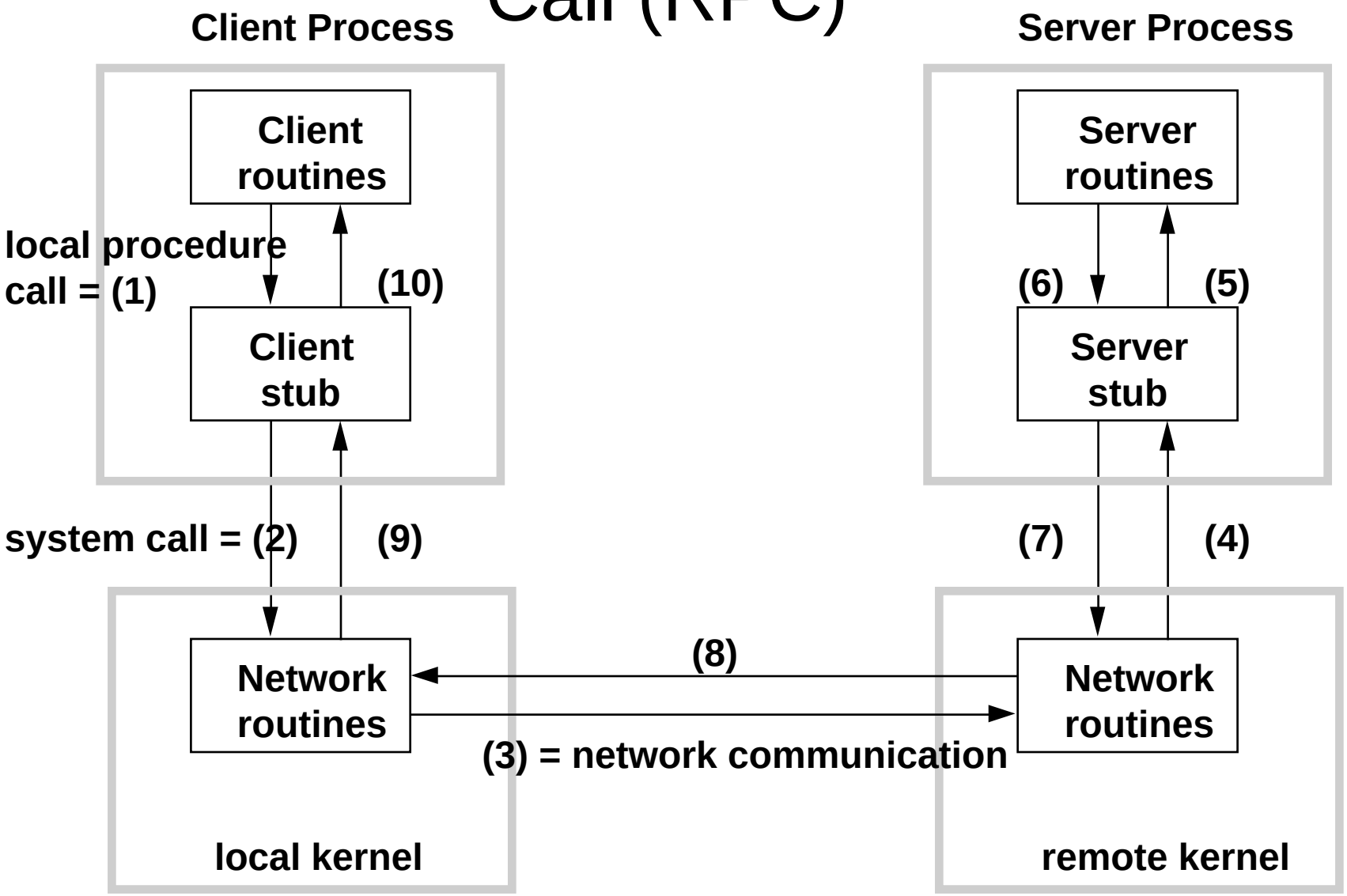
Laborator SPRC
RPC – Remote Procedure Calls

Remote Procedure Calls

- The procedure call (same as function call or subroutine call) is a well-known method for transferring control from one part of a process to another, with a return of control to the caller.
- Associated with the procedure call is the passing of arguments from the caller (the client) to the callee (the server).
- In most current systems the caller and the callee are within a single process on a given host system. This is what we called “local procedure calls”.
- In a remote procedure call (RPC), a process on the local system invokes a procedure on a remote system. The reason we call this a “procedure call” is because the intent is to make it appear to the programmer that a normal procedure call is taking place.
- We use the term “request” to refer to the client calling the remote procedure, and the term “response” to describe the remote procedure returning its result to the client.
- We will use the SUN RPC as our remote procedure call example. However, our example works for BSD as well with very minor modifications.
- But let’s look at the steps that is involved in a remote procedure call.

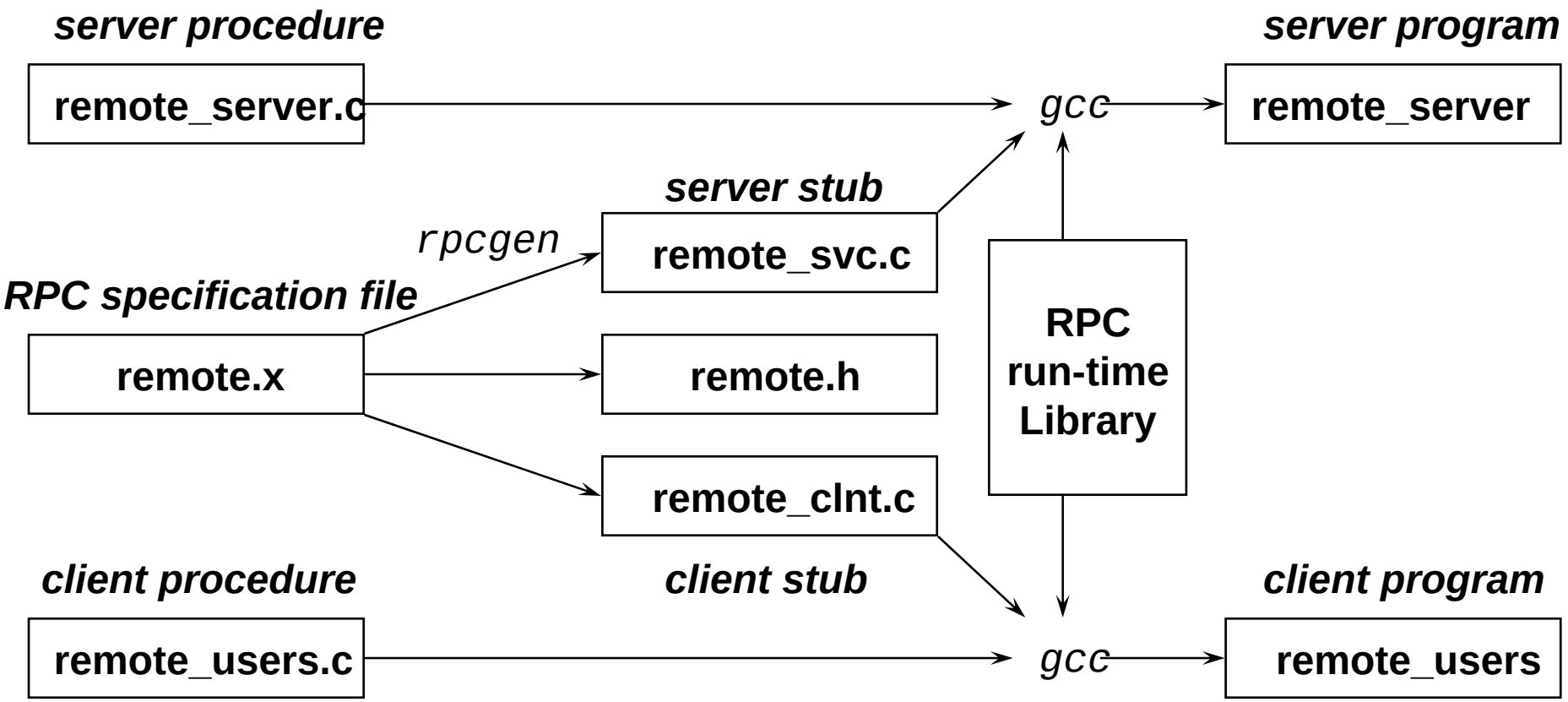
The 10 steps in a Remote Procedure Call (RPC)



The 10 steps in a Remote Procedure Call (continue)

1. The client calls a local procedure, called the client stub. It appears to the client that the client stub is the actual server procedure that it wants to call. The purpose of the stub is to package the arguments for the remote procedure, possibly put them into some standard format and then build one or more network messages. The packaging of the client's arguments into a network message is termed marshaling.
2. These network messages are sent to the remote system by the client stub. This requires a system call to the local kernel.
3. The network messages are transferred to the remote system. Either a connection-oriented or a connection-less protocol is used.
4. A server stub procedure is waiting on the remote system for the client's request. It unmarshals the arguments from the network message and possibly converts them.
5. The server stub executes a local procedure call to invoke the actual server function, passing it the arguments that it received in the network messages from the client stub.
6. When the server procedure is finished, it returns to the server stub with return values.
7. The server stub converts the return values, if necessary, and marshals them into one or more network messages to send back to the client stub.
8. The messages get transferred back across the network to the client stub.
9. The client stub reads the network messages from the local kernel.
10. After possibly converting the return values, the client stub finally returns to the client function. This appears to be a normal procedure return to the client.

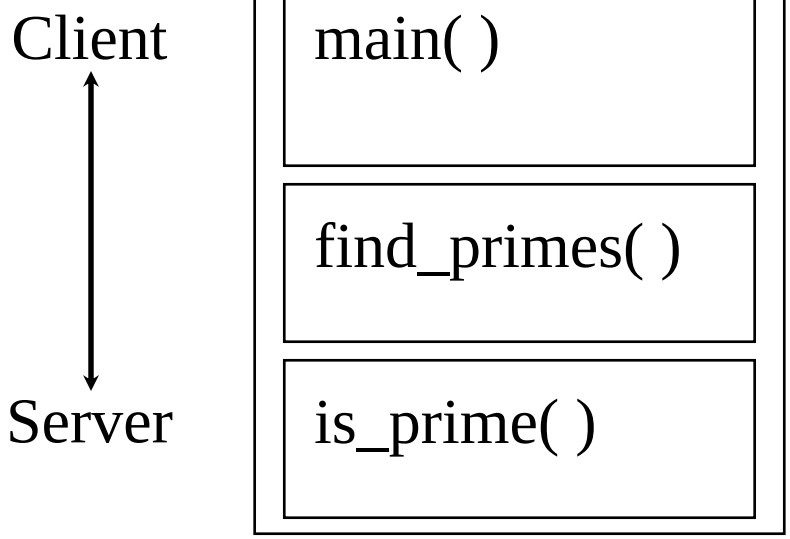
Client-Server Application using RPC



RPC example II -- finding prime numbers

```
/* print a list of primes between 1 and 1000 */
main()
{
 int I, how_many, primes[1000];
 how_many = find_primes(1, 1000, primes);
 for (I = 0; I < how_many; I++)
       printf("%d is prime\n", primes[I]);
}
/* Find all primes between min and max, return them
     in an array */
int find_primes(int min, int max, int *array)
{
 int I, count = 0;
 for (I = min; I <= max; I++)
      if (isprime(I)) array[count++] = I;
 return count;
}
```

```
/* Return TRUE If n is prime */
int isprime(int n)
{
 int I;
 for (I = 2; I*I <= n; I++){
      if ((n % I) == 0) return 0;
 }
 return 1;
}
```



primes.x

```
const MAXPRIMES = 1000;
struct prime_request{
 int min;
 int max;
};

struct prime_result{
 int array<MAXPRIMES>;
};

program PRIMEPROG{
 version PRIMEVERS{
  prime_result FIND_PRIMES(prime_request) = 1;
 } = 1;
} = 0x32345678;
```

p_server.c

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

prime_result *find_primes_1(prime_request
    *request)
{
 static prime_result result;
 static int prime_array[MAXPRIMES];
 int i, count = 0;

 for (i = request->min; i <= request->max; i++)
     if (isprime(i)) prime_array[count++] = i;
 result.array.array_len = count;
 result.array.array_val = prime_array;
 return(&result);
}
```

```
int isprime(int n)
{
 int i;
 for (i = 2; i*i <= n; i++){
     if ((n % i) == 0) return
0;
 }
 return 1;
}
```

p_client.c

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

main(int argc, char *argv[])
{
 int i;
 CLIENT *cl;
 prime_result *result;
 prime_request request;

 if (argc != 4){
    printf("usage: %s host min max\n", argv[0]);
    exit(1);
 }
cl = clnt_create(argv[1], PRIMEPROG,
      PRIMEVERS, "tcp");
 if (cl == NULL){
    clnt_pcreateerror(argv[1]);
    exit(2);
 }
 request.min = atoi(argv[2]);
 request.max = atoi(argv[3]);

 result = find_primes_1(&request, cl);
 if (result == NULL){
    clnt_perror(cl, argv[1]);
    exit(3);
 }

 for (i = 0; i < result->array.array_len; i++)
     printf("%d is prime\n", result-
>array.array_val[i]);
 printf("count of primes found = %d\n",
  result->array.array_len);

 xdr_free(xdr_prime_result, result);
 clnt_destroy(cl);
}
```

One More Example on RPC

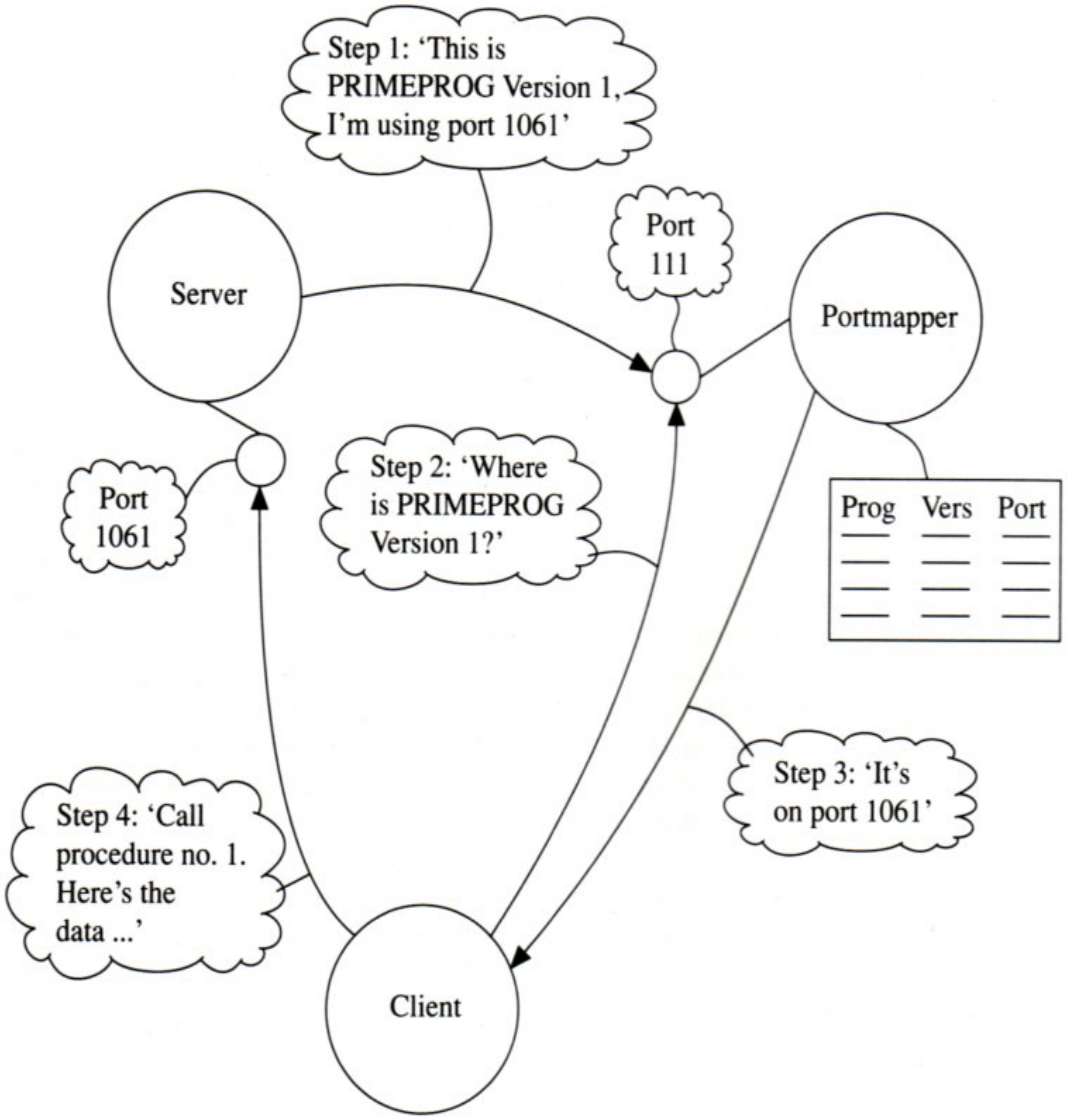
```
/* Program: shop.x */

/* Limitation of RPC:
   Remote Procedure Call just supports single parameter
   passing.
   Hence, one have to define a specific data structure for
   passing
   more than one parameters */

/* you can define your data structure here */
struct items{
 char name[10];
 int qty;
 float amount;
};

program SHOP_PROG{
   version SHOP_VERS{
      items ADD12IT(items) = 1;
   } = 1;
} = 0x32345678; /* please use your student ID */
```

Identifying an RPC Service

- There is a service registry called the port mapper, which keeps track of all RPC services registered on a machine, and their transport endpoint addresses.
- In later implementations the port mapper is called "*rpcbind*".
- Remote procedure is identified by THREE numbers:

  {*program number, version number, procedure number*}
- While {*program number, version number*} identify a specific server program, the *procedure number* identifies a procedure within that program
- The followings are the steps involved in locating an RPC service:
  - When server starts, it create a endpoint to accept connection, so it binds an arbitrary port number for this purpose;
  - It then send a message to the portmapper registering the service;
  - Portmapper add this mapping to its list;
  - When client wants to find the service, it ask the portmapper by passing it the program number, version number and protocol it uses;
  - Portmapper returns the port number to the client;
  - Client call the server procedure by sending an RPC call message to the port it just got from the portmapper;
  - The call message contains the serialized arguments and the procedure number

Identifying an RPC Service



Identifying an RPC Service

```
deepthought% rpcinfo -p
   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
    100007    3   udp   1029  ypbind
    100007    3   tcp   1027  ypbind
    100024    1   udp   1035  status
    100021    1   udp   1036  nlockmgr
    100024    1   tcp   1028  status
    100021    1   tcp   1029  nlockmgr
    100087   10   udp   1038
    100021    3   udp   1036  nlockmgr
    100021    3   tcp   1029  nlockmgr
    100083    1   tcp   4046
    100003    2   udp   2049  nfs
    100005    1   udp   1056  mountd
    100005    2   udp   1056  mountd
    100026    1   udp   1057  bootparam
    100005    1   tcp   4048  mountd
    100026    1   tcp   4047  bootparam
    100005    2   tcp   4048  mountd
```

Program numbers:
00000000-1FFFFFFF Admin by Sun Microsystems
20000000-3FFFFFFF User-defined
40000000-5FFFFFFF Transient
60000000-FFFFFFFF Reserved for future use