Ammar Khan
Feb 19, 2023
CPRE 550
Project 1: RPC

Introduction

The object of this project is to write a network management application that tracks user logins and host system statistics such as CPU Usage, memory usage, processes loaded per minute and the current date time. We are required to have a client server setup where the client and server communication is all RPC based. To tackle this project, I am going to use rpc-gen to generate the rpc stubs. This requires me to draft up an input file which is a remote program interface definition written in the RPC language. This input file provides the basic program definition for the rcp generator, such as the function headers of any function stubs need to be generated as well as any special data types or structs that need to be generated. This file will be the basis of my design analysis below.

Proposed Design Overview

You can see the contents of my input file below…

```
program NETWORK_MNGR_PROG {
    version NETWORK_MNGR {
        string DATE(long) = 1;
        double CPU_USAGE(void) = 2;
        double MEM_USAGE(void) = 3;
        double LOAD_PROCS_PER_MIN(void) = 4;
        string USER_LOGINS(void) = 5;
    } = 1;                          /* version number = 1     */
} = 0x31234567;            /* program number        */
```

Figure 1: RPC-GEN Input File

For my program, I chose to define five functions. The first function I have defined is the *DATE* function which takes in a long and returns a string. This function is largely unchanged from the function in the RPC sample code provided so I will not spend much time discussing it.

The second function is the *CPU_USAGE* function which takes in no inputs and returns a double which is meant to be the current cpu usage as a percentage. For example, this function would return 50.31 to denote that 50.31% of the CPU is currently being used.

The third function is the *MEM_USAGE* function which takes in no inputs and returns a double which is meant to be the current memory usage as a percentage. For example, this function would return 93.21 to denote that 93.21% of the main system memory is currently being used.

The fourth function is the *LOAD_PROCS_PER_MIN* function which takes no inputs and returns a double representing the load averages over the last minute. In other words, this represents the average load of processes in the run queue or waiting for disk I/O over the last minute.

The last function is a function meant for listing out all of the user logins in to the host system. This function will take no parameters and will return a string as a response. I will go into greater detail as to how I implemented each function in the section below, but will provide a brief overview as to how each of these functions work.

Implementation

In this section I will delve into a more in depth analysis of how I will actually implement each of the functions listed above. As I mentioned previously, the Date function is largely unchanged from the sample code provided so I will exclude it from this review.

Client

In this system, the client does not actually calculate or record any of the system statistics. It simply queries the Server and then outputs the response. As in the rpc sample code posted on Canvas, the client in my application prompts the user to select a number 1 through 8 depending on what query they want to execute. Options 1 through 3 are all various date/time formats, and options 4 through 6 represent CPU usage, Memory usage and Load Procs Per Min respectively. Finally, option 7 will list the usernames of users authenticated with the server and option 8 will quit the client task. Each respective option will query the corresponding rpc stub, all of which are defined below, and captures the pointer value returned and outputs it to the client.

Server

The server side is where all of the function stubs defined in the input file live, and I will break down how each function is calculating the necessary data as well as document any transformations we need to perform on the data before sending back to the client(s).

CPU_USAGE

I found that there was no one library that provided the current cpu usage for a system, so I had to find a way to calculate the host cpu usage manually. I accomplished this by reading the the “proc/uptime” file. As shown in the Red Hat documentation, this file outputs two numbers in the format below …

```
350735.47 234388.90
```

Figure 2: proc/uptime

The first number here represents the total number of seconds that the system has been up, whereas the second number represents the total number of seconds the system has been in the idle state. With this information in mind, I devised the following process for calculating the CPU usage at a given time…

1. Read proc/uptime file
2. Measure Uptime and Idletime
3. close file and sleep for 1 second
4. After 1 second has elapsed, read proc/uptime file again
5. Measure uptime and idletime again
6. Use following formula to calculate percentage of time CPU was in idle during that 1 second interval….
    1. IdlePercentage = ((Idletime2 – Idltime) / (Uptime2 – Uptime)) * 100
7. CPU Usage = 100 – IdlePercentage

The process listed above will provide the percentage of time over the last second that the CPU was being used, thus providing a usage metric for how heavily our CPU is being used. An important implementation detail to note here is that while the Uptime number presented in the file represents the total uptime of the system, the Idletime represents the sum of the idle times of every core of the CPU. Therefore, in multi-core systems the idle time number will be much higher than the uptime which would obviously cause issues with this approach. The simplest workaround for this is to use of the average idle time of all the cores which can be had by taking the recorded idle time metric and dividing by the number of CPU cores, which can simply be found by including unistd.h and then leveraging the following command …

sysconf(_SC_NPROCESSORS_ONLN)

MEM_USAGE

Calculating memory usage at a given time proved to be much simpler than caclulating the CPU Usage. For this we can include the header file *<sys/sysinfo.h>* and leverage the *sysInfo* struct. This struct provides a variety of system metrics, two of which are *totalram* and *freeram*. *SysInfo.totalram* tells how much total usable main memory is present in our system, and *sysinfo.freeram* tells how much of that memory is actually free and not currently being used up. With these two metrics, calculating the percentage of memory becomes a simple exercise with the following process …

1. Record *totalram* and *freeram*
2. *PercentageMemoryFree* = (*freeram* / *totalram*) * 100
3. *PercentageMemoryInUse* = 100 – *PercentageMemoryFree*

LOAD_PROCS_PER_MIN

In order to determine the process load, there were several options such as using the *<sys/sysinfo.h>* header or using *getloadavg()* function, but I chose to use the *proc/loadavg* file. This file reports stats in the following format…

```
0.20 0.18 0.12 1/80 11206
```

Figure 3: proc/loadavg

The first of these numbers is a load average figure representing the number of processes in the run queue or waiting for disk I/O averaged over the last minute. This is not quite load processes per minute but is the closest metric I could think of that did not require sleeping and waiting for 60 seconds.

USER_LOGINS

In order to track the users logged into a server at a given time, I imported the *<utmp.h>* header. This header enables one to interface with the /etc/utmp file, which is one of the files that is written to when a user logs in. The getuent() method in particular returns the current file position in the umtp file in the following shape …

```
struct utmp {
    short   ut_type;              /* Type of record */
    pid_t   ut_pid;               /* PID of login process */
    char    ut_line[UT_LINESIZE]; /* Device name of tty - "/dev/" */
    char    ut_id[4];             /* Terminal name suffix,
                                     or inittab(5) ID */
    char    ut_user[UT_NAMESIZE]; /* Username */
    char    ut_host[UT_HOSTSIZE]; /* Hostname for remote login, or
                                     kernel version for run-level
                                     messages */
    struct  exit_status ut_exit;  /* Exit status of a process
                                     marked as DEAD_PROCESS; not
                                     used by Linux init(1) */
    /* The ut_session and ut_tv fields must be the same size when
       compiled 32- and 64-bit.  This allows data files and shared
       memory to be shared between 32- and 64-bit applications. */
#if __WORDSIZE == 64 && defined __WORDSIZE_COMPAT32
    int32_t ut_session;           /* Session ID (getsid(2)),
                                     used for windowing */
    struct {
        int32_t tv_sec;           /* Seconds */
        int32_t tv_usec;          /* Microseconds */
    } ut_tv;                      /* Time entry was made */
#else
     long   ut_session;           /* Session ID */
     struct timeval ut_tv;        /* Time entry was made */
#endif

    int32_t ut_addr_v6[4];        /* Internet address of remote
                                     host; IPv4 address uses
                                     just ut_addr_v6[0] */
    char __unused[20];            /* Reserved for future use */
};
```

Figure 4: UTMP Struct

From this struct, one can look at the ut_type short to determine what type of record one is looking at. Once a user is authenticated and logged in, a process will transition from a ut_type 6 (LOGIN_PROCESS) to a 7 (USER_PROCESS). From these user processes, we can extract the username from the ut_user field. For my implementation, I loop through all of the processes until the getuent() call does not return any more. I then loop through these processes and return the ut_types to show a list of all the active users authenticated and logged into the server.

Citiation

[1] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-uptime

[2] https://man7.org/linux/man-pages/man2/sysinfo.2.html