# Tutorials

In this chapter, we will go through some sample powerRPC programs.

# A file server

In this tutorial, we show you how to write a file server, and then write a client that understands some simple commands similar to those in FTP, such as GET, PUT, LS and CD. Since you are completely relieved from the task of writing networking code by using powerRPC, this assignment become rather trivial - once you get familiar with how RPC works.

## Design of interface

Our file server should provide a set of RPC functions that allow a client to access files located at the server machine. The server would open the file on client's behalf, and subsequently, read from or write to the file when such operations are requested from the client.

It will be convenient to make our RPC functions look like existing file I/O functions. For example, we could make our RPC interface resembles the UNIX file I/O system calls, read(), write(), and chdir(), etc. For our demo, let's mimic the C stdio library functions, such as fread() and fwrite(). To distinguish our RPC from the C library functions, we prefix our functions with a lower case 'r', so our RPCes will be rfread(), rfwrite(), etc , we will define a type called rFILE corresponding to the FILE type. So the rfread() function is of the following prototype,

```
int rfread(void * buf, int size, int nmemb, rFILE* stream);
```

of course, we should also have a rfopen() function to open a remote file.

Assuming an RPC connection has been established, a client could read from a file "foo" sitting on the server's machine like this,

```
rFILE * fp;
char    buf[1024];
fp = rfopen("foo", "r");
rfread(buf, 1, 1024, fp);
```

Having decide the interface functions, we need to decide the transport protocol. UDP is unreliable and the size of datagrams are usually limited to 8K, so let's choose TCP.

Another decision we need to make is the statefulness of the server. A stateful server maintains the state for the client, whereas in a stateless server, the client supplies all the information (such as current file offset) at every RPC. In our demo, we choose to use a stateful implementation.

The stateful server works as follows. The server maintains a table of opened files, when the client makes a rfopen() call, the server uses fopen() to open the file and record the FILE* pointer in the table, the index to that table entry is return to the client. The client then use the index to reference the file when it makes rfread(), rfwrite() and rfclose() calls later. Obviously, This index must be a field of our rFILE structure.

To implement our client program (simple FTP), we also need an RPC to list the contents of the remote directory. Thus we have the rlistdir() function, which returns a linked list of directory entries in its argument.

Combining these ideas we come up with the following interface declaration,

```
typedef  char [size=strlen(*this)+1, 1024]  str1024;
typedef  char  c_arr1024 [size=strlen(*this)+1, 1024];
typedef  unsigned long size_T;

typedef struct fHandle {
       int fd; // identify the file on server
}rFILE;

typedef struct dentry {
                c_arr1024      name;
                struct dentry *next;
}DENTRY;

interface rfile {
     property TRANSPORT_PROTOCOL= tcp;
     property INIT_BEFORE_REGISTER= init_fdtable;

     rFILE* rfopen(in const str1024 filename,
                   in const str1024 mode)
            {
                property     TIMEOUT_VALUE = 2;
            };

     int    rfread
            (out void [maxsize=size*nm, size=return>0?return:0] ptr,
             size_T size, size_T nm, in rFILE* stream
             );
     int    rfwrite
            (in const void [maxsize=size*nm, size=nm*size] ptr,
             size_T size, size_T nm, in rFILE* stream);
     int    rfclose(rFILE*stream) ;
     int    rlistdir(in str1024 path, out DENTRY * pent);
     int    rchdir(in str1024 path) ;
} 0x5555 ;
```

Note we defined the property *INIT_BEFORE_REGISTER*, which is the function to initialize the table of FILE* pointers to 0 on the server.

# Server implementation

You need to write the server implementation of the rfopen(), rfread(), etc. The rfopen() functions merely fopen()s the file, and record the FILE* pointer in the global *fd_table* and return the index to the client.

The *rfread()* function is listed below,

```
int   rfread(void *ptr, int size, int nm, rFILE * stream)
{
    FILE           *fp;
    int             index = stream->fd;

    /* first check if the index is valid,
       if true, get the FILE pointer */
    if (index < 0 || index >= MAXFILE
        || (fp = fd_table[index]) == 0)
    {
        fprintf(stderr, "Invalid rFILE pointer!\n");
        return -1;
    }
    return fread(ptr, size, nm, fp);
}
```

That is it! The rfwrirte() function is defined by replacing the word read in the above with write.

The rchdir() function is even simpler.

```
int   rchdir(char *path)
{
    return chdir(path);
}
```

The rlistdir() function is a bit more complicated, but probaly you can make it simpler by writing a more elegant linked list.

# The client

Now the server code is complete. When it is compiled and executed, it makes the six RPC functions to be callable from anywhere in a network. A programmer can make use use of these functions and write whatever applications he/she wants. Given the available RPC functions, we can easily write a file transfer client program which supports GET, PUT, LS and CD commands.

The get_file() function, which reads a remote file *src* and saves it in file *local*, is listed below,

```
int   get_file(char *src, char *local)
{
    rFILE          *fp = 0;
    FILE           *localfp;
    char            buf[1024];
    int             cnt;

    localfp = fopen(local, "w");
    if (!localfp) {
        perror(local);
        return -1;
    }
    fp = rfopen(src, "r");

    if (fp == 0) {
        fprintf(stderr, "Fail to open remote file!\n");
        fclose(localfp);
```

```
        return -1;
    }
    while ((cnt = rfread(buf, 1, 1024, fp)) > 0) {
        fwrite(buf, 1, cnt, localfp);
    }

    fclose(localfp);
    rfclose(fp);
    free(fp);
    return 0;
}
```

The code above is almost exactly what one would do to copy one local file to another using the fread(), fwrite() functions. The only difference is in the *free(fp)* call. PowerRPC always allocates the memory for a return value of reference type, it is the caller's responsibility to free that memory.

# A talk program

In this example, we will write a talk program that allow two users to send messages to each other.

## Design of the interface

A true talk program must have some daemon to notify a user that someone else is trying to initiate talk with him/her. We can certainly write such as server using powerRPC, however, this is not the purpose of this demo, since you can already write such a server after learning the powerRPC from the material above.

Our talk system will be just one program named talk2. One user starts talk2 first, and another user executes the same program to communicate with the first user, knowing he/she is there waiting. Talk2 must be both a server and a client of the same RPC interface, when sending message to the peer, it is a client, when receiving message, the role is reversed, and it becomes the server.

What we are going to do requires more from both powerRPC and the programmer.

The talk RPC interface contains a single function: *send_msg()*.

```
interface talk2 {

  property   TRANSPORT_PROTOCOL = udp;
  property   GEN_MAIN_FUNC = false;
  property   SERV_CALL_PREFIX = s_;

  void  send_msg(
        unsigned long sender_program_no,
        char [size = strlen(sender_host) + 1, 1024] sender_host,
        char [size = strlen(sender_name) + 1, 1024] sender_name,
        char [size = strlen(msg) + 1, 4096] msg
  ) = 1;

} = 0x9999;
```

The *send_msg()* RPC takes four arguments, the first three identifies the sender, the last one is the message being sent. Since talk2 is both a server and a client, we must define the property *SERV_CALL_PREFIX*, so the server implementation of the *send_msg* is actually named *s_send_msg*. We also need to write our *main()* function, therefore the *GEN_MAIN_FUNC* property is set to false.

# Implementation

Our code for *s_send_msg()* is just a little more than a few *printf* s.

```
void   s_send_msg(u_long sno, char *host, char *sender, char *msg)
{
    printf("\n...............%s@%d@%s................\n%s",
           sender, sno, host, msg);
    printf("..................over.................\n");
    talk2_unbind(0);
    talk2_bind(host, sno, 0, 0);
}
```

Besides writing the message from the client onto the terminal, we also establish an RPC connection to the sender.

To receive messages from a peer, the Talk2 program needs to be an RPC server, to send messages it must read stdin and acts as an RPC client. This requires the Talk2 program to do I/O multiplexing, in our case, the Talk2 program must handle the input from both the network communication channel for RPCes and the terminal input. powerRPC accommodates this easily by providing a set of server library functions to set up I/O handlers for a particular file descriptor. Although we could use the *INIT_AFTER_REGISTER* property to insert all of the code, we take this chance to write the server *main()* function ourselves using the powerRPC generated code and libraries.

Thus we have the following code,

```
void    handle_stdin(int fd)
{
    char            msg[1024];
    int             cnt;
    cnt = read(fd, msg, 1023);
    if (cnt <= 0)
        exit(1);

    msg[cnt] = '\0';
    send_msg(myprog, myhost, myname, msg);
}


int     main(int argc, char **argv)
{
    fd_set          fds;
    SVCXPRT        *mytxp;
    int             serv_sock;
    int             dtsz;
    char            msg[4096];

    if (argc < 2) {
        printf("Usage: %s -n talker -p program# [-c peer -P perr_prog#]\n",
               argv[0]);
        exit(0);
```

```
    }
    getoption(argc, argv);

    gethostname(myhost, 1023);

    if (strlen(peer_host)) {
        if (!talk2_bind(peer_host, peer_no, 0, 0)) {
            printf("Can not find peer to talk.\n");
            exit(1);
        }
    }
    pw_serv_init();

    if (!talk2_1_reg(RPC_ANYSOCK, myprog, 1, IPPROTO_UDP))
        exit(1);

    myprog = myprog == 0 ? TALK : myprog;

    printf("My program number is %d\n", myprog);
    signal(SIGINT, unset_myprog);

    pw_serv_input_handler(0, handle_stdin);

    pw_serv_mainloop(0, 0, 0, 0);
}
```

As you can the see, the *handle_stdin()* function simply reads something from stdin and call the *send_msg()* RPC to deliver it to a connected peer.

The main server function is more interesting. First, it reads some options. To allow two talk2 program to sit on the same machine, we let them to use whatever program number the user gives (if none is given at the command line option then the one defined in the IDL will be used). The first instance of Talk2 has no one to talk with, however, a subsequent Talk2 can talk to it by supplying the peer hostname and program number, through the ‘‘-c" and ‘‘-P" options respectively. So when the *peer_host* variable is set, we try to make connection to another Talk2. Then we initialize the server code by calling *pw_serv_init()* function. Then we register our server by calling the generated *talk2_1_reg()* function, with the program number defined in *myprog* variable. We then set up the input handler for stdin. Finally, Talk2 enters its mainloop by calling *pw_serv_mainloop()* with default arguments.

# Usage

The first Talk2 program would be started like this

```
 % talk2 -n Mike -p 1234
```

It will announce ‘‘My program number is 1234".

Suppose the first one is on machine **host1**, we can talk with it by

```
%talk2 -n Dave -p 2345 -P 1234 -c host1
```

Now both Mike and Dave can type in messages on their terminal and talk to each other.

Our 100 line Talk2 is not intended to be a replacement for the existing talk program. But you can make it comparable in functionality with talk by writing more code. How good it can be is only limited by your C/C++ skills, not by your knowledge of networking or things like that.

# Asynchronous RPC

Sometimes, we want an RPC to return immediately, and let the server return the results to the client asynchronously. For example, if it takes the *quote* server a long time to query a database to get the data, for reasons such as the large number of requests, we should let the client to continue to do other things and get the result later when it is ready.

With powerRPC this can be achieved easily. By setting the *NON_BLOCKING* property to *true*, an RPC call would return before the server function gets called. To receive the result later, the client can register a collection service, which is an RPC interface. When the server get the result, it calls upon this RPC registered by the client to send the result back.

In this example, we demonstrate how to make our *quote* RPC asynchronous. We have two RPC interfaces defined in *back.idl* and *over.idl*. The client main function looks like this,

```
#include "over.h"
#include "back.h"
#include <signal.h>
#include <rpc/pmap_clnt.h>

main(int argc, char **argv)
{
    char            myhost[1024];
    if (argc < 3) {
        printf("usage: %s hostname ticker\n", argv[0]);
        exit(1);
    }
    pw_serv_init();

    back_1_unmap(0, 0);


    if (!back_1_reg(RPC_ANYSOCK, BACK, BACK_1, IPPROTO_TCP)) {
        printf("fail registering \n");
        exit(0);
    }
    pw_serv_async_enable();

    if (!over_bind(argv[1], 0, 0, 0)) {
        printf("fail connect to OVER server.\n");
        exit(0);
    }
    gethostname(myhost, 1023);
    getQuote(myhost, argv[2]);
    printf("Returned from 1st getQuote... now do something else ..\n");
    getQuote(myhost, argv[2]);
    printf("Returned from 2st getQuote... now do something else ..\n");
    getQuote(myhost, argv[2]);
    printf("Returned from 3rd getQuote... now do something else ..\n");

    while (1) {
        printf("happily doing ....\n");
```

```
        sleep(10);
    }
}
```

In the above, the client set up an asynchronous RPC *back*. Then it makes two calls of *getQuote()*, both return immediately. This *getQuote()* call is different from the one we studied earlier: it has an additional argument *myhost*. When the server receives the call, it gets the quote, and calls back *myhost* to send the results. The client enters a **while** loop, pretending to do other work, when the server's callback arrives, the client will be interrupted, and the *returnQuote()* implementation is called. The asynchronous behavior of the client's RPC is enabled by the *pw_serv_async_enable()* powerRPC library call {.

Below is the server's code.

```
#include "back.h"
#include <stdlib.h>
/*
This is an asynchronous call, the client sends over its hostname,
after receiving the request, the server replies back to the client
immediately.  Then client and server reverse their roles, the
server make the RPC returnQuote() to send the result to the client,
who is supposed to acting as a BACK server waiting for the result.
*/

void   getQuote(char *caller_host, char Ticker[8])
{

    stkQuote        quote;

    printf("Called by %s\n", caller_host);
    printf("sleep for a while ....\n");

    sleep(6);

    if (!back_bind(caller_host, 0, 0, 0)) {
        printf("client is not there!\n");
        return;
    }
    strcpy(quote.Ticker, Ticker);

    /* find the quotes */
    quote.Low = rand() % 100;
    quote.High = rand() % 100;
    quote.Close = rand() % 100;
    printf("now sending back quote ....\n");
    printf("%s %f %f %f\n", quote.Ticker,
           quote.Low, quote.High, quote.Close);

    returnQuote(&quote);

    back_unbind(0);
}
```

# Other sample programs

The powerRPC distribution conatins other sample programs for demonstration purposes.