Programming Information, Hints, and Tips for the SRPC Assignment

Prof. Patrick G. Bridges

October 19, 2015

1 Introduction

The CS587 SRPC programming assignment is a non-trivial C programming assignment intended to help you understand what is required to implement remote procedure call with at-most-once semantics and to help you gain some experience with network programming. The specification given in the handout is relatively open-ended—while you are required to implement RPC with at-most-once semantics using UDP datagrams, retries, and timeouts, you are not told how you must do so in terms of the protocol itself. The amount of networking expertice required is minimal; the purpose of this document is to give you guidelines to help you implement the SRPC runtime successfully and sufficient information on how to send and receive UDP datagrams.

2 Analysis of Requirements

The first step toward implementing the RPC runtime is to read Birrell and Nelson's paper on implementing remote procedure calls. This paper is linked from the class reading list and should give you a rough idea of what you will have to implement. There are a number of important simplifications from what that paper describes to what you must implement, however:

- You do not need to implement a location service—`Srpc_Bind()` directly specifies the host and port to connect to to bind an RPC. Simply look up the appropriate port
- You do not need to handle multiple *concurrent* requests, and you may block other requests entirely (causing them to timeout) while handling a long-running RPC.
- You do not need to implement stub compilers—the client and server programs use a special calling convention into the runtime and have already packed arguments
- You do not need to use a heavily optimized protocol—while Birrell and Nelson went to great lengths using implicit acknowledgements to remove unnecessary packets in the common case of small and fast RPCs, you do not need to do so.

3 Tips

3.1 Use a single-threaded client and server

Debugging concurrent programs can be difficult. Since you do not need to handle concurrent requests and the client or server, you may use a single thread for processing all requests. This avoids many synchronization pitfalls.

You will still need a small amount of concurrency for one case - you should implement either probe packets or keepalive notification from the server to keep the client from timing out in long-running RPCs. If the server is blocked executing a long-running procedure, it will not be able to receive probe packets in a single-threaded server. As a result, it may be desirable to have the server send "I'm still computing" packets to the client while it executes a long-running procedure. This can be done by scheduling and handling the ALARM signal and calling sendto to the current client from the signal handler. `setitimer()` is the appropriate function call to use to start a periodic timer to send keepalive packets.

3.2 Separate the RPC and networking portions of your code

You life will be much easier if you separate the portion of your code responsible for packing and unpacking RPC requests into flat buffers from the code responsible for exchanging requests and replies with at-most-once semantics. This will allow you to implement marshalling and unmarshalling of data on a simple unreliable network layer and then to later replace that layer with one that provides at-most-once semantics.

3.3 Argument handling

Argument handling is another element of the project that is specified in a way that may be somewhat unclear or confusing to some of you. In particular, `Srpc_Arg` uses the "`void *`" type for passing generic values in C, limitations on the size of arguments (i.e. `SRPC_MAX_ARG_SIZE`) may be unclear to you, and you may not know how to pack and unpack arguments into buffers for transmission over the network.

Storing values in `Srpc_Arg` structures The "`void *`" type is frequently used to store generic values in C programs, and the `Srp_Arg` structure makes use of this practice. To store a 4-byte integer into the first element of an arg array and a string into the second element, the following code might be used:

```
Srpc_arg arglist[3];
char *string = "Here is a string";
int myint = 89, myint2;
arglist[0].type = SRPC_TYPE_INT;
arglist[0].size = 4;
arglist[0].val = (void *)myint; /* Stored by value */
arglist[1].type = SRPC_TYPE_DATA;
arglist[1].size = strlen(string) + 1; /* +1 for the NULL */
arglist[1].val = (void *)strdup(string); /* Stored by reference,
    copy freed by Srpc_FreeArgs */
arglist[2].type = SRPC_TYPE_NONE; /* terminate the list */
myint2 = (int)arglist[0].val;
```

So, integers are stored by value in the arg structure casting them to pointers, and then referenced by casting them back to integers. Strings and other "longer" data, on the other hand are stored by reference.

Sending `Srpc_Arg` arrays over the network To transmit an arg array over the network, you're going to have to convert it a flat buffer that can be unpacked by the receiving process. I did this by making the first 4 bytes of the packed buffer be the number of arguments packed into the buffer, and then successive data be of the form (type of data (4 bytes), length of data (4 bytes), data (as specified)). Note that there is a fixed limit on how large this can be. The total amount of data in

an arg array in SRPC is no more than `SRPC_MAX_ARG_SIZE` bytes, *including the information on types and length in the array.* This guarantees that your data will fit in a UDP packet, and also makes it easy to malloc space for incoming data. My code for packing an `Srpc_Arg` into a buffer looked something like this:

```
Srpc_Status
SrpcPackArgs(Srpc_Arg *pa, char *buf)
{
    /* Pack an arg array into a preallocated buffer */
    int nErr = SRPC_ERR_OK;
    char *p = buf;
    while (pa->type != SRPC_TYPE_NONE) {
        *(unsigned int *)p = htonl(pa->type);
        p += sizeof(unsigned int);
        *(unsigned int *)p = htonl(pa->size);
        p += sizeof(unsigned int);
        switch (pa->type) {
          case SRPC_TYPE_INT:
              switch (pa->size) {
                case 1:
                    *p = htonl((unsigned char)(unsigned int)pa->value);
                    break;
                case 2:
                    *(unsigned short *)p
                        = htons((unsigned short)(unsigned int)pa->value);
                    break;
                case 4:
                    *(unsigned int *)p = htonl((unsigned int)pa->value);
                    break;
                default:
                    nErr = SRPC_ERR_INVALID_ARG_TYPE;
                    goto error;
              }
      break;
          case SRPC_TYPE_DATA:
              memcpy(p, pa->value, pa->size);
              break;
          default:
              nErr = SRPC_ERR_INVALID_ARG_TYPE;
              goto error;
        }
        p += pa->size;
        pa++;
    }
    return SRPC_ERR_OK;
  error:
    return nErr;

}
```

4 UDP Programming

To communicate over UDP, you need to use *sockets*, endpoints for communication in UNIX. Sockets are created in UNIX using the `socket()` system call. To create a socket that uses UDP, you would make the following system call:

```
int sock;
sock = socket(AF_INET, SOCK_DGRAM, 0);
```

Once this is done, the socket needs to be bound to a local address. As described in class, the `bind()` system call does this:

```
struct sockaddr_in cli_addr;
cli_addr.sin_family = AF_INET;
cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
cli_addr.sin_port = htons(0); /* any local port - on the server, you
                                 would use the port specified in
                                 Srpc_ServerInit() instead of 0 */
bind(psns->nSock, (struct sockaddr *)&cli_addr, sizeof(cli_addr));
```

Once you have sockets set up on both the client and the server, communication between them is done using the `sendto()` and `recvfrom()` system calls. For example:

```
struct sockaddr_in dest_addr, src_addr;
struct hostent *ph;
char *buf = "Hello", bufIn[15];
ph = gethostbyname(hostname);

dest_addr.sin_family = AF_INET;
dest_addr.sin_addr = **(struct in_addr **)ph->h_addr_list;
dest_addr.sin_port = htons(server_port);
sendto(sock, buf, strlen(buf) + 1, &dest_addr, sizeof(dest_addr);
recvfrom(sock, buf, 15, &src_addr, NULL);
```