

Bart Brock

Dr. Lin

CSCI 531

4/23/18

Project 2, Remote Procedure Call

This is my submission for Project 2. It uses the remote procedure call (RPC) protocol to implement data passing and distributed computing. The paper will start by introducing the remote procedure call as implemented on a Linux system, giving a simple working example of how to build and run an RPC session. In part II a more complex module will be created to show the advantages of distributed computing. Here the module will use a quicksort algorithm to sort an array of random numbers using distributed computing via the RPC implementation. Data collected will be given and analyzed at the end of the paper.

The test environment for this project utilized virtual machines running Ubuntu 16.10 as both the client and server. One of the virtual machines was setup in a data center connected via Home Telecom business fiber and configured with a public IP in order to provide an over the internet connection. This was setup as the server computer and hosted on a Dell 610 server with dual Xeon processors running Microsoft Hyper-V server. The host computers used for the clients were hosted on a Dell R620 server with dual Xeon processors running VMWare ESXi and a Dell 3558 notebook with an i3 processor running windows 10 and VMWare workstation 14. The guest operating systems were Ubuntu 16.10 server configured with 1 CPU, 8GB of RAM, and 40GB hard drive and Ubuntu 16.10 desktop with 1 CPU, 1GB RAM, and 30GB hard drive respectively.

Section I Task 1

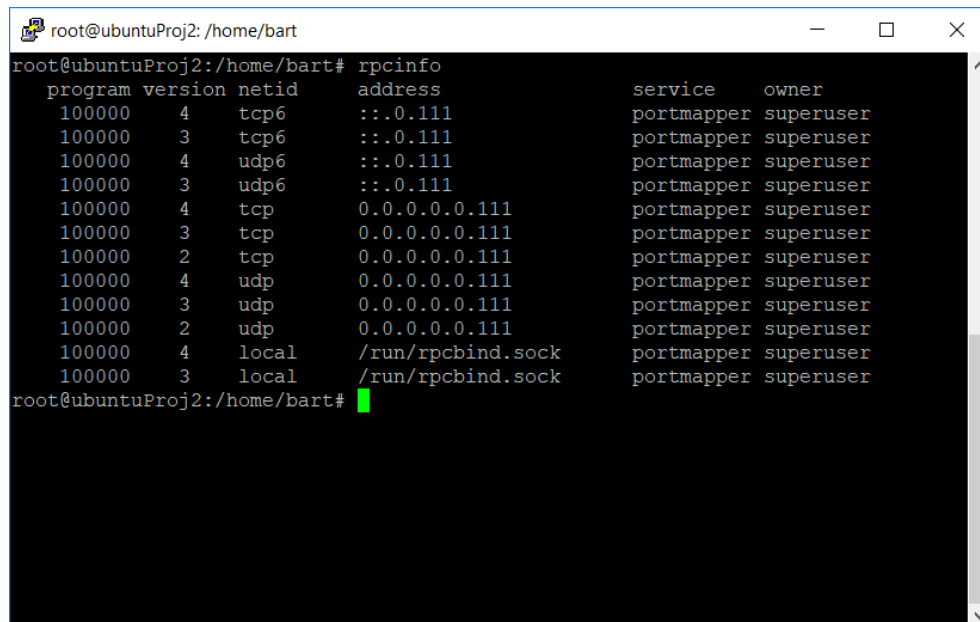
Introduction

In this section we will test the host computer to see if RPC is installed. If not, we will setup RPC on the computer.

Explanation

The following steps were performed to test for RPC installation.

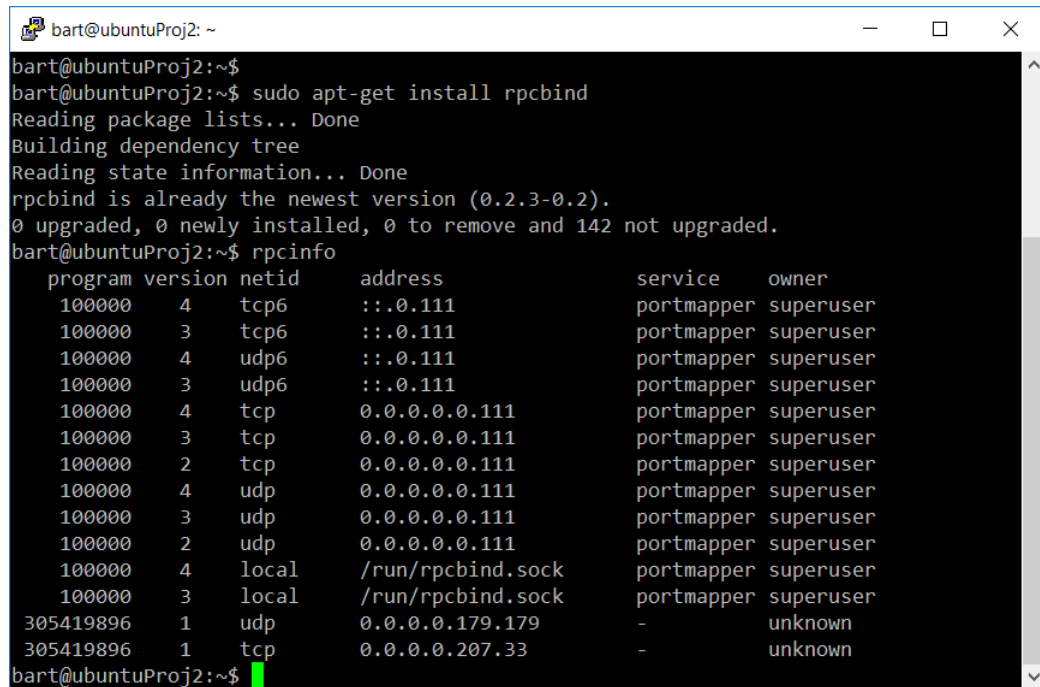
1. Run the **rpcinfo** command. On many installations this is installed by default. The output is shown in figure 1. You can see *rpcbind* is installed.



```
root@ubuntuProj2: /home/bart
root@ubuntuProj2: /home/bart# rpcinfo
  program version netid  address          service  owner
    100000     4   tcp6   :::.0.111        portmapper superuser
    100000     3   tcp6   :::.0.111        portmapper superuser
    100000     4   udp6   :::.0.111        portmapper superuser
    100000     3   udp6   :::.0.111        portmapper superuser
    100000     4    tcp   0.0.0.0.0.111    portmapper superuser
    100000     3    tcp   0.0.0.0.0.111    portmapper superuser
    100000     2    tcp   0.0.0.0.0.111    portmapper superuser
    100000     4    udp   0.0.0.0.0.111    portmapper superuser
    100000     3    udp   0.0.0.0.0.111    portmapper superuser
    100000     2    udp   0.0.0.0.0.111    portmapper superuser
    100000     4   local  /run/rpcbind.sock portmapper superuser
    100000     3   local  /run/rpcbind.sock portmapper superuser
root@ubuntuProj2: /home/bart#
```

Figure 1. rpcinfo Screen Capture

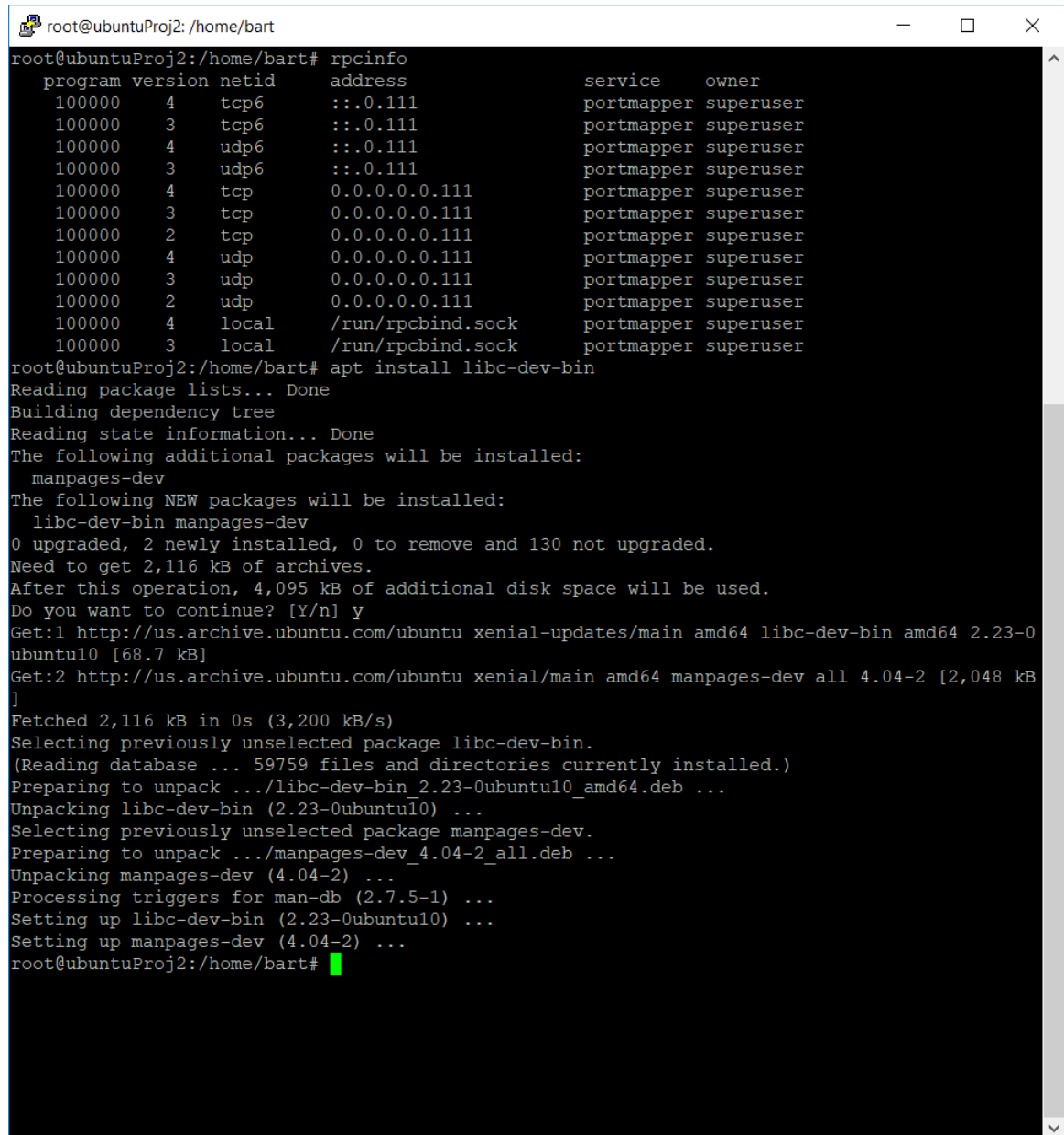
2. Even though RPC is installed we will execute the **sudo apt-get rpcbind** command to see the results. Run **rpcinfo** again to verify. This is shown in figure 2.



```
bart@ubuntuProj2: ~  
bart@ubuntuProj2:~$  
bart@ubuntuProj2:~$ sudo apt-get install rpcbind  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
rpcbind is already the newest version (0.2.3-0.2).  
0 upgraded, 0 newly installed, 0 to remove and 142 not upgraded.  
bart@ubuntuProj2:~$ rpcinfo  
program version netid address service owner  
100000 4 tcp6 ::.0.111 portmapper superuser  
100000 3 tcp6 ::.0.111 portmapper superuser  
100000 4 udp6 ::.0.111 portmapper superuser  
100000 3 udp6 ::.0.111 portmapper superuser  
100000 4 tcp 0.0.0.0.0.111 portmapper superuser  
100000 3 tcp 0.0.0.0.0.111 portmapper superuser  
100000 2 tcp 0.0.0.0.0.111 portmapper superuser  
100000 4 udp 0.0.0.0.0.111 portmapper superuser  
100000 3 udp 0.0.0.0.0.111 portmapper superuser  
100000 2 udp 0.0.0.0.0.111 portmapper superuser  
100000 4 local /run/rpcbind.sock portmapper superuser  
100000 3 local /run/rpcbind.sock portmapper superuser  
305419896 1 udp 0.0.0.0.179.179 - unknown  
305419896 1 tcp 0.0.0.0.207.33 - unknown  
bart@ubuntuProj2:~$
```

Figure 2. rpcbind Install Screenshot

3. Install the development library by entering **sudo apt install libc-dev-bin**. This output is shown in figure 3.



```

root@ubuntuProj2: /home/bart
root@ubuntuProj2:/home/bart# rpcinfo
  program version netid      address                service  owner
    100000     4   tcp6      :::.0.111             portmapper superuser
    100000     3   tcp6      :::.0.111             portmapper superuser
    100000     4   udp6      :::.0.111             portmapper superuser
    100000     3   udp6      :::.0.111             portmapper superuser
    100000     4    tcp      0.0.0.0.0.111         portmapper superuser
    100000     3    tcp      0.0.0.0.0.111         portmapper superuser
    100000     2    tcp      0.0.0.0.0.111         portmapper superuser
    100000     4    udp      0.0.0.0.0.111         portmapper superuser
    100000     3    udp      0.0.0.0.0.111         portmapper superuser
    100000     2    udp      0.0.0.0.0.111         portmapper superuser
    100000     4   local    /run/rpcbind.sock     portmapper superuser
    100000     3   local    /run/rpcbind.sock     portmapper superuser
root@ubuntuProj2:/home/bart# apt install libc-dev-bin
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  manpages-dev
The following NEW packages will be installed:
  libc-dev-bin manpages-dev
0 upgraded, 2 newly installed, 0 to remove and 130 not upgraded.
Need to get 2,116 kB of archives.
After this operation, 4,095 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://us.archive.ubuntu.com/ubuntu xenial-updates/main amd64 libc-dev-bin amd64 2.23-0ubuntu10 [68.7 kB]
Get:2 http://us.archive.ubuntu.com/ubuntu xenial/main amd64 manpages-dev all 4.04-2 [2,048 kB]
Fetched 2,116 kB in 0s (3,200 kB/s)
Selecting previously unselected package libc-dev-bin.
(Reading database ... 59759 files and directories currently installed.)
Preparing to unpack .../libc-dev-bin 2.23-0ubuntu10_amd64.deb ...
Unpacking libc-dev-bin (2.23-0ubuntu10) ...
Selecting previously unselected package manpages-dev.
Preparing to unpack .../manpages-dev_4.04-2_all.deb ...
Unpacking manpages-dev (4.04-2) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up libc-dev-bin (2.23-0ubuntu10) ...
Setting up manpages-dev (4.04-2) ...
root@ubuntuProj2:/home/bart#

```

Figure 3. RPC Development Library Install Screenshot

4. Install other tools to be used in the development by executing **sudo apt-get install build-essential**. This output is similar to that of step 3.

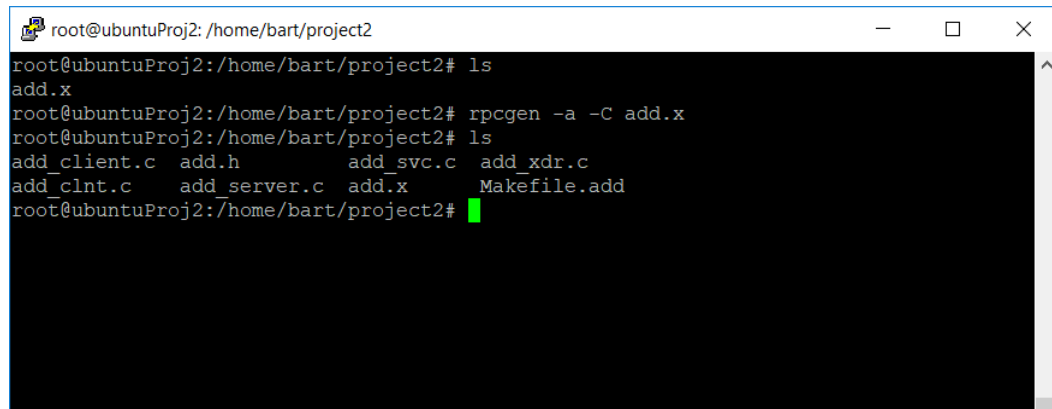
Section II Task 2

Introduction

Next, we will start the development process by creating a RPC template file and using that with the *rpcgen* command to create the executable stub programs for the client and server machines. Our initial RPC program will pass 100 elements from an array, add 100 to each element, and then send them back.

Explanation

1. First create the template. This holds the data structure to be passed by the RPC messages. It also includes version number and ID to distinguish it from other RPC messages. Our template file, *add.x*, contains the information as shown in the project handout.
2. We will need programs at the server and the client to accomplish this task. The stubs are created for us by the *rpcgen* command. We enter **rpcgen -a -C add.x** and it creates eight files. These are shown in figure 4. As you can see there are c files for both the server and client, a header file and a make configuration file called *Makefile.add*. *rpcgen* is a protocol compiler. It takes the RPC language file (similar to C) and creates the output files as defined by the parameters. The -a parameter tells it to generate all the files including sample code for client and server. The -C option tells the RPC compiler to generate code in ANSI C. This option also generates code that could be compiled with the C++ compiler and is the default.



```

root@ubuntuProj2: /home/bart/project2
root@ubuntuProj2:/home/bart/project2# ls
add.x
root@ubuntuProj2:/home/bart/project2# rpcgen -a -C add.x
root@ubuntuProj2:/home/bart/project2# ls
add_client.c  add.h          add_svc.c  add_xdr.c
add_clnt.c   add_server.c  add.x      Makefile.add
root@ubuntuProj2:/home/bart/project2#

```

Figure 4. rpcgen Command Screenshot

3. Let's take the RPC server first. The sample code generated by the *rpcgen* is shown in in the handout. We add code to read the 100-element integer stream add 100 to each element and then send back the results.
4. Now it's time to tackle the RPC client. The original file is also shown below in the handout. The *add_prog_1* function creates an RPC session with *clnt_create* and then calls it with the *print_1 function*. We updated the code to generate the array and then print out the results returned by the server.

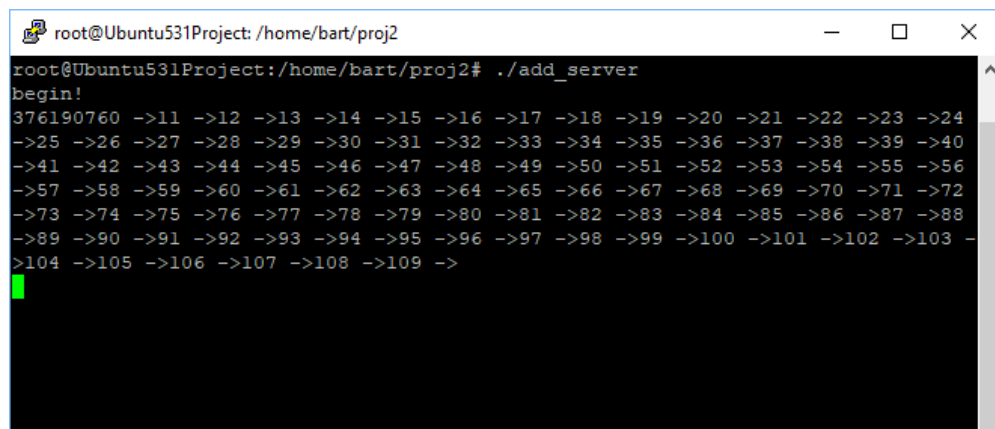
Section III Task 5

Introduction

Now that we have updated the code for both the server and the client, we will compile the c files using the Makefile generated by the *rpcgen* command. Afterwards we can run our programs and watch the numbers fly. The steps for this are shown below.

Explanation

1. Use the make command to compile the C code files, **make -f Makefile.add.** the object files are created along with the executables.
2. In order to run the executables, we must start the server side first. It will simply wait for an RPC message from the client. Start the server program by running **./add_server** on the server computer. Figure 5 shows the server command.



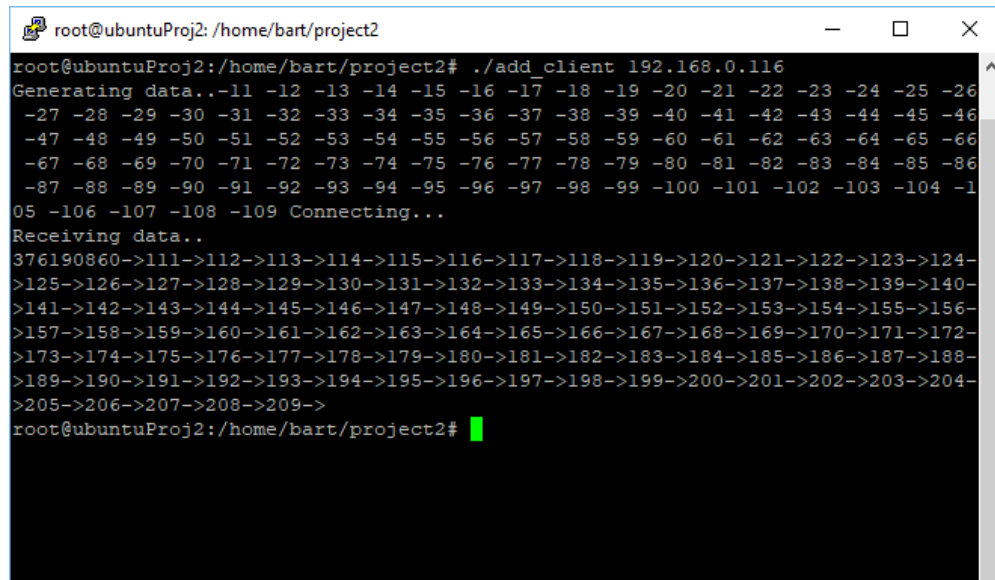
```

root@Ubuntu531Project: /home/bart/proj2
root@Ubuntu531Project:/home/bart/proj2# ./add_server
begin!
376190760 ->11 ->12 ->13 ->14 ->15 ->16 ->17 ->18 ->19 ->20 ->21 ->22 ->23 ->24
->25 ->26 ->27 ->28 ->29 ->30 ->31 ->32 ->33 ->34 ->35 ->36 ->37 ->38 ->39 ->40
->41 ->42 ->43 ->44 ->45 ->46 ->47 ->48 ->49 ->50 ->51 ->52 ->53 ->54 ->55 ->56
->57 ->58 ->59 ->60 ->61 ->62 ->63 ->64 ->65 ->66 ->67 ->68 ->69 ->70 ->71 ->72
->73 ->74 ->75 ->76 ->77 ->78 ->79 ->80 ->81 ->82 ->83 ->84 ->85 ->86 ->87 ->88
->89 ->90 ->91 ->92 ->93 ->94 ->95 ->96 ->97 ->98 ->99 ->100 ->101 ->102 ->103 -
>104 ->105 ->106 ->107 ->108 ->109 ->

```

Figure 5. RPC Server Program Output

3. Once the server is running start the client program on the client computer. The server and client can run on the same computer as we have done in this example. A hostname must be passed to the client program so we will use localhost since we are running both on the same machine. The command for the client side is **./add_client localhost**. A screenshot is shown in figure 6 for the client program and its output.



```

root@ubuntuProj2: /home/bart/project2
root@ubuntuProj2:/home/bart/project2# ./add_client 192.168.0.116
Generating data..-11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26
-27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 -40 -41 -42 -43 -44 -45 -46
-47 -48 -49 -50 -51 -52 -53 -54 -55 -56 -57 -58 -59 -60 -61 -62 -63 -64 -65 -66
-67 -68 -69 -70 -71 -72 -73 -74 -75 -76 -77 -78 -79 -80 -81 -82 -83 -84 -85 -86
-87 -88 -89 -90 -91 -92 -93 -94 -95 -96 -97 -98 -99 -100 -101 -102 -103 -104 -1
05 -106 -107 -108 -109 Connecting...
Receiving data..
376190860->111->112->113->114->115->116->117->118->119->120->121->122->123->124-
>125->126->127->128->129->130->131->132->133->134->135->136->137->138->139->140-
>141->142->143->144->145->146->147->148->149->150->151->152->153->154->155->156-
>157->158->159->160->161->162->163->164->165->166->167->168->169->170->171->172-
>173->174->175->176->177->178->179->180->181->182->183->184->185->186->187->188-
>189->190->191->192->193->194->195->196->197->198->199->200->201->202->203->204-
>205->206->207->208->209->
root@ubuntuProj2:/home/bart/project2#

```

Figure 6. RPC Client Program Output

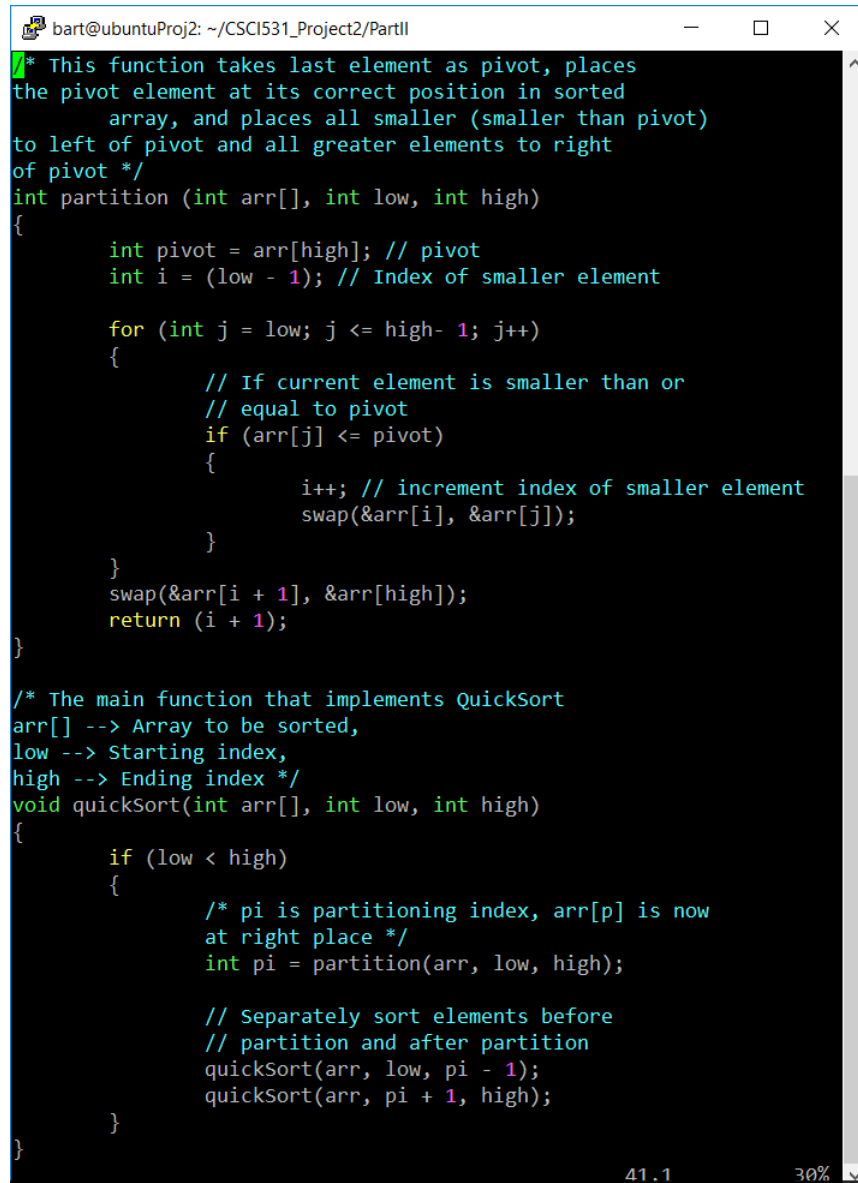
Section IV Task 6

Introduction

Now that we have the basics out of the way let's try something a little more challenging. We will develop a distributed quick sort project. The project will entail having two computers that will distribute the processing of the quicksort algorithm between them.

Explanation

1. First, we will need a quick sort program. Here is one of the commonly utilized quick sort models in a C program file shown in figure 7.



```

bart@ubuntuProj2: ~/CSCI531_Project2/PartII
/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

```

Figure 7. Quicksort C Code

2. Next, we will need to create the array using a random number generator. In addition to the number generator we know that the requirements state that duplicates are not allowed so a helper function is also created to check the array for a duplicate.

Section V Task 7

Introduction

Now that we have the basic sort functions we need to add threading so that the tasks can run in parallel and also bring in the RPC functions for both the client side and the server side.

Explanation

1. To create the multiple threads, we use the POSIX Thread standard implemented in Linux C using pthreads. The threads are created with the call to *pthread_create()* function. The threads are synchronized using the *pthread_join()* function, and the threads are destroyed with a call to the *pthread_exit()* function. More information for these functions can be viewed by using the Linux man command, e.g. `man pthread_create`.
2. We will need a new RPC configuration to pass the larger arrays for the distributed quicksort. We decided on the numbers structure used in the *qsRPC.x* file that is shown in figure 8. This creates a structure with an array of 5000 integers and two integer variables showing the low and high values of the array. This was done so we could define the maximum size array but also use it for smaller arrays without having to recompile each time.



```

struct numbers
{
    int low;
    int high;
    int a[5000];
};

program QSRPC_PROG {
    version QSRPC_V1 {
        numbers qsort(numbers)=1;
    }=1;
} = 0x12345678;

```

Figure 8. qsRPC.x File

3. We now have all the building blocks, the quicksort algorithm, the random number generator, the threading functions and the RPC generated files. All that's left is to put them all together.

Section VI Task 8

Introduction

Combining all the building blocks mentioned in the previous paragraph means creating a client-side module that will create the array and sort the array using the quicksort algorithm using a multi-threaded schema. The server side will only need to receive the array, sort it, then return it to the client where it will be combined with all other portions and printed. The test scenarios (these are described in the next section) must be taken into account during the development. These include various array sizes, sorting all on local machine and sorting using the distributed schema. Now let's walk through the development in the next paragraph.

Explanation

1. Start by creating the local sort program inside the RPC generated sample client code, *rsRPC_client.c*, and test. We decided to set the number of elements of the array to 5000; that way we could test on smaller arrays and move to the larger without recompiling.
2. Modify the local sort program to use two threads to conduct the sort using the POSIX threading methodology discussed previously.
3. Create the server module that will receive and sort the array data. It was at this point that all sort functions were moved to a header file called *quicksort.h* so the server program *qsRPC_server.c* could utilize the same functions by including the header file.
4. Update the client program to perform an initial split of the array to be sorted, spawn a thread to sort the upper portion, and send the lower portion to the server to be processed. When the both the thread is complete and the server responds with the sort result print the combined data to the screen.
5. In order to test the different array length scenarios, the portion of the main program was modified to repeat the above procedure for each array size. At this point threading support functions were moved to the header file to simplify the client program file.
6. To test the different local and distributed scenarios the client file was copied to files *rsLocal_client.c* and *rsThread_client.c*. These were modified to sort only on the local machine and sort on the local machine using two threads. The results will be presented and discussed below in the next section. A copy of the source code is

included in the appendices. All the files are staged at

https://github.com/bartbrock/CSCI531_Project2/.

Section VII Results and Analysis

Scenario 1

Description

This scenario is the simple quick sort conducted entirely on the local machine. The sort is done on the entire array using one thread. Trials were conducted for array sizes of 100, 1000 and 5000.

Data collected

The data collected is shown in table 1. A total of twenty test runs were conducted using the two different client computers identified in the introduction. The times ranged from 0.0514 ms milliseconds (ms) for a 100-element trial to 101.489 ms for a 5000-element trial. Table 1 shows the minimum, maximum, and average times for each of the different array sizes.

	100 Elements	1000 Elements	5000 Elements
Min	0.051	1.920	58.817
Max	0.266	3.355	101.489
Average	0.084	2.629	79.525

Table 1. Runtimes in milliseconds

Scenario 1A

Description

This test is basically the same as above except that the sort was divided into two threads so that the sort could be conducted in parallel. For this test the array had to be divided initially as in the following distributed test in order to run the threaded sort.

Data collected

The data collected is shown in table 2. A total of twenty test runs using the two client computers identified in the introduction were conducted. The times ranged from 0.22 milliseconds (ms) to 103.068 ms.

	100 Elements	1000 Elements	5000 Elements
Min	0.220	1.971	58.414
Max	0.605	5.328	103.068
Average	0.343	2.945	81.183

Table 2. Thread Results

Scenario 2Description

This test shows the execution times for the distributed sort method. The array is divided into two portions. One is sorted locally while the other is sorted by the server and then returned.

Data collected

The execution time for twenty trials of this test is shown in table 3. The values range from 76.314 ms to 368.717 ms. Much larger than the local sort.

	100 Elements	1000 Elements	5000 Elements
Min	76.314	72.132	168.627
Max	301.649	313.532	368.717
Average	184.036	189.285	264.915

Table 3. RPC Results

Scenario 2A

Description

This test is the same as the previous except that the server and client are being run as different processes on the same computer. This was added to provide a comparison to outline the transmission times associated with the distributed sort.

Data collected

The execution time for twenty trials of this test is shown in table 4. The values range from 2.62 ms to 104.73 ms.

	100 Elements	1000 Elements	5000 Elements
Min	2.262	4.222	60.723
Max	7.728	8.657	104.730
Average	3.345	5.401	84.840

Table 4. RPC-Localhost Results

Comparison of Data

Comparing the data from the different trials we see that the local sort is significantly faster than the RPC results for all array sizes. We deduce that this is due to the high processor speeds and the latency involved with the transmission time of the RPC messages. Figures 9 and 10 below show the execution times for scenarios 1 and 2, the local sorting and the distributed sorting.

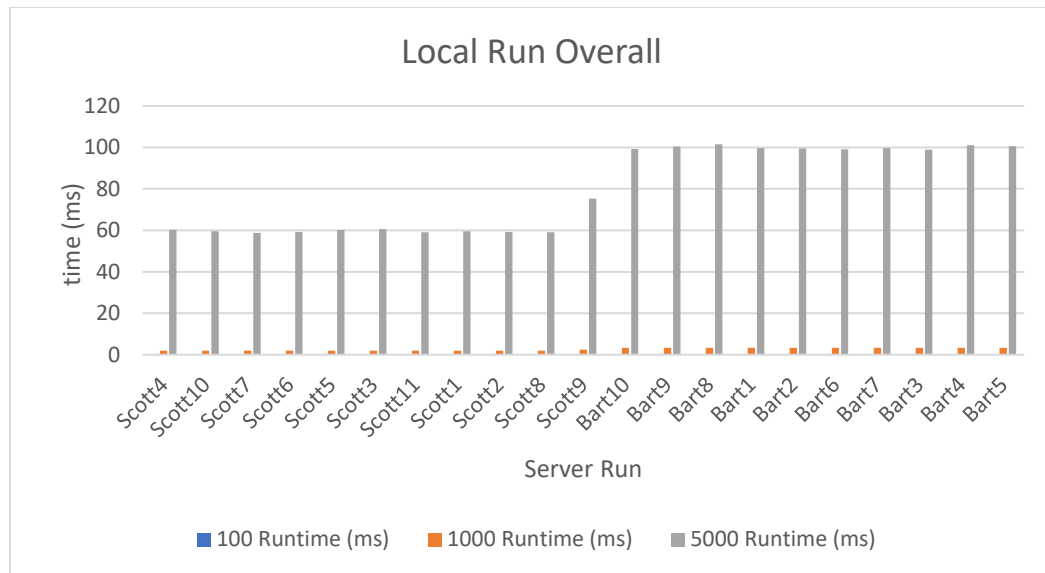


Figure 9. Local Results Graph

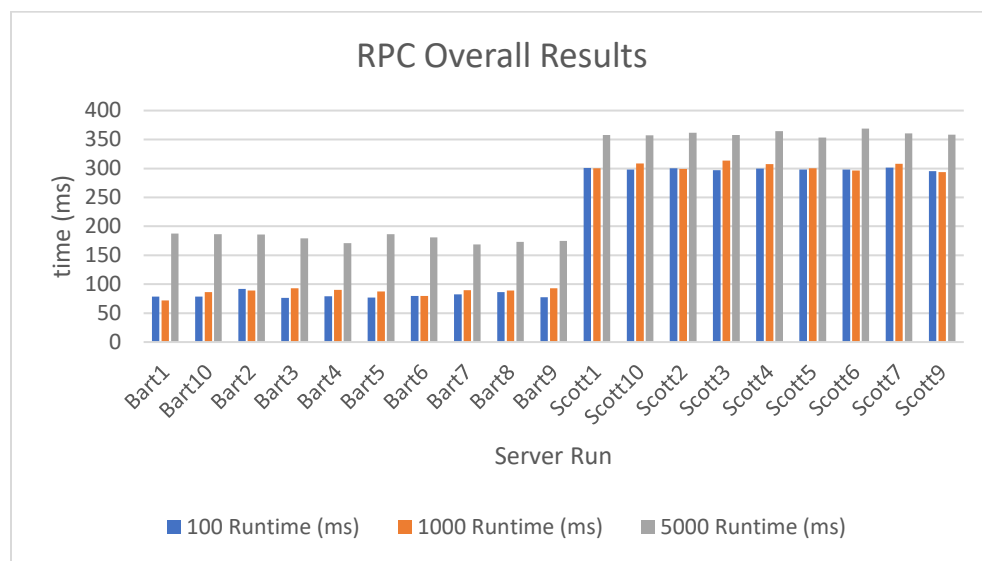


Figure 10. RPC Results Graph

Analysis of Data

As mentioned earlier the times for RPC distributed sort are much greater than that for the local sort for all array sizes. Figures 11 and 12 show all scenarios of the 5000 element trials. This clearly shows how the RPC under performs even at these larger data sets. The transmission

times can be estimated by comparing the execution times of the RPC trial with the localhost trial. Both of these test the RPC protocol but localhost executes both server and client on a single machine. By subtracting these two times we can get a rough estimation of the transmission times. They ranged from 150ms to 250ms.

One thing of interest is the difference in the time values for RPC between the first half of the graph and the second. There is almost a 200 ms difference in the times. That is due to these tests being run on separate computers located inside the Home Telecon area and outside the Home Telecon provider area. Other than that, the times for all test are fairly constant for each client computer.

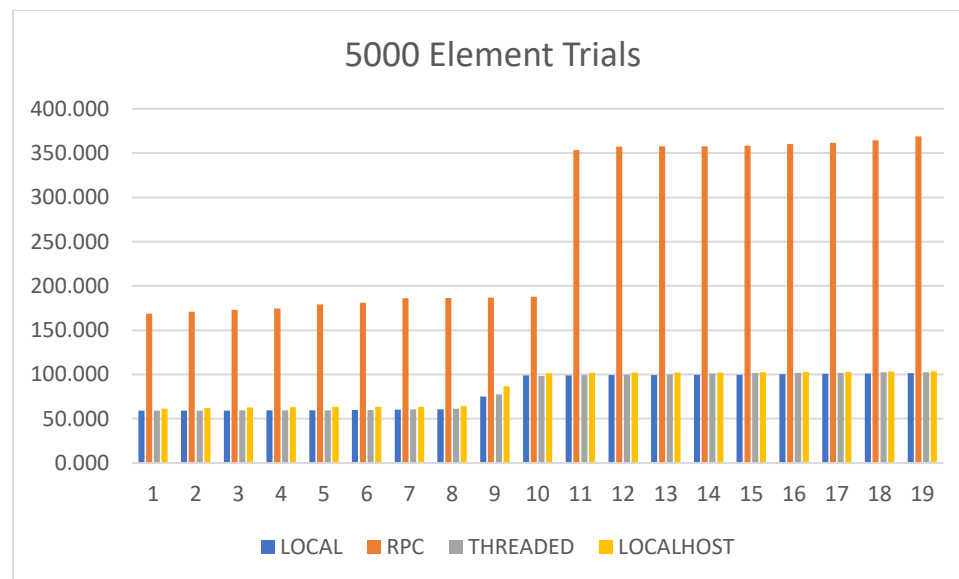


Figure 11. Comparison of all 5000 Element Test Runs

Another point of interest is the difference between the local, threading, and localhost execution times. These are shown in figure 12 without the RPC values in order to obtain a better scale. We thought that the dual threaded would beat out the local single threaded execution but the data shows that, although they are very close the local seems the edge out the threaded

executions most of the time. We have to assume that this is due to the overhead involved with managing the threads. The Localhost runtimes are larger but still very close to these values as well. This shows further that the greatest latency is not in the operating system but in the transmission over IP.

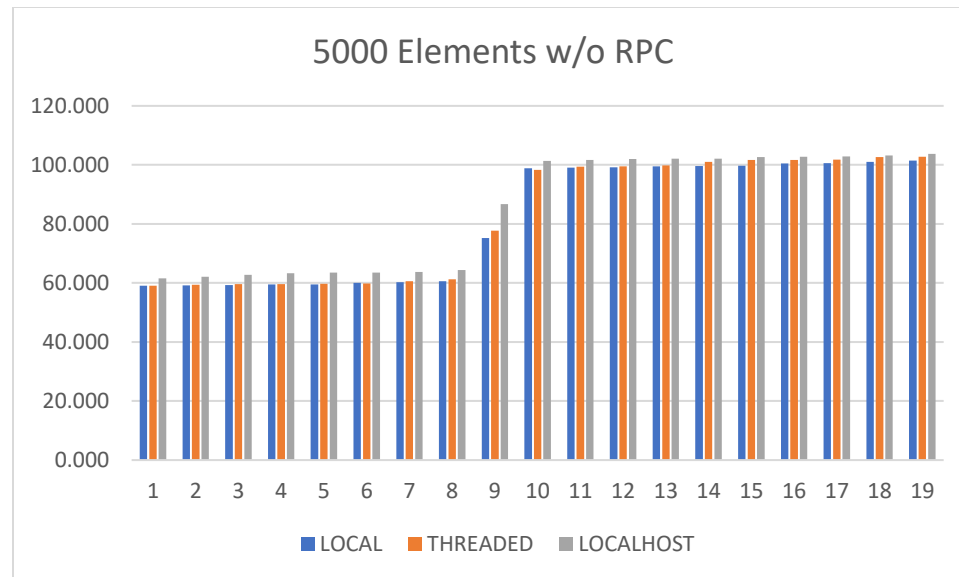


Figure 12. Comparison of 5000 Element Test Runs w/o RPC

There was a question of how the different size distributions would affect the execution times since the initial split of the array was based on a random pivot value. The figures below show the correlation between the actual size of the RPC data and the total execution time. At first glance it looks like the times are all over the chart but if you look closely the times are actually going back and forth between two near constant set of values for each data size. This corresponds to the difference seen between the two computer locations and is a result of transmission times. So, it appears that the data size has little effect on the overall time and transmission time a much greater effect.

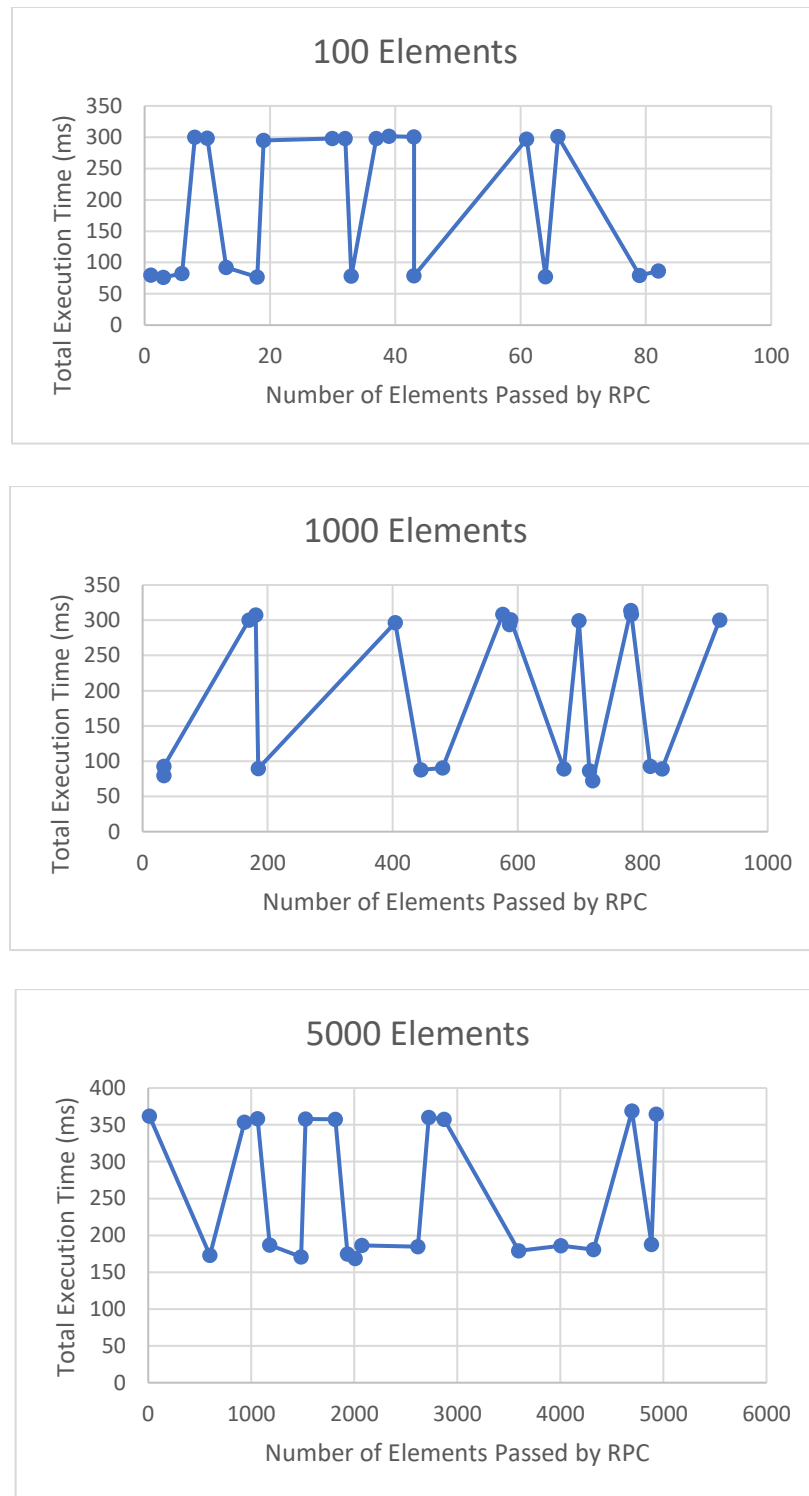


Figure 14. Comparison of Execution Time vs Size of RPC Data

Section VIII Conclusion

It was fairly evident that the speed of the modern computer won out over the distributed processing model for the data sizes under test. Of course, at some point (large sizes of data arrays) where the distributed processing would have the better performance. For the sizes tested here the communications overhead involved greatly overshadowed any performance gain from multiprocessing.

Appendix A, Listing of qsRPC.h File

```

/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _QSRPC_H_RPCGEN
#define _QSRPC_H_RPCGEN

#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

struct numbers {
    int low;
    int high;
    int a[5000];
};
typedef struct numbers numbers;

#define QSRPC_PROG 0x12345678
#define QSRPC_V1 1

#if defined(__STDC__) || defined(__cplusplus)
#define qsort_1
extern numbers * qsort_1(numbers *, CLIENT *);
extern numbers * qsort_1_svc(numbers *, struct svc_req *);
extern int qsrpc_prog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define qsort_1
extern numbers * qsort_1();
extern numbers * qsort_1_svc();
extern int qsrpc_prog_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern bool_t xdr_numbers (XDR *, numbers*);

#else /* K&R C */
extern bool_t xdr_numbers ();

#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_QSRPC_H_RPCGEN */

```

Appendix B, Listing of quicksort.h File

```
#pragma once
/* QuickSort Multithreaded */
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdbool.h>
#include<time.h>
#include <sys/time.h>
#include <stdint.h>

#include "qsRPC.h"

#define ZERO 0

// type definition of the thread/rpc parameters
// same as numbers in .h file
//typedef struct arrayParamsStruct{
//    int low;
//    int high;
//    int a[5000];
//} arrayParamsType;

// declare structure variables for global use
numbers arrayParams, arrayParamsLow, arrayParamsHigh;
// initial working array for quicksort
int arr[5000];
// thread id array
pthread_t tid[2];

// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
```

```

high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* quickSortThread is used to create thread for quick sort function
the parameter structure is passed by pointer
*/
void* quickSortThread(void *arg)
{
    numbers *aP = (numbers *)arg;
    //printf("Enter quickSortThread, aP.a[0]= %d \n", aP->a[0]);
    int low = aP->low;
    int high = aP->high;
    int *arr = aP->a;

    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
    //printf("Exit quickSortThread \n");
    pthread_exit(NULL);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    //printf("\n");
}

void extractArrayStruct(numbers *params, int arrOld[], int start, int size)
{
    //printf("Enter extractArrayStruct \n");
    for (int i = start; i < size+start; i++)
    {
        params->a[i-start] = arrOld[i];
        //printf("i= %d, arrOld= %d, arrayParams= %d, address= %d \n", i,
arrOld[i], params->a[i-start], &params->a[i-start]);
    }
    params->low = 0;
    params->high = size-1;
}

void extractArray(int arrNew[], int arrOld[], int start, int size)
{
    //printf("Enter extractArray \n");
    for (int i = start; i < size+start; i++)
    {

```

```

        //printf("i= %d, start= %d, size= %d, arrOld= %d \n", i, start, size,
arrOld[i]);
        arrNew[i-start] = arrOld[i];
    }
}

bool isDupElement(int testNum, int aSize)
{
    bool result = false;
    for (int i = 0; i < aSize; i++)
    {
        if (arr[i] == testNum) result = true;
    }
    return result;
}

void createRandomArray(int size)
{
    int element;
    for (int pos = 0; pos < size; pos++ )
    {
        element = rand() % 10001;
        if (isDupElement(element, pos))
            pos--;
        else
            arr[pos] = element;
    }
}

uint64_t get_posix_clock_time()
{
    struct timespec ts;

    if (clock_gettime(CLOCK_MONOTONIC, &ts) == 0)
        return (uint64_t)(ts.tv_sec * 1000000000 + ts.tv_nsec);
    else
        return 0;
}

```


Appendix C, Listing of qsRPC_server.c File

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "quicksort.h"

numbers *
qsort_1_svc(numbers *argp, struct svc_req *rqstp)
{
    static numbers result;
    int *arr = argp->a;
    int low = argp->low;
    int high = argp->high;

    printf("Start qsort_1_svc \n");
    printf("argp.a[0]= %d, arr[0]= %d \n", argp->a[0], arr[0]);
    // insert server code here
    quickSort(arr, low, high);

    result = *argp;
    printf("After sort \n");
    printf("argp.a[0]= %d, arr[0]= %d result.a[0]= %d \n", argp->a[0], arr[0], result.a[0]);

    return &result;
}
```

Appendix D, Listing of qsRPC_client.c File

```

#include "quicksort.h"

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

numbers *
qsrpc_prog_1(char *host, numbers *params)
{
    CLIENT *clnt;
    numbers *result_1;
    numbers* qsort_1_arg = params;

#ifdef DEBUG
    // setup remote connection
    clnt = clnt_create (host, QSRPC_PROG, QSRPC_V1, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    // call to remote
    result_1 = qsort_1(qsort_1_arg, clnt);
    if (result_1 == (numbers *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    // get return values from result
    //
#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
    return result_1;
}

int
main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];

    int n = 0, err = 0;
    int arraySize100 = 100, arraySize1000 = 1000, arraySize5000 = 5000;
    int pivot = 0;
    int arrHighSize = 0;

    // set up timers
    srand(time(0));
    time_t s100, e100, s1000, e1000, s5000, e5000; // start and end times for all runs
    uint64_t start_time_value1, end_time_value1, start_time_value2, end_time_value2,
start_time_value3, end_time_value3;
    uint64_t time_diff1, time_diff2, time_diff3;

    /*****
        Trial #1  Array size = 100
        *****/
    printf("\n***** Start Trial 1 Array Size = 100 ***** \n");
    // mark start time
    time(&s100);
    start_time_value1 = get_posix_clock_time();

```

```

// set up array
n = arraySize100;
createRandomArray(n);

// start quicksort process
//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
pivot = partition(arr, 0, n-1);
arrHighSize = n - pivot;

printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
pivot, arrHighSize);

// create sub arrays
extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
//printf("\npre-sorted arrayParamsLow: \n");
//printArray(arrayParamsLow.a, pivot);
//printf("pre-sorted arrayParamsHigh: \n");
//printArray(arrayParamsHigh.a, arrHighSize);

//send upper portion to thread
if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
    printf("\ncan't create thread :[%s] \n", strerror(err));

//send lower portion to remote
// call to rpc stub ++++++++
arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

//if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
//    printf("\ncan't create thread :[%s] \n", strerror(err));

// process upper portion
//quickSort(arrHigh, 0, arrHighSize-1);

//wait for thread to complete
//pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);

printf("\nSorted Array\n");
printArray(arrayParams.a, arrayParamsLow.high + 1);
printArray(arrayParamsHigh.a, arrHighSize);

// mark end time
time(&e100);
end_time_value1 = get_posix_clock_time();
time_diff1 = end_time_value1 - start_time_value1;
//printf("\nStart time: %s\n", ctime(&s100));
//printf("End time : %s\n", ctime(&e100));
//printf("Runtime in seconds : %f \n", difftime(e100, s100));
printf("Runtime in milliseconds : %f \n", (double)time_diff1 / 1000000);
// ----- End Run # 1 -----
printf("\n----- End Run # 1 ----- \n\n");

/*****
Trial #2 Array size = 1000
*****/
printf("\n***** Start Trial 2 Array Size = 1000 ***** \n");
// mark start time
time(&s1000);
start_time_value2 = get_posix_clock_time();

// set up array
n = arraySize1000;
createRandomArray(n);

// start quicksort process

```

```

//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
pivot = partition(arr, 0, n - 1);
arrHighSize = n - pivot;

printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
pivot, arrHighSize);

// create sub arrays
extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
//printf("\npre-sorted arrayParamsLow: \n");
//printArray(arrayParamsLow.a, pivot);
//printf("pre-sorted arrayParamsHigh: \n");
//printArray(arrayParamsHigh.a, arrHighSize);

//send upper portion to thread
if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
    printf("\ncan't create thread :[%s] \n", strerror(err));

//send lower portion to remote
// call to rpc stub ++++++++
arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

//if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
//    printf("\ncan't create thread :[%s] \n", strerror(err));

// process upper portion
//quickSort(arrHigh, 0, arrHighSize-1);

//wait for thread to complete
//pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);

printf("\nSorted Array\n");
printArray(arrayParams.a, arrayParamsLow.high + 1);
printArray(arrayParamsHigh.a, arrHighSize);

// mark end time
time(&el000);
end_time_value2 = get_posix_clock_time();
time_diff2 = end_time_value2 - start_time_value2;
//printf("\nStart time: %s\n", ctime(&s1000));
//printf("End time : %s\n", ctime(&el000));
//printf("Runtime in seconds : %f \n", difftime(el000, s1000));
printf("Runtime in milliseconds : %f \n", (double)time_diff2 / 1000000);
// ----- End Run # 2 -----
printf("\n----- End Run # 2 ----- \n\n");

/*****
Trial #3 Array size = 5000
*****/
printf("\n***** Start Trial 3 Array Size = 5000 ***** \n");
// mark start time
time(&s5000);
start_time_value3 = get_posix_clock_time();

// set up array
n = arraySize5000;
createRandomArray(n);

// start quicksort process
//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
pivot = partition(arr, 0, n - 1);
arrHighSize = n - pivot;

```

```

    printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
    pivot, arrHighSize);

    // create sub arrays
    extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
    extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
    //printf("\npre-sorted arrayParamsLow: \n");
    //printArray(arrayParamsLow.a, pivot);
    //printf("pre-sorted arrayParamsHigh: \n");
    //printArray(arrayParamsHigh.a, arrHighSize);

    //send upper portion to thread
    if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
        printf("\ncan't create thread :[%s] \n", strerror(err));

    //send lower portion to remote
    // call to rpc stub ++++++++
    arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

    //if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
    //    printf("\ncan't create thread :[%s] \n", strerror(err));

    // process upper portion
    //quickSort(arrHigh, 0, arrHighSize-1);

    //wait for thread to complete
    //pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    printf("\nSorted Array\n");
    printArray(arrayParams.a, arrayParamsLow.high + 1);
    printArray(arrayParamsHigh.a, arrHighSize);

    // mark end time
    time(&e5000);
    end_time_value3 = get_posix_clock_time();
    time_diff3 = end_time_value3 - start_time_value3;
    //printf("\nStart time: %s\n", ctime(&s5000));
    //printf("End time : %s\n", ctime(&e5000));
    //printf("Runtime in seconds : %f \n", difftime(e5000, s5000));
    printf("Runtime in milliseconds : %f \n", (double)time_diff3 / 1000000);
    // ----- End Run # 3 -----
    printf("\n----- End Run # 3 ----- \n\n");

    // print out time results for all trials
    printf("TRIALS FOR RPC EXECUTION \n");
    printf("TRIAL 1, 100 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff1 / 1000000);
    printf("TRIAL 2, 1000 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff2 / 1000000);
    printf("TRIAL 3, 5000 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff3 / 1000000);

    exit (0);
}

```

Appendix E, Listing of qsLocal_client.c File

```

#include "quicksort.h"

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

numbers *
qsrpc_prog_1(char *host, numbers *params)
{
    CLIENT *clnt;
    numbers *result_1;
    numbers* qsort_1_arg = params;

#ifdef DEBUG
    // setup remote connection
    clnt = clnt_create (host, QSRPC_PROG, QSRPC_V1, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    // call to remote
    result_1 = qsort_1(qsort_1_arg, clnt);
    if (result_1 == (numbers *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    // get return values from result
    //
#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
    return result_1;
}

int
main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];

    int n = 0, err = 0;
    int arraySize100 = 100, arraySize1000 = 1000, arraySize5000 = 5000;
    int pivot = 0;
    int arrHighSize = 0;

    // set up timers
    srand(time(0));
    time_t s100, e100, s1000, e1000, s5000, e5000; // start and end times for all runs
    uint64_t start_time_value1, end_time_value1, start_time_value2, end_time_value2,
start_time_value3, end_time_value3;
    uint64_t time_diff1, time_diff2, time_diff3;

    /*****
        Trial #1  Array size = 100
        *****/
    printf("\n***** Start Trial 1 Array Size = 100 ***** \n");
    // mark start time
    time(&s100);
    start_time_value1 = get_posix_clock_time();

```

```

// set up array
n = arraySize100;
createRandomArray(n);

// start quicksort process
//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
//pivot = partition(arr, 0, n-1);
//arrHighSize = n - pivot;

//printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
pivot, arrHighSize);

// create sub arrays
//extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
//extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
//printf("\npre-sorted arrayParamsLow: \n");
//printArray(arrayParamsLow.a, pivot);
//printf("pre-sorted arrayParamsHigh: \n");
//printArray(arrayParamsHigh.a, arrHighSize);

//send upper portion to thread
//if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
//    printf("\ncan't create thread :[%s] \n", strerror(err));

//send lower portion to remote
// call to rpc stub ++++++++
//arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

//if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
//    printf("\ncan't create thread :[%s] \n", strerror(err));

// process upper portion
quickSort(arr, 0, n-1);

//wait for thread to complete
//pthread_join(tid[0], NULL);
//pthread_join(tid[1], NULL);

printf("\nSorted Array\n");
printArray(arr, n);
//printArray(arrayParamsHigh.a, arrHighSize);

// mark end time
time(&e100);
end_time_value1 = get_posix_clock_time();
time_diff1 = end_time_value1 - start_time_value1;
//printf("\nStart time: %s\n", ctime(&s100));
//printf("End time : %s\n", ctime(&e100));
//printf("Runtime in seconds : %f \n", difftime(e100, s100));
printf("Runtime in milliseconds : %f \n", (double)time_diff1 / 1000000);
// ----- End Run # 1 -----
printf("\n----- End Run # 1 ----- \n\n");

/*****
Trial #2 Array size = 1000
*****/
printf("\n***** Start Trial 2 Array Size = 1000 ***** \n");
// mark start time
time(&s1000);
start_time_value2 = get_posix_clock_time();

// set up array
n = arraySize1000;
createRandomArray(n);

// start quicksort process

```

```

//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
//pivot = partition(arr, 0, n - 1);
//arrHighSize = n - pivot;

//printf("\npivotIndex = %d, pivotVaule= %d, n = %d arrHighSize = %d \n", pivot,
arr[pivot], n, arrHighSize);

// create sub arrays
//extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
//extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
//printf("\npre-sorted arrayParamsLow: \n");
//printArray(arrayParamsLow.a, pivot);
//printf("pre-sorted arrayParamsHigh: \n");
//printArray(arrayParamsHigh.a, arrHighSize);

//send upper portion to thread
//if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
//printf("\ncan't create thread :[%s] \n", strerror(err));

//send lower portion to remote
// call to rpc stub ++++++++
//arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

//if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
//printf("\ncan't create thread :[%s] \n", strerror(err));

// process upper portion
quickSort(arr, 0, n-1);

//wait for thread to complete
//pthread_join(tid[0], NULL);
//pthread_join(tid[1], NULL);

printf("\nSorted Array\n");
printArray(arr, n);
//printArray(arrayParamsHigh.a, arrHighSize);

// mark end time
time(&el000);
end_time_value2 = get_posix_clock_time();
time_diff2 = end_time_value2 - start_time_value2;
//printf("\nStart time: %s\n", ctime(&s1000));
//printf("End time : %s\n", ctime(&el000));
//printf("Runtime in seconds : %f \n", difftime(el000, s1000));
printf("Runtime in milliseconds : %f \n", (double)time_diff2 / 1000000);
// ----- End Run # 2 -----
printf("\n----- End Run # 2 ----- \n\n");

/*****
Trial #3 Array size = 5000
*****/
printf("\n***** Start Trial 3 Array Size = 5000 ***** \n");
// mark start time
time(&s5000);
start_time_value3 = get_posix_clock_time();

// set up array
n = arraySize5000;
createRandomArray(n);

// start quicksort process
//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
//pivot = partition(arr, 0, n - 1);
//arrHighSize = n - pivot;

```



```

    //printf("\npivotIndex = %d, pivotVaule= %d, n = %d arrHighSize = %d \n", pivot,
arr[pivot], n, arrHighSize);

    // create sub arrays
    //extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
    //extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
    //printf("\npre-sorted arrayParamsLow: \n");
    //printArray(arrayParamsLow.a, pivot);
    //printf("pre-sorted arrayParamsHigh: \n");
    //printArray(arrayParamsHigh.a, arrHighSize);

    //send upper portion to thread
    //if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
        //printf("\ncan't create thread :[%s] \n", strerror(err));

    //send lower portion to remote
    // call to rpc stub ++++++++
    //arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

    //if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
    //    printf("\ncan't create thread :[%s] \n", strerror(err));

    // process upper portion
    quickSort(arr, 0, n-1);

    //wait for thread to complete
    //pthread_join(tid[0], NULL);
    //pthread_join(tid[1], NULL);

    printf("\nSorted Array\n");
    printArray(arr, n);
    //printArray(arrayParamsHigh.a, arrHighSize);

    // mark end time
    time(&e5000);
    end_time_value3 = get_posix_clock_time();
    time_diff3 = end_time_value3 - start_time_value3;
    //printf("\nStart time: %s\n", ctime(&s5000));
    //printf("End time : %s\n", ctime(&e5000));
    //printf("Runtime in seconds : %f \n", difftime(e5000, s5000));
    printf("Runtime in milliseconds : %f \n", (double)time_diff3 / 1000000);
    // ----- End Run # 3 -----
    printf("\n----- End Run # 3 ----- \n\n");

    // print out time results for all trials
    printf("TRIALS FOR LOCAL EXECUTION \n");
    printf("TRIAL 1, 100 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff1 / 1000000);
    printf("TRIAL 2, 1000 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff2 / 1000000);
    printf("TRIAL 3, 5000 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff3 / 1000000);

    exit (0);
}

```

Appendix F, Listing of qsThread_client.c File

```
#include "quicksort.h"

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

numbers *
qsrpc_prog_1(char *host, numbers *params)
{
    CLIENT *clnt;
    numbers *result_1;
    numbers* qsort_1_arg = params;

#ifdef DEBUG
    // setup remote connection
    clnt = clnt_create (host, QSRPC_PROG, QSRPC_V1, "tcp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    // call to remote
    result_1 = qsort_1(qsort_1_arg, clnt);
    if (result_1 == (numbers *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    // get return values from result
    //
#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
    return result_1;
}

int
main (int argc, char *argv[])
{
    char *host;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];

    int n = 0, err = 0;
    int arraySize100 = 100, arraySize1000 = 1000, arraySize5000 = 5000;
    int pivot = 0;
    int arrHighSize = 0;

    // set up timers
    srand(time(0));
    time_t s100, e100, s1000, e1000, s5000, e5000; // start and end times for all runs
    uint64_t start_time_value1, end_time_value1, start_time_value2, end_time_value2,
    start_time_value3, end_time_value3;
    uint64_t time_diff1, time_diff2, time_diff3;

    /*****
        Trial #1  Array size = 100
        *****/
    printf("\n***** Start Trial 1 Array Size = 100 ***** \n");
    // mark start time
    time(&s100);
    start_time_value1 = get_posix_clock_time();

```

```

// set up array
n = arraySize100;
createRandomArray(n);

// start quicksort process
//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
pivot = partition(arr, 0, n-1);
arrHighSize = n - pivot;

printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
pivot, arrHighSize);

// create sub arrays
extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
//printf("\npre-sorted arrayParamsLow: \n");
//printArray(arrayParamsLow.a, pivot);
//printf("pre-sorted arrayParamsHigh: \n");
//printArray(arrayParamsHigh.a, arrHighSize);

//send upper portion to thread
if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
    printf("\ncan't create thread :[%s] \n", strerror(err));

//send lower portion to remote
// call to rpc stub ++++++++
//arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
    printf("\ncan't create thread :[%s] \n", strerror(err));

// process upper portion
//quickSort(arrHigh, 0, arrHighSize-1);

//wait for thread to complete
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);

printf("\nSorted Array\n");
printArray(arrayParamsLow.a, arrayParamsLow.high + 1);
printArray(arrayParamsHigh.a, arrHighSize);

// mark end time
time(&e100);
end_time_value1 = get_posix_clock_time();
time_diff1 = end_time_value1 - start_time_value1;
//printf("\nStart time: %s\n", ctime(&s100));
//printf("End time : %s\n", ctime(&e100));
//printf("Runtime in seconds : %f \n", difftime(e100, s100));
printf("Runtime in milliseconds : %f \n", (double)time_diff1 / 1000000);
// ----- End Run # 1 -----
printf("\n----- End Run # 1 ----- \n\n");

/*****
Trial #2 Array size = 1000
*****/
printf("\n***** Start Trial 2 Array Size = 1000 ***** \n");
// mark start time
time(&s1000);
start_time_value2 = get_posix_clock_time();

// set up array
n = arraySize1000;
createRandomArray(n);

// start quicksort process

```

```

//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
pivot = partition(arr, 0, n - 1);
arrHighSize = n - pivot;

printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
pivot, arrHighSize);

// create sub arrays
extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
//printf("\npre-sorted arrayParamsLow: \n");
//printArray(arrayParamsLow.a, pivot);
//printf("pre-sorted arrayParamsHigh: \n");
//printArray(arrayParamsHigh.a, arrHighSize);

//send upper portion to thread
if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
    printf("\ncan't create thread :[%s] \n", strerror(err));

//send lower portion to remote
// call to rpc stub ++++++++
//arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
    printf("\ncan't create thread :[%s] \n", strerror(err));

// process upper portion
//quickSort(arrHigh, 0, arrHighSize-1);

//wait for thread to complete
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);

printf("\nSorted Array\n");
printArray(arrayParamsLow.a, arrayParamsLow.high + 1);
printArray(arrayParamsHigh.a, arrHighSize);

// mark end time
time(&el000);
end_time_value2 = get_posix_clock_time();
time_diff2 = end_time_value2 - start_time_value2;
//printf("\nStart time: %s\n", ctime(&s1000));
//printf("End time : %s\n", ctime(&el000));
//printf("Runtime in seconds : %f \n", difftime(el000, s1000));
printf("Runtime in milliseconds : %f \n", (double)time_diff2 / 1000000);
// ----- End Run # 2 -----
printf("\n----- End Run # 2 ----- \n\n");

/*****
Trial #3 Array size = 5000
*****/
printf("\n***** Start Trial 3 Array Size = 5000 ***** \n");
// mark start time
time(&s5000);
start_time_value3 = get_posix_clock_time();

// set up array
n = arraySize5000;
createRandomArray(n);

// start quicksort process
//printf("\nOriginal array, size= %d \n", n);
//printArray(arr, n);

// divide array into two
pivot = partition(arr, 0, n - 1);
arrHighSize = n - pivot;

```

```

    printf("\nPivot Vaule = %d, Low Array Size = %d, High Array Size = %d \n", arr[pivot],
    pivot, arrHighSize);

    // create sub arrays
    extractArrayStruct(&arrayParamsLow, arr, ZERO, pivot);
    extractArrayStruct(&arrayParamsHigh, arr, pivot, arrHighSize);
    //printf("\npre-sorted arrayParamsLow: \n");
    //printArray(arrayParamsLow.a, pivot);
    //printf("pre-sorted arrayParamsHigh: \n");
    //printArray(arrayParamsHigh.a, arrHighSize);

    //send upper portion to thread
    if (pthread_create(&(tid[1]), NULL, &quickSortThread, (void *)&arrayParamsHigh))
        printf("\ncan't create thread :[%s] \n", strerror(err));

    //send lower portion to remote
    // call to rpc stub ++++++++
    //arrayParams = *(qsrpc_prog_1(host, &arrayParamsLow));

    if (pthread_create(&(tid[0]), NULL, &quickSortThread, (void *)&arrayParamsLow))
        printf("\ncan't create thread :[%s] \n", strerror(err));

    // process upper portion
    //quickSort(arrHigh, 0, arrHighSize-1);

    //wait for thread to complete
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    printf("\nSorted Array\n");
    printArray(arrayParamsLow.a, arrayParamsLow.high + 1);
    printArray(arrayParamsHigh.a, arrHighSize);

    // mark end time
    time(&e5000);
    end_time_value3 = get_posix_clock_time();
    time_diff3 = end_time_value3 - start_time_value3;
    //printf("\nStart time: %s\n", ctime(&s5000));
    //printf("End time : %s\n", ctime(&e5000));
    //printf("Runtime in seconds : %f \n", difftime(e5000, s5000));
    printf("Runtime in milliseconds : %f \n", (double)time_diff3 / 1000000);
    // ----- End Run # 3 -----
    printf("\n----- End Run # 3 ----- \n\n");

    // print out time results for all trials
    printf("TRIALS FOR THREADED EXECUTION \n");
    printf("TRIAL 1, 100 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff1 / 1000000);
    printf("TRIAL 2, 1000 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff2 / 1000000);
    printf("TRIAL 3, 5000 elements \n");
    printf("Runtime in milliseconds : %f \n\n", (double)time_diff3 / 1000000);

    exit (0);
}

```