Porting an RPC Application to TI-RPC

F

This appendix is a tutorial on: porting an existing RPC application to TI-RPC. It discusses porting issues and differences in writing TI-RPC applications compared to RPC applications, with examples.

Porting an RPC Application

To port an RPC application, you need:

- do nothing, if your application works in binary compatibility mode and you're satisfied that it only uses UDP and/or TCP transports. You will eventually have to port your application to TI-RPC; the binary compatibility package is not a permanent solution.
- to regenerate (via `rpcgen`) and recompile automatically-generated RPC modules, if your application uses simple RPC calls and no BSD socket or TCP or UDP specifics
- minimal editing and some new code, if the application depends on any BSD socket semantics or any features specific to TCP or UDP (examples of the latter would include knowing the format of host addresses or relying on the Berkeley UNIX concept of privileged ports)
- more than minimal revision, if the application is dependent on the internals of the RPC library, the sockets implementation, or has ingrained assumptions about transport addressing

Advantages of Porting Over Using Binary Compatibility

- Porting makes applications transport-independent, and thus able to run over many more transports than before
- Porting enables you to take advantage of new interfaces to make applications more efficient
- Binary compatibility will be less efficient than native mode
- Binary compatibility and old interfaces may be dropped in future releases

---

Note – Applications that open their own sockets and use them in RPC calls such as `clnt{tcp,udp}_create`, `clntudp_bufcreate`, `svc{tcp,udp}_create`, and `svcudp_bufcreate` are not portable. Such applications should be linked with the `librpcsoc` library.

Porting Issues

- New library: `libnsl`. Networking functions no longer a part of `libc`.
- Many old interfaces are present in the new library, but they work only with TCP and/or UDP transports. To take advantage of new transports, you must use the new interfaces
- Entirely new name-to-address and network selection mechanisms for transport-independence
- New transports are based on TLI. TLI's notion of opaque addressing may have implications for applications that previously knew how to interpret addresses. Specifically, the lowest level address format is now a `struct netbuf`. Formerly, it was a `struct sockaddr`.

Using `rpcgen`

- The most painless way to port code: use `rpcgen` to generate the new support code, then compile and link with the new library
- Input files need not be changed
- Many improvements (support for ANSI C, multiple arguments, more client and server-side skeletons)

Creating Client Handles

The following calls have the same functionality as in SunOS 4.x, although those marked with an asterisk may disappear in future releases:

clnt_destroy
clnt_pcreateerror
*clntraw_create
clnt_spcreateerror
*clnttcp_create
*clntudp_bufcreate
*clntudp_create

Added functionality, same interface:

`clnt_control` added `CLGET_SVC_ADDR`
`clnt_create` last argument is now a nettype
`clnt_create_vers` last argument is now a nettype

New functionality:

clnt_dg_create
clnt_raw_create
clnt_tli_create
clnt_tp_create
clnt_vc_create

Client Creation Code Examples

If you have existing client creation code written for RPC:

```
clnt_create("hostname", PROGRAM, VERSION, "udp");
```

If you rewrite it in this manner, you may use it for TI-RPC:

```
clnt_create("hostname", PROGRAM, VERSION, "datagram_n");
```

Or if you have more extensive client creation code written for RPC:

```
struct hostent *h;
struct sockaddr_in sin;
int sock = RPC_ANYSOCK;
u_short port;
struct timeval wait;
```

```
if ((h = gethostbyname("host")) == (struct hostent *) NULL) {
    syslog(LOG_ERR, “gethostbyname failed”);
    exit(1);
}
sin.sin_addr.s_addr = *(u_long *) hp->h_addr;
if ((port = pmap_getport(&sin, PROGRAM, VERSION, "udp")) == 0) {
    syslog(LOG_ERR, "pmap_getport failed");
    exit(1);
} else
    sin.sin_port = htons(port);
wait.tv_sec = 25;
wait.tv_usec = 0;
clntudp_create(&sin, PROGRAM, VERSION, wait, &sock);
```

If you rewrite it in this manner, it will run on TI-RPC:

```
struct netconfig *nconf;
struct netconfig *getnetconfigent();
struct t_bind *tbind;
struct timeval wait;

nconf = getnetconfigent("udp");
fd = t_open(nconf->nc_device, O_RDWR, (struct t_info *) NULL);
if (fd == -1) {
    syslog(LOG_ERR, "t_open failed %m");
    exit(1);
}
tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
if (tbind == (struct t_bind *) NULL) {
    syslog(LOG_ERR, "t_bind failed %m");
    exit(1);
}
if (rpcb_getaddr(PROGRAM, VERSION, nconf, &tbind->addr, "host")
            == FALSE) {
    syslog(LOG_ERR, "rpcb_getaddr failed");
    exit(1);
}
cl = clnt_tli_create(fd, nconf, &tbind->addr, PROGRAM, VERSION,
          0, 0);
(void) t_free((char *) tbind, T_BIND);
if (cl == (CLIENT *) NULL) {
    syslog(LOG_ERR, "clnt_tli_create failed");
    exit(1);
}
```

```
wait.tv_sec = 25;
wait.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, (char *) &wait);
```

Creating and Destroying Services

The following calls have the same functionality as in SunOS 4.x, although those marked with an asterisk may disappear in future releases:

svc_destroy
svcfd_create
*svcraw_create
*svctcp_create
*svcudp_create
*svcudp_bufcreate

New functionality:

svc_create
svc_dg_create
svc_fd_create
svc_raw_create
svc_tli_create
svc_tp_create
svc_vc_create

Registering and Unregistering Services

The following calls have the same functionality as in SunOS 4.x, although those marked with an asterisk may disappear in future releases:

*registerrpc
*svc_register
*svc_unregister
xprt_register
xprt_unregister

New functionality:

rpc_reg
svc_reg
svc_unreg

Service Registration Code Example

You might have service registration code written for RPC:

```
SVCXPRT *transp;
extern void svcfunc();

if ((transp = svcudp_create (RPC_ANYSOCK)) == (SVCXPRT *) NULL) {
    syslog(LOG_ERR, "svcudp_create failed");
    exit(1);
}
if (!svc_register(transp, PROGRAM, VERSION, svcfunc,
                  IPPROTO_UDP)) {
    syslog(LOG_ERR, "svc_register IPPROTO_UDP failed");
    exit(1);
}
if ((transp = svctcp_create(RPC_ANYSOCK)) == (SVCXPRT *) NULL) {
    syslog(LOG_ERR, "svctcp_create failed");
    exit(1);
}
if (!svc_register(transp, PROGRAM, VERSION, svcfunc,
                  IPPROTO_TCP)) {
    syslog(LOG_ERR, "svc_register IPPROTO_TCP failed");
    exit(1);
}
```

You would rewrite it thus for TI-RPC:

```
extern void svcfunc();

if (!svc_create(svcfunc, PROGRAM, VERSION, "netpath") {
    syslog(LOG_ERR, "svc_create failed");
    exit(1);
}
```

Calls

The following calls have the same functionality as in SunOS 4.x:

callrpc
clnt_call
svc_getcaller

New functionality:

rpc_call
svc_rpcgetcaller

Broadcasting

The following call has the same functionality as in SunOS 4.x, although it is supported for backward compatibility only:

`clnt_broadcast`

New functionality:

`rpc_broadcast` broadcasts to both `portmap` and `rpcbind`

Broadcasting Code Example

If you have broadcasting code written for RPC:

```
statstime sw;
extern int collectnames();

    ...
    clnt_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
        xdr_void, NULL, xdr_statstime, &sw, collectnames);
...
collectnames(resultsp, raddrp)
    char *resultsp;
    struct sockaddr_in *raddrp;
{
    u_long addr;
    struct entry *entryp, *lim;
    struct hostent *hp;
    extern int curentry;

    /*
     * weed out duplicates
     */
    addr = raddrp->sin_addr.s_addr;
    lim = entry + curentry;
    for (entryp = entry; entryp < lim; entryp++)
        if (addr == entryp->addr)
            return (0);

    /*
     * print the host's name (if possible) or address
```

```
     */
    hp = gethostbyaddr(&raddrp->sin_addr.s_addr, sizeof(u_long),
                       AF_INET);
    if (hp == (struct hostent *) NULL)
        printf("0x%x\n", addr);
    else
        printf("%s\n", hp->h_name);
}
```

You would rewrite it in this way for TI-RPC:

```
statstime sw;
extern int collectnames();

    ...
    rpc_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
         xdr_void, NULL, xdr_statstime, &sw, collectnames,
         (char *) 0);
...
collectnames(resultsp, taddr, nconf)
    char *resultsp;
    struct t_bind *taddr;
    struct netconfig *nconf;
{
    struct entry *entryp, *lim;
    struct nd_hostservlist *hs;
    extern int curentry;
    extern int netbufeq();

    /*
     * weed out duplicates
     */
    lim = entry + curentry;
    for (entryp = entry; entryp < lim; entryp++)
        if (netbufeq(&taddr->addr, entryp->addr))
            return (0);
    ...
    /*
     * print the host's name (if possible) or address
     */
    if (netdir_getbyaddr(nconf, &hs, &taddr->addr) == ND_OK)
        printf("%s\n", hs->h_hostservs->h_host);
    else {
        char *uaddr = taddr2uaddr(nconf, &taddr->addr);
        if (uaddr) {
            printf("%s\n", uaddr);
            (void) free(uaddr);
```

```
            } else
                printf("unknown\n");
        }
    }
    netbufeq(a, b)
        struct netbuf *a, *b; {
        return (a->len == b->len && !memcmp(a->buf, b->buf, a->len));
    }
```

The RPC library is capable of speaking with either the previous RPC's `portmap` program, or its replacement, `rpcbind`. The following functions are used when communicating with `portmap`:

pmap_set
pmap_unset
pmap_getport
pmap_getmaps
pmap_rmtcall

New applications should be using the new `rpcbind` functions:

rpcb_set
rpcb_unset
rpcb_getaddr
rpcb_getmaps
rpcb_rmtcal

Note that `rpcb_getaddr` can be used to talk to `portmap` as well as `rpcbind`.

`rpcbind` now provides a simple time service (primarily for use by secure RPC client:server time synchronization), available via the `rpcb_gettime` function. This function returns the server's time.

Recompilation Issues

Must include `-lnsl` in the link step

If recompiling an application that uses any of the following functions or macros:

callrpc
clntraw_create
clnttcp_create
clntudp_bufcreate

clntudp_create
svc_getcaller
svc_register
svc_unregister
svcfd_create
svcraw_create
svctcp_create
svcudp_bufcreate
svcudp_create

you must compile using the -DPORTMAP preprocessor control.

Authentication

The following calls have the same functionality as in SunOS 4.x, although they are supported for backward compatibility only:

authdes_create
authunix_create
authunix_create_default

New functionality:

authdes_seccreate
authsys_create
authsys_create_default

Name-to-Address Translation

The following calls have the same functionality as in SunOS 4.x:

gethostbyname
gethostbyaddr
gethostent
getrpcbyname
getrpcbynumber
getservbyname
getservbyaddr

New functionality:

netdir_free
netdir_getbyname
netdir_getbyaddr
netdir_options
netdir_perror
netdir_sperror
taddr2uaddr
uaddr2taddr