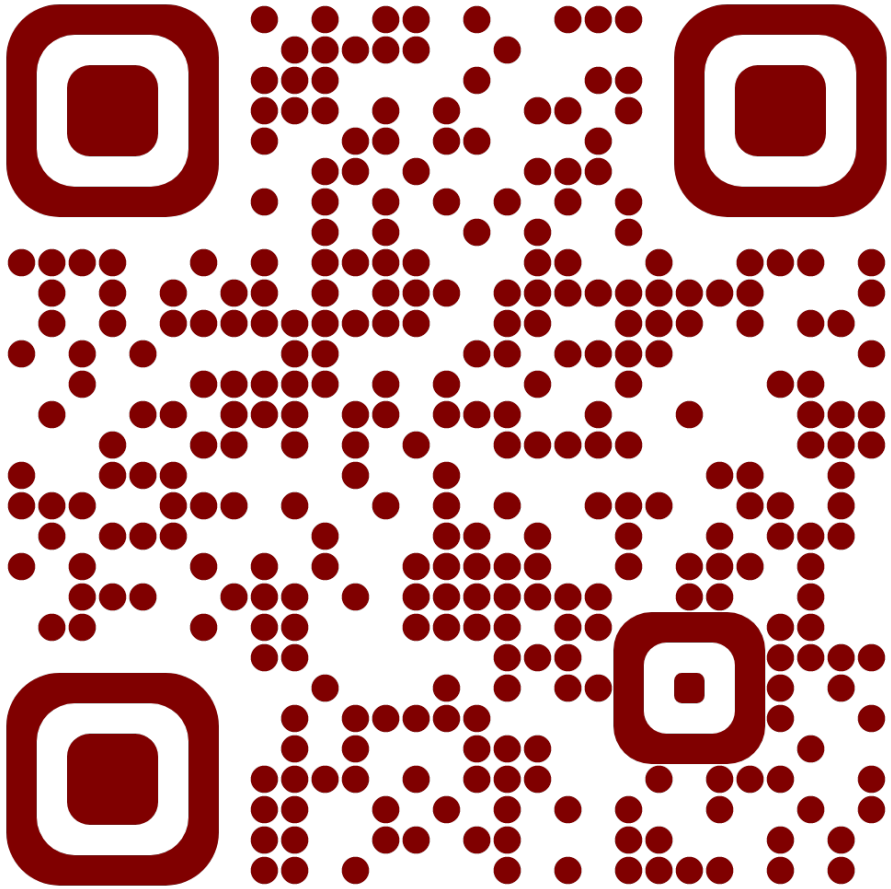
Socket Programming

Rathindra Nath Dutta

Senior Research Fellow
Advanced Computing & Microelectronics Unit
Indian Statistical Institute, Kolkata

November 4, 2022

https://ratcoinc.github.io/Networks/



WEB PAGE

Remote Procedure Call

- Commonly known as **RPC**
- A mechanism to invoke a function call on a remote host with local parameters, and get back the computed result

Remote Procedure Call

- Commonly known as **RPC**
- A mechanism to invoke a function call on a remote host with local parameters, and get back the computed result

- Extension of conventional/local procedure call
- The called procedure need not exist in the same address space as the calling Procedure
- Two processes may be on the same host, or on different hosts connected in the same network

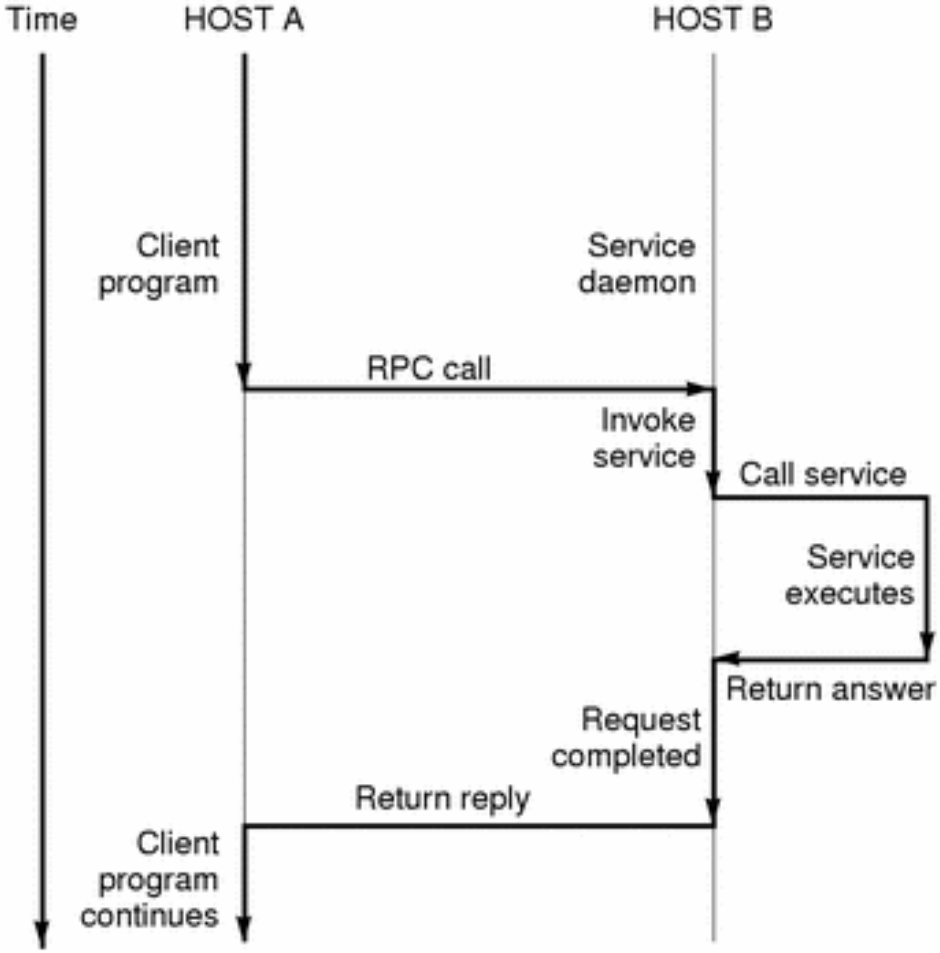
Remote Procedure Call

- Commonly known as **RPC**
- A mechanism to invoke a function call on a remote host with local parameters, and get back the computed result

- Extension of conventional/local procedure call
- The called procedure need not exist in the same address space as the calling Procedure
- Two processes may be on the same host, or on different hosts connected in the same network

- Primarily used for distributed client server based applications

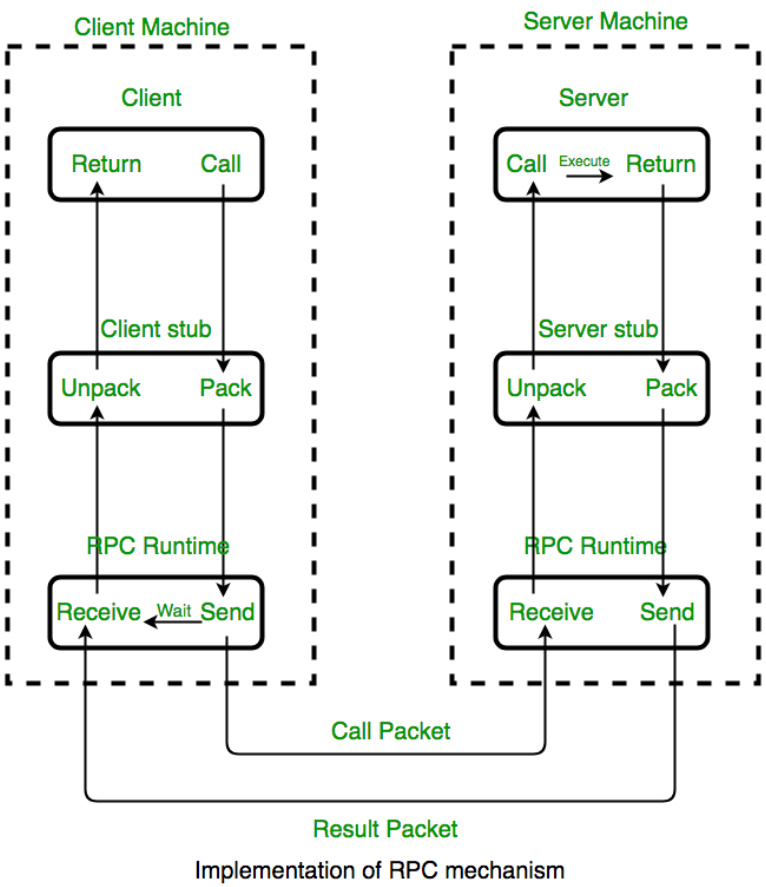
How RPC Works



- Server runs a listener daemon service
- Upon receiving an RPC request from client, server executes the procedure and returns the result
- From invoking an RPC call, until the reply returns, the client process is blocked

[1] image src:https://docs.oracle.com/cd/E19455-01/805-7224/images/5865.epsi.gif

RPC Application Development

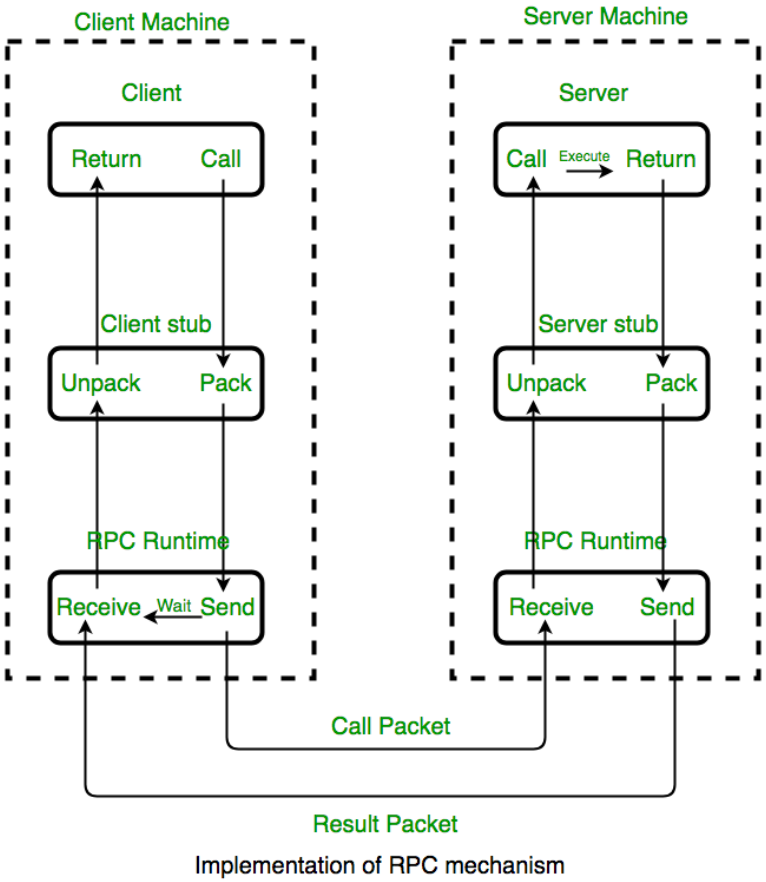


Implementation of RPC mechanism

- Client calls a local (stub) version of the remote procedure
- It then packs the arguments etc. for a network communication
- The RPC runtime routines does the actual network communication
- The server stub then unpacks the procedure details, arguments etc. and invokes the actual procedure

[1] image src:https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/

RPC Application Development



Implementation of RPC mechanism

- The computed result is returned to the client in similar fashion
- This packing/unpacking business is formally known as **Marshall/Unmarshall** - deals with serialization of data, byte ordering etc.

[1] image src:https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/

RPC Application Development

It involves three main steps:

- Specify the protocol - write stubs, RPC runtimes etc
- Write the server program
- Write the client program

RPC Application Development

It involves three main steps:

- Specify the protocol - write stubs, RPC runtimes etc
- Write the server program
- Write the client program
- Fortunately there is `rpcgen` compiler to rescue us

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs
- It uses its own language, very similar to C preprocessor directives

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs
- It uses its own language, very similar to C preprocessor directives
- In the protocol definition, specify the name of the service procedures, data types of parameters and return arguments along with unique version and ID numbers

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs
- It uses its own language, very similar to C preprocessor directives
- In the protocol definition, specify the name of the service procedures, data types of parameters and return arguments along with unique version and ID numbers
- The definition is written in a special file with a `.x` extension

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs
- It uses its own language, very similar to C preprocessor directives
- In the protocol definition, specify the name of the service procedures, data types of parameters and return arguments along with unique version and ID numbers
- The definition is written in a special file with a `.x` extension
- Invoke the compiler as: `rpcgen rpcprogdef.x`

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs
- It uses its own language, very similar to C preprocessor directives
- In the protocol definition, specify the name of the service procedures, data types of parameters and return arguments along with unique version and ID numbers
- The definition is written in a special file with a `.x` extension
- Invoke the compiler as: `rpcgen rpcprogdef.x`
- It will generate four files:
  - `rpcprogdef_clnt.c` – the client stub
  - `rpcprogdef_svc.c` – the server stub
  - `rpcprogdef.h` – header file of definitions, common to server & client
  - `rpcprogdef_xdr.c` – XDR routines that translate each data type defined in the header file (if required)

Using `rpcgen` Compiler

- A standalone executable program that reads a protocol definition and automatically generates client and server stubs
- It uses its own language, very similar to C preprocessor directives
- In the protocol definition, specify the name of the service procedures, data types of parameters and return arguments along with unique version and ID numbers
- The definition is written in a special file with a `.x` extension
- Invoke the compiler as: `rpcgen rpcprogdef.x`
- It will generate four files:
  - `rpcprogdef_clnt.c` – the client stub
  - `rpcprogdef_svc.c` – the server stub
  - `rpcprogdef.h` – header file of definitions, common to server & client
  - `rpcprogdef_xdr.c` – XDR routines that translate each data type defined in the header file (if required)
- The external data representation (XDR) provides the abstraction needed for machine independent communication

Example of rpcgen

contents of `calc.x`:

```
struct intpair {
  int a;
  int b;
};

program CALC_PROG {
  version CALC_VERS {
    int ADD(intpair) = 1;
    int SUB(intpair) = 2;
  } = 1;
} = 0x23456789;
```

- The procedures are allowed to have only a single argument[1]
- Use a wrapper for multiple arguments

[1] The *newstyle* of `rpcgen` allows procedures to have multiple arguments; use `-N` option: `https://linux.die.net/man/1/rpcgen`

Example of rpcgen

contents of `calc.x`:

```
struct intpair {
  int a;
  int b;
};

program CALC_PROG {
  version CALC_VERS {
    int ADD(intpair) = 1;
    int SUB(intpair) = 2;
  } = 1;
} = 0x23456789;
```

- The procedures are allowed to have only a single argument[1]
- Use a wrapper for multiple arguments
- A remote procedure is uniquely identified by the triple:
  (program no., version no., procedure no.)

[1] The *newstyle* of `rpcgen` allows procedures to have multiple arguments; use `-N` option:
`https://linux.die.net/man/1/rpcgen`

Example of rpcgen

contents of `calc.x`:

```
struct intpair {
  int a;
  int b;
};

program CALC_PROG {
  version CALC_VERS {
    int ADD(intpair) = 1;
    int SUB(intpair) = 2;
  } = 1;
} = 0x23456789;
```

- The procedures are allowed to have only a single argument[1]
- Use a wrapper for multiple arguments
- A remote procedure is uniquely identified by the triple:
  (program no., version no., procedure no.)
- Program numbers are 32-bit numbers, written in hex, choose any number between `0x20000000` - `0x3FFFFFFF` used for unique assignment of IP ports

[1] The *newstyle* of `rpcgen` allows procedures to have multiple arguments; use `-N` option:
`https://linux.die.net/man/1/rpcgen`

Example of rpcgen

contents of `calc.x`:

```
struct intpair {
  int a;
  int b;
};

program CALC_PROG {
  version CALC_VERS {
    int ADD(intpair) = 1;
    int SUB(intpair) = 2;
  } = 1;
} = 0x23456789;
```

- The procedures are allowed to have only a single argument[1]
- Use a wrapper for multiple arguments
- A remote procedure is uniquely identified by the triple: (program no., version no., procedure no.)
- Program numbers are 32-bit numbers, written in hex, choose any number between `0x20000000` - `0x3FFFFFFF` used for unique assignment of IP ports
- Version number and procedure number are integers, starting from 1
- Program and procedure names are declared with all capital letters

[1] The *newstyle* of `rpcgen` allows procedures to have multiple arguments; use `-N` option: `https://linux.die.net/man/1/rpcgen`

Experiment

use `rpcgen` compile the `calc.x` file

```
rpcgen calc.x
```

inspect the generated files

Defining the RPC Server and Client

The service side will have to register the procedures that may be called by the client and receive and return any data required for processing

The client application call the remote procedure pass any required data and will receive the returned data

[1] The `-a` option generates all the files including sample code for client and server side and also a make file: `https://linux.die.net/man/1/rpcgen`

Defining the RPC Server and Client

The service side will have to register the procedures that may be called by the client and receive and return any data required for processing

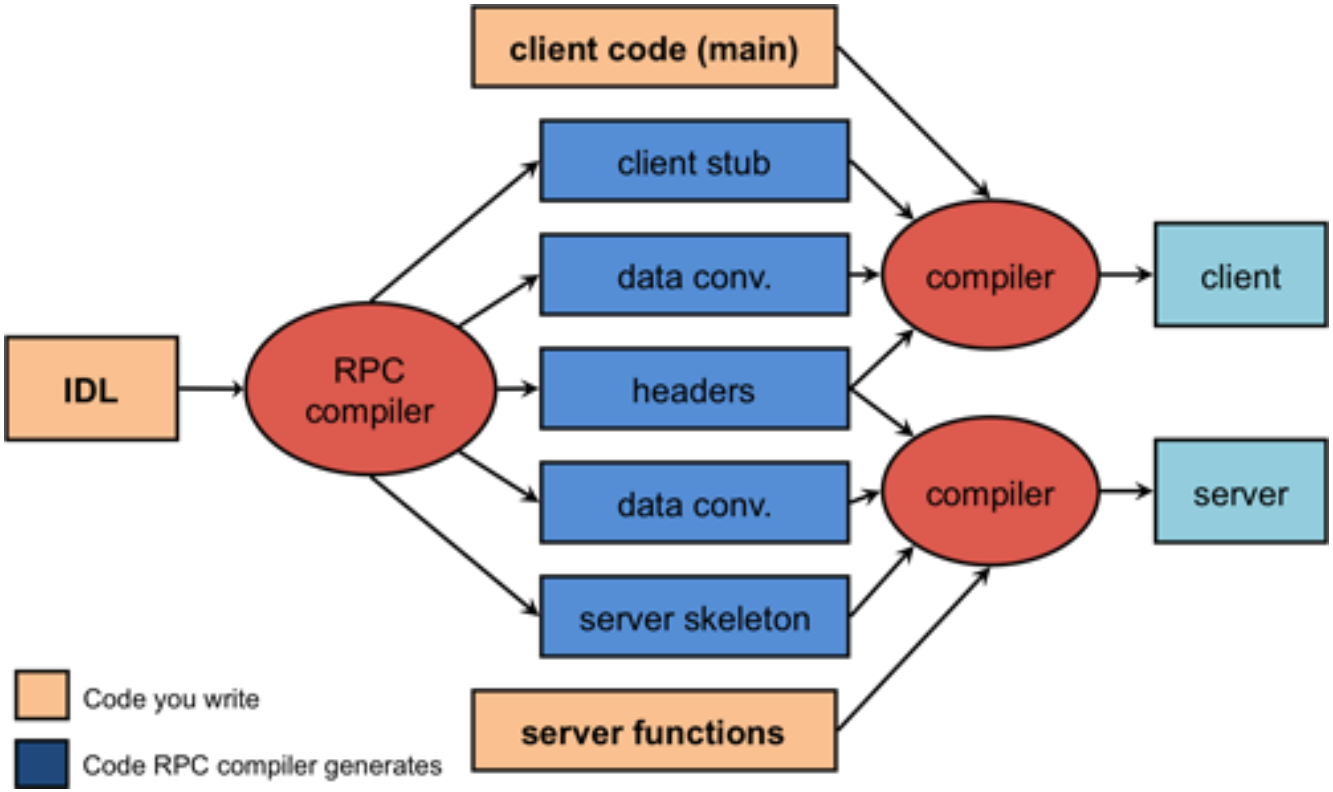
The client application call the remote procedure pass any required data and will receive the returned data

to get a template for client and server, run:
`rpcgen -a calc.x`

inspect the new files

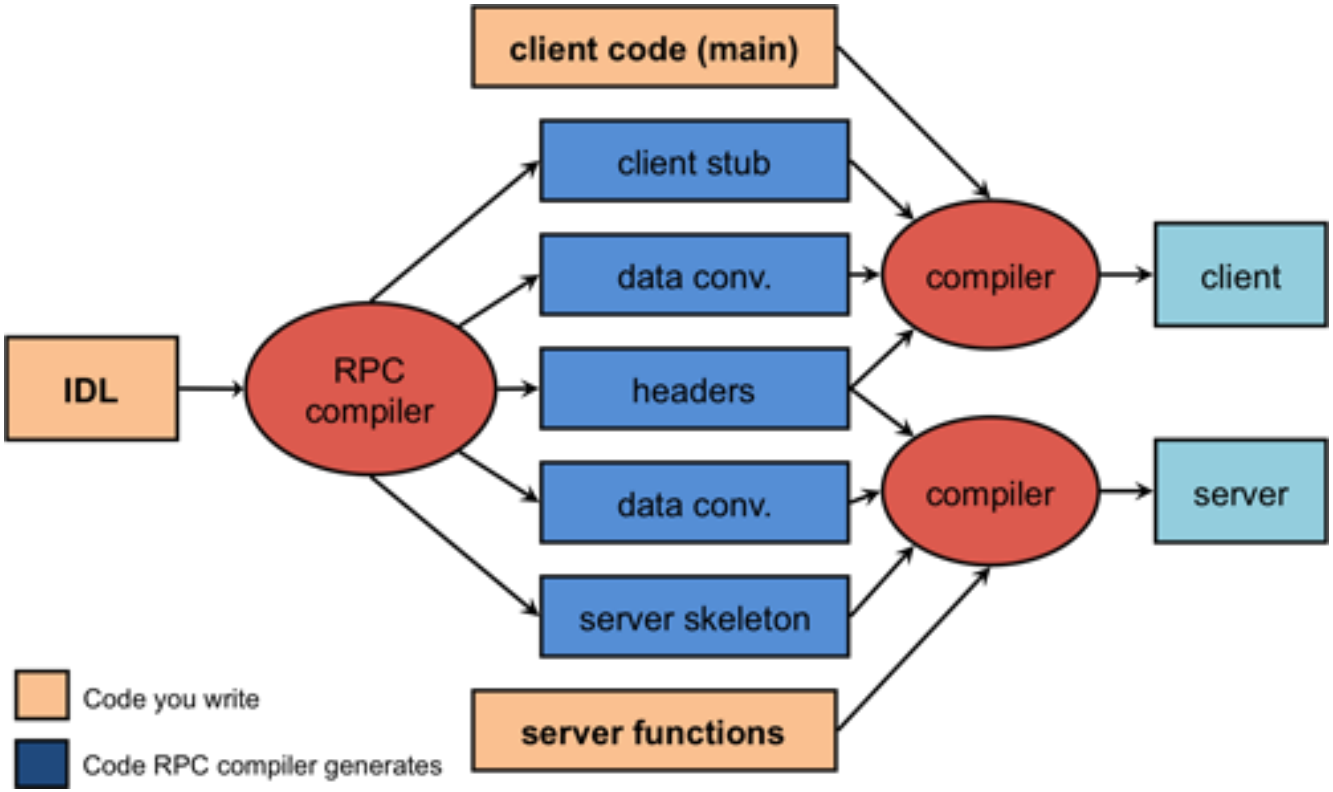
[1] The `-a` option generates all the files including sample code for client and server side and also a make file: `https://linux.die.net/man/1/rpcgen`

Compiling the RPC Server and Client



[1] image src: https://people.cs.rutgers.edu/~pxk/417/notes/rpc.html

Compiling the RPC Server and Client



Fortunately, the `-a` option of `rpcgen` also generates a *makefile*

[1] image src: `https://people.cs.rutgers.edu/~pxk/417/notes/rpc.html`

Compiling the RPC Server and Client

Things to keep in mind:

- Glibc's RPC support was deprecated and has been removed in newer version of UNIX/Linux
- There is replacement implementations based on TI-RPC, which additionally support IPv6

  can be installed via: `sudo apt install libtirpc*`
- Modify the generated makefile to add the following two lines:

```
CFLAGS += -DRPC_SVC_FG
CFLAGS += -I/usr/include/tirpc
LDLIBS += -ltirpc
```

- `rpcbind` is required to register an RPC service can be installed via:

  `sudo apt install rpcbind`
- Use `rpcinfo` to see running services

[1] `DRPC_SVC_FG` will cause our server to run in the foreground, for testing purposes; this is convenient since we'll be less likely to forget about it, and it will be easier to kill (no need to look up its process ID)

Compiling the RPC Server and Client

Things to keep in mind:

- Glibc's RPC support was deprecated and has been removed in newer version of UNIX/Linux
- There is replacement implementations based on TI-RPC, which additionally support IPv6

  can be installed via: `sudo apt install libtirpc*`
- Modify the generated makefile to add the following two lines:

```
CFLAGS += -DRPC_SVC_FG
CFLAGS += -I/usr/include/tirpc
LDLIBS += -ltirpc
```

- `rpcbind` is required to register an RPC service can be installed via:

  `sudo apt install rpcbind`
- Use `rpcinfo` to see running services

[1] `DRPC_SVC_FG` will cause our server to run in the foreground, for testing purposes; this is convenient since we'll be less likely to forget about it, and it will be easier to kill (no need to look up its process ID)

Compiling and Running the RPC Server and Client

- Edit the `calc_server.c` file to modify the definitions of our functions simply write a print statements like:
  `printf("add function called\n ");`

- Run the makefile to build both server and client
  `make -f Makefile.calc`

- If the `make` utility is not already installed:
  `sudo apt install make`
  or run: `sudo apt install build-essential`

- Run the server and client in two different terminals
  `./calc_server`
  `./calc_client 127.0.0.1`

Writing Actual Codes

- In `calc_client.c` file look for the line:
  ```
  result_1 = add_1(&add_1_arg, clnt);
  ```
- Load our `add_1_arg intpair` with values before the `add_1()` call:
  ```
  add_1_arg.a = 123;
  add_1_arg.b = 456;
  ```
- Write an `else` part of the following `if`:
  ```
  if (result_1 == (int *) NULL) {
      clnt_perror (clnt, "call failed");
  } else {
      printf("result = %d\n", *result_1);
  }
  ```

- In `calc_server.c` file replace our simple `printf` statement with:
  ```
  result = argp->a + argp->b;
  printf("returning: %d\n", result);
  ```
- Rebuild (make) and run the server and the client

The Final Codes

The protocol definition file: `calc.x`

Generate necessary files with `rpcgen -a calc.x`

The modified files: `calc_server.c` and `calc_client.c`

The modified makefile (if required): `Makefile.calc`

Only the `add()` part is done; `sub()` is left as an exercise

Sending an Array over RPC

- Define a structure containing a static[1] array (possibly larger size), and an integer for actual element count

- Save the following as `arr.x`

```
struct intarr {
  int arr[100];
  int n;
};
program SUM_PROG {
  version SUM_VERS {
    int ADD(intarr) = 1;
  } = 1;
} = 0x23456789;
```

- Do `rpcgen -a arr.x`

[1] sending dynamic array: `https://stackoverflow.com/questions/27460456/how-do-i-send-an-array`

Sending an Array over RPC

- In `sum_prog_1()` of `arr_client.c` initialize the `intarr` members before the RPC call and print the returned value after it

```
add_1_arg.n = 4;
add_1_arg.arr[0] = 10;
add_1_arg.arr[1] = 11;
add_1_arg.arr[2] = 32;
add_1_arg.arr[3] = 44;

result_1 = add_1(&add_1_arg, clnt);
if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
} else {
    printf("result = %d\n", *result_1);
}
```

Sending an Array over RPC

- In `arr_server.c` write the following as the body of `add_1_svc()`

```
static int result;
result = 0;
for(int i=0; i<argp->n; i++) {
    result += argp->arr[i];
}
return &result;
```

- Build (make) and run the server and the client

The Final Codes

The protocol definition file: `arr.x`

Generate necessary files with `rpcgen -a arr.x`

The modified files: `arr_server.c` and `arr_client.c`

The modified makefile (if required): `Makefile.arr`