# Live RPC Code Examples D

This appendix contains copies of the complete live code modules used in the `rpcgen` and RPC chapters of this book. They are compilable as they are written and will run (unless otherwise noted to be pseudo-code or the like). They are provided for informational purposes only. SunSoft assumes no liability from their use.

## Directory Listing Program and Support Routines (`rpcgen`)

*Code Example D-1* `rpcgen` Program: `dir.x`

```
/*
 * dir.x: Remote directory listing
 * protocol
 *
 * This source module is a rpcgen source module
 * used to demonstrate the functions of the rpcgen
 * tool.
 *
 * It is compiled with the rpcgen -h -T switches to
 * generate both the header (.h) file and the
 * accompanying data structures.
 *
 */
const MAXNAMELEN = 255;/*maxlengthofadirectoryentry*/

typedef string nametype<MAXNAMELEN>; /* directory entry */
typedef struct namenode *namelist;   /*linkinthelisting*/
```

```
/*
 * A node in the directory listing
 */
struct namenode {
   nametype name;       /* name of directory entry */
   namelist next;       /* next entry */
 };

/*
 * The result of a READDIR operation:
 * a truly portable application would use an agreed upon list of
 * error codes rather than, as this sample program does, rely upon
 * passing UNIX errno's back. In this example the union is used to
 *  discriminate between successful and unsuccessful remote calls.
 */
union readdir_res switch (int errno) {
   case 0:
      namelist list; /*no error: return directory listing*/
   default:
      void;      /*error occurred: nothing else to return*/
};

/*
 * The directory program definition
 */
program DIRPROG {
   version DIRVERS {
      readdir_res
      READDIR(nametype) = 1;
   } = 1;
 } = 0x20000076;
```

*Code Example D-2* Remote `dir_proc.c`

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>      /* Always needed */
#include <dirent.h>
#include "dir.h"          /* Created by rpcgen */

extern int errno;
extern char *malloc();
extern char *strdup();

/* ARGSUSED1*/
```

```
readdir_res *
readdir_1(dirname,req)
    nametype *dirname;
    struct svc_req *req;
{
    DIR *dirp;
    struct dirent *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

    /*
     * Open directory
     */
    dirp = opendir(*dirname);
    if (dirp == (DIR *)NULL) {
        res.errno = errno;
        return (&res);
    }
    /*
     * Free previous result
     */
    xdr_free(xdr_readdir_res, &res);
    /*
     * Collect directory entries. Memory allocated here is freed by
     * xdr_free the next time readdir_1 is called.
     */

    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *) malloc(sizeof(namenode));
        if (nl == (namenode *) NULL) {
            res.errno = EAGAIN;
            closedir(dirp);
            return(&res);
        }
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }
    *nlp = (namelist)NULL;
    /* Return the result */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

*Code Example D-3* `rls.c` Client

```
/*
 * rls.c: Remote directory listing client
 */

#include <stdio.h>
#include <rpc/rpc.h>/* always need this */
#include "dir.h"    /* generated by rpcgen */

extern int errno;

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n",
        argv[0]);
        exit(1);
    }
    server = argv[1];
     dir = argv[2];
/*
 * Create client "handle" used for calling MESSAGEPROG on the server
 * designated on the command line.
 */
    cl = clnt_create(server, DIRPROG, DIRVERS, "visible");
    if (cl == (CLIENT *)NULL) {
        clnt_pcreateerror(server);
        exit(1);
    }

    result = readdir_1(&dir, cl);
    if (result == (readdir_res *)NULL) {
        clnt_perror(cl, server);
        exit(1);
    }
```

```
/* Okay, we successfully called the remote procedure. */

    if (result->errno != 0) {
    /*
     * A remote system error occurred. Print error message and die.
     */
    }
    if (result->errno < sys_nerr)
        fprintf (stderr, "%s : %s\n", dir,
        sys_enlist[result->errno]);
        errno = result->errno;
        perror(dir);
        exit(1);
    }

/* Successfully got a directory listing. Print it out. */
    for(nl = result->readdir_res_u.list; nl != NULL; nl = nl->next) {
        printf("%s\n", nl->name);
    }
exit(0);
}
```

## Time Server Program (`rpcgen`)

*Code Example D-4* `rpcgen` Program: `time.x`

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
    version TIMEVERS {
            unsigned int TIMEGET(void) = 1;
    } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%   static int thetime;
%
%   thetime = time(0);
%   return (&thetime);
%}
#endif
```

## Add Two Numbers Program (rpcgen)

*Code Example D-5* rpcgen program: Add Two Numbers

```
/* This program contains a procedure to add 2 numbers to demonstrate
 * some of the features of the new rpcgen. Note that add() takes 2
 * arguments in this case.
 */
program ADDPROG {/* program number */
    version ADDVER {/* version number */
        int add ( int, int )  /* procedure */
            = 1;
    } = 1;
} = 199;
```

## Spray Packets Program (rpcgen)

Refer to the notes section on the spray(1M) man page for information about using this tool.

*Code Example D-6* rpcgen program: spray.x

```
/*
 * Copyright (c) 1987, 1991 by Sun Microsystems, Inc.
 */

/* from spray.x */

#ifdef RPC_HDR
#pragma ident "@(#)spray.h  1.2  91/09/17 SMI"
#endif

/*
 * Spray a server with packets
 * Useful for testing flakiness of network interfaces
 */

const SPRAYMAX = 8845;/* max amount can spray */

/*
 * GMT since 0:00, 1 January 1970
 */
struct spraytimeval {
    unsigned int sec;
    unsigned int usec;
```

```
};

/*
 * spray statistics
 */
struct spraycumul {
    unsigned int counter;
    spraytimeval clock;
};

/*
 * spray data
 */
typedef opaque sprayarr<SPRAYMAX>;

program SPRAYPROG {
    version SPRAYVERS {
        /*
         * Just throw away the data and increment the counter. This
         * call never returns, so the client should always time it out.
         */
        void
        SPRAYPROC_SPRAY(sprayarr) = 1;

        /*
         * Get the value of the counter and elapsed time since last
         * CLEAR.
         */
        spraycumul
        SPRAYPROC_GET(void) = 2;

        /*
         * Clear the counter and reset the elapsed time
         */
        void
        SPRAYPROC_CLEAR(void) = 3;
    } = 1;
} = 100012;
```

## Print Message Program With Remote Version

*Code Example D-7* `printmesg.c`

```
/* printmsg.c: print a message on the console */
#include <stdio.h>
```

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit(1);
    }
    message = argv[1];
    if( !printmessage(message) ) {
        fprintf(stderr, "%s: couldn’t print your message\n",
                  argv[0]);
        exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}

/* Print a message to the console. */

/*
 * Return a boolean indicating whether the message was actually
 * printed.
 */
printmessage(msg)
    char *msg;
{
    FILE *f;

    if = fopen("/dev/console","w");
        if (f == (FILE *)NULL)
            return (0);
    fprintf(f,"%sen”, msg);
    fclose(f);
    return (1);
}
```

*Code Example D-8* Remote Version of `printmesg.c`

```
/*  * rprintmsg.c: remote version of "printmsg.c”  */
#include <stdio.h>
#include <rpc/rpc.h> /* always needed */
#include "msg.h”     /* msg.h generated by rpcgen */

main(argc, argv)
```

```
    int argc;
    char *argv[];
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;
    extern int sys_nerr;
    extern char *sys_errlist[];

    if (argc != 3) {
        fprintf(stderr,"usage: %s host messagen", argv[0]);
        exit(1);
    }
    /*
     * Save values of command line arguments
     */
    server = argv[1];
    message = argv[2];
/*
 * Create client"handle" used for calling
 * MESSAGEPROG on the server
 * designated on the command line.
 */
    cl = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,
                    "visible");
    if (cl == (CLIENT *)NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }
    /* Call the remote procedure "printmessage" on the server */
    result = printmessage_1(&message, cl);
    if (result == (int *)NULL) {
    /*
     * An error occurred while calling the server.
     * Print error message and die.
     */
        clnt_perror(cl, server);
        exit(1);
    }
    /* Okay, we successfully called the remote procedure. */
    if (*result == 0) {
```

```
        /*
         * Server was unable to print our message.
         * Print error message and die.
         */
        fprintf(stderr,"%s"
    }
/* The message got printed on the server's console */
    printf("Message delivered to %s!\n", server);
    exit(0);
}
```

*Code Example D-9* `rpcgen` Program: `msg.x`

```
/* msg.x: Remote message printing protocol */
program MESSAGEPROG {
    version MESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
    } = 1;
} = 0x20000001;
```

*Code Example D-10* `mesg_proc.c`

```
/*
 * msg_proc.c: implementation of the remote
 * procedure "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h> /* always needed */
#include "msg.h" /* msg.h generated by rpcgen */

/*
 * Remote version of "printmessage"
 */
/*ARGSUSED1*/
int printmessage_1(msg, req)
    char **msg;
    struct svc_req *req;
{
    static int result; /* must be static! */
    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == (FILE *)NULL) {
        result = 0;
        return (&result);
```

```
    }
    fprintf(f, "%sen", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

## Batched Code Example

*Code Example D-11* Batched Client Program

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

main(argc, argv)
    int              argc;
    char           **argv;
{
    struct timeval  total_timeout;
    register CLIENT *client;
    enum clnt_stat  clnt_stat;
    char            buf[1000], *s = buf;

    if ((client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS,
                  "CIRCUIT_V")) == (CLIENT *) NULL) {
        clnt_pcreateerror("clnt_create");
        exit(1);
    }

    timerclear(&total_timeout);
    while (scanf("%s", s) != EOF) {
        clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
          &s,xdr_void, (caddr_t) NULL, total_timeout);
    }

    /* Now flush the pipeline */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void,
           (caddr_t) NULL, xdr_void, (caddr_t) NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(1);
    }
```

```
    clnt_destroy(client);
    exit(0);
}
```

*Code Example D-12* Batched Server Program

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

void            windowdispatch();
main()
{
    int num;

    num = svc_create(windowdispatch, WINDOWPROG, WINDOWVERS,
          "CIRCUIT_V");
    if (num == 0) {
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    svc_run();                          /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT        *transp;
{
    char           *s = NULL;

    switch (rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RENDERSTRING:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");
                /* Tell caller an error occurred */
                svcerr_decode(transp);
                break;
            }
            /* Code here to render the string s */
            if (!svc_sendreply(transp, xdr_void, (caddr_t) NULL))
```

```
            fprintf(stderr, "can't reply to RPC call\n");
        break;
    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /* Be silent in the face of protocol errors */
            break;
        }
        /* Code here to render string s, but send no reply! */
        break;
    default:
        svcerr_noproc(transp);
        return;
    }
    /* Now free string allocated while decoding arguments */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

## Non-Batched Example

This example is included for reference only. It is a version of the batched client string rendering service, Code Example D-11, written in as a non-batched program.

*Code Example D-13* Unbatched Version of Batched Client

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

main(argc, argv)
    int             argc;
    char          **argv;
{
    struct timeval  total_timeout;
    register CLIENT *client;
    enum clnt_stat  clnt_stat;
    char            buf[1000], *s = buf;

    if ((client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS,
                "CIRCUIT_V")) == (CLIENT *) NULL) {
        clnt_pcreateerror("clnt_create");
        exit(1);
    }
```

```
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    while (scanf("%s", s) != EOF) {
        if(clnt_call(client, RENDERSTRING, xdr_wrapstring, &s,
            xdr_void, (caddr_t) NULL, total_timeout) != RPC_SUCCESS) {
            clnt_perror(client, "rpc");
            exit(1);
        }
    }
    clnt_destroy(client);
    exit(0);
}
```