

# Zdalne wywoływanie procedur Sun RPC

Witold Paluszyński

witoldp@pwr.wroc.pl

<http://sequoia.ict.pwr.wroc.pl/~witold/>

Copyright © 2000,2004 Witold Paluszyński  
All rights reserved.

Niniejszy dokument zawiera materiały do wykładu na temat systemu zdalnego wywoływania procedur Sun RPC w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.



# Zdalne wywoływanie procedur — Sun RPC

- Schemat systemu zdalnego wywoływania procedur: klient wywołuje procedurę na zdalnym systemie przekazując jej argumenty, a serwer oblicza wyniki i je odsyła.
- Ten model został wprowadzony przez firmę Sun i następnie oddany do *public domain*, tzn. można go zainstalować i używać na dowolnym systemie. Istnieje inny system RPC o nazwie DCE również dostępny na wielu systemach, o bardzo podobnej konstrukcji, ale mniej popularny.
- Wywoływana procedura jest identyfikowana przez numer programu, numer procedury, i numer wersji.
- Przekazywanie parametru i wyniku: kodowanie i konwersja formatu danych: XDR.
- Model zapewnia wybór stylu autentykacji użytkownika, w tym styl *none* (tzn. brak autentykacji), styl unixowy (login i hasło), i inne, oparte na szyfrowaniu i kluczach publicznych.
- Wyższy poziom wywołań RPC: prostszy, mniejszy wybór możliwości.
- Niższy poziom RPC: więcej możliwości i generator interface'u `rpcgen`.

# Wyższy poziom RPC

- funkcja klienta wywołująca zdalną procedurę

```
callrpc(char *host, u_long prognum, u_long versnum, u_long procnum,  
        xdrproc_t inproc, char *in,  
        xdrproc_t outproc, char *out);
```

- xdrproc\_t inproc – filtr do kodowania argumentu
- char \*in – wskaźnik do argumentu zdalnej funkcji
- xdrproc\_t outproc – filtr do rozkodowania wyniku
- char \*out - wskaźnik do wyniku ze zdalnej funkcji

⇒ Oba filtry mają działanie dwukierunkowe: kodujące i rozkodujące.

- funkcje serwera

```
registerrpc(u_long prognum, u_long versnum, u_long procnum,  
            char *(*procname)(),  
            xdrproc_t inproc, xdrproc_t outproc);
```

```
void svc_run(void);
```

⇒ W podstawowej wersji działanie serwera jest iteracyjne.

# Wyższy poziom RPC — klient

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h> /* req'd for prog,vers definitions */

main(int argc, char **argv) {
    unsigned long rusers;
    int stat;

    if (argc != 2) {
        fprintf(stderr, "usage: rusers hostname\n");
        exit (1);
    }
    if (stat = callrpc(argv[1],
        RUSERSPROC, RUSERSVERS, RUSERSPROC_NUM,
        xdr_void, (char *)0,
        xdr_u_long, (char *)&rusers) != 0){
        clnt_perrno(stat);
        exit (1);
    }
    printf("Rusers on %s returned %ld\n", argv[1], rusers);
    exit (0);
}
```

# Wyższy poziom RPC — serwer

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h> /* req'd for prog,vers definitions */

char * ruser (char *indata) {
    static unsigned long rusers;
    /*
     * Code here to compute the number of users
     * and place result in variable rusers.
     */
    return((char *) &rusers);
}

int main () {
    if (registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
                   ruser, xdr_void, xdr_u_long) != 0) {
        perror("registerrpc failed");
    }
    svc_run(); /* Never returns */
    fprintf(stderr, "Error: svc run returned!\n");
    exit (1);
}
```

# Kodowanie danych w systemie XDR

- Kodowanie przesyłanych danych procedurami XDR (*eXternal Data Representation*) zapewnia izolację od różnic architektury maszyn, porządku bajtów w słowie, ułożenia danych struktury, itp.
- System XDR posiada gotowe funkcje konwersji dla wielu standardowych typów danych:

<code>xdr_int()</code>	<code>xdr_u_int()</code>	<code>xdr_enum()</code>
<code>xdr_long()</code>	<code>xdr_u_long()</code>	<code>xdr_bool()</code>
<code>xdr_short()</code>	<code>xdr_u_short()</code>	<code>xdr_wrapstring()</code>
<code>xdr_char()</code>	<code>xdr_u_char()</code>	

- Dla innych (złożonych) typów danych, procedury konwersji XDR trzeba napisać, zgodnie z obowiązującym schematem.

Mają one służyć do dekodowania i kodowania danych i posiadać dwa argumenty. Pierwszy jest wskaźnikiem na strukturę zakodowaną, a drugi na oryginalną daną. Funkcje zwracają 0 w przypadku porażki konwersji, a wppw wartość niezerową.

- Dla prostych typów danych użytkownika o określonej wielkości, takich jak struktury, funkcję konwersji można łatwo napisać posługując się gotowymi funkcjami wbudowanymi, na przykład:

```
struct simple {  
    int a;  
    short b;  
};
```

```
int xdr_simple(XDR *xdrsp, struct simple *simplep)  
{  
    if (!xdr_int(xdrsp, &simplep->a))  
        return(0);  
    if (!xdr_short(xdrsp, &simplep->b))  
        return(0);  
    return(1);  
}
```

- Dla tworzenia takich funkcji konwersji dla bardziej skomplikowanych typów danych system XDR dostarcza pewne gotowe prefabrykaty:

xdr_array()	xdr_bytes()	xdr_reference()
xdr_vector()	xdr_union()	xdr_pointer()
xdr_string()	xdr_opaque()	



- Dla tablic o stałej długości służy: `xdr_vector()`

```
int intarr[SIZE];

int xdr_intarr(XDR *xdrsp, int intarr[]) {
    return (xdr_vector(xdrsp, intarr,
                      SIZE, sizeof(int), xdr_int));
}
```

- Natomiast dla tablic o zmiennej długości można użyć następującego schematu z funkcją `xdr_array()`

```
struct varintarr {
    int *data;
    int arrlen;
};

int xdr_varintarr(XDR *xdrsp, struct varintarr *arrp) {
    return (xdr_array(xdrsp, &arrp->data, &arrp->arrlen,
                     MAXLEN, sizeof(int), xdr_int));
}
```

- Ponieważ system XDR zamienia wszystkie elementy na wielokrotności 4 bajtów, co byłoby niekorzystne w przypadku pojedynczych znaków, dlatego istnieje funkcja `xdr_bytes()`, podobna do funkcji `xdr_array()`, lecz upakowująca znaki.
- W następującym przykładzie użyto funkcji `xdr_string()` kodującej napisy znakowe zakończone znakiem NULL, oraz funkcji `xdr_reference()`, która podąża za wskaźnikami w strukturach:

```
struct finalexample {
    char *string;
    struct simple *simplep;
};

xdr_finalexample(XDR *xdrsp, struct finalexample *finalp) {
    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
        return (0);
    if (!xdr_reference (xdrsp, &finalp->simplep,
                        sizeof(struct simple), xdr_simple))
        return (0);
    return (1);
}
```

# Rejestracja usług RPC — serwer rpcbind

- portmapper (port 111) — program zarządzający procedurami dostępnymi na danym komputerze

```
% rpcinfo
      program version netid      address          service    owner
      100000    4      ticots    sequoia.rpc      rpcbind    superuser
      100000    3      ticots    sequoia.rpc      rpcbind    superuser
      100000    4      ticotsord sequoia.rpc      rpcbind    superuser
      100000    3      ticotsord sequoia.rpc      rpcbind    superuser
      100000    4      ticlts    sequoia.rpc      rpcbind    superuser
      100000    3      ticlts    sequoia.rpc      rpcbind    superuser
      100000    4      tcp       0.0.0.0.0.111    rpcbind    superuser
      100000    2      udp       0.0.0.0.0.111    rpcbind    superuser
```

- przykład zapytania do zdalnego portmappera:

```
% rpcinfo -p
      program vers proto  port  service
      100000    4    tcp   111   rpcbind
      100000    3    tcp   111   rpcbind
      100000    2    tcp   111   rpcbind
      100000    4    udp   111   rpcbind
      100000    3    udp   111   rpcbind
      100000    2    udp   111   rpcbind
      ...
      2100000000 1    udp  36680
```

- wykaz programów z dostępnymi mechanizmami transportu

```
% rpcinfo -s localhost
```

program	version(s)	netid(s)	service	owner
100011	1	ticlts,udp	rquotad	superuser
100024	1	ticots,ticotsord,ticlts,tcp,udp	status	superuser
100021	4,3,2,1	tcp,udp	nlockmgr	superuser
100005	3,2,1	ticots,ticotsord,tcp,ticlts,udp	mountd	superuser
100003	3,2	tcp,udp	nfs	superuser
100227	3,2	tcp,udp	nfs_acl	superuser

- sprawdzenie statystyk RPC zdalnego systemu

```
% rpcinfo -m diablo
```

- nazwy dobrze znanych usług RPC

```
% less /etc/rpc
```

- sprawdzenie konkretnego programu i wersji

```
% rpcinfo -l localhost 2100000000 1
```

program	vers	tp_family/name/class	address	service
2100000000	1	inet/udp/clts	156.17.9.3.143.72	-

- wywołanie procedury 0 w danym programie:

```
% rpcinfo -T tcp localhost 100024
program 100024 version 1 ready and waiting
```

# Ograniczenia wyższego poziomu RPC

- ograniczenia funkcji `callrpc()`:
  - brak kontroli nad czasem oczekiwania na odpowiedź (funkcja `callrpc()` ponawia próbę wywołania kilka razy)
  - brak możliwości wyboru protokołu warstwy transportowej, wyłącznie UDP (max. 8K, zawodny)
  - brak możliwości autentykacji, tzn. styl *none*
- ograniczenia funkcji `svc_run()`:
  - iteracyjne działanie serwera

# Niższy poziomu RPC

- funkcje niższego poziomu RPC:
  - zmiana protokołu warstwy transportowej (np. TCP)
  - ustawianie czasów czekania i retransmisji
  - użycie autentykacji przez przedstawienie akredytacji
  - jawne przydzielanie i zwalnianie pamięci w funkcjach XDR
- różnice w budowie serwera:
  - utworzenie *uchwyty* serwera z możliwością wyboru gniazdka i protokołu warstwy transportowej
  - rejestracja programu (i wersji), ale nie procedury (*dispatch procedure*)
  - możliwa praca bez rejestracji w portmapperze (na wybranym porcie)
  - jawne tworzenie procedury 0 (pustej)
  - jawne odbieranie argumentu
  - jawne wysyłanie wyniku
- różnice w budowie klienta:
  - dostęp do struktur adresowych i gniazdka
  - ustawianie czasu oczekiwania i retransmisji

# Niższy poziom RPC — serwer

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main() {
    SVCXPRT *transp;
    int nuser();

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
                     nuser, IPPROTO_UDP)) {
        fprintf(stderr, "can't register RUSER service\n");
        exit (1);
    }
    svc_run(); /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

```

nuser (struct svc_req *rqstp, SVCXPRT *transp)
{
    unsigned long nusers;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    case RUSERSPROC_NUM:
        /*
         * Code here to compute the number of users
         * and assign it to the variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}

```



# Niższy poziom RPC — klient

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/time.h>
#include <netdb.h>

main(int argc, char **argv) {
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit (-1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, "can't get addr for %s\n", argv[1]);
        exit(-1);
    }
}
```

```

pertry_timeout.tv_sec = 3;
pertry_timeout.tv_usec = 0;
bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
server_addr.sin_family = AF_INET;
server_addr.sin_port = 0;
if ((client = clntudp_create(&server_addr, RUSERSPROG, RUSERSVERS,
                           pertry_timeout, &sock)) == NULL) {
    clnt_pcreateerror("clntudp_create");
    exit(-1);
}
total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
                     0, xdr_u_long, &nusers, total_timeout);
if (clnt_stat != RPC_SUCCESS) {
    clnt_perror(client, "rpc");
    exit(-1);
}
printf("%d users on %s\n", nusers, argv[1]);
clnt_destroy(client);
exit(0);
}

```

# Narzędzie rpcgen

- generuje podstawowy kod RPC obsługujący zdalne wywołania i przesyłanie argumentów
- dodatkowo może wygenerować szkieletowe wersje programów klienta i serwera, oraz makefile
- umożliwia wywoływanie serwera bezpośrednio i za pośrednictwem innych programów, np. inetd, z opcją jedno- lub wielorazowego wywoływania i *timeout*-ami
- pozwala wybierać mechanizm(y) transportowy(e)
- opiera się na prostym języku specyfikacji funkcji zdalnego wywoływania
- wiersze % przesyłane wprost do plików wynikowych
- kontekstowo definiuje makra preprocesora `RPC_HDR`, `RPC_XDR`, `RPC_SVC`, `RPC_CLNT`, `RPC_TBL`

# Narzędzie rpcgen — przykład

Dla przykładowej specyfikacji programu RPC o nazwie `mini.x`:

```
struct seedstruct {
    bool newseed;
    long seed;
};

program MINISERVER {
    version ONE{
        double COMPUTE_SIN( double ) = 1;
        void DISPLAY_MSG( string ) = 2;
        long DRAW_LONG( seedstruct ) = 3;
    } = 1;
} = 0x23456789;
```

możemy otrzymać następujące elementy:

- moduły: `mini.h` `mini_xdr.c` `mini_clnt.c` `mini_svc.c`
- opcjonalnie, moduły: `mini_client.c` `mini_server.c`

- mini.h

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

#ifndef _MINI_H_RPCGEN
#define _MINI_H_RPCGEN

#include <rpc/rpc.h>

struct draw {
    int seconds;
    char *strng;
};
typedef struct draw draw;

#define minis    0x23456789
#define one      1
#define sleep_and_draw  1
extern long * sleep_and_draw_1();
extern int minis_1_freeresult();

/* the xdr functions */
```

```
extern bool_t xdr_draw();  
  
#endif /* !_MINI_H_RPCGEN */
```

- mini\_xdr.c

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

#include "mini.h"

bool_t
xdr_seedstruct(register XDR *xdrs, seedstruct *objp)
{

    #if defined(_LP64) || defined(_KERNEL)
        register int *buf;
    #else
        register long *buf;
    #endif

    if (!xdr_bool(xdrs, &objp->newseed))
        return (FALSE);
    if (!xdr_long(xdrs, &objp->seed))
        return (FALSE);
    return (TRUE);
}
```

## ● mini\_svc.c

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

#include "mini.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>
#include <rpc/pmap_clnt.h> /* for pmap_unset */
#include <string.h> /* strcmp */
#include <unistd.h> /* setsid */
#include <sys/types.h>
#include <memory.h>
#include <stropts.h>
#include <netconfig.h>
#include <sys/resource.h> /* rlimit */
#include <syslog.h>

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

#ifdef DEBUG
#define RPC_SVC_FG
#endif

#define _RPCSVC_CLOSEDOWN 120
static int _rpcpmstart;          /* Started by a port monitor ? */

/* States a server can be in wrt request */

#define _IDLE 0
#define _SERVED 1
```



```

static int _rpcsvcstate = _IDLE;          /* Set when a request is serviced */
static int _rpcsvccount = 0;              /* Number of requests being serviced */

static
void _msgout(char* msg)
{
#ifdef RPC_SVC_FG
    if (_rpcpmstart)
        syslog(LOG_ERR, msg);
    else
        (void) fprintf(stderr, "%s\n", msg);
#else
    syslog(LOG_ERR, msg);
#endif
}

static void
closedown(int sig)
{
    if (_rpcsvcstate == _IDLE && _rpcsvccount == 0) {
        int size;
        int i, openfd = 0;

        size = svc_max_pollfd;
        for (i = 0; i < size && openfd < 2; i++)
            if (svc_pollfd[i].fd >= 0)
                openfd++;
        if (openfd <= 1)
            exit(0);
    } else
        _rpcsvcstate = _IDLE;

    (void) signal(SIGALRM, (SIG_PF) closedown);
    (void) alarm(_RPCSVC_CLOSEDOWN/2);
}

```

```

static void
miniserver_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        double compute_sin_1_arg;
        char *display_msg_1_arg;
        seedstruct draw_long_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

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

    case COMPUTE_SIN:
        _xdr_argument = (xdrproc_t) xdr_double;
        _xdr_result = (xdrproc_t) xdr_double;
        local = (char *(*)(char *, struct svc_req *)) compute_sin_1_svc;
        break;

    case DISPLAY_MSG:
        _xdr_argument = (xdrproc_t) xdr_wrapstring;
        _xdr_result = (xdrproc_t) xdr_void;
        local = (char *(*)(char *, struct svc_req *)) display_msg_1_svc;
        break;

    case DRAW_LONG:
        _xdr_argument = (xdrproc_t) xdr_seedstruct;

```

```

        _xdr_result = (xdrproc_t) xdr_long;
        local = (char *(*)(char *, struct svc_req *)) draw_long_1_svc;
        break;

default:
    svcerr_noproc(transp);
    _rpcsvccount--;
    _rpcsvcstate = _SERVED;
    return;
}
(void) memset((char *)&argument, 0, sizeof (argument));
if (!svc_getargs(transp, _xdr_argument, (caddr_t) &argument)) {
    svcerr_decode(transp);
    _rpcsvccount--;
    _rpcsvcstate = _SERVED;
    return;
}
result = (*local)((char *)&argument, rqstp);
if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
    svcerr_systemerr(transp);
}
if (!svc_freeargs(transp, _xdr_argument, (caddr_t) &argument)) {
    _msgout("unable to free arguments");
    exit(1);
}
_rpcsvccount--;
_rpcsvcstate = _SERVED;
return;
}

main()
{
    pid_t pid;
    int i;

    (void) sigset(SIGPIPE, SIG_IGN);

```

```

/*
 * If stdin looks like a TLI endpoint, we assume
 * that we were started by a port monitor. If
 * t_getstate fails with TBADF, this is not a
 * TLI endpoint.
 */
if (t_getstate(0) != -1 || t_errno != TBADF) {
    char *netid;
    struct netconfig *nconf = NULL;
    SVCXPRT *transp;
    int pmclose;

    _rpcpmstart = 1;
    openlog("mini", LOG_PID, LOG_DAEMON);

    if ((netid = getenv("NLSPROVIDER")) == NULL) {
        /* started from inetd */
        pmclose = 1;
    } else {
        if ((nconf = getnetconfigent(netid)) == NULL)
            _msgout("cannot get transport info");

        pmclose = (t_getstate(0) != T_DATAXFER);
    }
    if ((transp = svc_tli_create(0, nconf, NULL, 0, 0)) == NULL) {
        _msgout("cannot create server handle");
        exit(1);
    }
    if (nconf)
        freenetconfigent(nconf);
    if (!svc_reg(transp, MINISERVER, ONE, miniserver_1, 0)) {
        _msgout("unable to register (MINISERVER, ONE).");
        exit(1);
    }
    if (pmclose) {

```

```

                (void) signal(SIGALRM, (SIG_PF) closedown);
                (void) alarm(_RPCSVC_CLOSEDOWN/2);
            }
            svc_run();
            exit(1);
            /* NOTREACHED */
        }
        else {
#ifdef RPC_SVC_FG
            int size;
            struct rlimit rl;
            pid = fork();
            if (pid < 0) {
                perror("cannot fork");
                exit(1);
            }
            if (pid)
                exit(0);
            rl.rlim_max = 0;
            getrlimit(RLIMIT_NOFILE, &rl);
            if ((size = rl.rlim_max) == 0)
                exit(1);
            for (i = 0; i < size; i++)
                (void) close(i);
            i = open("/dev/null", 2);
            (void) dup2(i, 1);
            (void) dup2(i, 2);
            setsid();
            openlog("mini", LOG_PID, LOG_DAEMON);
#endif
        }
        if (!svc_create(miniserver_1, MINISERVER, ONE, "netpath")) {
            _msgout("unable to create (MINISERVER, ONE) for netpath.");
            exit(1);
        }

        svc_run();

```

```
    _msgout("svc_run returned");  
    exit(1);  
    /* NOTREACHED */  
}
```

## ● mini\_clnt.c

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

#include <memory.h> /* for memset */
#include "mini.h"
#ifdef _KERNEL
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#endif /* !_KERNEL */

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

double *
compute_sin_1(double *argp, CLIENT *clnt)
{
    static double clnt_res;

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

```

}

void *
display_msg_1(char **argp, CLIENT *clnt)
{
    static char clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, DISPLAY_MSG,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_void, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return ((void *)&clnt_res);
}

long *
draw_long_1(seedstruct *argp, CLIENT *clnt)
{
    static long clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, DRAW_LONG,
        (xdrproc_t) xdr_seedstruct, (caddr_t) argp,
        (xdrproc_t) xdr_long, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

```



}

## ● makefile.mini

```
# This is a template makefile generated          by rpcgen

# Parameters

CLIENT = mini_client
SERVER = mini_server

SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = mini.x

TARGETS_SVC.c = mini_svc.c mini_server.c mini_xdr.c
TARGETS_CLNT.c = mini_clnt.c mini_client.c mini_xdr.c
TARGETS = mini.h mini_xdr.c mini_clnt.c mini_svc.c mini_client.c mini_server.c

OBJECTS_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECTS_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
# Compiler flags

CFLAGS += -g
LDLIBS += -lnsl
RPCGENFLAGS =

# Targets

all : $(CLIENT) $(SERVER)

$(TARGETS) : $(SOURCES.x)
            rpcgen $(RPCGENFLAGS) $(SOURCES.x)

$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $(SOURCES_CLNT.h) $(TARGETS_CLNT.c)
```

```
$(OBJECTS_SVC) : $(SOURCES_SVC.c) $(SOURCES_SVC.h) $(TARGETS_SVC.c)
```

```
$(CLIENT) : $(OBJECTS_CLNT)  
            $(LINK.c) -o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)
```

```
$(SERVER) : $(OBJECTS_SVC)  
            $(LINK.c) -o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)
```

```
clean:
```

```
    $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)
```

## ● mini\_server.c

```
/*
 * 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 "mini.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>

double *
compute_sin_1_svc(double *argp, struct svc_req *rqstp)
{
    static double  result;

    /*
     * insert server code here
     */

    return (&result);
}

void *
display_msg_1_svc(char **argp, struct svc_req *rqstp)
{
    static char * result;
```

```

    /*
     * insert server code here
     */

    return((void *) &result);
}

long *
draw_long_1_svc(seedstruct *argp, struct svc_req *rqstp)
{
    static long  result;

    /*
     * insert server code here
     */

    return (&result);
}

```

## ● mini\_client.c

```
/*
 * 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 "mini.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */

void
miniserver_1(char *host)
{
    CLIENT *clnt;
    double *result_1;
    double compute_sin_1_arg;
    void *result_2;
    char * display_msg_1_arg;
    long *result_3;
    seedstruct draw_long_1_arg;

#ifdef DEBUG
    clnt = clnt_create(host, MINISERVER, ONE, "netpath");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
#endif /* DEBUG */

    result_1 = compute_sin_1(&compute_sin_1_arg, clnt);
    if (result_1 == (double *) NULL) {
        clnt_perror(clnt, "call failed");
    }
}
```

```

    result_2 = display_msg_1(&display_msg_1_arg, clnt);
    if (result_2 == (void *) NULL) {
        clnt_perror(clnt, "call failed");
    }
    result_3 = draw_long_1(&draw_long_1_arg, clnt);
    if (result_3 == (long *) NULL) {
        clnt_perror(clnt, "call failed");
    }
#ifdef DEBUG
    clnt_destroy(clnt);
#endif
    /* DEBUG */
}

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

    if (argc < 2) {
        printf("usage:  %s server_host\n", argv[0]);
        exit(1);
    }
    host = argv[1];
    miniserver_1(host);
}

```

# Rozsyłanie (broadcast) RPC

- istnieje możliwość skierowania wywołań RPC do grupy serwerów za pomocą rozsyłania (*broadcast*); nadawcę nie ma wtedy kontroli czy i który serwer otrzyma wywołanie, a serwer nie ma gwarancji, że klient otrzyma akurat jego odpowiedź
- funkcja `clnt_broadcast` podobna do `callrpc`

```
enum clnt_stat clnt_broadcast(u_long prognum,  
                             u_long versnum, u_long procnum, xdrproc_t inproc,  
                             char *in, xdrproc_t outproc, char *out,  
                             resultproc_t eachresult);
```

- należy napisać własną funkcję `eachresult`, która zostanie wywołana dla każdej odebranej odpowiedzi i otrzyma zdekodowany wynik odebrany od jednego z serwerów

```
eachresult(char *out, struct sockaddr_in *addr);
```

- gdy `eachresult` zwróci `TRUE`, `clnt_broadcast` kończy pracę; do tej chwili czeka, okresowo powtarzając wywołanie RPC



- ograniczenia rozsyłania RPC
  - wyłącznie UDP
  - wywołania max 1400 bajtów
  - odpowiedzi max 8800 bajtów
  - tylko serwery zarejestrowane w portmapperach
  - należy oczekiwać i odebrać wiele odpowiedzi
  - błędne odpowiedzi nie zostaną odebrane (np. niezgodność wersji)

# Kolejkowanie (batching) RPC

- wywołania RPC mogą być kolejkowane
  - serwer nie odsyła odpowiedzi
  - klient nie oczekuje odpowiedzi (NULL)
  - timeout = 0
  - wymagane użycie niezawodnego transportu (TCP)
  - po serii wywołań kolejkowanych konieczne wywołanie zwykłe w celu „opróżnienia” kolejki

```
client = clnt_create(server, OUT_STRING_PROG, OUT_STRING_VERS, "tcp");
```

```
total_timeout.tv_sec = 0;
total_timeout.tv_usec = 0;
while (scanf("%s"), s) != EOF) {
    clnt_stat = clnt_call(client, OUT_STRING_B,
        xdr_wrapstring, &s, NULL, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "kolejkowanie RPC");
        exit(-1);
    }
}
```

```
total_timeout.tv_sec = 20;
clnt_stat = clnt_call(client, NULLPROC,
    xdr_void, NULL, xdr_void, NULL, total_timeout);
if (clnt_stat != RPC_SUCCESS) {
    clnt_perror(client, "kolejkowanie RPC");
    exit(-1);
}
```

# Autentykacja

W systemie Sun RPC możliwe są różne systemy autentykacji:

autentykacja = przedstawienie akredytacji przez klienta  
+  
jej weryfikacja przez serwer

Pakiet RPC zawiera sekcję *credentials* (informacje identyfikujące nadawcę) oraz *verifier* (informacje pozwalające potwierdzić akredytację).

Podstawowym schematem autentykacji jest jej brak (schemat AUTH\_NONE), w którym zarówno akredytacja jak i weryfikacja jest pusta. Istnieje również schemat AUTH\_SYS (albo AUTH\_UNIX) polegający na przedstawieniu danych użytkownika (UID, GID), jednak w tym schemacie tylko sekcja akredytacji jest wypełniona (brak możliwości weryfikacji tych informacji). Dopiero schematy oparte o szyfrowanie z użyciem kluczy publicznych (AUTH\_DES i AUTH\_KERB) pozwalają na weryfikację wywołania przez serwer.

```
clnt = clntudp_create(address, prognum, versnum, wait, sock);
```

```
clnt->cl_auth = authnone_create();    /* domyslne */
```

# Autentykacja stylu UNIX (SYS)

```
/* utworzenie i wypelnienie struktury akredytacji stylu Unix: */  
auth_destroy(cl->cl_auth);  
clnt->cl_auth = authsys_create_default();
```

```
/* tworzona jest nastepujaca struktura akredytacji: */  
struct authsys_parms {  
    u_long    aup_time;  
    char      *aup_machname;  
    uid_t     aup_uid;  
    gid_t     aup_gid;  
    u_int     aup_len;  
    gid_t     *aup_gids;  
};
```

# Autentykacja stylu UNIX — serwer

```
nuser (struct svc_req *rqstp, SVCXPRT *transp) {
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;

    /* we don't care about authentication for null proc */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
        return;
    }

    /* now get the uid */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        unix_cred = (struct authunix_parms *) rqstp->rq_clntcred;
        uid = unix_cred->aup_uid;
        break;
    case AUTH_NULL:
    default: /* return weak authentication error */
        svcerr_weakauth (transp);
        return;
    }
}
```

```

switch (rqstp->rq_proc) {
case RUSERSPROC_NUM:
    /* make sure caller is allowed to call this proc */
    if (uid == 16) {
        svcerr_systemerr (transp);
        return;
    }
    /* Code here to compute the number of users
     * and assign it to the variable nusers */
    if (!svc_sendreply(transp, xdr_u_long, &nusers))
        fprintf(stderr, "can't reply to RPC call\n");
    return;

default:
    svcerr_noproc (transp);
    return;
}
}

```

# Autentykacja stylu DES

```
/* akredytacja stylu DES (klucze publiczne): */  
clnt->cl_auth = authdes_seccreate(servername, 60, &servaddr, NULL);  
  
char servername[MAXNAMELEN]; /* nazwa procesu serwera */  
  
/* gdy serwer pracuje jako root */  
host2netname(servername, rhostname, NULL);  
  
/* gdy serwer pracuje jako zwykly uzytkownik */  
user2netname(servername, getuid(), NULL);
```





# Autentykacja stylu DES — serwer

```
#include <sys/time.h>
#include <rpc/auth_des.h>

nuser (struct svc_req *rqstp, SVCXPRT *transp) {
    struct authdes_cred *des_cred;
    int uid, gid, gidlen, gidlist[10];

    /* we don't care about authentication for null proc */

    if (rqstp->r_proc == NULLPROC) {
        /* same as before */
    }

    /* now get the uid */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_DES:
        des_cred = (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user(des_cred->adc_fullname.name,
                           &uid, &gid, &gidlen, gidlist)) {
            fprintf(stderr, "unknown user: %s\n", des_cred->adc_fullname.name);
            svcerr_systemerr (transp);
            return;
        }
    }
```

```
    break;

    case AUTH_NULL:
    default:
        svcerr_weakauth (transp);
        return;
    }
}
```