

# Remote Procedure Call Programming Interface Requirements Version 1.0

The following is a description of the Remote Procedure Call method of communications with the InfoSolutions<sup>®</sup> Medical Information Network. This method of communication is based on the Remote Procedure Call (RPC) standard developed by Sun Microsystems as defined in RFC1057. Data representation is maintained across platforms using the External Data Representation (XDR) standard developed by Sun Microsystems as defined in RFC1014.

Please familiarize yourself with these RFCs, which are available from the Internet. RFCs can be obtained via FTP from NIS.NSF.NET, NISC.JVNC.NET, VENERA.ISI.EDU, WUARCHIVE.WUSTL.EDU, SRC.DOC.IC.AC.UK, FTP.CONCERT.NET, DS.INTERNIC.NET, or NIC.DDN.MIL. Details on obtaining RFCs via FTP or EMAIL may be obtained by sending an EMAIL message to "rfc-info@ISI.EDU" with the message body "help: ways\_to\_get\_rfc".

For example:

To: rfc-info@ISI.EDU  
Subject: Getting rfc  
Help: Ways\_to\_get\_rfc

Requests for special distribution should be addressed to either the author of the RFC in question, or to NIC@NIC.DDN.MIL. Unless specifically noted otherwise on the RFC itself, all RFCs are for unlimited distribution.

Following is the code for an example client process. The InfoSolutions Network IP address, program number, version number and procedure number, along with vendor identification and password, will be provided by Blue Cross and Blue Shield of Alabama when testing commences.

## Client Process Programming Code Example

```
/*  
*****  
/* The CLIENT routine */  
*****  
#include <fcntl.h> /* open() */  
#include <sys/mode.h> /* open() */  
#include <malloc.h> /* Get/Free Of Main Storage */  
#include <stdio.h>  
#include <rpc/rpc.h>  
#include <netdb.h>  
#include <sys/socket.h>  
#include <sys/time.h>  
#include <sys/errno.h>  
#include "/infosol/include/aix/implode.h" /*header file for implode library*/  
#include "client.h" /*header for client*/  
  
FILE *zIn, *zOut; /*files for compression*/  
char *WorkBuff; /*working area for compression lib*/
```

```

/***** ROUTINES TO INTERFACE WITH PKZIP COMPRESSION LIBRARY
*****/
unsigned int ReadFile(char *buff, unsigned short int *size) /* Read From File*/
{
    unsigned iStatus;
    iStatus=fread(buff,1,*size,zIn);
    return(iStatus);
}

unsigned int WriteFile(char *buff, unsigned short int *size)/* Write To File*/
{
    unsigned iStatus; iStatus=fwrite(buff,1,*size,zOut);
    return(iStatus);
}
/*****/

main(int argc, char **argv)
{
    short dstatus; /*return from DES*/
    unsigned status; /*return from implode*/
    unsigned short int type = CMP_BINARY; /*type of data compressed*/
    unsigned short int dsize = 4096; /*dictionary size*/
    char sendfile_cmpFN[100]; /*sendfile compressed file name*/
    char sendfile_cmp_encFN[100]; /*sendfile compressed, encrypted file name*/
    char rcvfile_cmpFN[100]; /*rcv file compressed file name*/
    char rcvfile_cmp_encFN[100]; /*rcv file compressed, encrypted file name*/

    SUBM_REC pr; /*configuration record*/
    FILE *config; /*configuration file*/
    unsigned cread; \
    char unix_auth[13]; /*for unix-style authorization string*/

    struct hostent *hp;
    struct timeval total_timeout;
    struct sockaddr_in server_addr;

    int sock = RPC_ANYSOCK;
    FILE *stream_to_host; /*pointer to stream to host*/
    FILE *stream_from_host; /*pointer to stream from host*/
    unsigned long RCPPROG = 0x32000000; /*host program number*/
    unsigned long RCPPROC = 1; /*host procedure number*/
    unsigned long RCPVERS = 1; /*host program version number*/
    int xdr_send(); /*prototype of xdr transmit*/
    int xdr_receive(); /*prototype of xdr receive*/
    int err; /*return code from rpc*/
    enum clnt_stat clnt_stat; /*client call return code*/
    register CLIENT *client; /*pointer to client process*/
    /*test the command line for proper program usage*/
    if (argc < 2)

```

```

{
fprintf(stdout, "usage: %s config-file\n", argv[0]);
exit(-1);
}

fprintf(stdout, "Sendfile will be compressed then encrypted.\n");

/*read configuration file*/
if ((config=fopen(argv[1],"r")) == NULL)
{
fprintf(stdout, "Error opening configuration file.\n"); exit(-1);
}
fscanf(config, "%s\n%s\n%s\n%s\n%s\n%s\n%s\n",
pr.subm_nr,
pr.password,
pr.host_name,
pr.host_port,
pr.send_file,
pr.recv_file,
pr.cleanup);
fclose(config);

/*open sendfile for compression*/
if ((zIn=fopen(pr.send_file,"rb")) == NULL)
{
fprintf(stdout, "Error opening file to compress.\n"); exit(-1);
}

/*open/create file for compressed data*/ strncpy(sendfile_cmpFN,pr.send_file,strlen(pr.send_file));
strcat(sendfile_cmpFN, ".cmp");
if ((zOut=fopen(sendfile_cmpFN,"wb")) == NULL)
{
fprintf(stdout, "Error creating file for compression.\n");
exit(-1);
}

WorkBuff=(char *)malloc(65535); /* Work Area For PKZIP*/
if (WorkBuff == NULL)
{
fprintf(stdout, "Error allocating work buffer for compression.\n"); exit(-1); }
status=implode(ReadFile,WriteFile,WorkBuff,&type,&dsiz);
if (status)
{
fprintf(stdout, "Error compressing file...error %d\n",status); exit(-1);
}

free(WorkBuff);
fclose(zIn);
fclose(zOut);

```

```

/*****perform encryption*****/
/*Create filename for encrypted file*/
strncpy(sendfile_cmp_encFN,sendfile_cmpFN,strlen(sendfile_cmpFN));
strcat(sendfile_cmp_encFN, ".enc");

/*Call encryption routine*/ dstatus=DES(sendfile_cmpFN,sendfile_cmp_encFN,pr.password,'E');
if (dstatus != 0)
{
fprintf(stdout,"Error encrypting file...error %d\n",dstatus);
exit (-1);
}

/*Open stream to host*/
stream_to_host = fopen(sendfile_cmp_encFN, "r");
if (stream_to_host == NULL)
{
fprintf(stdout,"Error opening stream to host.\n");
exit(-1);
}

/*Create file name to receive from host*/
strncpy(recvfile_cmp_encFN,pr.recv_file,strlen(pr.recv_file));
strcat(recvfile_cmp_encFN, ".cmp.enc");

/*Open stream from host*/
stream_from_host = fopen(recvfile_cmp_encFN, "w");
if (stream_from_host == NULL)
{
fprintf(stdout,"Error opening stream from host.\n");
exit(-1);
}

/*Initialize socket structure*/
server_addr.sin_addr.s_addr = inet_addr(pr.host_name);
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(atoi(pr.host_port));

/*Connect to host*/
if ((client = clnttcp_create(&server_addr,RCPPROG, RCPVERS,&sock,0,0 )) == NULL)
{
fprintf(stdout,"Error creating client process...error %d\n",errno);
exit (-1);
}
/*Unix style authentication*/
strncpy(unx_auth,vendor_id,3); /*vendor id declared in client.h*/
strcat(unx_auth,"1"); /*for compressed and encrypted request*/
strcat(unx_auth,pr.subm_nr);
client->cl_auth = authunix_create(unx_auth,0,0,0,0);

```

```

/*Set-up timeouts for rpc call*/
total_timeout.tv_sec = 60;
total_timeout.tv_usec = 60;
fprintf(stdout, "Making remote call...\n");

/*Perform rpc call*/
clnt_stat = clnt_call(client,
RCPPROC,
xdr_send,
stream_to_host,
xdr_receive,
stream_from_host,
total_timeout);
if (clnt_stat != 0)
{
fprintf(stdout, "Error making call: ");
clnt_perror(client, "FROM CLNT_CALL");
}

/*Disconnect from host*/
clnt_destroy(client);

/*Close stream to host*/
if (fclose(stream_to_host) == EOF)
{
fprintf(stdout, "Error closing stream to host.\n");
}

/*Close stream from host*/
if (fclose(stream_from_host) == EOF)
{
fprintf(stdout, "Error closing stream from host.\n");
}

/*Create filename for compressed file*/ strncpy(recvfile_cmpFN, pr.recv_file, strlen(pr.recv_file));
strcat(recvfile_cmpFN, ".cmp");

/*Call to encryption routine to decrypt response*/
dstatus=DES(recvfile_cmp_encFN, recvfile_cmpFN, pr.password, 'D');
if (dstatus != 0)
{
fprintf(stdout, "Error decrypting response file...error %d\n", dstatus);
exit (-1);
}

/*Pprepare to decompress response file*/
/*Open recvfile for compression*/
if ((zIn=fopen(recvfile_cmpFN, "rb")) == NULL)

```

```

{
fprintf(stdout,"Error opening file to decompress.\n");
exit(-1);
}

/*open/create file for uncompressed data*/
if ((zOut=fopen(pr.recv_file,"wb")) == NULL)
{
fprintf(stdout,"Error creating file for decompression.\n");
exit(-1);
}

WorkBuff=(char *)malloc(65535);
/* Work Area For PKZIP*/
if (WorkBuff == NULL)
{
fprintf(stdout,"Error allocating work buffer for decompressing.\n");
exit(-1);
}

status=explode(ReadFile,WriteFile,WorkBuff);
if (status)
{
fprintf(stdout,"Unable to decompress file...error %d\n",status);
exit(-1);
}

free(WorkBuff);
fclose(zIn);
fclose(zOut);

/*clean up work files?*/
if (atoi(pr.cleanup))
{
fprintf(stdout,"Cleaning up work files...\n");
unlink(sendfile_cmpFN);
unlink(sendfile_cmp_encFN);
unlink(recvfile_cmpFN);
unlink(recvfile_cmp_encFN);
}

/*finished*/ exit(0);
}
/*****/
/* The xdr_send routine: */
/* read from fp, write onto wire */
/*****/

xdr_send(XDR *xdrs, FILE *fp)

```

```

{
unsigned long size; /*size of data to pass*/
char buf[BUFSIZ], *p; /*buffer and pointer to data*/
while (1)
{
fprintf(stdout, "top of send loop\n");

/*get send data from stream*/
if (((size = fread(buf, sizeof(char), BUFSIZ,
fp)) == 0) && ferror(fp))
{
fprintf(stdout, "can't fread\n");
return (1);
}

/*set-up pointer to data that was read*/
p = buf;

/*XDR convert the data*/
if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
{
fprintf(stdout, "!xdr_bytes\n"); return 0;
}

/*test for end of data stream*/
if (size == 0)
{
fprintf(stdout, "size=0\n");
return 1;
}
}
}

/*****
/* The xdr_receive routine: */
/* read from wire, write onto fp */
*****/

xdr_receive(XDR *xdrs, FILE *fp)
{
unsigned long size; /*size of data received*/
char buf[BUFSIZ], *p; /*buffer and pointer to data*/
while (1)
{
fprintf(stdout, "top of receive loop\n");

/*set-up data and xdr convert*/
p = buf;
if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))

```

```
{
fprintf(stdout,"!xdr_bytes\n");
return 0;
}
```

```
/*check for end of data*/
```

```
if (size == 0)
{
```

```
fprintf(stdout,"size = 0\n");
return 1;
}
```

```
/*write the received data*/
```

```
if (xdrs->x_op == XDR_DECODE)
{
fprintf(stdout,"XDR DECODE\n");
if (fwrite(buf, sizeof(char),size,fp) != size)
{
fprintf(stdout, "can't fwrite\n");
return (1);
}
}
}
```

```
/* client.h header file
```

```
char vendor_id[4]="";
typedef struct
```

```
{
```

```
char subm_nr[9]; /*TEST SUBMITTER NUMBER*/
char password[9]; /*TEST PASSWORD*/
char host_name[15]; /*HOST IP ADDRESS*/
char host_port[5]; /*HOST PORT NUMBER*/
char send_file[100]; /*FILE TO SEND TO HOST*/
char rcv_file[100]; /*FILE TO RECEIVE FROM HOST*/
char cleanup[2]; /*CLEANUP WORK FILES?*/
}SUBM_REC;
```

```
/*prototypes*/
```

```
short DES(); /*prototype for des encrypt decrypt*/
```

```
/* Test information loaded in your configuration file should follow the following format to match
the structure SUBM_REC above
```

```
SUBMID (8)
```

```
PASSWORD (8)
HOST ADDRESS
PORT
SENDFILE
RCVFILE
1
*/
```

```
#include <fcntl.h>
#include <sys/mode.h>
#include <stdio.h>
```

```
/******
/*this function will perform the encryption/decryption routine */
/*Encryption/decryption will be performed on the */
/*input_file with the result placed in the output_file. */
/*An operation type of E denotes encryption and D denotes */
/*decryption */ /******
short DES(char input_file[100], char output_file[100],
char key_value[9], char operation_type)
{
int byte_count; /* Byte Count Read/Write */
int diagcode; /* Diagnostic Return Code */
int i; /* Working Indexing Variable */
char szMsg[80]; /* Logfile message area*/
char sys_command[200]; /* System command storage area*/

/*validate the operation type*/
if ((operation_type != 'E') && (operation_type != 'D'))
{
fprintf(stdout, "Improper operation requested from DES()\n");
return(1);
}

/*if decryption decrypt, otherwise encrypt*/
if (operation_type == 'D')
{
/*set up decrypt command*/
sprintf(sys_command, "cipher -d -k %s < %s > %s 2>/dev/null", key_value, input_file, output_file);

/*perform command, if failure return*/
if (system(sys_command) != 0)
{
fprintf(stdout, "Error decrypting input file\n");
return (2);
}
}
else
{
```

```
/*set up encrypt command*/
sprintf(sys_command,
"cipher -e -k %s < %s > %s 2>/dev/null",
key_value,input_file,output_file);
fprintf(stdout,"%s\n",sys_command);
/*perform command, if failure return*/
if (system(sys_command) != 0)
{
fprintf(stdout,"Error encrypting input file.\n");
return(3);
}
}

return(0); /*return successful execution*/

} /* end of DES.C() main function */
```