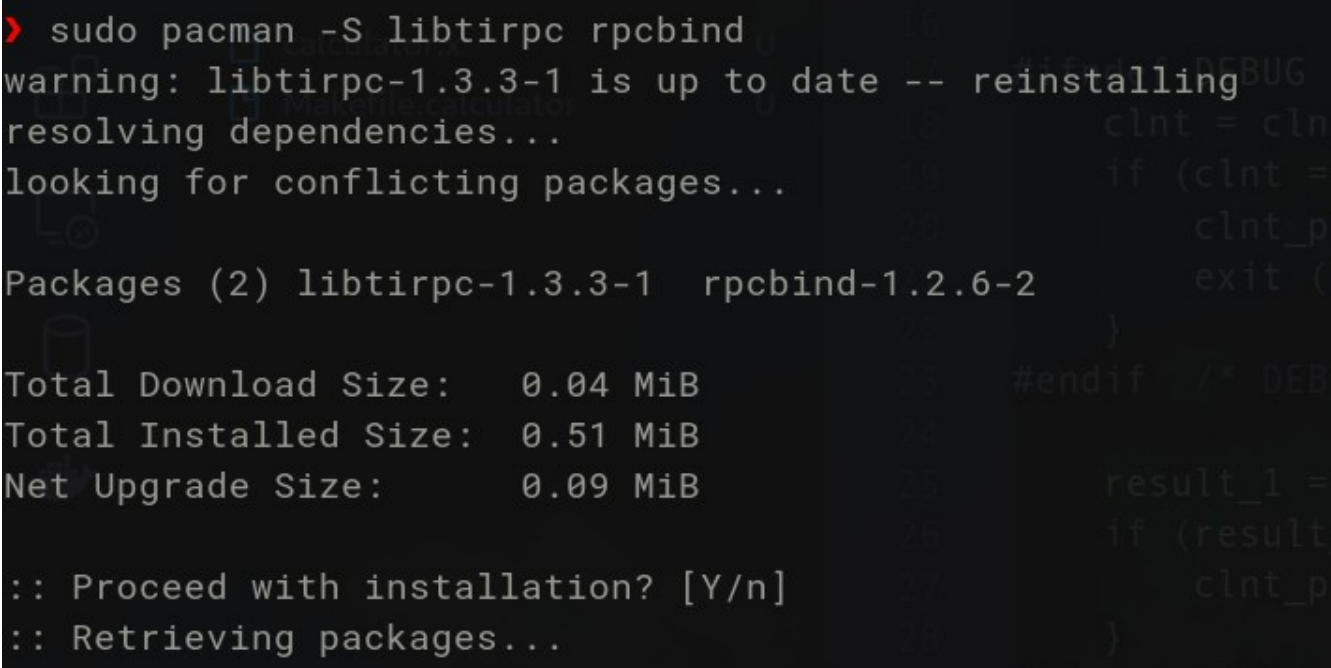


RPC в LINUX

Для работы с библиотекой RPC необходимо:

1. Установить необходимые пакеты. Для этого в дистрибутиве Arch Linux в командной строке необходимо ввести команду `sudo pacman -S libtirpc rpcbind`. Данная команда устанавливает необходимые для работы с RPC библиотеки и самая главная из них это – `tirpc` (первая строка рис. 1). В других дистрибутивах имена пакетов могут отличаться, найти их можно путем поиска в репозитории. Для этого необходимо найти информацию о данных пакетах для своего дистрибутива в поисковике.

rpcbind (Remote Procedure Call bind) – является механизмом, при котором порты интернет-адресов могут быть назначены программе, запущенной на удаленном компьютере, чтобы действовать так, как если бы она была запущена на локальном компьютере.



```
> sudo pacman -S libtirpc rpcbind
warning: libtirpc-1.3.3-1 is up to date -- reinstalling
resolving dependencies...
looking for conflicting packages...

Packages (2) libtirpc-1.3.3-1  rpcbind-1.2.6-2

Total Download Size:    0.04 MiB
Total Installed Size:  0.51 MiB
Net Upgrade Size:       0.09 MiB

:: Proceed with installation? [Y/n]
:: Retrieving packages...
```

рис 1.

2. Установить *rpcgen*. В дистрибутиве Arch Linux компилятор `rpcgen` находится в пакете `rpcsvc-proto`, а не пакете `rpcgen`, как написано в большинстве источников.

Ниже, на рис 2. показан скриншот, содержащий возможные команды установки данного пакета в различных дистрибутивах. Если установка не удалась по причине того, что в репозитории нет пакета с таким названием, необходимо выполнить поиск по репозиторию дистрибутива, установленного на компьютере.

Debian	<code>apt-get install libc-dev-bin</code>	libc-dev-bin
Ubuntu	<code>apt-get install libc-dev-bin</code>	GNU C Library: Development binaries This package contains utility programs related to the GNU C Library development package.
Alpine	<code>apk add rpcgen</code>	glibc-common
Arch Linux	<code>pacman -S rpcgen</code>	Common binaries and locale data for glibc
Kali Linux	<code>apt-get install libc-dev-bin</code>	rpcgen
CentOS	<code>yum install glibc-common</code>	Remote Procedure Call (RPC) protocol compiler
Fedora	<code>dnf install rpcgen</code>	glibc-arm-linux-gnu
Windows (WSL2)	<code>sudo apt-get update sudo apt-get install libc-dev-bin</code>	at arm-linux-gnu
OS X	<code>brew install rpcgen</code>	rpsecv-proto
Raspbian	<code>apt-get install libc-dev-bin</code>	rpsecv protocol definitions from glibc
Dockerfile	<code>dockerfile.run/rpcgen</code>	
Docker	<code>docker run cmd.cat/rpcgen rpcgen</code> powered by Commando	

рис 2.

в дистрибутиве Arch Linux (версия ядра 6.0.2) rpcgen устанавливается командой (первая строка рис 3.):

```
> sudo pacman -S rpsecv-proto
resolving dependencies...
looking for conflicting packages...

Packages (1) rpsecv-proto-1.4.3-1

Total Download Size:    0.06 MiB
Total Installed Size:  0.21 MiB

:: Proceed with installation? [Y/n]
:: Retrieving packages...
```

рис 3.

3. Создать файл *calculator.x* (листинг 1):

Листинг 1.

```
/*
 * filename: calculator.x
 * function: Define constants, non-standard data types and the calling
process in remote calls
 */

const ADD = 0;
const SUB = 1;
const MUL = 2;
const DIV = 3;

struct CALCULATOR
{
    int op;
    float arg1;
    float arg2;
    float result;
};

program CALCULATOR_PROG
{
    version CALCULATOR_VER
    {
        struct CALCULATOR CALCULATOR_PROC(struct CALCULATOR) = 1;
    } = 1; /* Version number = 1 */
} = 0x20000001; /* RPC program number */
```

4. Сгенерировать следующие файлы, используя компилятор *rpcgen*:

`rpcgen -a calculator.x`

Makefile.calculator: используется для компиляции кода клиента и кода сервера;

calculator.h: объявление используемых переменных и функций;

calculator_xdr.c: “кодировка” нестандартных типов данных;

calculator_clnt.c: прокси-сервер удаленного вызова вызывается локальной операционной системой, параметры вызова упакованы в сообщение. Далее сообщение посылается на сервер. Данный файл генерируется с помощью *rpcgen* и не нуждается в модификации;

calculator_svc.c: преобразует запрос, вводимый по сети, в вызов локальной процедуры. Файл ответственен за распаковку полученного сервером сообщения и вызов фактической реализации на стороне сервера и на уровне приложения. Не нуждается в модификации;

calculator_client.c: скелетон программы клиента, требуется модифицировать;

calculator_server.c: скелетон программы сервера, требуется модифицировать;

Исходные коды скелетонов:

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculator.h"
#include <stdio.h>

void
calculator_prog_1(char *host)
{
    CLIENT *clnt;
    struct CALCULATOR *result_1;
    struct CALCULATOR calculator_proc_1_arg;

#ifdef    DEBUG
    clnt = clnt_create (host, CALCULATOR_PROG, CALCULATOR_VER, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif    /* DEBUG */

    result_1 = calculator_proc_1(&calculator_proc_1_arg, clnt);
    if (result_1 == (struct CALCULATOR *) NULL) {
        clnt_perror (clnt, "call failed");
    }
#ifdef    DEBUG
    clnt_destroy (clnt);
#endif    /* DEBUG */
} // calculator_prog_1

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    calculator_prog_1 (host); // вызов
    exit (0);
}

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculator.h"
```

```

struct CALCULATOR *
calculator_proc_1_svc(struct CALCULATOR *argp, struct svc_req *rqstp)
{
    static struct CALCULATOR result;

    /*
     * insert server code here
     */

    return &result;
}

```

5. Модификация скелетонов клиента и сервера:

calculator_client.c:

Листинг 2.

```

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculator.h"
#include <stdio.h>

void
calculator_prog_1(char *host)
{
    CLIENT *clnt;
    struct CALCULATOR *result_1;
    struct CALCULATOR calculator_proc_1_arg;

#ifdef DEBUG
    clnt = clnt_create (host, CALCULATOR_PROG, CALCULATOR_VER, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    /* -<<< Add to test */
    char c;

    printf("choose the operation:\n\t0---ADD\n\t1---SUB\n\t2---MUL\n\t3---DIV\n");
    c = getchar();

    if (c > '3' || c < '0')
    {
        printf("error:operate\n");
        exit(1);
    }

    calculator_proc_1_arg.op = c-'0';

```

```

    printf("input the first number: ");
    scanf("%f", &calculator_proc_1_arg.arg1);

    printf("input the second number:");
    scanf("%f", &calculator_proc_1_arg.arg2);

    /* -<<< Add to test */

    result_1 = calculator_proc_1(&calculator_proc_1_arg, clnt);
    if (result_1 == (struct CALCULATOR *) NULL) {
        clnt_perror (clnt, "call failed");
    }
#ifdef      DEBUG
    clnt_destroy (clnt);
#endif      /* DEBUG */

    /* -<<< Add to test */
    printf("The Result is %.3f\n", result_1->result);
    /* -<<< Add to test */

} // calculator_prog_1

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    calculator_prog_1 (host); // вызов
    exit (0);
}

```

calculator_server.c

Листинг 3.

```

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calculator.h"

struct CALCULATOR *
calculator_proc_1_svc(struct CALCULATOR *argp, struct svc_req *rqstp)
{
    static struct CALCULATOR  result;

    /*
     * insert server code here
     */

    /* -<<< Add to test */

```

```

switch(argp->op)
{
    case ADD:
        result.result = argp->arg1 + argp->arg2;
        break;
    case SUB:
        result.result = argp->arg1 - argp->arg2;
        break;
    case MUL:
        result.result = argp->arg1 * argp->arg2;
        break;
    case DIV:
        result.result = argp->arg1 / argp->arg2;
        break;
    default:
        break;
}
/* -<<< Add to test */

return &result;
}

```

6. Выполнить сборку проекта с помощью Makefile.calculator, для этого необходимо в командной строке выполнить следующую команду: `make -f Makefile.calculator`. Для корректной сборки необходимо предварительно изменить файл Makefile.calculator, добавив флаг компиляции **-I/usr/include/tirpc** и флаг линковки **-ltirpc**. Листинг данного файла сборки представлен ниже.

Листинг 4.

```

# This is a template Makefile generated by rpcgen

# Parameters

CLIENT = calculator_client
SERVER = calculator_server

SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = calculator.x

TARGETS_SVC.c = calculator_svc.c calculator_server.c calculator_xdr.c
TARGETS_CLNT.c = calculator_clnt.c calculator_client.c calculator_xdr.c
TARGETS = calculator.h calculator_xdr.c calculator_clnt.c calculator_svc.c
calculator_client.c calculator_server.c

OBJECTS_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECTS_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
# Compiler flags

CFLAGS += -g -I/usr/include/tirpc
LDLIBS += -lnsl -ltirpc
RPCGENFLAGS =

# Targets

```

```

all : $(CLIENT) $(SERVER)

$(TARGETS) : $(SOURCES.x)
            rpcgen $(RPCGENFLAGS) $(SOURCES.x)

$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $(SOURCES_CLNT.h) $(TARGETS_CLNT.c)

$(OBJECTS_SVC) : $(SOURCES_SVC.c) $(SOURCES_SVC.h) $(TARGETS_SVC.c)

$(CLIENT) : $(OBJECTS_CLNT)
            $(LINK.c) -o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)

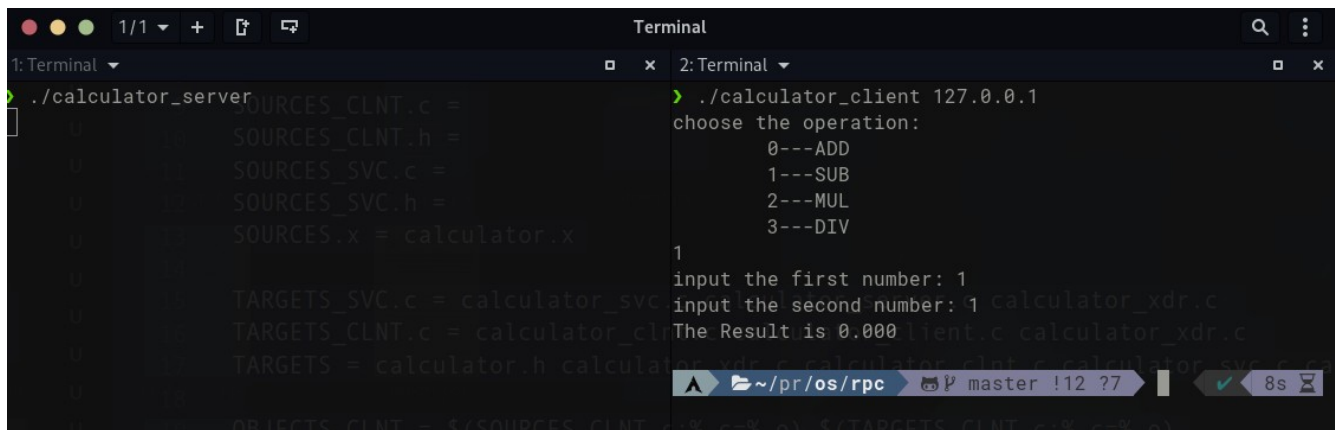
$(SERVER) : $(OBJECTS_SVC)
            $(LINK.c) -o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)

clean:
        $(RM) core $(TARGETS) $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)

```

7. Запустить сервер командой `./calculator_server`. Затем запустить клиент, указав в качестве параметра ip-адрес хоста (localhost в случае выполнения на ПК), выполнив команду:
`./calculator_client 127.0.0.1`

Пример работы программы представлен на рис. 4:



```

Terminal 1:
> ./calculator_server
SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = calculator.x
TARGETS_SVC.c = calculator_svc.c
TARGETS_CLNT.c = calculator_clnt.c
TARGETS = calculator.h calculator_xdr.c
OBJECTS_CLNT = $(SOURCES_CLNT).o $(SOURCES_SVC).o $(TARGETS_CLNT.o) $(TARGETS_SVC.o)

Terminal 2:
> ./calculator_client 127.0.0.1
choose the operation:
0---ADD
1---SUB
2---MUL
3---DIV
1
input the first number: 1
input the second number: 1
The Result is 0.000

```

рис. 5

***Замечание:** в случае если при запуске `./calculator_server` терминал печатает сообщение об ошибке вида “unable to register (TESTPROG, VERSION, udp).

необходимо перезапустить сервис `rpcbind` командой `systemctl restart rpcbind`

Заключение:

RPC являются средством взаимодействия параллельных процессов (IPC - Inter-process communication). Как известно, в самом общем случае различаются два вида взаимодействия параллельных процессов (IPC):

1. Вызов локальных процедур (LPC – local procedure call)

2. Вызов удаленных процедур (RPC – remote procedure call)

Вызов удаленных процедур сделан максимально подобным вызову локальных процедур. В ядре RPC не используют протокол, в чем и состоит принципиальное различие с сокетами. Действия фактически выполняются не по протоколу. Действия выполняются в сети путем связывания портов между собой.

Задание на лабораторную работу

Необходимо реализовать алгоритм Лампорта “Булочная”: процессы клиенты обращаются к серверу для получения номера. Сервер каждому приходящему клиенту выдает номер – максимальный из выданных + единица. Этот номер используется клиентом для получения нужной услуги от сервера: услуга предоставляется клиентам в порядке очереди.

Дополнительно:

Парадигма RPC делает удаленные вызовы расширением знакомого механизма вызова локальной процедуры. В частности, сам вызов выполняется как вызов локальной процедуры, а базовый механизм RPC прозрачно обрабатывает удаленность. Таким образом, программирование интерфейса сервера аналогично программированию вызова локальной процедуры, за исключением того, что обработчик вызова выполняется в отдельном адресном пространстве и домене безопасности.

С этой точки зрения вызов локальной процедуры является частным простым случаем более общего механизма вызова, предоставляемого RPC. Семантика RPC расширяет семантику вызова локальной процедуры различными способами:

Надежность (Reliability)

Сетевые транспортные средства могут обеспечивать различную степень надежности. Система времени выполнения RPC прозрачно обрабатывает эту транспортную семантику, но спецификации вызовов RPC включают спецификацию семантики выполнения, которая указывает протоколам RPC требуемые гарантии успеха и допустимость многократного выполнения на возможно ненадежном транспорте. Код серверного приложения должен соответствовать указанной семантике выполнения.

Связующий (Binding)

Привязка RPC происходит во время выполнения и находится под управлением программы. Использование механизма привязки RPC клиентом и сервером подробно обсуждается в этой главе.

Нет общей памяти (No Shared Memory)

Поскольку вызывающая и не вызываемые процедуры не используют одно и то же адресное пространство, удаленные вызовы процедур с параметрами ввода/вывода используют семантику ввода-вывода. По той же причине RPC не имеет понятия о "глобальных структурах данных", совместно используемых вызывающей стороной и вызываемым объектом; данные должны передаваться через параметры вызова.

Режимы сбоя (Failure Modes)

Ряд возможных сбоев возникает, когда вызывающий абонент и вызываемый объект находятся на физически разных компьютерах. К ним относятся сбои удаленной системы или сервера, сбои связи, проблемы безопасности и несовместимости протоколов. RPC включает механизм возврата таких удаленных ошибок вызывающему абоненту.

Отменяет (Cancels)

RPC расширяет механизм локальной отмены, перенаправляя отмены, возникающие во время RPC, на сервер, обрабатывающий вызов, позволяя коду серверного приложения обрабатывать отмену. RPC добавляет механизм тайм-аута отмены, гарантирующий, что вызывающий абонент сможет восстановить управление в течение указанного промежутка времени, если отмененный вызов не перезвонит.

Безопасность (Security)

Выполнение процедур за пределами физических машин и по сети создает дополнительные требования к безопасности. RPC API включает в себя интерфейс к базовым службам безопасности.

RPC API предоставляет программистам средства для применения этой расширенной семантики, но он защищает приложения от сложностей программирования отправки и получения на транспортном уровне. Парадигма программирования RPC предоставляет программисту контроль над удаленной семантикой в двух точках: в спецификации интерфейса и через RPC API.

- Спецификация интерфейса, в то время как она в основном используется для определения синтаксиса локального вызова интерфейса, также позволяет программистам указывать желаемую семантику выполнения, степень, в которой привязка находится под контролем программы, и семантику ошибок. Спецификация интерфейса описана на языке определения интерфейса ([Interface Definition Language](#)).
- RPC API предоставляет приложениям доступ к различным службам времени выполнения и управляет многими взаимодействиями клиента и сервер. Другие функции включают аутентификацию, параллелизм сервера и управление сервером.

Binding

A remote procedure call requires a remote binding. The calling client must bind to a server that offers the interface it wants, and the client's local procedure call must invoke the correct manager operation on the bound-to server. Because the various parts of this process occur at run time, it becomes possible to exercise nearly total programmatic control of binding. The RPC API provides access to all aspects of the binding process.

Each binding consists a set of components that can be separately manipulated by applications, including protocol and addressing information, interface information and object information. This allows servers to establish many binding paths to their resources and allows clients to make binding choices based on all of the components. These capabilities are the basis for defining a variety of server resource models.

Name Services

Servers need to make their resources widely available, and clients need some way to find them without knowing the details of network configuration and server installation. Hence, the RPC mechanism supports the use of name services, where servers can advertise their bindings and clients can find them, based on appropriate search criteria. The RPC API provides clients and servers with a variety of routines that can be used to export and import bindings to and from name services.

Что такое команды `rpcinfo` и `rpcbind` в Linux?

В следующих разделах подробно обсуждаются команды `rpcinfo` и `rpcbind`.

Команда `rpcinfo` в Linux



РЕКЛАМА



exclusivkuzk.ru

Домашние иконостасы для икон прямые.

[Узнать больше](#)



РЕКЛАМА



kiot.ru

Изготовление киотов, иконостасов

[Узнать больше](#)



РЕКЛАМА



vseikony.ru

Иконописная мастерская «ВсеИконы» - Скидки до 5000р!

от 8 000 ₪

[Узнать больше](#)

вы не укажете версию, то функция rpcinfo найдет все зарегистрированные версии программы.

Синтаксис команды rpcinfo

Синтаксис команды rpcinfo показан ниже:

```
~$ rpcinfo [options]
```

Опции для команды rpcinfo: -a, -b, -d, -l, -m, -n номер порта, -p и т. д. IP-адрес и порт хоста предоставляются с помощью опции -a. Параметр -b используется для выполнения широковещательной передачи RPC процедуре 0 и возврата всех ответивших хостов. Параметр -d используется для удаления регистрации службы RPC, указанной в параметрах versum и prognum. Опция -l используется для вывода списка записей versum и prognum для указанного хоста.

Опция -m печатает таблицу операций portmap. Параметр -n использует номер порта в качестве номера порта. Параметр -p проверяет службу portmap на хосте. Все параметры команды rpcinfo могут быть представлены с помощью команды --help. Следующая команда будет использоваться для отображения списка параметров, доступных для команды rpcinfo:

```
linux@linux-VirtualBox:~$ rpcinfo --help
```

Это даст вам полный список опций с их описанием. См. список ниже:

```
Usage: rpcinfo [-m | -s] [host]
       rpcinfo -p [host]
       rpcinfo -T netid host prognum [versnum]
       rpcinfo -l host prognum versnum
       rpcinfo [-n portnum] -u | -t host prognum [versnum]
       rpcinfo -a serv_address -T netid prognum [version]
       rpcinfo -b prognum versnum
       rpcinfo -d [-T netid] prognum versnum
```

Но прежде чем выполнять какую-либо команду RPC, необходимо убедиться, что она предварительно установлена в вашей системе. Если в вашей системе он не предустановлен, вы можете сделать это с помощью команды «sudo apt install». Используйте следующую команду и установите набор инструментов RPC в вашей системе:

```
linux@linux-VirtualBox:~$ sudo apt install rpcbind
```

Это установит rpcinfo, rpcbind, а также все другие команды RPC в вашей системе. Смотрите результат ниже:


```

Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  rpcbind
0 upgraded, 1 newly installed, 0 to remove and 49 not upgraded.
Need to get 46.6 kB of archives.
After this operation, 160 kB of additional disk space will be used.
Get:1 http://pk.archive.ubuntu.com/ubuntu jammy/main amd64 rpcbind amd
64 1.2.6-2build1 [46.6 kB]
Fetched 46.6 kB in 6s (7,335 B/s)
Selecting previously unselected package rpcbind.
(Reading database ... 166192 files and directories currently installed
.)
Preparing to unpack .../rpcbind_1.2.6-2build1_amd64.deb ...
Unpacking rpcbind (1.2.6-2build1) ...
Setting up rpcbind (1.2.6-2build1) ...
Created symlink /etc/systemd/system/multi-user.target.wants/rpcbind.se
rvice → /lib/systemd/system/rpcbind.service.
Created symlink /etc/systemd/system/sockets.target.wants/rpcbind.socke
t → /lib/systemd/system/rpcbind.socket.
Processing triggers for man-db (2.10.2-1) ...

```

Теперь ваша система готова к запуску команды `rpcinfo`. Посмотрим, что нам вернет команда `rpcinfo`:

```

linux@linux-VirtualBox:~$ rpcinfo

```

Если вы не используете какую-либо опцию или не указываете ничего с помощью команды `rpcinfo`, то она просто вернет все службы RPC локального хоста. Давайте подтвердим это выводом, приведенным ниже:

program	version	netid	address	service	owner
100000	4	tcp6	:::0.111	portmapper	superuser
100000	3	tcp6	:::0.111	portmapper	superuser
100000	4	udp6	:::0.111	portmapper	superuser
100000	3	udp6	:::0.111	portmapper	superuser
100000	4	tcp	0.0.0.0.0.111	portmapper	superuser
100000	3	tcp	0.0.0.0.0.111	portmapper	superuser
100000	2	tcp	0.0.0.0.0.111	portmapper	superuser
100000	4	udp	0.0.0.0.0.111	portmapper	superuser
100000	3	udp	0.0.0.0.0.111	portmapper	superuser
100000	2	udp	0.0.0.0.0.111	portmapper	superuser
100000	4	local	/run/rpcbind.sock	portmapper	superuser
100000	3	local	/run/rpcbind.sock	portmapper	superuser

Давайте отобразим все службы RPC, зарегистрированные с помощью протокола `rpcbind` на локальной машине. Вот как вы можете это сделать:

```
linux@linux-VirtualBox:~$ rpcinfo -p
```

Параметр -p даст вам следующий результат:

program	vers	proto	port	service
100000	4	tcp	111	portmapper
100000	3	tcp	111	portmapper
100000	2	tcp	111	portmapper
100000	4	udp	111	portmapper
100000	3	udp	111	portmapper
100000	2	udp	111	portmapper

Команда *rpcbind* в Linux

rpcbind — это команда операционной системы Linux для привязки программы RPC к универсальным адресам. Это утилита RPC, которая используется для преобразования номера программы RPC в определенные универсальные адреса. Утилита *rpcbind* должна быть запущена на хосте, чтобы она могла выполнять вызовы RPC к серверам. Служба RPC предоставляет утилите *rpcbind* две части информации при запуске: номер программы, которая готова к обслуживанию, и адрес, по которому она прослушивается. Когда клиенту необходимо выполнить вызов RPC, он связывается с *rpcbind*, чтобы получить адрес, по которому должен быть выполнен вызов RPC. Здесь следует помнить, что утилита *rpcbind* должна быть в активном состоянии, прежде чем вы попытаетесь использовать любую другую службу RPC.

Синтаксис команды *rpcbind*

Синтаксис команды *rpcbind* в операционной системе Linux следующий:

```
~$ rpcbind [options]
```

Доступные параметры для команды *rpcbind*: -a, -d, -f, -h, -i и т. д. Параметр -a используется для прерывания службы при ошибках во время отладки. Когда вы запускаете команду *rpcbind* с параметром -a, система прервет работу в случае возникновения какой-либо ошибки. Опция -d команды *rpcbind* позволяет запускать утилиту в режиме отладки. Параметр -f заставит процесс работать в фоновом режиме, а не разветвляться. Параметр -h используется для определения IP-адресов для привязки. Чтобы использовать любую из опций с *rpcbind*, вам нужно убедиться, что служба *rpcbind* активна. Вы можете сделать это, проверив состояние утилиты *rpcbind* с помощью следующей команды:

```
linux@linux-VirtualBox:~$ service rpcbind status
```

Это сообщит вам о состоянии утилиты *rpcbind*, чтобы вы могли перезапустить ее, если она не активна. А если он находится в активном состоянии, то вы получите следующий результат:

```

● rpcbind.service - RPC bind portmap service
   Loaded: loaded (/lib/systemd/system/rpcbind.service; enabled; vendor pr>
   Active: active (running) since Mon 2022-09-12 01:49:42 PKT; 4min 25s ago
   TriggeredBy: ● rpcbind.socket
     Docs: man:rpcbind(8)
    Main PID: 11297 (rpcbind)
      Tasks: 1 (limit: 5861)
     Memory: 552.0K
        CPU: 6ms
    CGroup: /system.slice/rpcbind.service
            └─11297 /sbin/rpcbind -f -w

01:49:42 12 ستمبر linux-VirtualBox systemd[1]: Starting RPC bind portmap serv>
01:49:42 12 ستمبر linux-VirtualBox systemd[1]: Started RPC bind portmap serv>

```

Library Functions Manual

rpc(3)

NAME [top](#)

rpc - library routines for remote procedure calls

LIBRARY [top](#)

Standard C library (*libc*, *-lc*)

SYNOPSIS AND DESCRIPTION [top](#)

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service, and then sends back a reply. Finally, the procedure call returns to the client.

To take use of these routines, include the header file *<rpc/rpc.h>*.

Краткие теоретические сведения

Итак, в рамках технологии RPC узел (клиент или сервер) идентифицируется при помощи следующих четырех параметров:

- имя узла сети (имя хоста);
- номер программы на этом узле;
- номер версии программы;

- номер процедуры в программе.

Для того, чтобы программа-сервер была доступна для взаимодействия, она должна быть зарегистрирована на соответствующем узле сети (т.е. на сервере) в службе RPC соответствующей операционной системы.

Удаленный RPC-клиент, обращаясь к службе RPC сервера, может получить сетевой адрес процедуры, который будет в дальнейшем использован для обращений к программе-серверу.

Процедура-сервер, создаваемая службой RPC, должна иметь один единственный аргумент и единственный результат. Если есть необходимость возвращать, скажем, несколько параметров, то для этой цели возможно использовать, например, структуру, массив и т.д.

Минимальная реализация сетевых (распределенных) приложений на базе технологии RPC среднего уровня возможна при использовании всего трех функций:

- `register_rpc, svc_run` (на стороне сервера)
- `call_rpc` (на стороне клиента)

Т.е. средний уровень технологии RPC является достаточно ограниченным в своем функционале.

При создании распределенных приложений, кроме собственно функций RPC, следует также использовать функции преобразования данных во внешнее представление согласно стандарту XDR (так называемые XDR-функции).

Описание основных функций интерфейса RPC

Регистрации процедуры-сервера на узле сети

Она осуществляется при помощи функции `register_rpc()`:

```
#include <sys/types.h>
#include <rpc/rpc.h>

int register_rpc (prognum, vernum, procnum, procname, inproc, outproc)
```

- `u_long prognum;`
- `u_long vernum;`
- `u_long procnum;`
- `char *(*procname)();`
- `xdrproc_t inproc;`
- `xdrproc_t outproc;`

Параметры `prognum`, `vernum`, `procnum` определяют номера программы, версии и процедуры соответственно. Номера версии и процедуры могут быть заданы произвольно (например, равны 1).

Тогда как номер программы, находящейся в стадии разработки, должен назначаться из диапазона номеров 0x20000000...0x3fffffff.

процпаме определяет функцию C, регистрируемую в качестве сервера. Эта функция вызывается с указателем на ее аргумент и должна возвращать указатель на свой результат, располагаемый в статической или динамически выделенной (при помощи malloc или calloc) памяти. Для хранения результата нельзя использовать автоматически выделяемую память (напоминаем, что локальные переменные функций располагаются именно в такой памяти).

Аргументы inproc и outproc задают XDR-функции преобразования, аргумента и результата функции registerrpc, соответственно.

При успешном выполнении функция registerrpc возвращает 0, в случае ошибки – возвращает "-1".

Прием запросов сервером от клиента

Функция svc_run

```
#include <rpc/rpc.h>

void svc_run();
```

Эта функция не имеет аргументов, вызывается после того, как осуществлена регистрация процедуры-сервера в службе RPC. Обратите внимание, что при успешном выполнении функция svc_run() никогда не возвращает управление в вызвавшую ее программу. Это избавляет от необходимости реализовывать бесконечный цикл, как это, как правило, делается при программировании серверов на сокетах. А вот если возникнет ошибка – тогда (и только тогда) сработает функция, идущая следующей в программном коде, после svc_run().

Запрос к серверу

Запрос выполняет функция callrpc():

```
#include <sys/types.h>
#include <rpc/rpc.h>

int callrpc (host, prognum, vernum, procnum, inproc, in, outproc, out)
```

- char *host (имя хоста);
- u_long prognum (номер программы);
- u_long vernum (номер версии);
- u_long procnum (номер процедуры);
- xdrproc_t inproc (входная программа);
- char *in (входные параметры);
- xdrproc_t outproc (выходная программа);

- `char *out` (выходные параметры);

Аргументы:

- `host` - имя узла, на котором функционирует сервер. Если программа сервер работает на локальном компьютере (т.е. на том же самом, что и клиент), то в качестве имени узла будет фигурировать, как правило, `localhost` (а можно ввести и его имя).
- Узнать имя хоста можно при помощи консольной команды `hostname -s`
- `prognum`, `vernum` и `procnum` – это номера программы, версии и процедуры-сервера. Как обычно, сервер должен быть запущен на соответствующем узле (имеющем имя `host`) до обращения клиента к нему (иначе будет ошибка вида «Connection refused» или «В соединении отказано»)
- `in` должен указывать на данные, передаваемые серверу в качестве аргумента
- `out` указывает на область памяти, предназначенную для размещения в ней результата работы сервера.
- `inproc` и `outproc` представляют собой XDR-функции преобразования, аргумента процедуры-сервера и ее результата, соответственно.

Технология RPC функционирует по технологии «клиент-сервер». Соответственно, после того, как клиент послал запрос, он будет ожидать ответа от сервера.

При успешном выполнении вызова удаленной процедуры-сервера функция `callrpc()` вернет 0, в случае ошибки возвращается "-1".

XDR-функции

Эти функции предназначены для преобразования данных в/из XDR-формат. Среди них можно упомянуть такие функции, как:

- `xdr_int` - для преобразования целых;
- `xdr_u_int` - для преобразования беззнаковых целых;
- `xdr_short` - для преобразования коротких целых;
- `xdr_u_short` - для преобразования беззнаковых коротких целых;
- `xdr_long` - для преобразования длинных целых;
- `xdr_u_long` - для преобразования беззнаковых длинных целых;
- `xdr_char` - для преобразования символов;
- `xdr_u_char` - для преобразования беззнаковых символов;

- `xdr_wrapstring` - для преобразования строк символов (заканчивающихся символом `'\0'`).

Внимание! Вышеприведенные функции должны использоваться в качестве входной, выходной программ в функциях `register_rpc()` `call_rpc()`, соответственно (см. примеры листингов сервера и клиента ниже).

Если аргумент в процедуру-сервер не передается (или когда она не возвращает результат), целесообразно использовать функцию-"заглушку" `xdr_void()`.

XDR-функции являются универсальными. В зависимости от контекста их использования, они способны преобразовывать данные из внутреннего представления (характерного для компьютера конкретной архитектуры), во внешнее, согласно стандарту XDR и, наоборот; эти функции могут также динамически выделять или освобождать память для данных. Стандарт этих функций одинаков для всех (по крайней мере, наиболее известных) операционных систем. Т.е. вполне можно при помощи клиента из *Linux* направить сообщение на сервер, например, в *Windows* или даже в *OpenVMS*. При необходимости, можно также написать свои XDR-функции на основе имеющихся.

Итак, в рамках данного задания необходимо реализовать клиент и сервер, взаимодействующие друг с другом, с использованием технологии RPC.

Общее описание программ

Программы клиент и сервер, реализуемые в рамках данного задания, будут взаимодействовать по технологии «клиент-сервер», т.е. «запрос клиента – ответ сервера». Это означает, что:

- Сервер должен быть запущен РАНЕЕ клиента, при этом, до поступления запроса от клиента, он находится в состоянии ожидания. При поступлении запроса сервер обрабатывает его и направляет ответ клиенту;
- Клиент должен быть запущен ПОСЛЕ сервера, при этом он может направлять запросы к серверу и ожидает поступления ответа от него;
- Перед запуском сервера обязательно должна быть запущена служба RPC (иначе система вернет сообщение о недоступности соединения).

Иными словами, общая технология взаимодействия между клиентом и сервером – точно такая же, как и в технологии сокетов. Только сами программы и их интерфейс реализованы, в случае технологии RPC, проще. Однако, в отличие от технологии сокетов, на обоих узлах (как на клиенте, так и на сервере) должна быть запущена служба RPC, представляющая собой, своего рода, промежуточный (системный) сервер. Именно эта служба, в частности, выбирает порт для взаимодействия между клиентом и сервером. Порты назначаются динамически, выбираются среди свободных на конкретном узле.

Порт назначается, в качестве соответствия номерам RPC-программ. Это осуществляется регистратором - преобразователем портов (port mapper). По сути, последний представляет из себя (системный) сервер, который и осуществляет такое преобразование. Если этот сервер не запущен, исполнение (и, следовательно, ответ) RPC вызова будет невозможно.

Примечание. В технологиях RPC, основанных на транспортном интерфейсе TLI, вместо `port mapper` используется `rpcbind`.

Тогда как для технологии сокетов нет необходимости в реализации дополнительной службы. Зато ряд параметров (например, порт), которые используются в процессе клиент-серверного взаимодействия, необходимо устанавливать «вручную» (т.е. необходимо прописывать задание соответствующих параметров в программе). При этом (при использовании сырых сокетов - `SOCK_RAW`) возможно также «ручное» задание ряда других свойств сетевого соединения. Тогда как возможности технологии RPC в этом аспекте - ограничены. Для передачи сетевого адреса и номера порта используется служба (демон) `portmap()`.

Вызов службы RPC возможен, по-видимому, из любого языка, который поддерживает реализацию сетевых соединений. Соответственно этой службе, есть и протокол RPC. Он основан на протоколах транспортного уровня, т.е. работает поверх них. При этом создание логического соединения, обеспечение надежности и т.п. реализуют транспортные протоколы (*TCP, UDP*).

Remote Procedure Calls

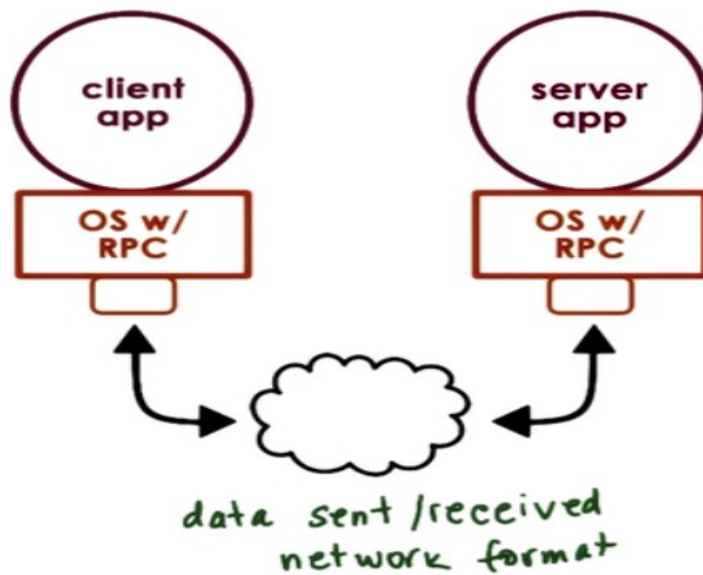
Example : GetFile App

- Client Server
- Create and init sockets
- Allocate and populate buffers
- Include 'protocol' info
 - GetFile, size
- Copy data into buffers
 - filename, file
- common steps related to remote IPC

Remote Procedure Calls (RPC)

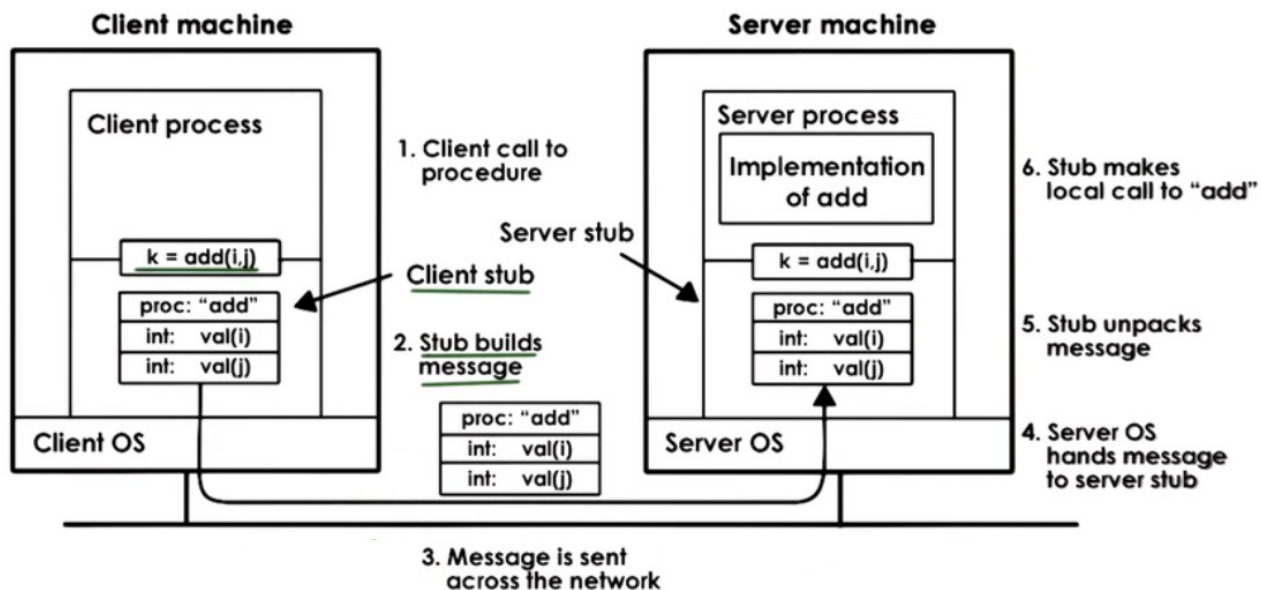
- Intended to simplify the development of cross address space and cross machine interactions
- + Higher-level interface for data movement and communication
+ Error handling
+ Hiding complexities of cross machine interactions

RPC requirements



1. Client/Server interactions
2. Procedure Call Interface => RPC
 - o sync call semantics
3. Type checking
 - o error handling
 - o packet bytes interpretation
4. Cross machine conversion
 - o e.g. big/little endian
5. Higher level protocol
 - o access control, fault tolerance, different transport protocols

Structure of RPC



RPC Steps:

(-1.) register : server registers procedure, arg types, location

(0.) bind : client finds and binds to desired server

1. call : client make RPC call; control passed to stub, client code blocks
2. marshal : client stub "marshals" args (serialize args into buffer)
3. send : client sends message to server
4. receive : server receives message; passes message to server stub; access control
5. unmarshal : server stub "unmarshals" args (extract args from buffer)
6. actual call : server stub calls local procedure implementation
7. result : server performs operation and computes result of RPC operation

(same on return <=)

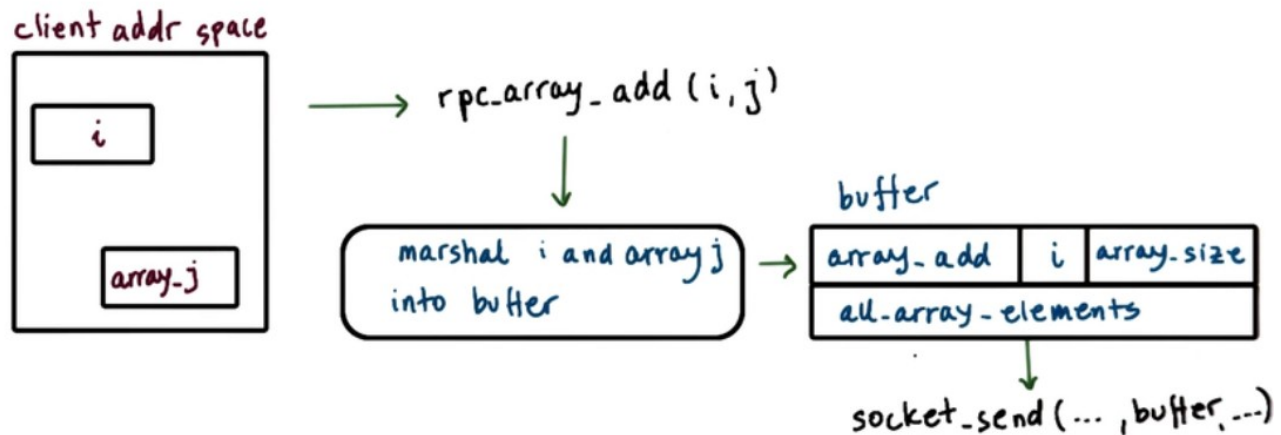
Interface definition Language (IDL)

- Used to describe the interface the server expects
 - procedure name, args, 2 result types
 - version number

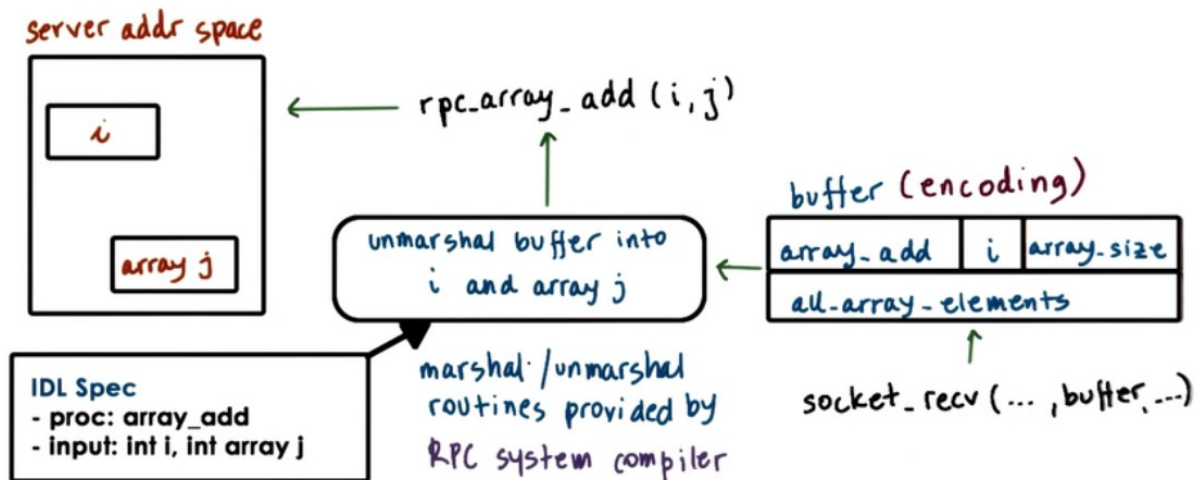
RPC can use IDL that is

1. Language agnostic
 - o XDR in SunRPC
2. Language specific
 - o Java in JavaRMI

Marshalling



Unmarshalling



Marshalling/Unmarshalling routines are provided by RPC system compiler.

Binding and Registry

- Client determines

- o **which** server to connect to?
 - service name. version number
 - o **how** to connect to that server?
 - IP address, network protocol
- Registry : database of available services
 - o search for service name to find server(which) and contact details(how)
 - o distributed
 - any RPC service can register
 - o machine-specific
 - for services running on same machine
 - clients must know machine addresses
 - registry provides port number needed for connection
- Who can provide a service?
 - o lookup registry for image processing
- What services do they provide?
 - o compress/filter.. version number => IDL
- How will they ship package?
 - o TCP / UDP -> registry

Pointers

- Procedure interface : foo(int,int)
- in Local Calls : foo(x,y) => okay
- in Remote Calls : foo(x,y) => ?

here, y points to location in caller address space

- Solutions:
 - o No pointers
 - o Serialize pointers; copy referenced ("points to") data structure to send buffer

Handling Partial Failures

- Special RPC error notification (signal, exception..)
 - Catch all possible ways in which RPC can (partially) fail

RPC Design choice

- Binding => How to find the server
- IDL => How to talk to server; how to package data
- Pointers as args => Disallow or serialize pointer data
- Partial failures => Special error notifications