifically, you should provide a server-stub and a client-stub so that an application programmer can make functions available via the server-stub, and another application programmer can invoke functions via the client-stub. The file `ece454rpc_types.h` that is in this module on Learn contains the signatures of all the 3 functions you to implement. The first three go into the server-stub and the last one goes into the client-stub.

Stuff you need to realize

Here are the three functions you need to realize. Functions (1)–(2) are for the server-stub. Function (3) is for the client-stub.

1.
```
bool register_procedure(const char *procedure_name,
                        const int nparams,
                        fp_type fnpointer);
```

   The application programmer that uses your stuff for providing the functions that can be invoked remotely first invokes `register_procedure()` to register his function with your server-stub. That is, your server-stub maintains a simple database of all the functions that are registered with it. The database is looked up by name, i.e., the argument `procedure_name`. Every function that is registered has the same signature, which is indicated by the `fp_type`, which is defined in `ece454rpc_types.h`.

   ```
   typedef return_type (*fp_type)(const int, arg_type *);
   ```

   The "`fp`" in `fp_type` stands for "function pointer." The way the server-stub can locate the application programmer's function to be called is using the function pointer `fnpointer`. What the above typedef says is that `fp_type` is a pointer to a function that takes as arguments a `const int` and `arg_type *`, and returns something of type `return_type`. Both `arg_type` and `return_type` are also defined in `ece454rpc_types.h`.

   The function `register_procedure()` returns true on success and false on failure.

2.
```
void launch_server();
```

   The call to `launch_server()` starts the server listening for requests for function calls, for functions that are registered with it via `register_procedure()`. It never returns. That is, all is does is wait for a client request, service it, and then go back to waiting for more client requests. And it does this forever.

   IMPORTANT: the first thing a call to `launch_server()` should do is print out, to `stdout` (e.g., using `printf()`) the IP address/domain name and UDP port number on which the server runs. This is so that a client can be informed as to how to reach the server.

Here is an example of the output we expect to see on `stdout` when `launch_server()` is invoked:

```
ecelinux3.uwaterloo.ca 5764
```

Your printout should look exactly like this (except that you can print out an IP address instead of domain name). That is, you should not embellish this printout in any way. Otherwise, our marking script that looks for this will break, and you will get an automatic 0 for the assignment.

3.
```
return_type make_remote_call(const char *servernameorip,
                             const int serverportnumber,
                             const char *procedure_name,
                             const int nparams,
                             ...);
```

This is used by the client application program to invoke a remote method. So you provide the implementation of `make_remote_call()` in the client-stub. It is a function that takes a variable number of arguments, as indicated by the `...` at the end. The variable number of arguments correspond to pairs of $\langle$`arg-size`, `arg`$\rangle$, where `arg-size` is the size (in bytes) of the argument, and `arg` is the argument, which must be of type `void *`. The number of pairs must be the value of the `nparams` argument. If there is any kind of error, a `return_val` of NULL and a `return_size` of 0 should be returned to the client application.

Sample server application

Following is a sample server that would use your implementation. This is available as part of this module in Learn.

```
#include <stdio.h>
#include "ece454rpc_types.h"

int ret_int;
return_type r;

return_type add(const int nparams, arg_type* a)
{
    if(nparams != 2) {
        /* Error! */
        r.return_val = NULL;
        r.return_size = 0;
        return r;
    }

    if(a->arg_size != sizeof(int) ||
       a->next->arg_size != sizeof(int)) {
        /* Error! */
        r.return_val = NULL;
        r.return_size = 0;
        return r;
    }
```

```
    int i = *(int *)(a->arg_val);
    int j = *(int *)(a->next->arg_val);

    ret_int = i+j;
    r.return_val = (void *)(&ret_int);
    r.return_size = sizeof(int);

    return r;
}

int main() {
    register_procedure("addtwo", 2, add);

    launch_server();

    /* should never get here, because
       launch_server(); runs forever. */

    return 0;
}
```

Sample client application

The following is a sample client application. This is available as part of this module in Learn.

```
#include <stdio.h>
#include "ece454rpc_types.h"

int main()
{
    int a = -10, b = 20;
    return_type ans = make_remote_call("ecelinux3.uwaterloo.ca",
                                       5673,
                                       "addtwo", 2,
                                       sizeof(int), (void *)(&a),
                                       sizeof(int), (void *)(&b));
    int i = *(int *)(ans.return_val);
    printf("client, got result: %d\n", i);

    return 0;
}
```

Transport

Obviously when the client app makes the `make_remote_call()`, you'll need to translate the call to a network message, and send it over the next to the server. Use UDP to do this. To use UDP, you need to use the `sock()` API. Type "man 2 socket" and "man 7 ip" in ecelinux. There should also be tons of examples

online on how to use the socket API to create a datagram (UDP) socket, and use it to send packets back and forth between a client and server. If you use one of these online sources, you should credit it appropriately in the comments of your code.

If I were you, I would start by first getting this transport to work. Then, you should be able to build the actual 3 functions that are required on top of it.

Memory management

You (and we, the users of your stuff) need to be careful about memory management, as several of you have noticed and talked to me about. Here are the rules you can follow:

— when your server stub invokes the function in our server application, your server stub owns the memory for the arguments. That is, your stub somehow constructs the instance of `arg_type *` and passes it to the invoked function. The invoked function treats it as a value parameter. Therefore, up on return, you should free that memory if it was allocated dynamically.

— when we return a `return_type` to your server stub, we no longer have control of it. **However,** you should not worry about freeing anything for this assignment. If you look at my simple example, you'll notice that all memory has been allocated statically.

– when our client code calls your client stub with `make_remote_call()`, you can assume that our code owns the memory for the arguments. So you do not have to free that memory.

— when you return a result from your client stub to our client application code, you can assume that you are handing over the memory to us, and not worry about freeing it.

Makefile

You must include a Makefile with your submission. It must have at least two targets: libstubs.a and clean. The latter target should remove all object, archive (.a) and executable files from the folder. That is, only source-code, the Makefile and the README must remain.

The target libstubs.a should build `libstubs.a`, which comprises all the 3 functions for the client and server stubs. See the sample Makefile I have included. In my Makefile, we first compile the server and client stubs to their respective object files, `client_stub.o` and `server_stub.o` respectively. We then use the `ar` command to create an archive (library) called `libstubs.a`. The client and server application programs can now link against `libstubs.a` to get access to the three functions you implement.

Coding Standard

Your code must follow a programming style that you can choose freely; but you have to explicitly specify which one in comments in your code towards the start. It is important that the code is easy to read, with meaningful variables names. Example coding conventions are at `https://code.google.com/p/google-styleguide/`.

Submission Instructions

- Your submission is made to the appropriate dropbox on Learn.
- It must be a single zip file. You can use the "`-r`" option to `zip` to recursively zip a folder.

- The name of your submission must be: `ece454a1.zip`. Our script will look for this, and if it does not find it for you, you get an automatic 0 (see marking scheme below).
- Unzipping your `ece454a1.zip` must result in a single folder called `ece454a1`, that has no sub-folders. All your source-code, Makefile and README must be in that single folder.
- Include a README file with the names of your two group-members if you are in 454. If you are in 750T10, then of course you need to do this assignment on your own. Include your name only in the README.

Evaluation

Our grading scheme:

a) Marking script breaks, e.g., because submission not structured as per instructions, makefile does not work, program crashes, etc. — automatic 0.

b) 5% if you implemented something meaningful that compiles and runs.

c) 30% if you implemented something meaningful that compiles and runs, and we are able to write simple server and client applications, and everything works.

d) 90% everything works for our more complex tests. E.g., we try multiple function registrations.

e) 100% if, in addition to meeting the functional requirements, your code is pretty and well-documented.