Ryan Lanciloti

CPRE 550

Project 1 Design Document

**Purpose:**

RPC, or remote procedure call, is a mechanism by which a user can transparently execute code on a remote machine. What this means is that the user can make a normal function call, but behind the scenes, this function is actually executed on another machine and the result is sent back the host machine as if the host was the one whom ran the function.

One possible use would be for a company like google to write an image recognition library which provides the developer a means of doing image recognition without actually doing image recognition. They would pass in their image to a function, and that function would send the image to a backend server completely hidden and abstracted away from the developers code and return the result. While very powerful and very useful, this would be an example of offloading CPU or GPU intensive work to a machine tailored to do this specific task. However, there’s another powerful use for RPC.

Instead of offloading work to another machine due to resource constraints, there’s also the case where we want to offload work because we want something from another machine. This would be the resource monitoring case, and the purpose of this project. RPC is an interesting mechanism it user here because it makes the function call transparent, so when we are making a call to, lets say get the CPU utilization, it may seem to the developer that they are getting the host machine’s CPU utilization instead of a remote machine’s CPU utilization. However, in this project we are not developing a library for other developers, and so this is not an issue.

In project 1, I will be developing an application which will query a remote system’s current time, CPU utilization, memory usage, loaded processes per minutes, and a list of users on the remote machine.

**Method:**

To develop this application, I will take the following steps:

1. Write an sys\_monitor.x (Remote Server Monitor) file which will contain the interface definitions for each of the remote functions.
2. Using the `rpcgen` command, generate the boiler plate code surrounding the RPC portion of the code.
3. Write a client application which will run on the host machine and take in user input to determine which functions to run and when.
4. Write a server application with concrete implementation of the remote functions. These will gather data from the remote machine and send the information back to the user.

**Structs:**

struct sysinfo {

    long uptime;              /\* Seconds since boot \*/

    unsigned long loads[3];   /\* 1, 5, and 15 minute load averages \*/

    unsigned long totalram;   /\* Total usable main memory size \*/

    unsigned long freeram;    /\* Available memory size \*/

    unsigned long sharedram;  /\* Amount of shared memory \*/

    unsigned long bufferram;  /\* Memory used by buffers \*/

    unsigned long totalswap;  /\* Total swap space size \*/

    unsigned long freeswap;   /\* swap space still available \*/

    unsigned short procs;     /\* Number of current processes \*/

    char \_f[22];              /\* Pads structure to 64 bytes \*/

};

typedef struct {

    struct mallinfo dynamic\_memory\_usage;   // Struct containing info about dynamic memory usage

    int page\_size;                          // Page size of the remote machine

    long phys\_page\_cnt;                     // Number of physical pages on the remote machine

    long available\_phys\_page\_cnt;           // Number of free physical pages on the remote machine

} mem\_usage\_ret;

**Enums:**

typedef enum {

    DATE,       // For getting only the date

    TIME,       // For getting only the time

    DATE\_TIME,  // For getting both the date and time together

    \_OPTION\_CNT // For getting the number of options in the enum, not

                // meant to be an option

} dt\_ops\_t;

typedef enum {

    ONE\_MINUTE,     // Maps for the 1 minute entry

    FIVE\_MINUTE,    // Maps for the 5 minute entry

    FIFTEEN\_MINUTE  // Maps for the 15 minute entry

} load\_time\_map\_t;

**Remote Functions:**

/\*\*

 \* FUNCTION: GetDateTime

 \*

 \* PARAMETERS:

 \*  dt\_ops\_t params: This parameter determines what is returned from the remote server.

 \*      Examples for each option can be found below.

 \*

 \* DESCRIPTION:

 \*  When called, this function will return either the date, time, or both from a

 \* remote machine.

 \*

 \* EXAMPLE: GetDateTime(DATE) -> February 6, 2023

 \* EXAMPLE: GetDateTime(TIME) -> 10:47:21

 \* EXAMPLE: GetDateTime(DATE\_TIME) -> February 6, 2023 - 10:47:21

 \*

 \* RETURNS: Returns a string representing the date time.

\*/

char\* GetDateTime(dt\_ops\_t params);

/\*\*

 \* FUNCTION: GetSystemInfo

 \*

 \* DESCRIPTION:

 \*  When called, this function will get a number of statistics about the remote CPU in the

 \*  form of the sysinfo struct defined earlier.

 \*

 \* RETURNS:

 \*  A struct containing information about the remote CPU. Description for what is

 \*  in the struct can be found above.

\*/

struct sysinfo GetSytemInfo();

/\*\*

 \* FUNCTION: GetMemoryUtilization

 \*

 \* DESCRIPTION:

 \*  When called, this function will get various statistics about memory utilization on the

 \*  remote computer.

 \*

 \* RETURNS:

 \*  A struct containing information about the remote computer's memory utilization. A full

 \*  description of what is in the struct can be found above.

\*/

mem\_usage\_ret GetMemoryUtilization();

/\*\*

 \* FUNCTION: GetLoadProcsPerMinute

 \*

 \* DESCRIPTION:

 \*  When called, this function will get the 1, 5, and 15 minute load averages of the remote

 \*  system.

 \*

 \* RETURNS:

 \*  A 3-element double array containing the 1, 5, and 15 minute load averages of the remote

 \*  system.

\*/

double\* GetLoadProcsPerMinute();

**Client Application:**

The client application will have a number of commands it can run. Some of the commands will also have subcommands. The top-level commands are the following:

1. Server
2. DateTime
3. SystemInfo
4. MemoryUsage
5. LoadInfo
6. Quit

Both **Server** and **DateTime** will have subcommands:

**Server**:

1. List
2. Add
3. Remove
4. Select

**DateTime:**

1. Date
2. Time
3. Both

The client will use console input from the user to determine what commands to execute and with what parameters, if any.

**Conclusion:**

With the data returned by these functions, I can output to the client the desired information in an easily digestible format. The RPC wrapper generated by `rpcgen` should handle the heavy lifting in terms of handling multiple clients or multiple servers.