Programming with SUN RPC

CS 417
Distributed Systems
Remote Procedure Call

- remote procedures appear local through stub functions
- stub functions have the same functional interface
  - communicate with server
  - await results from server

Remote procedure calls appear local because a local procedure exists that provides the same interface.

This procedure gathers up the parameters and converts them into a flat, pointerless representation that is sent as a network message to a server. This data conversion is known as marshaling.

N.B.: pointers are useless on the remote side since they refer to local memory locations.

The server, upon receiving the message, reassembles the parameters into a form that is readable on that machine (correct byte ordering, word sizes, etc.) and calls the user-written server function. Upon return from the server function, any return value is marshaled into a network message and sent back to the client.

The client receives the return message, unmarshals the data, and returns it back to the calling client code.
**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

RPC is a language construct, meaning it is a property of the programming language (since it deals with the semantics of function calls).

However, no languages support Sun RPC, so a pre-compiler must be used to generate the stub functions on the client and the server.

The Sun RPC compiler is called `rpcgen`. As input, it takes a list of remote procedures (interfaces) defined in an **interface definition language** (IDL).

The output from `rpcgen` is a set of files that include:

**server code**: main function that sets up a socket, registers the port with a name server, listens for and accepts connections, receives messages, unmarshals parameters, calls the user-written server function, marshals the return value, and sends back a network message.

**client stub**: code with the interface of the remote function that marshals parameters, sends the message to the server, and unmarshals the return value

**header**: contains definitions of symbols used by client and server as well as function prototypes

**data conversion functions**: a separate file may be generated if special functions need to be called to convert between local data types and their marshaled forms.
The Interface Definition Language (IDL) is the one bit of input needed by `rpcgen` to generate the stub functions.

The structure of IDL is vaguely similar to a set of C prototype definitions.

Note that any similarity to C is essentially coincidental: RPC IDL is a separate definition language that is not C.

Each IDL program contains the following structure:

- optional constant definitions and typedefs may be present
- the entire interface is enveloped in a `program` block. The sample on the right gives a name PROG to the set of interfaces and a numeric value of 0x3a3afeeb. Sun decreed that each collection of RPC interfaces is identified by a 32 bit value that you have to select. The restrictions given are:
  
  0x00000000-0x1fffffff: defined by sun  
  0x20000000-0x3fffffff defined by the user  
  0x40000000-0x5fffffff transient processes  
  0x60000000-0x7fffffff reserved

- within the program block, one or more sets of versions may be defined. A client program will always request an interface by asking for a `{program#, version#}` tuple. Each version contains a version name and number. In the sample on the right, the version name is PROG1 and the number is 1.

- within each version block, a set of functions is defined. These look similar to C prototypes and are tagged with a function number (each function gets a unique number within a version block).
Data types

• constants
  – may be used in place of an integer value - converted to \#define statement by \textit{rpcgen}

  \texttt{const MAXSIZE = 512;}

• structures
  – similar to C structures - \textit{rpcgen} transfers structure definition and adds a typedef for the name of the structure

  \texttt{struct intpair \{ int a, b \};}

  is translated to:

  \texttt{struct intpair \{ int a, b \};
  typedef struct intpair intpair;
Data types

- enumerations
  - similar to C
    ```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:
    ```c
    union identifier switch (declaration) {
      case_list
    }
    ```
  - for example:
    ```c
    const MAXBUF=30;
    union time_results switch (int status) {
      case 0: char timeval[MAXBUF];
      case 1: void;
      case 2: int reason;
    }
    ```

enumerations
- defines that state can have the value of one of the symbols: BUSY, IDLE, or TRANSIT. The symbols are defined to be the values 1, 2, and 3 respectively.

unions
- very different from C (similar to discriminated unions of Pascal or ADA)

The example shows that the union has a field of status. If status is set to 0, then the union also has a character array called timeval. If status is set to 1, then the union has no other fields, and if status is set to 2, then the union has an integer field called reason.
Data types

- type definitions
  - like C:
    ```c
    typedef long counter;
    ```
- arrays
  - like C but may have a fixed or variable length:
    ```c
    int proc_hits[100];
    ```
    defines a fixed size array of 100 integers.
    ```c
    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.
Data types

• 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 */
    }
Writing procedures using Sun 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 # 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

#include <stdio.h>
long bin_date(void);
char *str_date(long bintime);

main(int argc, char **argv)
{
    long lresult; /* return from bin_date */
    char *sresult; /* return from str_date */
    if (argc != 1) {
        fprintf(stderr, "usage: %s
", 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);
    exit(0);
}
**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;
}

/* 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;
}
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;
  } = 0x31423456;
  }
  ```

- IDL convention is to suffix the file with .x
  - we name the file date.x
  - it can be compiled with:
    ```
    rpcgen -C date.x
    ```

We have to envelope our two functions in a version block. This in turn has to be enveloped in a program block.

Each function is assigned an arbitrary number.

We pick a number for the program number and hope that nobody on our server will pick the same one.

When the file (date.x) is compiled with rpcgen -C date.x (the -C is to produce ANSI C function declarations), we get:

- **date.h**: header file
- **date_clnt.c**: client stub
- **date_svc.c**: server stub
Generating server functions: templates from rpcgen

- We can have `rpcgen` generate a template for the server code using the interface we defined:
  ```
rpcgen -C -Ss date.x >server.c
  ```
- This produces:
  ```
#include "date.h"
long *
bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
    static long result;
    /* insert server code here */
    return &result;
}

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

Note:
- the names we selected for the procedures have been modified: converted to lowercase and suffixed with an underscore, version, underscore, “svc”.
- each function has an extra parameter: `struct svc_req *rstp`. We generally won’t use this, but it’s the request structure that allows us to find out about where the request is coming from.
- the input parameter is a **pointer** to the type we asked for.
- the return parameter is also a **pointer** to the type we asked for.
- static results are generated. This is important because we return the *address* of the result. The address of a local variable lives on the stack and may be overwritten once a function has returned.
Generating server functions: plug in the code

- Now just copy the functions from the original stand-alone code

```c
long *
bin_date_1_svc(void *argp, struct svc_req *rqstp)
{
    static long result;
    long time();
    result = time((long *)0);
    return &result;
}

char **
str_date_1_svc(long *bintime, struct svc_req *rqstp)
{
    static char *result;
    char *ctime();
    result = ctime(bintime);
    return &result;
}
```

For the first function, we don’t use any incoming parameters, so we ignore the argument.

As guided by the template code, we return a `pointer` to the return type. The auto-generated code made `result` static so that the data won’t be out of scope when `bin_date_1_svc` returns.

For the second function, the first parameter is a `pointer` to the binary time (we renamed it to bintime, which is more meaningful than argp). The return value is stored in result.

Once again, we return a `pointer` to the data type we’re interested in. In this case, we want a char *, so we return a pointer to that – a char **.
Generating the client: get the server name

- We need to know the name of the server
  - use `getopt` library function to accept a `-h hostname` argument on the command line.

```c
extern char *optarg;
extern int optind;
char *server = "localhost"; /* default */
int err = 0;

while ((c = getopt(argc, argv, "h:")) != -1)
    switch (c) {
    case 'h':
        server = optarg;
        break;
    case '?':
        err = 1;
        break;
    }
/* exit if error or extra arguments */
if (err || (optind < argc)) {
    fprintf(stderr, "usage:  %s [-h hostname]\n", argv[0]);
    exit(1);
}
```

We now modify our main program (client) to accept a parameter `-h hostname`.

We’ll use `getopt` just to make life easier in the future when we may want to add more options.
Generating the client: add headers and create client handle

- We need a couple of extra #include directives:
  ```
  #include <rpc/rpc.h>
  #include "date.h"
  ```
- Before we can make any remote procedure calls, we need to initialize the RPC connection via `clnt_create`:
  ```
  CLIENT *cl; /* rpc handle */
  cl = clnt_create(server, DATE_PROG, DATE_VERS, "netpath");
  ```
- Program and version numbers are defined in `date.h`.
- “netpath” directs to read the NETPATH environment variable to decide on using TCP or UDP.
- The server’s RPC name server (port mapper) is contacted to find the port for the requested program/version/transport.

The main program needs a couple of headers: `rpc/rpc.h`, which defines rpc structures (such as the client handle) and `date.h`, which defines our remote functions.

Then we need to establish a connection with the server. To do this, we define a client handle and call `clnt_create`. This will contact the RPC name server to find the port number on the server for the requested program and version.

The “netpath” transport tells `clnt_create` to use look at the NETPATH environment variable to select the transport protocol (Linux does not support this). You can also explicitly state “tcp” or “udp”.
Generating the client: modify calls to remote functions

- Client’s calls to bin_date and str_date have to be modified:
  - add version number to the function
  - add a client handle as a parameter (from `clnt_create`)
  - always pass a single parameter (NULL if there is none)

```c
bin_date_1(NULL, cl);
str_date_1(&value, cl);
```

The names of the remote functions are changed to reflect the version number and to reflect the restriction of passing a single parameter and a client handle. Sun RPC now supports multiple parameters but the default is to use the original behavior – if you need multiple parameters, put them into a struct.
## Generating the client: check for RPC errors

- Remember: remote procedure calls may fail!
  - add code to check return value
  - a remote procedure call returns a *pointer* to the result we want
  - if the pointer is null, then the call failed.

```c
long *lresult; /* return from bin_date_1 */
if ((lresult=bin_date_1(NULL, cl))==NULL) {
    clnt_perror(cl, server); /* failed! */
    exit(1);
}
```

- if *bin_date_1* succeeds, the result can be printed:

```c
printf("time on %s is %ld\n", server, *lresult);
```
Generating the client: check for RPC errors (2)

- Same for the call to `str_date`:

```c
char **sresult; /* return from str_date_1 */
if ((sresult=str_date_1(lresult, cl)) == NULL) {
    /* failed! */
    clnt_perror(cl, server);
    exit(1);
}
```

- if the call to `str_date_1` succeeds, then print the result:

```c
printf("date is \%s", *sresult);
```
Compile - link - run

- Generate stubs
  
rpcgen -C date.x

- Compile & link the client and client stub
  
c -o client client.c date_clnt.c -lnsl

- Compile & link the server and server stub
  
c c -o server -DRPC_SVC_FG server.c date_svc.c -lnsl
  
  Note: defining RPC_SVC_FG compiles the server such that it will run in the foreground instead of running as a background process

- Run the server (e.g. on remus)
  
  $ ./server

- Run the client
  
  $ ./client -h remus
  
time on localhost is 970457832
date is Sun Oct 1 23:37:12 2000