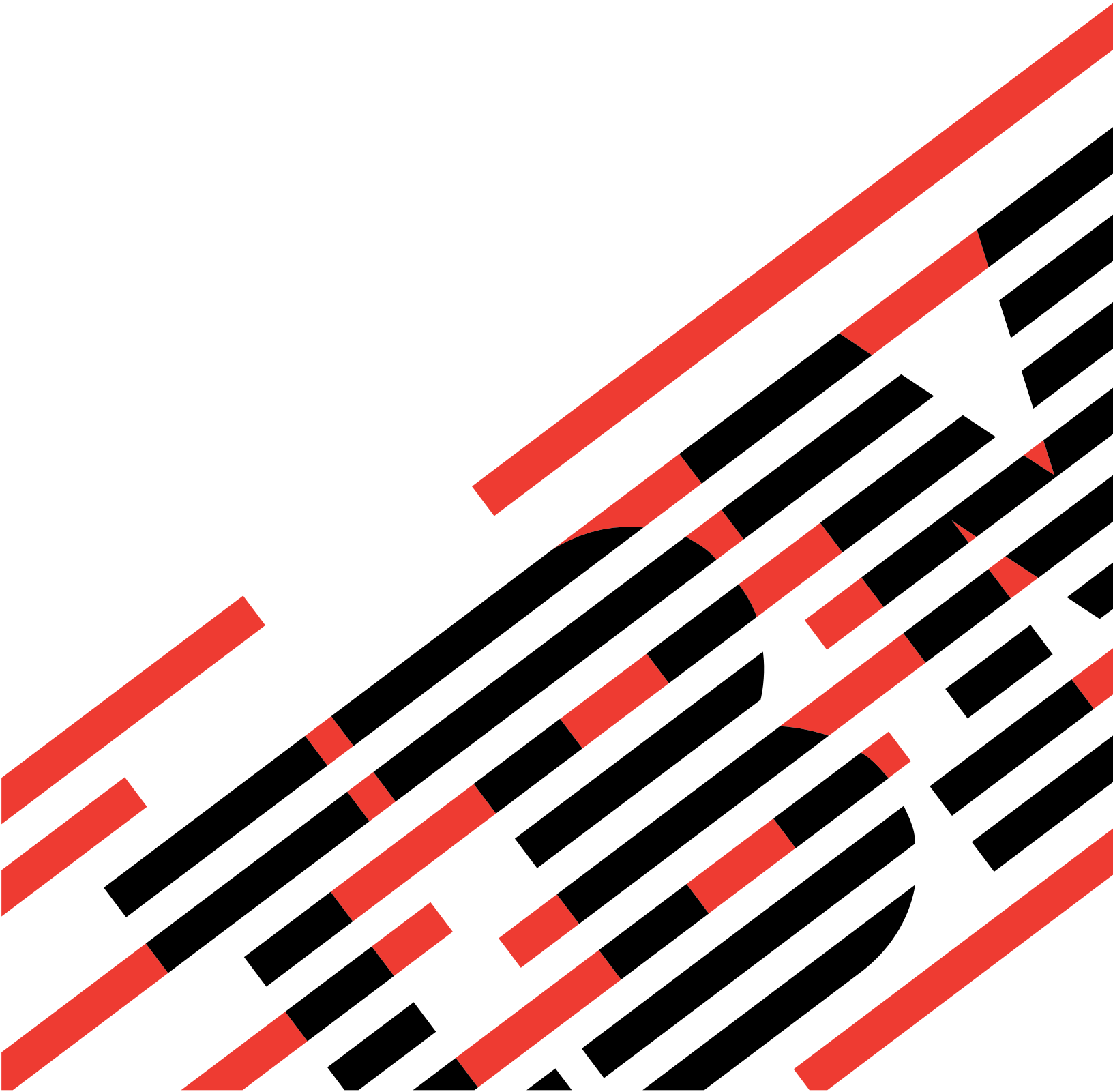
IBM

eserver

iSeries

Sun TI-RPC Distributed Applications



IBM

@server

iSeries

Sun TI-RPC Distributed Applications

Contents

© Copyright IBM Corp. 1998, 2002

Part 1. Using Sun TI-RPC to develop distributed applications

Remote procedure call (RPC) provides a high level paradigm, which allows distributed applications to communicate with one another. Sun Microsystems developed open networking computers (ONC) RPC to easily separate and distribute a client application from a server mechanism. Transport independent remote procedure call (TI-RPC) or ONC+ RPC is the latest version of RPC to be released. By providing a method for abstracting the underlying protocol used at the network layer, TI-RPC can provide a more seamless transition from one protocol to another.

To use TI-RPC on AS/400 to develop distributed applications, see the following topics:

- Chapter 2, “Using the rpcbind daemon” on page 5
- Chapter 3, “Using the rpcgen compiler” on page 7
- Chapter 4, “Using the network selection mechanism” on page 9
- Chapter 5, “Using data conversion support” on page 11

For detailed information about designing, implementing, and maintaining distributed applications by using TI-RPC, see the ONC+ Developer’s Guide  by Sun Microsystems, Inc. (1997).

Coding examples:

See the following topics for examples of service and client application programming interfaces:

- Chapter 6, “Examples: Developing service applications based on TI-RPC code” on page 13
- Chapter 7, “Developing client applications based on TI-RPC code examples” on page 27

© Copyright IBM Corp. 1998, 2002

Chapter 1. Print this topic

You can view or download a PDF version of this document for viewing or printing. You must have Adobe® Acrobat® Reader installed to view PDF files. You can download a copy from

http://www.adobe.com/prodindex/acrobat/readstep.html  .

To view or download the PDF version, select Programming Sun TI-RPC Distributed Applications (about 279 KB or 58 pages).

To save a PDF on your workstation for viewing or printing:

1. Open the PDF in your browser (click the link above).
2. In the menu of your browser, click **File**.
3. Click **Save As...**
4. Navigate to the directory in which you would like to save the PDF.
5. Click **Save**.

© Copyright IBM Corp. 1998, 2002

Chapter 2. Using the rpcbind daemon

When a client wants to connect to a remote procedure call (RPC) service, it contacts the RPCBIND daemon and requests the address of the service. In this way, addresses can be dynamic, and the client does not need to know which port the service is waiting on. Services need to be registered with the rpcbind daemon in order to be useful. If the rpcbind daemon is inactive, the services are unable to start up, and the clients are not able to find any of the services.

To use the rpcbind daemon on OS/400, complete the following tasks:

1. Ensure that the rpcbind daemon is running on OS/400.
2. Start the rpcbind daemon on OS/400 if it is not already running.

Ensuring that the rpcbind daemon is running on OS/400

To use the transport independent remote procedure call (TI-RPC) application programming interfaces (APIs), you need to ensure that the rpcbind daemon job (QNFSRPCD) is running on AS/400.

To ensure that the rpcbind daemon is running, complete the following steps:

1. On the OS/400 command line, type: `WRKACTJOB`
2. Look in the subsystem `QSYSWRK` for the existence of the following job:

   ```
   QNFSRPCD The rpcbind daemon
   ```

See “Starting and ending the rpcbind daemon on OS/400” for instructions on starting the rpcbind daemon if it is not running.

Starting and ending the rpcbind daemon on OS/400

The rpcbind daemon (RPCBIND) command starts the rpcbind daemon job (QNFSRPCD).

Starting the rpcbind daemon job:

To start the rpcbind daemon job, type the following command:

```
RPCBIND RTVRPCREG(*YES)
```

Note: An optional parameter for this command is RTVRPCREG, which specifies whether to retrieve previously recorded registration information when the rpcbind daemon is started. The default for this parameter is *NO. Select *YES if you want the rpcbind daemon to retrieve registration information when it starts. For more information about the parameter and value descriptions for this command, refer to the online help text.

Ending the rpcbind daemon job:

To end the rpcbind daemon job (QNFSRPCD), type the following command:

```
ENDRPCBIND
```

© Copyright IBM Corp. 1998, 2002

Chapter 3. Using the rpcgen compiler

The RPCGEN command generates C code from an input file that is written in the remote procedure call language (RPCL). You can use the generated C code to implement an RPC protocol.

To use the rpcgen compiler on OS/400, complete the following tasks:

1. Create your source input file in RPCL.

   Refer to the rpcgen Programmer's Guide  by Sun Microsystems, Inc. (1997) for details about using the rpcgen compiler.

2. Type the following command to run the rpcgen compiler on OS/400:

   ```
   RPCGEN
   ```

   Note: To see the parameter and value descriptions for this command, refer to the online help text.

3. Use a C language compiler on OS/400 to compile the output from the rpcgen compiler.

   Note: If you are using the Integrated Language Environment (ILE) C compiler on AS/400, you need to store the output files as source members.

© Copyright IBM Corp. 1998, 2002

Chapter 4. Using the network selection mechanism

The network selection mechanism allows you to choose the transport on which an application should run. The /etc/netconfig file is a database that lists the transports that are available to the host and identifies them by type. Transports are available in the /etc/netconfig file in the specified order in which they appear. For more information about network selection application programming interfaces (APIs), see System API Reference.

If you want to access the /etc/netconfig file on OS/400 in iSeries Navigator, complete the following steps:
1. Open iSeries Navigator if it is installed on your personal computer.
2. Expand the Network folder.
3. Expand the Servers folder.
4. Click on the OS/400 server.
5. Right click on RPC and select Properties from the pop-up menu.
6. Click on the RPC Transports tab

   Note: You must have *IOSYSCFG authority to view this information.

© Copyright IBM Corp. 1998, 2002

Chapter 5. Using data conversion support

All transport independent remote procedure call (TI-RPC) application programming interfaces (APIs) are enabled for National Language Support (NLS) on OS/400. This support has been added to a list of eXternal Data Representation (XDR) functions. These functions allow data to be communicated between clients and services that are in different code pages. The system administrator maintains the /etc/rpcnls file to associate the code pages with a remote client. The XDR functions use the information in the /etc/rpcnls file to provide an implicit data conversion. The following XDR functions have implicit data conversion routines built-in:

- `xdr_char() (single byte only)`
- `xdr_u_char() (single byte only)`
- `xdr_double_char (single and double byte)`
- `xdr_string() (single and double byte)`
- `xdr_wrapstring() (single and double byte)`

For more information about data conversion support in each of the transport independent remote procedure call (TI-RPC) application programming interfaces (APIs), see the System API Reference.

To access the /etc/rpcnls file on OS/400 in iSeries Navigator, complete the following steps:

1. Open iSeries Navigator if it is installed on your personal computer.
2. Expand the Network folder.
3. Expand the Servers folder.
4. Click on the OS/400 server.
5. Right click on RPC and select Properties from the pop-up menu.
6. Click on the Data Conversion Support tab.

   Note: You must have *IOSYSCFG authority to view this information.

© Copyright IBM Corp. 1998, 2002

Chapter 6. Examples: Developing service applications based on TI-RPC code

Transport Independent remote procedure call (TI-RPC) programming provides an effective method for developing distributed client-server based applications on OS/400.

To develop service applications on OS/400, use the following code examples as a guideline:

- "Example: TI-RPC simplified level service API"
- "Example: TI-RPC top level service API" on page 16
- "Example: TI-RPC intermediate level service API" on page 22
- "Example: TI-RPC expert level service API" on page 23
- "Example: Adding authentication to the TI-RPC service" on page 24

Related information:

- Part 1, "Using Sun TI-RPC to develop distributed applications" on page 1
- Chapter 7, "Developing client applications based on TI-RPC code examples" on page 27

Example: TI-RPC simplified level service API

The following code example illustrates one of the simplified level service application programming interfaces (APIs) that are used in developing transport independent remote procedure call (TI-RPC) services.

In this code example, notice how each procedure is registered independently from the rest. At the simplified level, a service may have multiple procedures, but each one must be registered separately. If the service is unregistered, all of the procedures are unregistered at once. There is no method that is provided to unregister an individual procedure while leaving the remaining procedures intact.

This level is a good choice for services with a small number of procedures. It is also useful for prototyping a much larger service by using a limited number of the final procedures. As with all the service levels, the final call in the service should be to svc_run(), which goes into Select Wait (waiting for a connection from a client).

```
#include <stdio.h>
#include <netconfig.h>
#include <rpc/rpc.h>
#include <errno.h>
#include "myapp.h"

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

 bool_t rslt; /* return value for rpc_call() */

 /* unregister any existing copy of this service */
 /* (void)svc_unreg(program, version) */
 svc_unreg(PROGNUM, VERSNUM);

 /* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
 /*                 xdr_in, xdr_out, nettype)            */
 rslt = rpc_reg(PROGNUM, VERSNUM, GET_UID, myapp_get_uid,
   xdr_wrapstring, xdr_u_int, NETTYPE);

 /* check for errors calling rpc_reg() */
 if (rslt == FALSE) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_UID");
  fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
```

© Copyright IBM Corp. 1998, 2002

```
    PROGNUM, VERSNUM, NETTYPE);
  /* clean up before exiting */
  svc_unreg(PROGNUM, VERSNUM);
  return 1;
  }

 /* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
 /*                 xdr_in, xdr_out, nettype)            */
 rslt = rpc_reg(PROGNUM, VERSNUM, GET_UID_STRING, myapp_get_uid_string,
   xdr_wrapstring, xdr_wrapstring, NETTYPE);

 /* check for errors calling rpc_reg() */
 if (rslt == FALSE) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_UID_STRING");
  fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  /* clean up before exiting */
  svc_unreg(PROGNUM, VERSNUM);
  return 1;
  }

 /* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
 /*                 xdr_in, xdr_out, nettype)            */
 rslt = rpc_reg(PROGNUM, VERSNUM, GET_SIZE, myapp_get_size,
   xdr_wrapstring, xdr_int, NETTYPE);

 /* check for errors calling rpc_reg() */
 if (rslt == FALSE) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_SIZE");
  fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  /* clean up before exiting */
  svc_unreg(PROGNUM, VERSNUM);
  return 1;
  }

 /* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
 /*                 xdr_in, xdr_out, nettype)            */
 rslt = rpc_reg(PROGNUM, VERSNUM, GET_MTIME, myapp_get_mtime,
   xdr_wrapstring, xdr_long, NETTYPE);

 /* check for errors calling rpc_reg() */
 if (rslt == FALSE) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_MTIME");
  fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  /* clean up before exiting */
  svc_unreg(PROGNUM, VERSNUM);
  return 1;
  }

 /* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
 /*                 xdr_in, xdr_out, nettype)            */
 rslt = rpc_reg(PROGNUM, VERSNUM, GET_MTIME_STRING, myapp_get_mtime_string,
   xdr_wrapstring, xdr_wrapstring, NETTYPE);

 /* check for errors calling rpc_reg() */
 if (rslt == FALSE) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_MTIME_STRING");
  fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  /* clean up before exiting */
  svc_unreg(PROGNUM, VERSNUM);
```

```
 return 1;
 }

/* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
/*                 xdr_in, xdr_out, nettype)            */
rslt = rpc_reg(PROGNUM, VERSNUM, GET_CODEPAGE, myapp_get_codepage,
  xdr_wrapstring, xdr_u_short, NETTYPE);

/* check for errors calling rpc_reg() */
if (rslt == FALSE) {
 /* print error messages and exit */
 fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_CODEPAGE");
 fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
   PROGNUM, VERSNUM, NETTYPE);
 /* clean up before exiting */
 svc_unreg(PROGNUM, VERSNUM);
 return 1;
 }

/* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
/*                 xdr_in, xdr_out, nettype)            */
rslt = rpc_reg(PROGNUM, VERSNUM, GET_OBJTYPE, myapp_get_objtype,
  xdr_wrapstring, xdr_wrapstring, NETTYPE);

/* check for errors calling rpc_reg() */
if (rslt == FALSE) {
 /* print error messages and exit */
 fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_OBJTYPE");
 fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
   PROGNUM, VERSNUM, NETTYPE);
 /* clean up before exiting */
 svc_unreg(PROGNUM, VERSNUM);
 return 1;
 }

/* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
/*                 xdr_in, xdr_out, nettype)            */
rslt = rpc_reg(PROGNUM, VERSNUM, GET_FILETYPE, myapp_get_filetype,
  xdr_wrapstring, xdr_wrapstring, NETTYPE);

/* check for errors calling rpc_reg() */
if (rslt == FALSE) {
 /* print error messages and exit */
 fprintf(stderr, "Error calling rpc_reg for %s\n", "GET_FILETYPE");
 fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
   PROGNUM, VERSNUM, NETTYPE);
 /* clean up before exiting */
 svc_unreg(PROGNUM, VERSNUM);
 return 1;
 }

/* (bool_t)rpc_reg(prognum, versnum, procnum, procname, */
/*                 xdr_in, xdr_out, nettype)            */
rslt = rpc_reg(PROGNUM, VERSNUM, END_SERVER, myapp_end_server,
  (xdrproc_t)xdr_void, (xdrproc_t)xdr_void, NETTYPE);

/* check for errors calling rpc_reg() */
if (rslt == FALSE) {
 /* print error messages and exit */
 fprintf(stderr, "Error calling rpc_reg for %s\n", "END_SERVER");
 fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
   PROGNUM, VERSNUM, NETTYPE);
 /* clean up before exiting */
 svc_unreg(PROGNUM, VERSNUM);
 return 1;
 }
```

```
 /* this should loop indefinitely waiting for client connections */
 svc_run();

 /* if we get here, svc_run() returned */
 fprintf(stderr, "svc_run() returned.  ERROR has occurred.\n");
 fprintf(stderr, "errno: %d\n", errno);

 /* clean up by unregistering.  then, exit */
 svc_unreg(PROGNUM, VERSNUM);

 return 1;

} /* end of main() */
```

Example: TI-RPC top level service API

The following code example illustrates a top level service application programming interface (API) used in developing transport independent remote procedure call (TI-RPC) services.

The development of a service is more complicated at the top level, because it requires the developer to write a dispatch routine. At this level, when a service request comes in, a dispatch routine is called. The dispatch routine must collect the arguments and call the correct local procedure, catch all errors and results, and return that information to the client. Once a dispatch function is written, it can be readily copied and used in other services with only slight modifications.

The top, intermediate, and expert layers can use the same dispatch function without modification. In the following example, the dispatch function is bundled with the other local functions in this file. Both files will need to be compiled and linked together before the service will run. The advantage of the top level over the other layers is the ability to specify the nettype as a string, instead of using the network selection APIs. After calling the top level API, the service is created, bound to the dispatch function, and registered with the rpcbind service.

```
#include <stdio.h>
#include <netconfig.h>
#include <rpc/rpc.h>
#include <errno.h>
#include "myapp.h"
int main(int argc, char *argv[]) {

 int num_svc; /* return value for the svc_create() API */

 /* unregister any existing copy of this service */
 /* (void)svc_unreg(program, version) */
 svc_unreg(PROGNUM, VERSNUM);

 /* (int)svc_create(dispatch, prognum, versnum, nettype); */
 num_svc = svc_create(myapp_dispatch, PROGNUM, VERSNUM, NETTYPE);

 /* check for errors calling svc_create() */
 if (num_svc == 0) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling %s.\n", "svc_create");
  fprintf(stderr, "PROG: %lu\nVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  fprintf(stderr, "errno: %d\n", errno);
  return 1;
  }

 /* this should loop indefinitely waiting for client connections */
 svc_run();

 /* if we get here, svc_run() returned */
 fprintf(stderr, "svc_run() returned.  ERROR has occurred.\n");
 fprintf(stderr, "errno: %d\n", errno);
```

```
 /* clean up by unregistering.  then, exit */
 svc_unreg(PROGNUM, VERSNUM);

 return 1;

} /* end of main() */
```

```
/* This is an example of the dispatch function */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <pwd.h>
#include <rpc/rpc.h>
#include <time.h>
#include "myapp.h"

char * myapp_get_uid(char *in) {

 u_int retval;              /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (u_int)-1;
  }
 else {
  retval = (u_int)(sbuf.st_uid);
  }

 return (char *)&retval;
}

char *myapp_get_uid_string(char *in) {

 char *retval;              /* return value for this procedure() */
 struct passwd *pbuf;       /* return value for getpwuid() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (char *)NULL;
  }
 else {

  pbuf = (struct passwd *)getpwuid((uid_t)(sbuf.st_uid));

  if (pbuf == NULL) {
   retval = (char *)NULL;
   }
  else {
   retval = (char *)(pbuf->pw_name);
   }
  }

 return (char *)&retval;
}

char * myapp_get_size(char *in) {
```

```
 int retval;                /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (int)-1;
  }
 else {
  retval = (int)(sbuf.st_size);
  }

 return (char *)&retval;
}

char * myapp_get_mtime(char *in) {

 long retval;               /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (long)-1;
  }
 else {
  retval = (long)(sbuf.st_mtime);
  }

 return (char *)&retval;

}

char *myapp_get_mtime_string(char *in) {

 char *retval;              /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (char *)NULL;
  }

 else {
  retval = (char *)ctime((time_t *)&(sbuf.st_mtime));
  }

 return (char *)&retval;
}

char * myapp_get_codepage(char *in) {

 u_short retval;            /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */
```

```
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (u_short)-1;
 }

 else {
  retval = (u_short)(sbuf.st_codepage);
 }

 return (char *)&retval;
}


char *myapp_get_objtype(char *in) {

 char *retval;              /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  retval = (char *)NULL;
 }

 else {
  retval = (char *)(sbuf.st_objtype);
 }

 return (char *)&retval;

}

char *myapp_get_filetype(char *in) {

 char *result = NULL;       /* return value for this procedure() */
 struct stat sbuf;          /* data storage area for stat() */
 int stat_ret;              /* return value for stat() */
 char *file = *(char **)in; /* input value for stat() */

 /* (int)stat(filename, struct stat *) */
 stat_ret = stat(file, &sbuf);

 if (stat_ret == -1) {
  return (char *)NULL;
  }

 if (S_ISDIR(sbuf.st_mode)) {
  result = "Directory";
  }

 if (S_ISREG(sbuf.st_mode)) {
  result = "Regulare File";
  }

 if (S_ISLNK(sbuf.st_mode)) {
  result = "Symbolic Link";
  }

 if (S_ISSOCK(sbuf.st_mode)) {
  result = "Socket";
  }
```

```
 if (S_ISNATIVE(sbuf.st_mode)) {
  result = "AS/400 Native Object";
  }

 if (S_ISFIFO(sbuf.st_mode)) {
  result = "FIFO";
  }

 if (S_ISCHR(sbuf.st_mode)) {
  result = "Character Special";
  }

 if (S_ISBLK(sbuf.st_mode)) {
  result = "Block Special";
  }

 return (char *)&result;

}

char * myapp_end_server(char *empty) {

 /* char *empty is not used      */
 /* function always returns NULL */

 svc_unreg(PROGNUM, VERSNUM);

 return (char *)NULL;

}

void myapp_dispatch(struct svc_req *request, SVCXPRT *svc) {

 union {
  /* all of the procedurs take a string */
  /* if there were other procedures, it */
  /* might look like this:              */
  /* int set_codepage_arg               */
  char *  filename_arg;
 } argument;

 char *result;            /* pointer to returned data from proc */
 xdrproc_t xdr_argument;  /* decodes data from client call */
 xdrproc_t xdr_result;    /* encodes data to return to client */
 char *(*proc)(char *);   /* pointer to local procedure to call */

 switch (request->rq_proc) {

  case NULLPROC:
   /* a special case.  always return void */
                        (void)svc_sendreply((SVCXPRT *)svc,
                                            (xdrproc_t)xdr_void,
                                            (char *)NULL);
   return;

  case GET_UID:
   /* takes a string argument (filename) */
   /* returns an u_int (uid of file ownder) */
   xdr_argument = xdr_wrapstring;
   xdr_result   = xdr_u_int;
   proc         = (char *(*)(char *))myapp_get_uid;
   break;

  case GET_UID_STRING:
   /* takes a string argument (filename) */
   /* returns a string (owner's name in string format) */
   xdr_argument = xdr_wrapstring;
```

```
 xdr_result   = xdr_wrapstring;
 proc         = (char *(*)(char *))myapp_get_uid_string;
 break;

case GET_SIZE:
 /* takes a string argument (filename) */
 /* returns an int (size of file in bytes) */
 xdr_argument = xdr_wrapstring;
 xdr_result   = xdr_int;
 proc         = (char *(*)(char *))myapp_get_size;
 break;

case GET_MTIME:
 /* takes a string argument (filename) */
 /* returns a long (time last modified) */
 xdr_argument = xdr_wrapstring;
 xdr_result   = xdr_long;
 proc         = (char *(*)(char *))myapp_get_mtime;
 break;

case GET_MTIME_STRING:
 /* takes a string argument (filename) */
 /* returns a string (time last modified, string format) */
 xdr_argument = xdr_wrapstring;
 xdr_result   = xdr_wrapstring;
 proc         = (char *(*)(char *))myapp_get_mtime_string;
 break;

case GET_CODEPAGE:
 /* takes a string argument (filename) */
 /* returns an u_short (codepage of file) */
 xdr_argument = xdr_wrapstring;
 xdr_result   = xdr_u_short;
 proc         = (char *(*)(char *))myapp_get_codepage;
 break;

case GET_OBJTYPE:
 /* takes a string argument (filename) */
 /* returns a string (object type) */
 xdr_argument = xdr_wrapstring;
 xdr_result   = xdr_wrapstring;
 proc         = (char *(*)(char *))myapp_get_objtype;
 break;

case GET_FILETYPE:
 /* takes a string argument (filename) */
 /* returns a string (file type) */
 xdr_argument = xdr_wrapstring;
 xdr_result   = xdr_wrapstring;
 proc         = (char *(*)(char *))myapp_get_filetype;
 break;

case END_SERVER:
 /* takes no arguments */
 /* returns no data */
 /* unregisters service with local rpcbind daemon */
 xdr_argument = (xdrproc_t)xdr_void;
 xdr_result   = (xdrproc_t)xdr_void;
 proc         = (char *(*)(char *))myapp_end_server;
 break;

default:
 /* fall through case.  return error to client */
 svcerr_noproc(svc);
 return;

} /* end switch(request->rq_proc) */
```

```
 /* clear the argument */
 memset((char *)&argument, (int)0, sizeof(argument));

 /* decode argument from client using xdr_argument() */
 if (svc_getargs(svc, xdr_argument, (char *)&argument) == FALSE) {
  /* if svc_getargs() fails, return RPC_CANTDECODEARGS to client */
  svcerr_decode(svc);
  return;
 }

 /* call local procedure, passing in pointer to argument */
 result = (char *)(*proc)((char *)&argument);

 /* check first that result isn't NULL */
 /* try to send results back to client.  check for failure */
 if ((result != NULL) && (svc_sendreply(svc, xdr_result, result) == FALSE))
{
  /* send error message back to client */
  svcerr_systemerr(svc);
 }

 /* free the decoded argument's space */
 if (svc_freeargs(svc, xdr_argument, (char *)&argument) == FALSE) {
  /* if unable to free, print error and exit */
  (void)fprintf(stderr, "unable to free arguments\n");
  exit(1);
 }

} /* end of myapp_dispatch() */
```

Example: TI-RPC intermediate level service API

The following code example illustrates an intermediate level service application programming interface (API) that is used in developing transport independent remote procedure call (TI-RPC) services.

The dispatch function may be reused without any changes. The only difference between the top and intermediate levels is the use of the network selection APIs . This level also creates, binds, and registers the service with the rpcbind daemon.

```
#include <stdio.h>
#include <netconfig.h>
#include <rpc/rpc.h>
#include <errno.h>
#include "myapp.h"

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

 struct netconfig *nconf;  /* pointer to nettype data */
 SVCXPRT *svc;             /* pointer to service handle */

 /* unregister any existing copy of this service */
 /* (void)svc_unreg(program, version) */
 svc_unreg(PROGNUM, VERSNUM);

 /* (struct netconfig *)getnetconfigent(nettype) */
 nconf = getnetconfigent(NETTYPE);
 if (nconf == (struct netconfig *)NULL) {
  fprintf(stderr, "Error calling getnetconfigent(%s)\n", NETTYPE);
  fprintf(stderr, "errno: %d\n", errno);
  return 1;
  }

 /* (SVCXPRT *)svc_tp_create(dispatch, prognum, versnum, netconf) */
 svc = svc_tp_create(myapp_dispatch, PROGNUM, VERSNUM, nconf);
```

```
       /* check for errors calling svc_tp_create() */
 if (svc == (SVCXPRT *)NULL) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling %s.\n", "svc_tp_create");
  fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  fprintf(stderr, "errno: %d\n", errno);
  return 1;
  }

 /* this should loop indefinitely waiting for client connections */
 svc_run();

 /* if we get here, svc_run() returned */
 fprintf(stderr, "svc_run() returned.  ERROR has occurred.\n");
 fprintf(stderr, "errno: %d\n", errno);

 /* clean up by unregistering.  then, exit */
 svc_unreg(PROGNUM, VERSNUM);

 return 1;

} /* end of main() */
```

Example: TI-RPC expert level service API

The following code example illustrates an expert level service application programming interface (API) that is used in developing transport independent remote procedure call (TI-RPC) services.

The expert level allows the programmer to specify Send and Receive buffer sizes for services. Calling svc_tli_create() creates the service handle, but it does not register or bind it with the rpcbind daemon. The programmer must call svc_reg() before the service works properly. Similar to the intermediate level, you have the option to use the network selection APIs to retrieve the transport information for the service.

```
#include <stdio.h>
#include <netconfig.h>
#include <rpc/rpc.h>
#include <errno.h>
#include "myapp.h"

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

 struct netconfig *nconf; /* pointer to nettype data */
 SVCXPRT *svc;            /* pointer to service handle */
 bool_t rslt;             /* return value for svc_reg() */

 /* unregister any existing copy of this service */
 /* (void)svc_unreg(program, version) */
 svc_unreg(PROGNUM, VERSNUM);

 /* (struct netconfig *)getnetconfigent(nettype) */
 nconf = getnetconfigent(NETTYPE);

 /* check for errors calling getnetconfigent() */
 if (nconf == (struct netconfig *)NULL) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling getnetconfigent(%s)\n", NETTYPE);
  fprintf(stderr, "errno: %d\n", errno);
  return 1;
  }

 /* (SVCXPRT *)svc_tli_create(filedes, netconfig, bindaddr, sendsz, recvsz) */
 svc = svc_tli_create(RPC_ANYFD, nconf, NULL, 0, 0);

 /* check for errors calling svc_tli_create() */
```

```
 if (svc == (SVCXPRT *)NULL) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling %s.\n", "svc_tli_create");
  fprintf(stderr, "errno: %d\n", errno);
  return 1;
  }

 /* (bool_t)svc_reg(svcxprt, prognum, versnum, dispatch, netconf) */
 rslt = svc_reg(svc, PROGNUM, VERSNUM, myapp_dispatch, nconf);

 /* check for errors calling svc_reg() */
 if (rslt == FALSE) {
  /* print error messages and exit */
  fprintf(stderr, "Error calling svc_reg\n");
  fprintf(stderr, "PROG: %lu\nVERS: %lu\tNET: %s\n",
    PROGNUM, VERSNUM, NETTYPE);
  fprintf(stderr, "errno: %d\n", errno);
  return 1;
  }

 /* this should loop indefinitely waiting for client connections */
 svc_run();

 /* if we get here, svc_run() returned */
 fprintf(stderr, "svc_run() returned.  ERROR has occurred.\n");
 fprintf(stderr, "errno: %d\n", errno);

 /* clean up by unregistering.  then, exit */
 svc_unreg(PROGNUM, VERSNUM);

 return 1;

} /* end of main() */
```

Example: Adding authentication to the TI-RPC service

The following code snippets display how the authentication system works in remote procedure call (RPC). System is the only authentication method that is provided on OS/400. The following information is set up and passed from the client to the service with every clnt_call(). In the following code snippets, notice that rpc_call() is not sufficient when using authentication information , because it defaults to using `authnone` (an empty authentication token):

- `aup_time - authentication information timestamp`
- `aup_machname - the hostname of the remote client`
- `aup_uid - the UID of the remote user of the client`
- `aup_gid - the primary GID of the remote user`
- `aup_gids - an array of the secondary groups of the remote user`

The authentication information comes directly into the service as part of the remote request. It is up to the server to parse this information and verify that the client is from a trusted machine and a trusted user. If the authentication type is incorrect, or too weak for the server to accept, it sends back an error, using svcerr_weakauth(), to indicate this to the client.

```
#include <sys/types.h> /* needed for gid_t and uid_t */
#include <stdlib.h>    /* misc. system auth APIs     */
#include <errno.h>

struct authsys_parms *credentials;  /* authentication information */
char *remote_machine;               /* machine name (from the credentials) */
uid_t remote_user;                  /* remote user's UID (from credentials) */

/* make sure we got the correct flavor of authentication */
if (request->rq_cred.oa_flavor != AUTH_UNIX) {
   /* if not, send back a weak authentication message and return */
```

```
   svcerr_weakauth(svc);
   return;
}

/* get our credentials */
credentials = (struct authsys_parms *)(request->rq_clntcred);

/* get the remote user's GID */
remote_user = credentials->aup_uid;

/* get the remote hostname of the client */
remote_machine = credentials->aup_machname;

/* check to see if this machine is "trusted" by us */
if ((strcmpi("remote1", remote_machine) != 0) &&
    (strcmpi("remote2", remote_machine) != 0)) {

   /* not from a machine we trust */
   /* send back an authentication error the client */
   svcerr_weakauth(svc);
   return;

} /* end of if (!trusted hostname) */

else {

   /* now check the user id for one we trust */
   /* information can be gotten from DSPUSRPRF */
   if ((remote_user != 568) &&
       (remote_user != 550) &&
       (remote_user != 528)) {

      /* not a user id we trst */
      /* send back an authentication error the client */
      svcerr_weakauth(svc);
      return;

   } /* end of if (!trusted uid) */

} /* end of else (tusted hostname) */

/* we fall out of the loop if the hostname and uid are trusted */
```

Chapter 7. Developing client applications based on TI-RPC code examples

Transport independent remote procedure call (TI-RPC) programming provides an effective method for developing distributed client-server based applications on OS/400. For more information about TI-RPC service application programming interfaces (APIs), see System API Reference.

To develop client applications on OS/400, use the following code examples as a guideline:

- "Example: TI-RPC simplified level client API"
- "Example: TI-RPC top level client API" on page 32
- "Example: TI-RPC intermediate level client API" on page 36
- "Example: TI-RPC expert level client API" on page 40
- "Example: Adding authentication to the TI-RPC client" on page 45

Related information:

- Part 1, "Using Sun TI-RPC to develop distributed applications" on page 1
- Chapter 6, "Examples: Developing service applications based on TI-RPC code" on page 13

Example: TI-RPC simplified level client API

The following code example illustrates a simplified level client application programming interface (API) that is used in developing transport independent remote procedure call (TI-RPC) applications.

The simplified level client API is the quickest and shortest set of code, because the client creation, control, use, and destruction are all in one call. This is convenient, but it does not allow the customization that can be done with a client handle. Defaults are accepted for timeout and buffer sizes, which is the most significant difference between the simplified level and the other levels.

```
#include <stdio.h>
#include <errno.h>
#include "myapp.h"

#define EXIT 100

int main(void) {

   enum clnt_stat rslt;    /* return value of rpc_call() */
   char hostname[256];     /* buffer for remote service's hostname */
   unsigned long procnum;  /* procedure to call */
   char filename[512];     /* buffer for filename */
   char *arg = filename;   /* pointer to filename buffer */

   union {
      u_int   myapp_get_uid_result;
      char *  myapp_get_uid_string_result;
      int     myapp_get_size_result;
      long    myapp_get_mtime_result;
      char *  myapp_get_mtime_string_result;
      u_short myapp_get_codepage_result;
      char *  myapp_get_objtype_result;
      char *  myapp_get_filetype_result;
   } result; /* a union of all the possible results */

   /* get the hostname from the user */
   printf("Enter the hostname where the remote service is running: \n");
   scanf("%s", (char *)&hostname);

   myapp_print_menu(); /* print out the menu choices */
```

© Copyright IBM Corp. 1998, 2002

```
/* get the procedure number to call from the user */
printf("\nEnter a procedure number to call: \n");
scanf("%lu", &procnum);

/* get the filename from the user */
printf("\nEnter a filename to stat: \n");
scanf("%s", (char *)&arg);

/* switch on the input */
switch (procnum) {

   case NULLPROC:

      /* rpc_call(host, prognum, versnum, procnum,    */
      /*          xdr_in, in, xdr_out, out, nettype); */
      rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                      (xdrproc_t)xdr_void, (char *)NULL, /* xdr_in  */
                      (xdrproc_t)xdr_void, (char *)NULL, /* xdr_out */
                      NETTYPE);

      /* check return value of rpc_call() */
      if (rslt != RPC_SUCCESS) {
         fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
         fprintf(stderr, "clnt_stat: %d\n", rslt);
         fprintf(stderr, "errno: %d\n", errno);
         return 1;
      }

      /* print results and exit */
      printf("NULLRPOC call succeeded\n");
      break;

   case GET_UID:

      /* rpc_call(host, prognum, versnum, procnum,    */
      /*          xdr_in, in, xdr_out, out, nettype); */
      rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                      xdr_wrapstring, (char *)&arg, /* xdr_in  */
                      xdr_u_int, (char *)&result,        /* xdr_out */
                      NETTYPE);

      /* check return value of rpc_call() */
      if (rslt != RPC_SUCCESS) {
         fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
         fprintf(stderr, "clnt_stat: %d\n", rslt);
         fprintf(stderr, "errno: %d\n", errno);
         return 1;
      }

      /* print results and exit */
      printf("uid of %s: %u\n",
             filename, result.myapp_get_uid_result);
      break;

   case GET_UID_STRING:

      /* rpc_call(host, prognum, versnum, procnum,    */
      /*          xdr_in, in, xdr_out, out, nettype); */
      rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                      xdr_wrapstring, (char *)&arg, /* xdr_in  */
                      xdr_wrapstring, (char *)&result, /* xdr_out */
                      NETTYPE);

      /* check return value of rpc_call() */
      if (rslt != RPC_SUCCESS) {
         fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
```

```
      fprintf(stderr, "clnt_stat: %d\n", rslt);
      fprintf(stderr, "errno: %d\n", errno);
      return 1;
   }

   /* print results and exit */
   printf("owner of %s: %s\n",
          filename, result.myapp_get_uid_string_result);
   break;

case GET_SIZE:

   /* rpc_call(host, prognum, versnum, procnum,    */
   /*          xdr_in, in, xdr_out, out, nettype); */
   rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                   xdr_wrapstring, (char *)&arg, /* xdr_in  */
                   xdr_int, (char *)&result, /* xdr_out */
                   NETTYPE);

   /* check return value of rpc_call() */
   if (rslt != RPC_SUCCESS) {
      fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
      fprintf(stderr, "clnt_stat: %d\n", rslt);
      fprintf(stderr, "errno: %d\n", errno);
      return 1;
   }

   /* print results and exit */
   printf("size of %s: %d\n",
          filename, result.myapp_get_size_result);
   break;

case GET_MTIME:

   /* rpc_call(host, prognum, versnum, procnum,    */
   /*          xdr_in, in, xdr_out, out, nettype); */
   rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                   xdr_wrapstring, (char *)&arg, /* xdr_in  */
                   xdr_long, (char *)&result, /* xdr_out */
                   NETTYPE);

   /* check return value of rpc_call() */
   if (rslt != RPC_SUCCESS) {
      fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
      fprintf(stderr, "clnt_stat: %d\n", rslt);
      fprintf(stderr, "errno: %d\n", errno);
      return 1;
   }

   /* print results and exit */
   printf("last modified time of %s: %ld\n",
          filename, result.myapp_get_mtime_result);
   break;

case GET_MTIME_STRING:

   /* rpc_call(host, prognum, versnum, procnum,    */
   /*          xdr_in, in, xdr_out, out, nettype); */
   rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                   xdr_wrapstring, (char *)&arg, /* xdr_in  */
                   xdr_wrapstring, (char *)&result, /* xdr_out */
                   NETTYPE);

   /* check return value of rpc_call() */
   if (rslt != RPC_SUCCESS) {
      fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
      fprintf(stderr, "clnt_stat: %d\n", rslt);
```

```
      fprintf(stderr, "errno: %d\n", errno);
      return 1;
   }

   /* print results and exit */
   printf("last modified time of %s: %s\n",
          filename, result.myapp_get_mtime_string_result);
   break;

case GET_CODEPAGE:

   /* rpc_call(host, prognum, versnum, procnum,    */
   /*          xdr_in, in, xdr_out, out, nettype); */
   rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                   xdr_wrapstring, (char *)&arg, /* xdr_in  */
                   xdr_u_short, (char *)&result, /* xdr_out */
                   NETTYPE);

   /* check return value of rpc_call() */
   if (rslt != RPC_SUCCESS) {
      fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
      fprintf(stderr, "clnt_stat: %d\n", rslt);
      fprintf(stderr, "errno: %d\n", errno);
      return 1;
   }

   /* print results and exit */
   printf("codepage of %s: %d\n",
          filename, result.myapp_get_codepage_result);
   break;

case GET_OBJTYPE:

   /* rpc_call(host, prognum, versnum, procnum,    */
   /*          xdr_in, in, xdr_out, out, nettype); */
   rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                   xdr_wrapstring, (char *)&arg, /* xdr_in  */
                   xdr_wrapstring, (char *)&result, /* xdr_out */
                   NETTYPE);

   /* check return value of rpc_call() */
   if (rslt != RPC_SUCCESS) {
      fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
      fprintf(stderr, "clnt_stat: %d\n", rslt);
      fprintf(stderr, "errno: %d\n", errno);
      return 1;
   }

   /* print results and exit */
   printf("object type of %s: %s\n",
          filename, result.myapp_get_objtype_result);
   break;

case GET_FILETYPE:

   /* rpc_call(host, prognum, versnum, procnum,    */
   /*          xdr_in, in, xdr_out, out, nettype); */
   rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                   xdr_wrapstring, (char *)&arg, /* xdr_in  */
                   xdr_wrapstring, (char *)&result, /* xdr_out */
                   NETTYPE);

   /* check return value of rpc_call() */
   if (rslt != RPC_SUCCESS) {
      fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
      fprintf(stderr, "clnt_stat: %d\n", rslt);
      fprintf(stderr, "errno: %d\n", errno);
```

```
         return 1;
      }

      /* print results and exit */
      printf("file type of %s: %s\n",
             filename, result.myapp_get_filetype_result);
      break;

   case END_SERVER:

      /* rpc_call(host, prognum, versnum, procnum,    */
      /*          xdr_in, in, xdr_out, out, nettype); */
      rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                      (xdrproc_t)xdr_void, (char *)NULL, /* xdr_in  */
                      (xdrproc_t)xdr_void, (char *)NULL, /* xdr_out */
                      NETTYPE);

      /* check return value of rpc_call() */
      if (rslt != RPC_SUCCESS) {
         fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
         fprintf(stderr, "clnt_stat: %d\n", rslt);
         fprintf(stderr, "errno: %d\n", errno);
         return 1;
      }

      /* print results and exit */
      printf("Service has been unregistered.\n");
      printf("You must still kill the job in QBATCH\n");
      break;

   case EXIT:

      /* do nothing and exit */
      printf("Exiting program now.\n");
      return 1;
      break;

   default:

      /* an invalid procedure number was entered */
      /* we could just exit here */
      printf("Invalid choice.  Issuing NULLRPOC instead.\n");
      procnum = NULLPROC;

      /* rpc_call(host, prognum, versnum, procnum,    */
      /*          xdr_in, in, xdr_out, out, nettype); */
      rslt = rpc_call(hostname, PROGNUM, VERSNUM, procnum,
                      (xdrproc_t)xdr_void, (char *)NULL, /* xdr_in  */
                      (xdrproc_t)xdr_void, (char *)NULL, /* xdr_out */
                      NETTYPE);

      /* check return value of rpc_call() */
      if (rslt != RPC_SUCCESS) {
         fprintf(stderr, "Error calling rpc_call(%lu)\n", procnum);
         fprintf(stderr, "clnt_stat: %d\n", rslt);
         fprintf(stderr, "errno: %d\n", errno);
         return 1;
      }

      /* print results and exit */
      printf("NULLRPOC call succeeded\n");
      break;

} /* end of switch(procnum) */

/* no cleanup is required for rpc_call() */
return 0;
```

```
}

void myapp_print_menu(void) {

   /* print out the procedure choices */
   printf("%.2ld - GET_UID            %.2ld - GET_UID_STRING\n",
          GET_UID, GET_UID_STRING);
   printf("%.2ld - GET_SIZE           %.2ld - GET_MTIME\n",
          GET_SIZE, GET_MTIME);
   printf("%.2ld - GET_MTIME_STRING   %.2ld - GET_CODEPAGE\n",
          GET_MTIME_STRING, GET_CODEPAGE);
   printf("%.2ld - GET_OBJTYPE        %.2ld - GET_FILETYPE\n",
          GET_OBJTYPE, GET_FILETYPE);
   printf("%.2ld - END_SERVER         %.2d - EXIT\n",
          END_SERVER, EXIT);

}
```

Example: TI-RPC top level client API

The following code example illustrates a top level client application programming interface (API) that is used in developing transport independent remote procedure call (TI-RPC) applications.

At the top level, you must create a client handle before you can use it or modify it. Top level APIs are easy to use, and they allow more manipulation and error handling than the simplified level.

```
#include <stdio.h>
#include <netconfig.h>
#include <netdir.h>
#include <errno.h>
#include "myapp.h"

#define EXIT 100

int main(void) {

   enum clnt_stat rslt;    /* return value of clnt_call() */
   char hostname[256];    /* buffer for remote service's hostname */
   unsigned long procnum;  /* procedure to call */
   char filename[512];     /* buffer for filename */
   xdrproc_t xdr_argument; /* xdr procedure to encode arguments */
   xdrproc_t xdr_result;   /* xdr procedure to decode results */
   CLIENT *clnt;           /* pointer to client handle */
   struct timeval tout;    /* timeout for clnt_call() */
   char *arg = filename;   /* pointer to filename buffer */

   union {
      u_int   myapp_get_uid_result;
      char *  myapp_get_uid_string_result;
      int     myapp_get_size_result;
      long    myapp_get_mtime_result;
      char *  myapp_get_mtime_string_result;
      u_short myapp_get_codepage_result;
      char *  myapp_get_objtype_result;
      char *  myapp_get_filetype_result;
   } result; /* a union of all the possible results */

   tout.tv_sec  = 30;  /* set default timeout to 30.00 seconds */
   tout.tv_usec =  0;

   /* get the hostname from the user */
   printf("Enter the hostname where the remote service is running: \n");
   scanf("%s", (char *)&hostname);

   myapp_print_menu(); /* print out the menu choices */
```

```
/* get the procedure number to call from the user */
printf("\nEnter a procedure number to call: \n");
scanf("%lu", &procnum);

/* get the filename from the user */
printf("\nEnter a filename to stat: \n");
scanf("%s", (char *)&filename);

/* clnt_create(host, prognum, versnum, nettype);    */
clnt = clnt_create(hostname, PROGNUM, VERSNUM, NETTYPE);

/* check to make sure clnt_create() didn't fail */
if (clnt == (CLIENT *)NULL) {
   /* if we failed, print out all appropriate error messages and exit */
   fprintf(stderr, "Error calling clnt_create()\n");
   fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
           PROGNUM, VERSNUM, NETTYPE);
   fprintf(stderr, "clnt_stat: %d\n", rpc_createerr.cf_stat);
   fprintf(stderr, "errno: %d\n", errno);
   fprintf(stderr, "re_errno: %d\n", rpc_createerr.cf_error.re_errno);
   return 1;
}

/* switch on the input */
switch (procnum) {

   case NULLPROC:
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

   case GET_UID:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_u_int;
      break;

   case GET_UID_STRING:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_SIZE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_int;
      break;

   case GET_MTIME:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_long;
      break;

   case GET_MTIME_STRING:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
```

```
      break;

   case GET_CODEPAGE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_u_short;
      break;

   case GET_OBJTYPE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_FILETYPE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case END_SERVER:
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

   case EXIT:
      /* we're done.  clean up and exit */
      clnt_destroy(clnt);
      return 1;
      break;

   default:
      /* invalid procedure number entered.  defaulting to NULLPROC */
      printf("Invalid choice.  Issuing NULLRPOC instead.\n");
      procnum = NULLPROC;
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

} /* end of switch(procnum) */

/* clnt_call(client, procnum, xdr_inproc, in, xdr_outproc, out, timeout) */
rslt = clnt_call(clnt, procnum, xdr_argument, (char *)&arg,
                 xdr_result, (char *)&result, tout);

/* check to make sure clnt_call() succeeded */
if (rslt != RPC_SUCCESS) {
   /* if clnt_call() failed, print errors and exit */
   printf("An error occurred calling %lu procedure\n", procnum);
   printf("clnt_stat: %d\terrno: %d\n", rslt, errno);
   clnt_destroy(clnt);
   return 1;
}

/* clnt_call() succeeded.  switch on procedure and print results */
switch (procnum) {

   case NULLPROC:
      /* print results and exit */
      printf("NULLRPOC call succeeded\n");
```

```
      break;

   case GET_UID:
      /* print results and exit */
      printf("uid of %s: %u\n",
             filename, result.myapp_get_uid_result);
      break;

   case GET_UID_STRING:
      /* print results and exit */
      printf("owner of %s: %s\n",
             filename, result.myapp_get_uid_string_result);
      break;

   case GET_SIZE:
      /* print results and exit */
      printf("size of %s: %d\n",
             filename, result.myapp_get_size_result);
      break;

   case GET_MTIME:
      /* print results and exit */
      printf("last modified time of %s: %ld\n",
             filename, result.myapp_get_mtime_result);
      break;

   case GET_MTIME_STRING:
      /* print results and exit */
      printf("last modified time of %s: %s\n",
             filename, result.myapp_get_mtime_string_result);
      break;

   case GET_CODEPAGE:
      /* print results and exit */
      printf("codepage of %s: %d\n",
             filename, result.myapp_get_codepage_result);
      break;

   case GET_OBJTYPE:
      /* print results and exit */
      printf("object type of %s: %s\n",
             filename, result.myapp_get_objtype_result);
      break;

   case GET_FILETYPE:
      /* print results and exit */
      printf("file type of %s: %s\n",
             filename, result.myapp_get_filetype_result);
      break;

   case END_SERVER:
      /* print results and exit */
      printf("Service has been unregistered.\n");
      printf("You must still kill the job in QBATCH\n");
      break;

   default:
      /* we should never get the default case.  */
      /* the previous switch should catch it.   */
      break;

} /* end of switch(procnum) */


/* clean up and exit */
clnt_destroy(clnt);
```

```
   return 0;

}

void myapp_print_menu(void) {

   /* print out the procedure choices */
   printf("%.2ld - GET_UID             %.2ld - GET_UID_STRING\n",
          GET_UID, GET_UID_STRING);
   printf("%.2ld - GET_SIZE            %.2ld - GET_MTIME\n",
          GET_SIZE, GET_MTIME);
   printf("%.2ld - GET_MTIME_STRING    %.2ld - GET_CODEPAGE\n",
          GET_MTIME_STRING, GET_CODEPAGE);
   printf("%.2ld - GET_OBJTYPE         %.2ld - GET_FILETYPE\n",
          GET_OBJTYPE, GET_FILETYPE);
   printf("%.2ld - END_SERVER          %.2d - EXIT\n",
          END_SERVER, EXIT);

}
```

Example: TI-RPC intermediate level client API

The following code example illustrates an intermediate level client application programming interface (API) that is used in developing transport independent remote procedure call (TI-RPC) applications.

The intermediate level for the client follows the same path as it does for the service. For example, the programmer is responsible for using the network selection APIs to obtain transport information instead of passing a simple text string.

```
#include <stdio.h>
#include <netconfig.h>
#include <netdir.h>
#include <errno.h>
#include "myapp.h"

#define EXIT 100

int main(void) {

   enum clnt_stat rslt;    /* return value of clnt_call() */
   char hostname[256];    /* buffer for remote service's hostname */
   unsigned long procnum;  /* procedure to call */
   char filename[512];     /* buffer for filename */
   xdrproc_t xdr_argument; /* xdr procedure to encode arguments */
   xdrproc_t xdr_result;   /* xdr procedure to decode results */
   CLIENT *clnt;           /* pointer to client handle */
   struct timeval tout;    /* timeout for clnt_call() */
   struct netconfig *nconf;/* transport information */
   char *arg = filename;   /* pointer to filename buffer */

   union {
      u_int   myapp_get_uid_result;
      char *  myapp_get_uid_string_result;
      int     myapp_get_size_result;
      long    myapp_get_mtime_result;
      char *  myapp_get_mtime_string_result;
      u_short myapp_get_codepage_result;
      char *  myapp_get_objtype_result;
      char *  myapp_get_filetype_result;
   } result; /* a union of all the possible results */


   tout.tv_sec  = 30;  /* set default timeout to 30.00 seconds */
   tout.tv_usec =  0;

   /* get the hostname from the user */
```

```
printf("Enter the hostname where the remote service is running: \n");
scanf("%s", (char *)&hostname);

myapp_print_menu(); /* print out the menu choices */

/* get the procedure number to call from the user */
printf("\nEnter a procedure number to call: \n");
scanf("%lu", &procnum);

/* get the filename from the user */
printf("\nEnter a filename to stat: \n");
scanf("%s", (char *)&filename);

/* getnetconfigent(nettype) */
nconf = getnetconfigent(NETTYPE);

/* check to make sure getnetconfigent() didn't fail */
if (nconf == NULL) {
   /* if getnetconfigent() failed, print error messages and exit */
   fprintf(stderr, "Error calling getnetconfigent(%s)\n", NETTYPE);
   fprintf(stderr, "errno: %d\n", errno);
   return 1;
}

/* clnt_tp_create(host, prognum, versnum, netconf) */
clnt = clnt_tp_create(hostname, PROGNUM, VERSNUM, nconf);

/* check to make sure clnt_tp_create() didn't fail */
if (clnt == (CLIENT *)NULL) {
   fprintf(stderr, "Error calling clnt_tp_create()\n");
   fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
           PROGNUM, VERSNUM, NETTYPE);
   fprintf(stderr, "clnt_stat: %d\n", rpc_createerr.cf_stat);
   fprintf(stderr, "errno: %d\n", errno);
   fprintf(stderr, "re_errno: %d\n", rpc_createerr.cf_error.re_errno);
   return 1;
}

/* switch on the input */
switch (procnum) {

   case NULLPROC:
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

   case GET_UID:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_u_int;
      break;

   case GET_UID_STRING:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_SIZE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_int;
```

```
      break;

   case GET_MTIME:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_long;
      break;

   case GET_MTIME_STRING:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_CODEPAGE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_u_short;
      break;

   case GET_OBJTYPE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_FILETYPE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case END_SERVER:
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

   case EXIT:
      /* we're done.  clean up and exit */
      clnt_destroy(clnt);
      return 1;
      break;

   default:
      /* invalid procedure number entered.  defaulting to NULLPROC */
      printf("Invalid choice.  Issuing NULLRPOC instead.\n");
      procnum = NULLPROC;
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

} /* end of switch(procnum) */

/* clnt_call(client, procnum, xdr_inproc, in, xdr_outproc, out, timeout) */
rslt = clnt_call(clnt, procnum, xdr_argument, (char *)&arg,
                 xdr_result, (char *)&result, tout);

/* check to make sure clnt_call() succeeded */
```

```
if (rslt != RPC_SUCCESS) {
   /* if clnt_call() failed, print errors and exit */
   printf("An error occurred calling %lu procedure\n", procnum);
   printf("clnt_stat: %d\terrno: %d\n", rslt, errno);
   clnt_destroy(clnt);
   return 1;
}

/* clnt_call() succeeded.  switch on procedure and print results */
switch (procnum) {

   case NULLPROC:
      /* print results and exit */
      printf("NULLRPOC call succeeded\n");
      break;

   case GET_UID:
      /* print results and exit */
      printf("uid of %s: %u\n",
             filename, result.myapp_get_uid_result);
      break;

   case GET_UID_STRING:
      /* print results and exit */
      printf("owner of %s: %s\n",
             filename, result.myapp_get_uid_string_result);
      break;

   case GET_SIZE:
      /* print results and exit */
      printf("size of %s: %d\n",
             filename, result.myapp_get_size_result);
      break;

   case GET_MTIME:
      /* print results and exit */
      printf("last modified time of %s: %ld\n",
             filename, result.myapp_get_mtime_result);
      break;

   case GET_MTIME_STRING:
      /* print results and exit */
      printf("last modified time of %s: %s\n",
             filename, result.myapp_get_mtime_string_result);
      break;

   case GET_CODEPAGE:
      /* print results and exit */
      printf("codepage of %s: %d\n",
             filename, result.myapp_get_codepage_result);
      break;

   case GET_OBJTYPE:
      /* print results and exit */
      printf("object type of %s: %s\n",
             filename, result.myapp_get_objtype_result);
      break;

   case GET_FILETYPE:
      /* print results and exit */
      printf("file type of %s: %s\n",
             filename, result.myapp_get_filetype_result);
      break;

   case END_SERVER:
      /* print results and exit */
      printf("Service has been unregistered.\n");
```

```
         printf("You must still kill the job in QBATCH\n");
         break;

      default:
         /* we should never get the default case.  */
         /* the previous switch should catch it.   */
         break;

   } /* end of switch(procnum) */


   /* clean up and exit */

   /* free the netconfig struct */
   freenetconfigent(nconf);
   /* free the universal address buffer */
   free(svcaddr.buf);
   /* destroy the client handle */
   clnt_destroy(clnt);


   return 0;

}

void myapp_print_menu(void) {

   /* print out the procedure choices */
   printf("%.2ld - GET_UID             %.2ld - GET_UID_STRING\n",
          GET_UID, GET_UID_STRING);
   printf("%.2ld - GET_SIZE            %.2ld - GET_MTIME\n",
          GET_SIZE, GET_MTIME);
   printf("%.2ld - GET_MTIME_STRING    %.2ld - GET_CODEPAGE\n",
          GET_MTIME_STRING, GET_CODEPAGE);
   printf("%.2ld - GET_OBJTYPE         %.2ld - GET_FILETYPE\n",
          GET_OBJTYPE, GET_FILETYPE);
   printf("%.2ld - END_SERVER          %.2d - EXIT\n",
          END_SERVER, EXIT);

}
```

Example: TI-RPC expert level client API

The following code example illustrates an expert level client application programming interface (API) that is used in developing transport independent remote procedure call (TI-RPC) applications.

The expert level for the development of a client API is the most complicated. It is also offers the most customization. This is the only level where the buffer size can be tuned for the client API. This level requires the programmer to set up the universal address for the client to connect to, either by using the name-to-address translation APIs or one of the other expert level APIs. Either way, this level requires more work, but it allows the programmer the ability to tailor the client application to the environment it runs in.

```
#include <stdio.h>
#include <netconfig.h>
#include <netdir.h>
#include <errno.h>
#include "myapp.h"

#define EXIT 100

int main(void) {

   enum clnt_stat rslt;    /* return value of clnt_call() */
   char hostname[256];    /* buffer for remote service's hostname */
   unsigned long procnum;  /* procedure to call */
   char filename[512];     /* buffer for filename */
```

```
xdrproc_t xdr_argument; /* xdr procedure to encode arguments */
xdrproc_t xdr_result;   /* xdr procedure to decode results */
CLIENT *clnt;           /* pointer to client handle */
struct timeval tout;    /* timeout for clnt_call() */
struct netconfig *nconf;/* transport information */
struct netbuf svcaddr;  /* universal address of remote service */
bool_t rpcb_rslt;       /* return value for rpcb_getaddr() */
char *arg = filename;   /* pointer to filename buffer */

union {
   u_int   myapp_get_uid_result;
   char *  myapp_get_uid_string_result;
   int     myapp_get_size_result;
   long    myapp_get_mtime_result;
   char *  myapp_get_mtime_string_result;
   u_short myapp_get_codepage_result;
   char *  myapp_get_objtype_result;
   char *  myapp_get_filetype_result;
} result; /* a union of all the possible results */

/* initialize the struct netbuf space */
svcaddr.maxlen = 16;
svcaddr.buf = (char *)malloc(svcaddr.maxlen);

if (svcaddr.buf == (char *)NULL) {
   /* if malloc() failed, print error messages and exit */
   fprintf(stderr, "Error calling malloc() for struct netbuf\n");
   fprintf(stderr, "errno: %d\n", errno);
   return 1;
}

tout.tv_sec  = 30;  /* set default timeout to 30.00 seconds */
tout.tv_usec =  0;

/* get the hostname from the user */
printf("Enter the hostname where the remote service is running: \n");
scanf("%s", (char *)&hostname);

myapp_print_menu(); /* print out the menu choices */

/* get the procedure number to call from the user */
printf("\nEnter a procedure number to call: \n");
scanf("%lu", &procnum);

/* get the filename from the user */
printf("\nEnter a filename to stat: \n");
scanf("%s", (char *)&filename);

/* getnetconfigent(nettype) */
nconf = getnetconfigent(NETTYPE);

/* check to make sure getnetconfigent() didn't fail */
if (nconf == NULL) {
   /* if getnetconfigent() failed, print error messages and exit */
   fprintf(stderr, "Error calling getnetconfigent(%s)\n", NETTYPE);
   fprintf(stderr, "errno: %d\n", errno);
   return 1;
}

/* rpcb_getaddr(prognum, versnum, nconf, output netbuf, hostname) */
/* this sets the universal address svcaddr */
rpcb_rslt = rpcb_getaddr(PROGNUM, VERSNUM, nconf, &svcaddr, hostname);

/* check to make sure rpcb_getaddr() didn't fail */
if (rpcb_rslt == FALSE) {
   /* if rpcb_getaddr() failed, print error messages and exit */
   fprintf(stderr, "Error calling rpcb_getaddr()\n");
```

```
   fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
           PROGNUM, VERSNUM, NETTYPE);
   fprintf(stderr, "clnt_stat: %d\n", rpc_createerr.cf_stat);
   fprintf(stderr, "errno: %d\n", errno);
   fprintf(stderr, "re_errno: %d\n", rpc_createerr.cf_error.re_errno);
   return 1;
}

/* clnt_tli_create(filedes, netconfig, netbuf,           */
/*                 prognum, versnum, sendsz, recvsz);    */
clnt = clnt_tli_create(RPC_ANYFD, nconf, &svcaddr,
                       PROGNUM, VERSNUM, 0, 0);

/* check to make sure clnt_tli_create() didn't fail */
if (clnt == (CLIENT *)NULL) {
   /* if we failed, print out all appropriate error messages and exit */
   fprintf(stderr, "Error calling clnt_tli_create()\n");
   fprintf(stderr, "PROG: %lu\tVERS: %lu\tNET: %s\n",
           PROGNUM, VERSNUM, NETTYPE);
   fprintf(stderr, "clnt_stat: %d\n", rpc_createerr.cf_stat);
   fprintf(stderr, "errno: %d\n", errno);
   fprintf(stderr, "re_errno: %d\n", rpc_createerr.cf_error.re_errno);
   return 1;
}

/* switch on the input */
switch (procnum) {

   case NULLPROC:
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

   case GET_UID:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_u_int;
      break;

   case GET_UID_STRING:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_SIZE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_int;
      break;

   case GET_MTIME:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_long;
      break;

   case GET_MTIME_STRING:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
```

```
      xdr_result   = xdr_wrapstring;
      break;

   case GET_CODEPAGE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_u_short;
      break;

   case GET_OBJTYPE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case GET_FILETYPE:
      /* set the encode procedure */
      xdr_argument = xdr_wrapstring;
      /* set the decode procedure */
      xdr_result   = xdr_wrapstring;
      break;

   case END_SERVER:
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

   case EXIT:
      /* we're done.  clean up and exit */
      clnt_destroy(clnt);
      return 1;
      break;

   default:
      /* invalid procedure number entered.  defaulting to NULLPROC */
      printf("Invalid choice.  Issuing NULLRPOC instead.\n");
      procnum = NULLPROC;
      /* set the encode procedure */
      xdr_argument = (xdrproc_t)xdr_void;
      /* set the decode procedure */
      xdr_result   = (xdrproc_t)xdr_void;
      break;

} /* end of switch(procnum) */

/* clnt_call(client, procnum, xdr_inproc, in, xdr_outproc, out, timeout) */
rslt = clnt_call(clnt, procnum, xdr_argument, (char *)&arg,
                 xdr_result, (char *)&result, tout);

/* check to make sure clnt_call() succeeded */
if (rslt != RPC_SUCCESS) {
   /* if clnt_call() failed, print errors and exit */
   printf("An error occurred calling %lu procedure\n", procnum);
   printf("clnt_stat: %d\terrno: %d\n", rslt, errno);
   clnt_destroy(clnt);
   return 1;
}

/* clnt_call() succeeded.  switch on procedure and print results */
switch (procnum) {

   case NULLPROC:
      /* print results and exit */
```

```
      printf("NULLRPOC call succeeded\n");
      break;

   case GET_UID:
      /* print results and exit */
      printf("uid of %s: %u\n",
             filename, result.myapp_get_uid_result);
      break;

   case GET_UID_STRING:
      /* print results and exit */
      printf("owner of %s: %s\n",
             filename, result.myapp_get_uid_string_result);
      break;

   case GET_SIZE:
      /* print results and exit */
      printf("size of %s: %d\n",
             filename, result.myapp_get_size_result);
      break;

   case GET_MTIME:
      /* print results and exit */
      printf("last modified time of %s: %ld\n",
             filename, result.myapp_get_mtime_result);
      break;

   case GET_MTIME_STRING:
      /* print results and exit */
      printf("last modified time of %s: %s\n",
             filename, result.myapp_get_mtime_string_result);
      break;

   case GET_CODEPAGE:
      /* print results and exit */
      printf("codepage of %s: %d\n",
             filename, result.myapp_get_codepage_result);
      break;

   case GET_OBJTYPE:
      /* print results and exit */
      printf("object type of %s: %s\n",
             filename, result.myapp_get_objtype_result);
      break;

   case GET_FILETYPE:
      /* print results and exit */
      printf("file type of %s: %s\n",
             filename, result.myapp_get_filetype_result);
      break;

   case END_SERVER:
      /* print results and exit */
      printf("Service has been unregistered.\n");
      printf("You must still kill the job in QBATCH\n");
      break;

   default:
      /* we should never get the default case.  */
      /* the previous switch should catch it.   */
      break;

} /* end of switch(procnum) */


/* clean up and exit */
```

```
   /* free the netconfig struct */
   freenetconfigent(nconf);
   /* free the universal address buffer */
   free(svcaddr.buf);
   /* destroy the client handle */
   clnt_destroy(clnt);


   return 0;

}

void myapp_print_menu(void) {

   /* print out the procedure choices */
   printf("%.2ld - GET_UID             %.2ld - GET_UID_STRING\n",
          GET_UID, GET_UID_STRING);
   printf("%.2ld - GET_SIZE            %.2ld - GET_MTIME\n",
          GET_SIZE, GET_MTIME);
   printf("%.2ld - GET_MTIME_STRING    %.2ld - GET_CODEPAGE\n",
          GET_MTIME_STRING, GET_CODEPAGE);
   printf("%.2ld - GET_OBJTYPE         %.2ld - GET_FILETYPE\n",
          GET_OBJTYPE, GET_FILETYPE);
   printf("%.2ld - END_SERVER          %.2d - EXIT\n",
          END_SERVER, EXIT);

}
```

Example: Adding authentication to the TI-RPC client

The following code snippets display how the authentication system works in remote procedure call (RPC). System is the only authentication method that is provided on OS/400. The following information is setup and passed from the client to the service with every clnt_call(). In the following code snippets, notice that rpc_call() is not sufficient when using authentication information , because it defaults to using `authnone` (an empty authentication token):

- `aup_time - authentication information timestamp`
- `aup_machname - the hostname of the remote client`
- `aup_uid - the UID of the remote user of the client`
- `aup_gid - the primary GID of the remote user`
- `aup_gids - an array of the secondary groups of the remote user`

It is up to the client to set up the authentication information and make it part of the client handle. After that, all subsequent calls to clnt_call() will pass that authentication information along. It is up to the server to report on unauthorized clients. RPC only provides a simple method of communicating the information. The data that is sent by the client is authenticated, but not encrypted. The reply from the service is not encrypted either. Authentication provides a simple way of verifying the remote hostname and the user identification. It cannot be considered a secure and private method of communication.

```
#include <sys/types.h> /* needed for gid_t and uid_t */
#include <stdlib.h>    /* misc. system auth APIs     */
#include <unistd.h>    /* misc. system auth APIs     */
#include <errno.h>

#ifndef NGROUPS_MAX
   #define NGROUPS_MAX 16
#endif

char hostname[256];         /* hostname for credentials */
int rslt;                   /* return value of gethostname() */
gid_t groups[NGROUPS_MAX];  /* array of groups set by getgroups() */
gid_t *aup_gids;            /* pointer to array of gid_t */
uid_t uid;                  /* uid, return value for geteuid() */
```

```
gid_t gid;                  /* gid, return value for getegid() */
int num_groups;             /* return value for getgroups(), number of groups set
*/

aup_gids = groups;          /* point to the array of groups */
uid = geteuid();            /* get the effective uid of the user */
gid = getegid();            /* get the effect primary gid of the user */

/* get a list of other groups the user is a member of */
/* (int)getgroups(maxgropus, array) */
num_groups = getgroups(NGROUPS_MAX, groups);

/* check return value of getgroups() for error */
if (num_groups == -1) {
    /* print error message and exit */
    fprintf(stderr, "getgroups() failed for %d\n", uid);
    fprintf(stderr, "errno: %d\n", errno);
    return 1;
}

/* (int)gethostname(buffer, buflen) */
rslt = gethostname(hostname, 256);

/* check return value of gethostname() for error */
if (rslt == -1) {
    /* print error message and exit */
    fprintf(stderr, "gethostname() failed\n");
    fprintf(stderr, "errno: %d\n", errno);
    return 1;
}

/* insert just before clnt_call() */
/* (AUTH *)authsys_create(hostname, uid, gid, num_groups, gid[]); */
clnt->cl_auth = authsys_create(hostname, uid, gid, num_groups, aup_gids);

if (clnt->cl_auth == NULL) {
    /* print error messages and exit */
    fprintf(stderr, "authsys_create() failed\n");
    fprintf(stderr, "errno: %d\n", errno);
    /* clean up */
    clnt_destroy(clnt);
    return 1;
}
```

Chapter 8. Other information about TI-RPC

For detailed information about designing, implementing, and maintaining distributed applications by using TI-RPC, see the ONC+ Developer's Guide  by Sun Microsystems, Inc. (1997).

Refer to the rpcgen Programmer's Guide  by Sun Microsystems, Inc. (1997) for details about using the rpcgen compiler.

For more information about TI-RPC service application programming interfaces (APIs), see the System API Reference.

About the coding examples

This web page contains small programs that are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you ″AS IS″. THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

© Copyright IBM Corp. 1998, 2002

Printed in U.S.A.