Name: Rushikesh Karbhari Palve
Roll No. 31258

: Date :	Page No :
/ /20	

Assignment No. 7

DOP: 11-11-2021 **DOS: 16-11-2021**

Problem Definition :-

Implementation (Unix C progeamming / Java) of RPC Merhanism.

Objectives :-

(i) To understand the concept of RPC mechanism.
(ii) To implement RPC merhanism in Unix C progeamming / Java.

Learning Outcomes :-

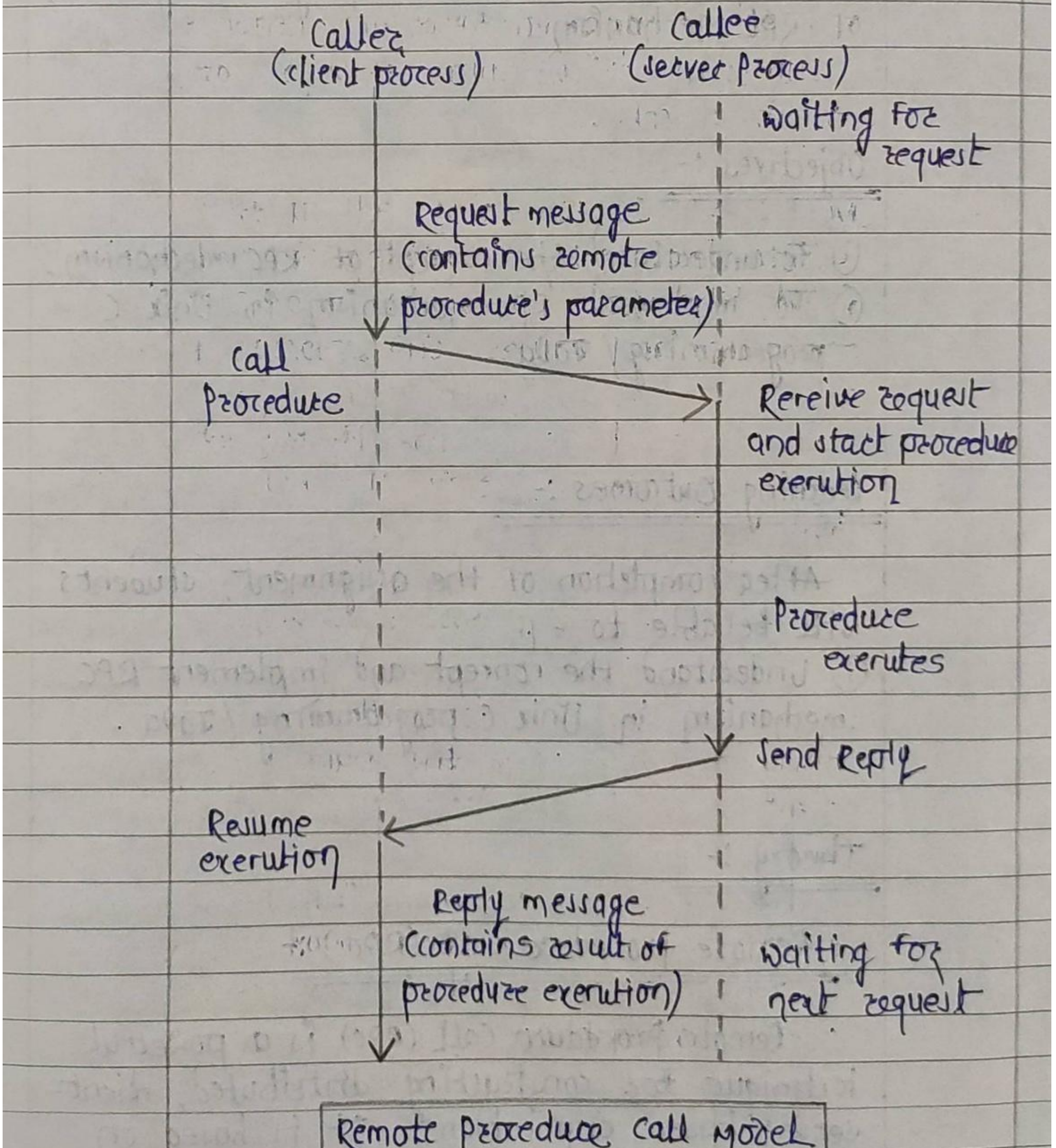
After completion of the assignment, students will be able to -
(i) Understand the concept and implement RPC merhanism in Unix C programming / Java.

Theory :-

Remote Procedure Call (RPC) :-

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on

extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.



Remote Procedure Call model

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there -

2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call -

NOTE: RPC is especially well suited for client-server (e.g. query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

The following steps take place during a RPC:

1. A client invokes a client stub procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub marshalls (packs) the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format and copying each parameter into the message -

3. The client stub passes the message to the transport layer, which sends ~~to~~ it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub which demarshalls (unpack) the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub, which marshalls the return values into a message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

Working of Rpc :-

ROLL NO. 31258

Date: / /20 | Page No.:

RPC ISSUES :-

Issues that must be addressed :-

1. RPC Runtime :-

Rpc run-time system is a library of routines and a set of services that handle the network communications that underline the RPC mechanism. In the course of an RPC call, client-side and server-side run-time system's code handle binding, establish communications over an appropriate protocol, pass call data between the client and server, and handle communications errors.

2. Stub :-

The function of the stub is to provide transparency to the programmer-written application code.

- On the client side, the stub handles the interface between the client's local procedure call and the run-time system, marshalling and unmarshaling data, invoking the RPC run-time protocol and if requested, carrying out some of the binding steps.
- On the server side, the stub provides a similar interface between the run-time system and the local manager procedures that are executed by the server.

3. Binding :- How does the client know who to call and where the service resides?

The most flexible solution is to use dynamic binding and find the server at run time when the RPC is first made. The first time the client stub is invoked, it contacts a name server to determine the transport address at which the server resides.
Binding consists of two parts :

- Naming.
- Locating :

1. A server having a service to offer exports an interface for it. Exporting an interface registers it with the system so that clients can use it.

2. A client must import an (exported) interface before communication can begin.

4. The call semantics associated with RPC :-

It is mainly classified into following choices :-

- Retry Request message -

Whether to retry sending a request message when a server has failed or the receiver didn't receive the message.

- Duplicate Filtering :-

Remove the duplicate server requests.

- Retransmission of results -

To resend lost messages without re-executing the operations at the server-side.

ADVANTAGES :-

1. RPC provides ABSTRACTION i.e. message-passing nature of network communication is hidden from the user -
2. RPC often omits many of the protocol layers to improve performance. Even ~~perfom~~ a small performance improvement is important because a program may invoke RPCs often.
3. RPC enables the usage of the applications in the distributed environment, not only in the local environment.
4. With RPC code re-writing / re-developing effort is minimized -
5. Process-oriented and thread oriented models supported by RPC.

CONCLUSION :-

We have successfully understood the concept and implemented RPC mechanism -

CODE :-

avg.x

```
const MAXAVGSIZE  = 200;

struct input_data {
  double input_data<200>;
};

typedef struct input_data input_data;

program AVERAGEPROG {
    version AVERAGEVERS {
        double AVERAGE(input_data) = 1;
    } = 1;
} = 22855;
```

avg.h

```
#ifndef _AVG_H_RPCGEN
#define _AVG_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif

#define MAXAVGSIZE 200

struct input_data {
   struct {
      u_int input_data_len;
      double *input_data_val;
   } input_data;
};
typedef struct input_data input_data;


#define AVERAGEPROG 22855
#define AVERAGEVERS 1

#if defined(__STDC__) || defined(__cplusplus)
#define AVERAGE 1
extern  double * average_1(input_data *, CLIENT *);
extern  double * average_1_svc(input_data *, struct svc_req *);
extern int averageprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else
#define AVERAGE 1
extern  double * average_1();
extern  double * average_1_svc();
extern int averageprog_1_freeresult ();
```

```
#endif

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_input_data (XDR *, input_data*);
extern  bool_t xdr_input_data (XDR *, input_data*);

#else
extern bool_t xdr_input_data ();
extern bool_t xdr_input_data ();

#endif

#ifdef __cplusplus
}
#endif

#endif
```

avg_clnt.c

```
#include <memory.h>
#include "avg.h"

static struct timeval TIMEOUT = { 25, 0 };

double *
average_1(input_data *argp, CLIENT *clnt)
{
   static double clnt_res;

   memset((char *)&clnt_res, 0, sizeof(clnt_res));
   if (clnt_call (clnt, AVERAGE,
      (xdrproc_t) xdr_input_data, (caddr_t) argp,
      (xdrproc_t) xdr_double, (caddr_t) &clnt_res,
      TIMEOUT) != RPC_SUCCESS) {
      return (NULL);
   }
   return (&clnt_res);
}
```

avg_svc.c

```
#include "avg.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
```

```
averageprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
   union {
      input_data average_1_arg;
   } argument;
   char *result;
   xdrproc_t _xdr_argument, _xdr_result;
   char *(*local)(char *, struct svc_req *);

   switch (rqstp->rq_proc) {
   case NULLPROC:
      (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
      return;

   case AVERAGE:
      _xdr_argument = (xdrproc_t) xdr_input_data;
      _xdr_result = (xdrproc_t) xdr_double;
      local = (char *(*)(char *, struct svc_req *)) average_1_svc;
      break;

   default:
      svcerr_noproc (transp);
      return;
   }
   memset ((char *)&argument, 0, sizeof (argument));
   if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t)
&argument)) {
      svcerr_decode (transp);
      return;
   }
   result = (*local)((char *)&argument, rqstp);
   if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result,
result)) {
      svcerr_systemerr (transp);
   }
   if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t)
&argument)) {
      fprintf (stderr, "%s", "unable to free arguments");
      exit (1);
   }
   return;
}

int
main (int argc, char **argv)
{
   register SVCXPRT *transp;

   pmap_unset (AVERAGEPROG, AVERAGEVERS);

   transp = svcudp_create(RPC_ANYSOCK);
   if (transp == NULL) {
      fprintf (stderr, "%s", "cannot create udp service.");
      exit(1);
   }
   if (!svc_register(transp, AVERAGEPROG, AVERAGEVERS, averageprog_1,
IPPROTO_UDP)) {
      fprintf (stderr, "%s", "unable to register (AVERAGEPROG,
```

```
AVERAGEVERS, udp).");
      exit(1);
   }

   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   if (transp == NULL) {
      fprintf (stderr, "%s", "cannot create tcp service.");
      exit(1);
   }
   if (!svc_register(transp, AVERAGEPROG, AVERAGEVERS, averageprog_1,
IPPROTO_TCP)) {
      fprintf (stderr, "%s", "unable to register (AVERAGEPROG,
AVERAGEVERS, tcp).");
      exit(1);
   }

   svc_run ();
   fprintf (stderr, "%s", "svc_run returned");
   exit (1);

}
```

avg_xdr.c

```
#include "avg.h"

bool_t
xdr_input_data (XDR *xdrs, input_data *objp)
{
   register int32_t *buf;

    if (!xdr_array (xdrs, (char **)&objp->input_data.input_data_val,
(u_int *) &objp->input_data.input_data_len, 200,
      sizeof (double), (xdrproc_t) xdr_double))
       return FALSE;
   return TRUE;
}
```

client.c

```
#include "avg.h"
#include <stdlib.h>

void
averageprog_1( char* host, int argc, char *argv[])
{
   CLIENT *clnt;
   double  *result_1, *dp, f;
      char *endptr;
      int i;
   input_data  average_1_arg;
     average_1_arg.input_data.input_data_val =
     (double*) malloc(MAXAVGSIZE*sizeof(double));
     dp = average_1_arg.input_data.input_data_val;
     average_1_arg.input_data.input_data_len =
         argc - 2;
```

```
    for (i=1;i<=(argc - 2);i++) {
      f = strtod(argv[i+1],&endptr);
      printf("value   = %e\n",f);
      *dp = f;
      dp++;
    }
 clnt = clnt_create(host, AVERAGEPROG,
AVERAGEVERS, "udp");
 if (clnt == NULL) {
    clnt_pcreateerror(host);
    exit(1);
 }
 result_1 = average_1(&average_1_arg, clnt);
 if (result_1 == NULL) {
    clnt_perror(clnt, "call failed:");
    }
    clnt_destroy( clnt );
       printf("average = %e\n",*result_1);
}


main( int argc, char* argv[] )
{
    char *host;

    if(argc < 3) {
      printf(
       "usage: %s server_host value ...\n",
       argv[0]);
       exit(1);
    }
         if(argc > MAXAVGSIZE + 2) {
           printf("Two many input values\n");
           exit(2);
         }
    host = argv[1];
    averageprog_1( host, argc, argv);
}
```

server.c

```
#include <rpc/rpc.h>
#include "avg.h"
#include <stdio.h>

static double sum_avg;

double * average_1(input_data *input,
   CLIENT *client) {

  double *dp = input->input_data.input_data_val;
  u_int i;
  sum_avg = 0;
  for(i=1;i<=input->input_data.input_data_len;i++) {
    sum_avg = sum_avg + *dp; dp++;
  }
```

```
  sum_avg = sum_avg /
   input->input_data.input_data_len;
  return(&sum_avg);
}

double * average_1_svc(input_data *input,
   struct svc_req *svc) {
  CLIENT *client;
  return(average_1(input,client));
}
```