Remote Procedure Calls (RPC)

Remote Procedure Call

- Remote procedures appear local through stub functions
- Stub functions have the same functional interface
  - Communicate with server
  - Await results from server
Stub Function Generation

- Programming languages do not support Sun RPC
  - A separate pre-compiler, `rpcgen`, must be used
- Input:
  - Interface definition language
- Output:
  - Server main routine
  - Client stub functions
  - Header file
  - Data conversion functions, if needed
  - Makefile

Interface Definition Language

- Used by `rpcgen` to generate stub functions
- Defines an RPC program: collection of RPC procedures
- Structure:

```
 type definitions

 program identifier {
   version version_id {
     procedure list
     } = value;
   ... 
     } = value;

 program PROG {
   version PROG1 {
     void PROC_A(int) = 1;
   } = 1;
   } = 0x3a3afebb;
```
Data Types

- **Constants**
  - May be used in place of an integer value - converted to `#define` statement by `rpcgen`
    
    ```
    const MAXSIZE = 512;
    ```

- **Structures**
  - Similar to C structures - `rpcgen` transfers structure definitions and adds a typedef for the name of the structure
    
    ```
    struct intpair { int a, b; }
    ```
    
    is translated to:
    
    ```
    typedef struct intpair intpair;
    ```

- **Enumerations**
  - Similar to C
  - ```
    enum state { BUSY=1, IDLE=2, TRANSIT=3; }
    ```

- **Unions**
  - Not like C
  - A union is a specification of data types based on some criteria:
    ```
    union identifier switch (declaration) {
      case_list
    }
    ```
  - For example:
    ```
    const MAXBUF=30;
    union time_results switch (int status) {
      case 0: char timeval[MAXBUF];
      case 1: void;
      case 2: int reason;
    }
    ```
Data Types

- **Type definitions**
  - Like C:
    
    ```
    typedef long counter;
    ```

- **Arrays**
  - Like C but may have a fixed or variable length:
    
    ```
    int proc_hits[100];
    ```
     defines a fixed size array of 100 integers.
    
    ```
    long x_vals<50>
    ```
     defines a variable-size array of a maximum of 50 longs

- **Pointers**
  - Like C, but not sent over the network. What is sent is a Boolean value (true for pointer, false for null) followed by the data to which the pointer points

- **Strings**
  - Declared as if they were variable length arrays
    
    ```
    string name<50>;
    ```
     Declares a string of at most 50 characters.
    
    ```
    string anyname<>
    ```
     Declares a string of any number of characters.

- **Boolean**
  - Can have the value of TRUE or FALSE:
    
    ```
    bool busy;
    ```

- **Opaque data**
  - Untyped data that contains an arbitrary sequence of bytes - may be fixed or variable length:
    
    ```
    opaque extra_bytes[512];
    ```
    ```
    opaque more<512>;
    ```
    
    Latter definition is translated to C as:
    
    ```
    struct {
    uint more_len; /* length of array */
    char *more_val; /* space used by array */
    }
    ```
Using rpcgen

- `rpcgen` has the following options that are nice:
  - `a` – generate all the files including sample code for the server and client
  - `C` – generate ANSI C code. This should be used if you are using the C++ compiler
  - `N` – use the new style of `rpcgen`. This allows procedures to have multiple arguments. It also uses the style of parameter passing that closely resembles C

Writing Functions using RPC

- Create a procedure whose name is the name of the RPC definition
  - In lowercase
  - Followed by an underscore, version number, underscore, “svc”
  - For example, BLIP → blip_1_svc
- Argument to procedure is a `pointer` to the argument data type specified in the IDL
- Default behavior: only one parameter to each function
  - If you want more, use a struct
  - This was relaxed in later versions of `rpcgen` but remains the default
- Procedure must return a `pointer` to the data type specified in the IDL
- The server stub uses the procedure’s return value after the procedure returns, so the return address must be that of a `static` variable
Sample RPC Program

- Start with stand-alone program that has two functions:
  - `bin_date` returns system date as # seconds since Jan 1 1970 0:00 GMT
  - `str_date` takes the number of seconds as input and returns a formatted data string

- **Goal**
  - Move `bin_date` and `str_date` into server functions and call them via RPC.

---

Stand-alone Program

```c
#include <stdio.h>
long bin_date(void);
char *str_date(long bintime);
int main(int argc, char **argv)
{
    long lresult; /* return from bin_date */
    char *sresult; /* return from str_date */
    if ( argc != 1 )
    {
        fprintf(stderr, "usage: %s\n", argv[0]);
        exit(1);
    }
    /* call the procedure bin_date */
    lresult = bin_date();
    printf("time is %ld\n", lresult);
    /* convert the result to a date string */
    sresult = str_date(lresult);
    printf("date is %s", sresult);
} /* main */
```
Stand-alone Program: Functions

/* bin_date returns the system time in binary format */
long bin_date( void )
{
    long timeval;
    long time(); /* Unix time function; returns time */
    timeval = time( (long *)0 );
    return timeval;
} /* bin_date */

/* str_date converts a binary time into a date string */
char *str_date( long bintime )
{
    char *ptr;
    char *ctime(); /* Unix library function that does the work */
    ptr = ctime( &bintime );
    return ptr;
} /* str_date */

Define Remote Interface (IDL)

- Define two functions that run on server:
  - bin_date has no input parameters and returns a long
  - str_date accepts a long as input and returns a string

- IDL:
  program DATE_PROG {
    version DATE_VERS {
      long BIN_DATE(void) = 1;
      string STR_DATE(long) = 2;
    } = 1;  
    } = 55;  

- IDL convention is to suffix the file with .x
  - We name the file date.x
  - It can be compiled with:
    rpcgen -C -N -a date.x
date_server.c

#include "date.h"
long *
bin_date_1_svc( struct svc_req *rqstp )
{
    static long result;
    /* insert server code here */
    return &result;
}

char **
str_date_1_svc( long arg1, struct svc_req *rqstp )
{
    static char *result;
    /* insert server code here */
    return &result;
}

Fixing date_server.c

- Now just copy the functions from the original stand-alone code
  long *
  bin_date_1_svc( struct svc_req *rqstp )
  {
      static long result;

      result = time( (long *)0 );
      return &result;
  }
  char **
  str_date_1_svc( long arg1, struct svc_req *rqstp )
  {
      static char *result;

      result = ctime( &arg1 );
      return &result;
  }
Fixing data_client.c

- We need to know the name of the server
  - Use `getopt` library function to accept a `-h` **hostname** argument on the command line.
  - The following is placed in the main function of data_client.c
    ```c
    char *host = "localhost"; /* default */
    char c;
    int err = 0;
    while ( (c = getopt(argc, argv, "h:" )) != -1 )
    switch (c)
    {  case 'h': server = optarg;
        break;
        case '?': err = 1;
        break;
    } /* switch */
    /* exit if error or extra arguments */
    if (err || (optind < argc))
    {   fprintf(stderr, "usage: %s [-h hostname]
    exit(1);
    } /* if */
    ```

Fixing date_client.c (cont)

- Before we can make any remote procedure calls, we need change the last parameter from udp to tcp:
  ```c
  cl = clnt_create(server, DATE_PROG, DATE_VERS, "tcp");
  ```
- Program and version numbers are defined in date.h.
- “tcp” directs to transport using TCP.
- The server’s RPC name server (port mapper) is contacted to find the port for the requested program/version/transport.
  - Remember, you may have to start or stop the port mapper
    - `/etc/init.d/portmap start`
    - `/etc/init.d/portmap stop`
Fixing date_client.c (cont)

- Fix the call to str_date to take the result from bin_date
- Change
  result_2 = str_date_1( str_date_1_arg1, clnt);
- To
  result_2 = str_date_1( *result_1, clnt);

Fixing date_client.c (cont)

- If the call to bin_date_1 succeeds, print the result:
  printf( “Time on %s is %ld\n” server, *result_1);
- If the call to str_date_1 succeeds, print the result:
  printf( “Date is %s”, *result_2 );
Compile – Link – Run

- Generate the stubs
  `rpcgen -a -N -C date.x`

- Compile and link the client code
  `gcc -g -c date_clnt.c`
  `gcc -g -c date_client.c`
  `gcc -g -o date_client date_clnt.o date_client.o -lnsl`

- Compile and link the server code
  `gcc -g -c date_svc.c`
  `gcc -g -c date_server.c`
  `gcc -g -o date_server date_svc.o date_server.o -lnsl`

- Run the server
  `$ ./date_server`

- Run the client
  `$ ./date_client -h kiev.cs.usu.edu`
  Time on localhost is 1067637751
  Date is Fri Oct 31 15:02:31 2003