CS 348: Computer Networks

- RPC; 16th – 18th Oct 2012

Instructor: Sridhar Iyer
IIT Bombay

Traditional distributed applications

- Application logic

- Transport interface code:
  - Makes the appropriate network calls to send and receive the messages
  - Usually divided into transport-independent and transport-dependent parts

- Middleware provides transparency of the transport interface code

Middleware

- Interfaces allow applications to be distributed and to take advantage of other services provided over the network
- Middleware is a set of services that are accessible to application programmers through an API
- Example: Sockets, RPC, CORBA

Middleware design

Middleware support for distributed applications has two approaches:

- Communication Oriented: sockets
  - begin with the communication protocol
  - design client and server by specifying how each reacts to incoming messages and how each generates outgoing messages
  - can be restrictive for application designer

- Application Oriented

Middleware design

- Application Oriented: RPC
  - design a conventional application
  - divide the application into multiple parts (client/server)
  - add communication protocols that allow each to execute on a separate machine
  - can be restrictive for programming

Remote Procedure Call (RPC)

- Birell and Nelson in 1984
- Goal:
  - make distributed programming as simple as possible
  - distributed programming should be similar to normal sequential programming

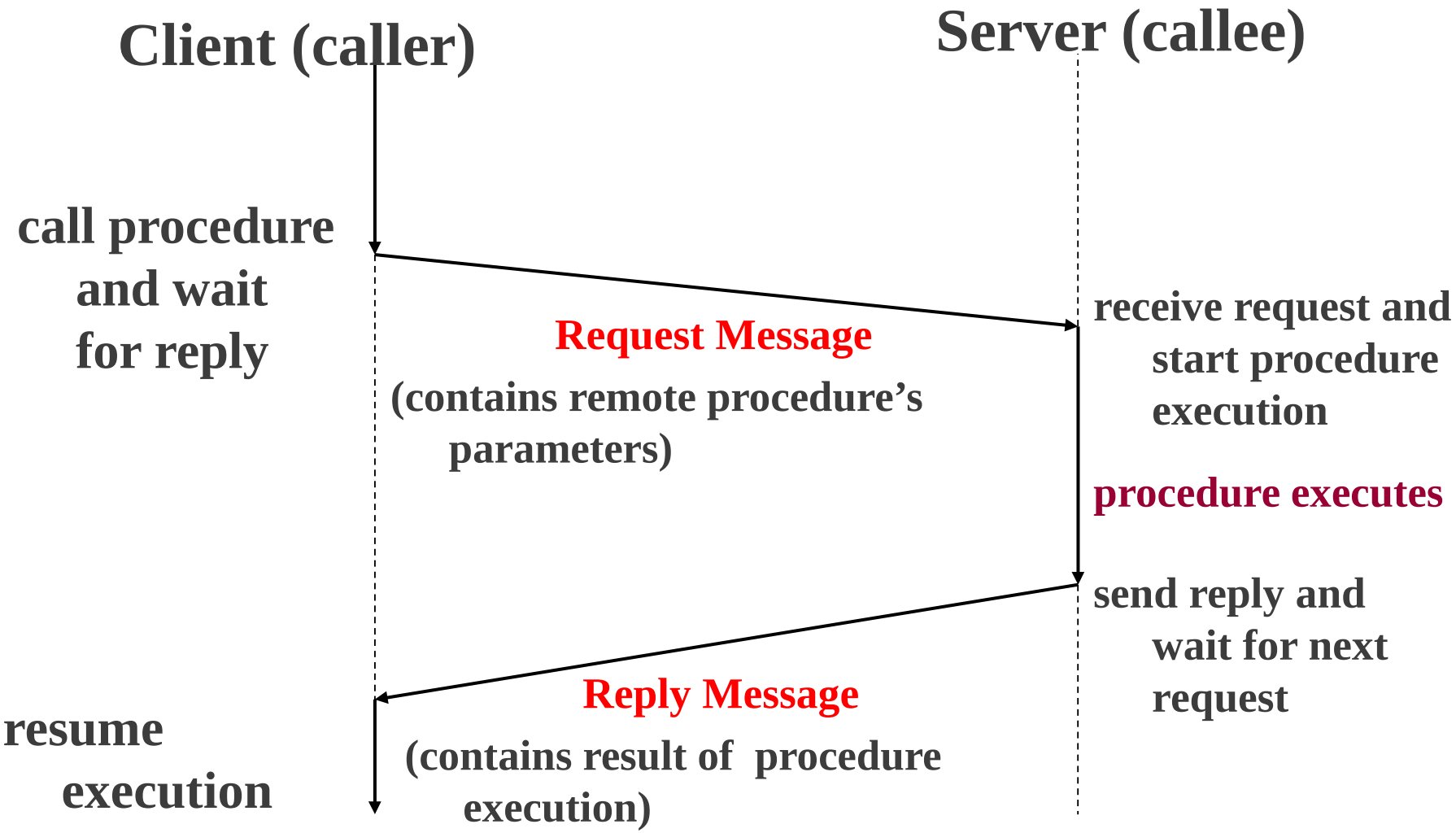
RPC motivation

- Sequential programming: to get something done, call a function
- Distributed programming: to get something done, send a message to the server, i.e., do I/O
- RPC tries to bridge this gap

RPC: Central idea

- Allow programs to call procedures located on other hosts
  - when program on host A calls a procedure on host B, the calling process on A is suspended
  - the execution of the called procedure takes place on B
  - when the procedure returns, the process on A resumes
- Information exchange takes place via procedure arguments and return value

RPC functionality



RPC: Advantages

- Extends the conventional procedure call to the client/server model
  - Allows a client to execute procedures on other computers
  - Remote procedures accept arguments and return results
- Helps programmer to focus on the application instead of communication

RPC forms the foundation for many distributed utilities, like NFS and NIS

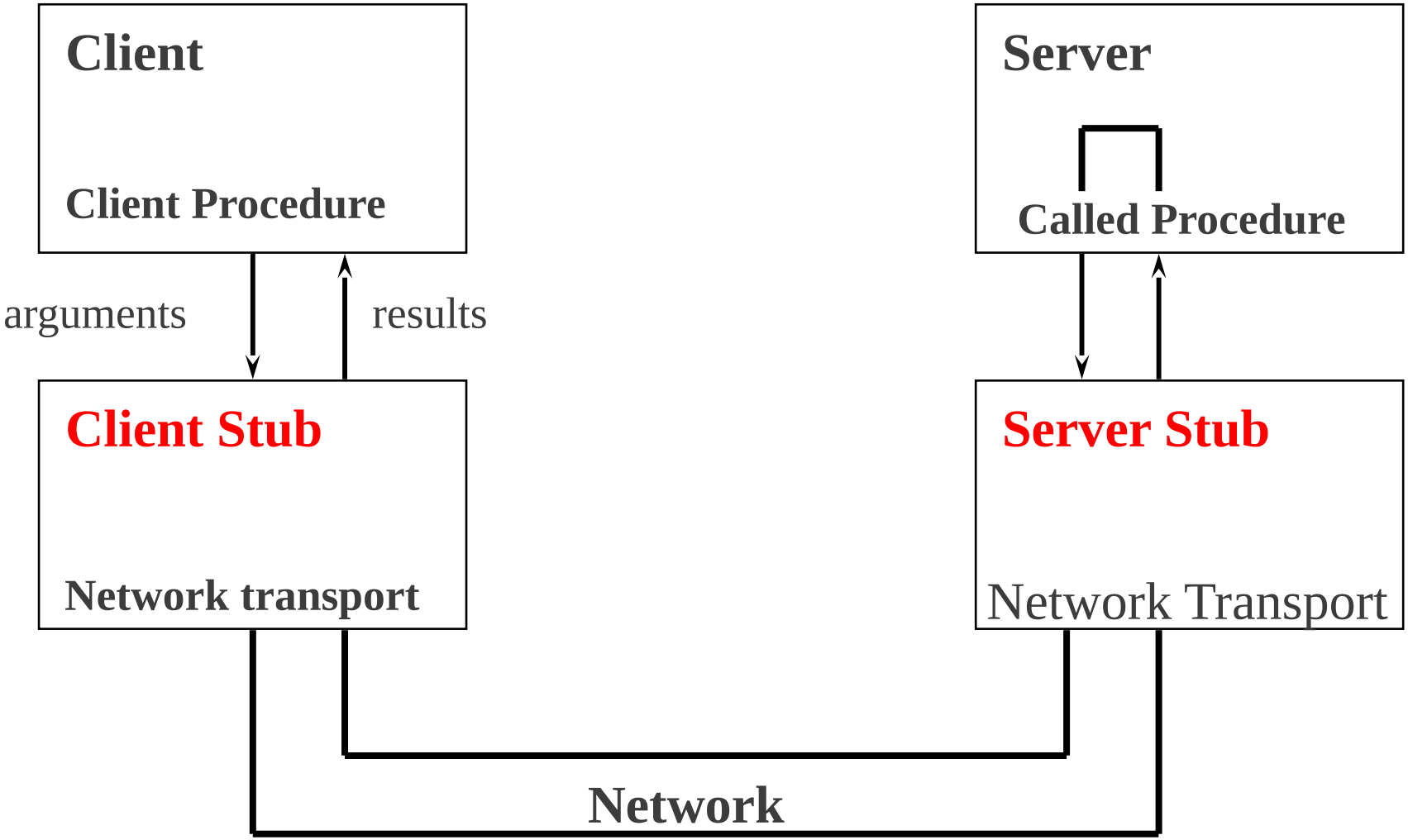
RPC: Design Issues

- Transparency:
  - Syntactic Transparency:
    - RPC should have same syntax as local procedure call
  - Semantic Transparency:
    - RPC semantics should be identical to local procedure call

RPC: Design Issues

- Standard Representation:
  - client machine may be little-endian and server machine may be big-endian
  - representation of floating point numbers on the two machines may be different
  - n client architectures and m server architectures => n * m versions
  - need for eXternal Data Representation (XDR) for all data types

RPC: Overview



RPC: Call semantics

- Possibly or maybe: No retry of call; no certainty of results
- At-least once: Retry implemented but no duplicate filtering
- Exactly once: Retry and duplicate filtering are implemented
- Others: At-most once; last-one call semantics etc

RPC: Communication protocol

- Request (R): No value returned; no acknowledgement
- Request-Reply (RR): Replies are used as acknowledgements
- Request-Reply-Acknowledge (RRA): Client specifically acknowledges each reply from server

RPC: Parameter passing

stubs do marshalling (packing) of arguments and results

- *call-by-value* is the most common semantics
- Pointers, global variables etc cannot be used for communication
- all arguments are marshalled into a structure and a single argument is passed

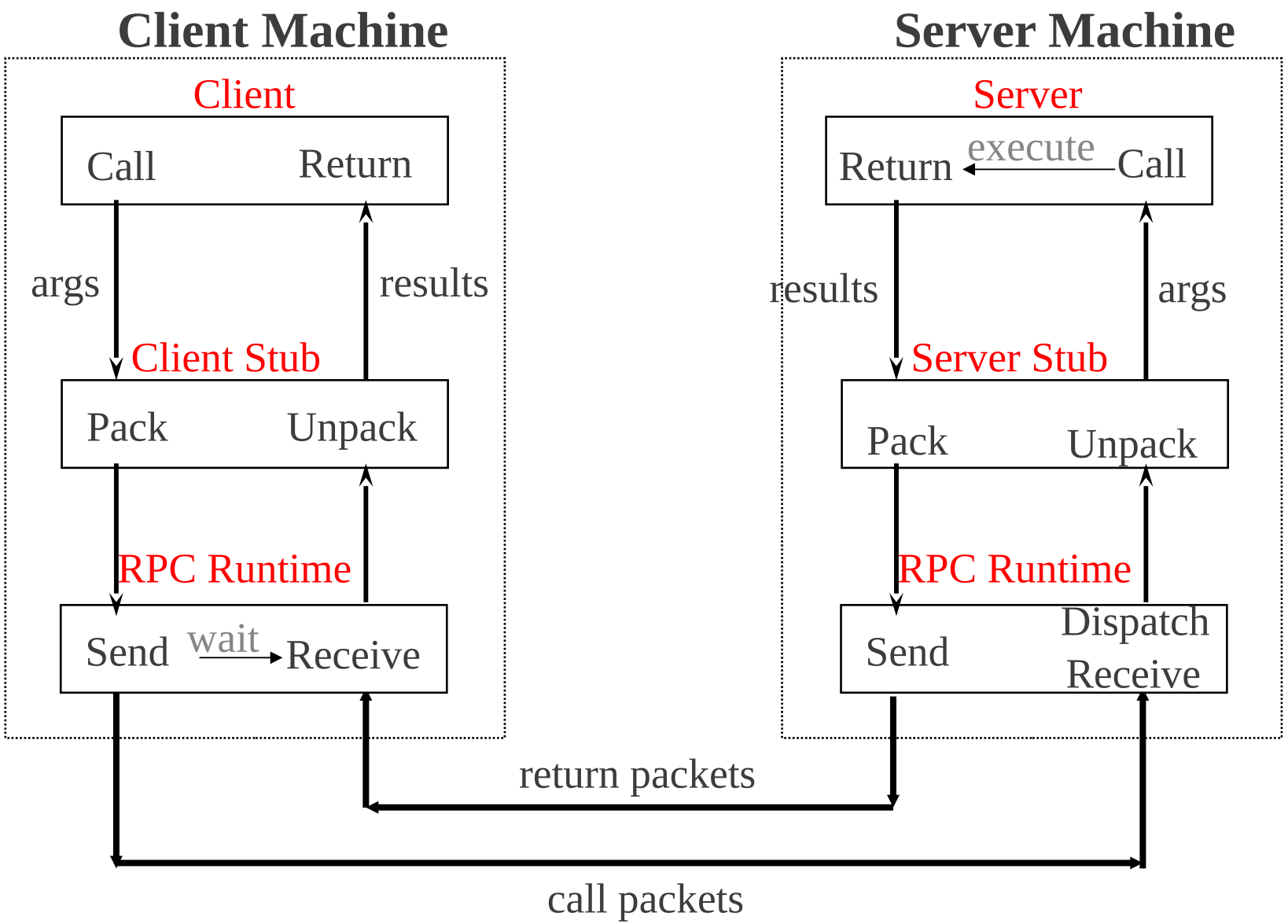
Why call-by-value?

- *call by copy-in/copy-out:* pass value of the variable pointed to; copy the returned value back into the variable

```
void myfunc (int *x, int *y) { (*x)++; (*y)++; }
main() { int a = 1; myfunc (&a, &a);}
```

  - normal execution: value of a after the call will be 3
  - if myfunc is a remote procedure, to which arguments are passed by copy-in/copy-out, the result will be 2!

RPC: Implementation



Client stub functions

- Pack the arguments into a message (also known as parameter marshalling)
- Send the message to the server and wait for reply
- Upon reply, extract the return value from the message
- Do a normal return

Client stub

- To generate the client stub, one needs to know the argument and return value types
- Client program will have one stub for each remote procedure

Server stub functions

- Wait for a message
- Extract information about which server procedure has been called (one server may ``export'' several procedures)
- Unpack the parameters and call the requested procedure
- Upon procedure return, pack the return value in a message
- Send message back to client

Server stub

- To generate the server stub, one needs to know the parameter and return value types of the exported procedure
- Typically there is one server stub per exported procedure; sometimes a single stub may serve many procedures

RPC runtime functions

- Relieves client/server stubs from handling low-level network communication
- Handles transmission of messages across the network between client and server
- Performs the necessary application-level retransmissions, acknowledgements, encryption etc

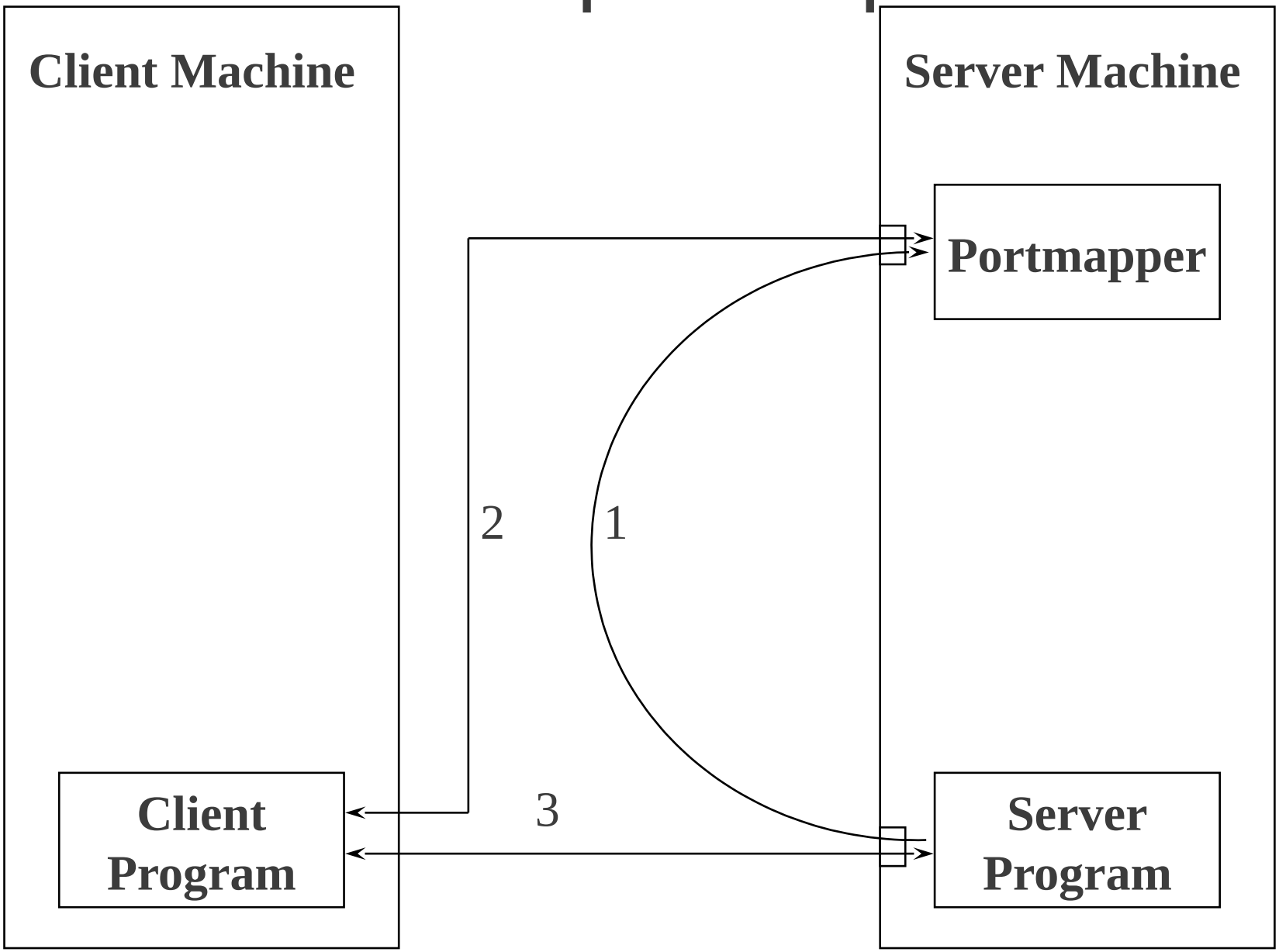
RPC: Client-Server binding

- Server exports the interface name and registers the service port with the portmapper (binder);
- Client imports the interface and performs a lookup with the portmapper to get the service port
- Client and server open a communication path to execute remote procedures

Portmapper

- Portmapper may be available at a fixed host address or client/server may use broadcast to locate the binder
- Binding may be at compile-time, link-time or call-time

RPC portmap



Protocol port (16 bit) ←→ RPC port (32 bit)

RPC: Crash recovery

- Server crash: retransmit request
  - procedure may get executed twice when server recovers!
  - at-least-once or at-most-once semantics
  - stateful v/s stateless servers
- Client crash: orphan computation
  - computation being done in a server on behalf of such a client is useless

Sun RPC

- First commercial implementation of RPC
- data representation format known as XDR (eXternal Data Representation)
- parameter passing only by copy-in (no copy-out); Return value is the only way of passing information back
- at least once semantics, no handling of orphans
- portmapper allows each RPC program to choose an arbitrary unused port

The XDR library

- The XDR library provides a conventional way of translating built-in C data types and pointer-based structures into an external bit-string representation
- The XDR library has primitive and complex filters

Primitive XDR filters

xdr_int

xdr_char

xdr_long

xdr_float

xdr_void

xdr_enum

Complex XDR filters

xdr_array : variable-length array with arbitrary element size

xdr_bytes : variable-length array of bytes

xdr_opaque : Fixed length data (uninterpreted)

xdr_reference : Object references

xdr_string : Null-terminated character arrays

Parameter marshalling

- standard representation for basic data types such as int, char, float..
- standard ways of packing compound types
  - For example: arrays will be packed as: size, e_0, e_1, e_2,…
- given these conventions, the stubs on either side can pack/unpack information correctly

RPC programming

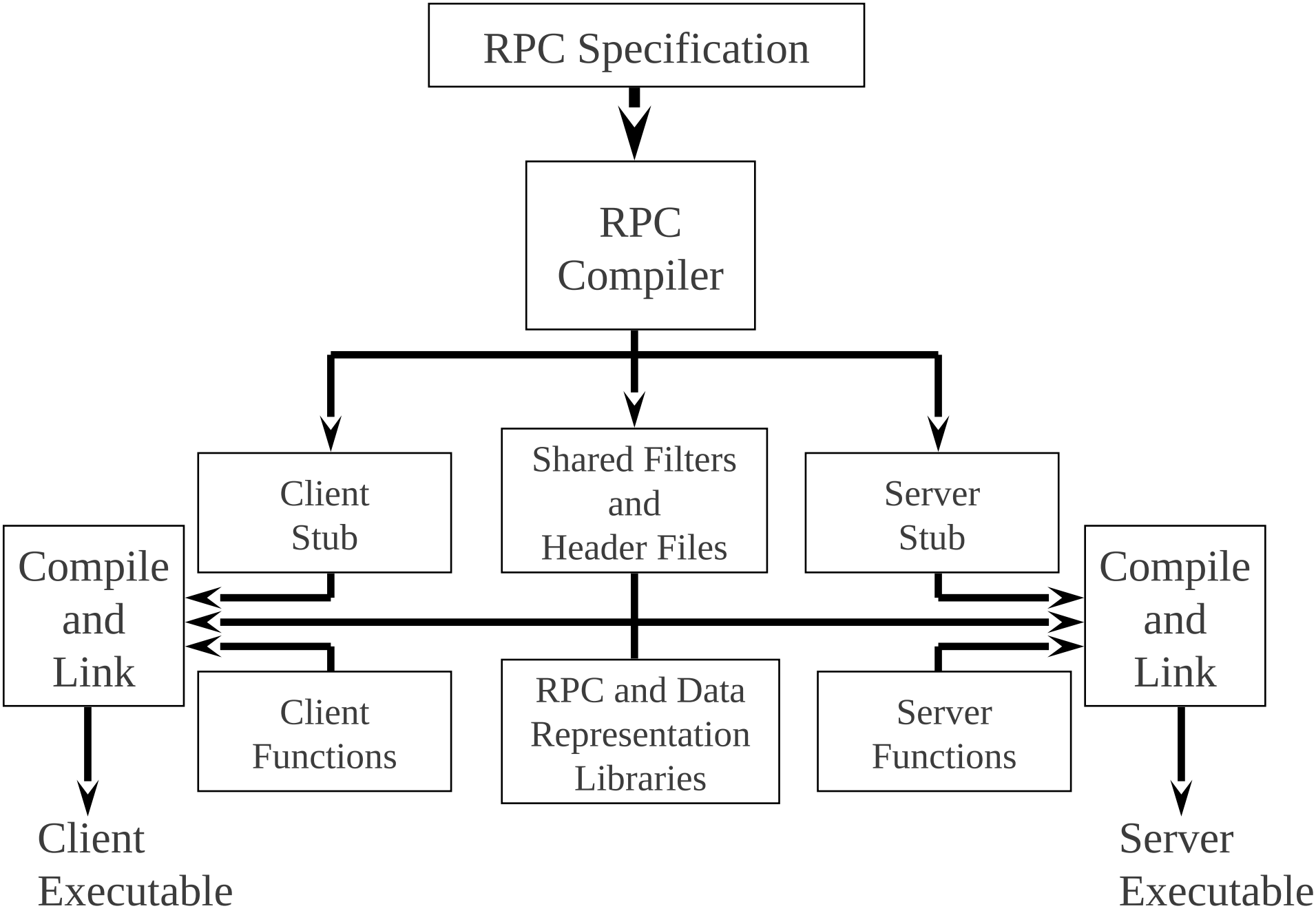
- Using library stubs: Stubs may be provided as part of the RPC RunTime library or may be created by a user
- Using RPCGEN: RPCGEN compiler generates client and server stubs automatically
- Most of the programming is done using RPCGEN

RPCGEN: Protocol compiler

- Not much is gained if the programmer has to write the client and the server stubs
- Protocol specification (procedure names, argument types, return type etc.) is specified in RPCGEN language (IDL)
- RPCGEN generates client, server stubs, and XDR routines which do parameter packing and unpacking

Using RPCGEN

- Programmer has to write:
  - Client main program
  - Implementation of the procedure (for the server)
  - Compile the client program with the client stub, and procedure implementation with the server stub, into two independent client and server programs


RPC Specification
RPC Compiler
Client Stub
Shared Filters and Header Files
Server Stub
Compile and Link
Compile and Link
Client Functions
RPC and Data Representation Libraries
Server Functions
Client Executable
Server Executable

Compiling with rpcgen

rpcgen file.x

client stub: file_clnt.c

server stub: file_svc.c

XDR filters: file_xdr.c

RPC Example: StatelessFS.x

```
/* Interface definition for a stateless file service */
const File_Name_Size = 16
const Buffer_Size = 1024

typedef string FileName<File_Name_Size>;
typedef long Position;
typedef long Nbytes;

struct Data { long n; char buffer[Buffer_Size]; };
struct readargs {FileName filename; Position position; Nbytes
    n; };
struct writeargs {FileName filename; Position position; Data
    data; };
```

StatelessFS.x (contd)

```
…
program Stateless_FS {
    version Stateless_FS_Vers {
            Data READ (readargs) = 1;
            /* first procedure */
            Nbytes WRITE (writeargs) = 2;
            /* second procedure */
    } = 1; /* version number */
} = 0x20000000; /* unique identifier; program
    number */
```

RPC Example: client.c

/* Client program for stateless file service */

```
#include <stdio.h>
#include <rpc/rpc.h>
#include “Stateless_FS.h”
main (argc, argv) int argc; char **argv;
{
    CLIENT   *client_handle;
    char          *server_host_name = “akash”;
    readargs     read_args;
    writeargs   write_args;
    Data         *read_result;
    Nbytes     *write_result;
```

client.c (contd)

```
…
client_handle = clnt_create (server_host_name,
   Stateless_FS, Stateless_FS_Vers, “udp”); /* Get a
   client handle; creates a socket */
if (client_handle == NULL) {
   clnt_pcreateerror (server_host_name);
   return(1); /* Cannot contact server */
};
```

client.c (contd)

```
    ....
    /* Prepare parameters and make an RPC to read procedure */
    read_args.filename = “example”;
    read_args.position = 0;
    read_args.n = 500;
    read_result = read_1 (&read_args, client_handle);
    ....
    /* Similar code for RPC to write procedure */
    ....
    clnt_destroy (client_handle); /* destroy client handle; close
      socket */
}
```

Client calls

- clnt_create():

  CLIENT *clnt_create(host, prognum, versnum, protocol)

  - creates the CLIENT structure for the specified server host, program, and version number
  - can use tcp or udp protocol
- clnt_destroy():
  - deallocates the memory associated with CLIENT

Client calls (contd)

- clnt_control():
  - used to change or retrieve the characteristics of an RPC, such as the timeout value, socket descriptors, and status
- clnt_call():
  - used by the client to make an RPC call
  - It is blocking and uses a timeout value

RPC Example: server.c

```
/* Server program for stateless file service */
#include <stdio.h>
#include <rpc/rpc.h>
#include “Stateless_FS.h”

/* READ PROCEDURE */
Data *read_1 (args)
    readargs *args;
{
 static Data result; /* Must be declared as static */
```

server.c (contd)

```
    /* Statements for reading args.n bytes from the file
       args.filename starting from position args.position,
       and for putting the data read in &result.buffer and
       the actual number of bytes read in result.n */

    return (&result); /* Return the result as a single
       argument */
}
```

server.c (contd)

```
....
/* WRITE PROCEDURE */
Nbytes *write_1 (args)
    writeargs *args;
{
    static Nbytes result;
     /* Statements for writing args.data.n bytes from the buffer
        &args.data.buffer into the file args.filename starting from
        position args.position,  and for putting the actual number of
        bytes written in result */
     return (&result);
}
```

Server calls

- svctcp_create():

  SVCXPRT *svctcp_create(sock, sendz, recvz)

  - Creates and returns a TCP RPC service transport at a particular socket
  - TCP-based RPC uses buffered I/O
- svcudp_create():

  SVCXPRT *svcudp_create(sock);

  - Creates and returns a pointer to a UDP/IP- based server transport

Generating the Client

- Compile the client code:
  - cc -c -o client.o -g client.c
- Compile the client stub:
  - cc -c StatelessFS_clnt.c
- Compile the XDR filters:
  - cc -c StatelessFS_xdr.c
- Build the client executable:
  - cc -o Client client.o StatelessFS_clnt.o StatelessFS_xdr.o

Generating the Server:

- Compile the service procedures:
  - cc -c -o server.o server.c
- Compile the server stub:
  - cc -c StatelessFS_svc.c
- Build the server executable:
  - cc -o Server server.o StatelessFS_svc.o StatelessFS_xdr.o

NFS: Network File System

- Distributed file system that provides transparent access to remote disks
- Independent of architecture, operating system and network transport protocol
- Emulates the UNIX file system
- Built using the RPC/XDR mechanism

- Introduced by SUN in 1985; de facto standard since 1989

NFS: Design goals

- Access Transparency:
  - NFS client module provides an API that is identical to the local operating system's interface
  - No modifications required to existing programs to enable them to operate with remote files

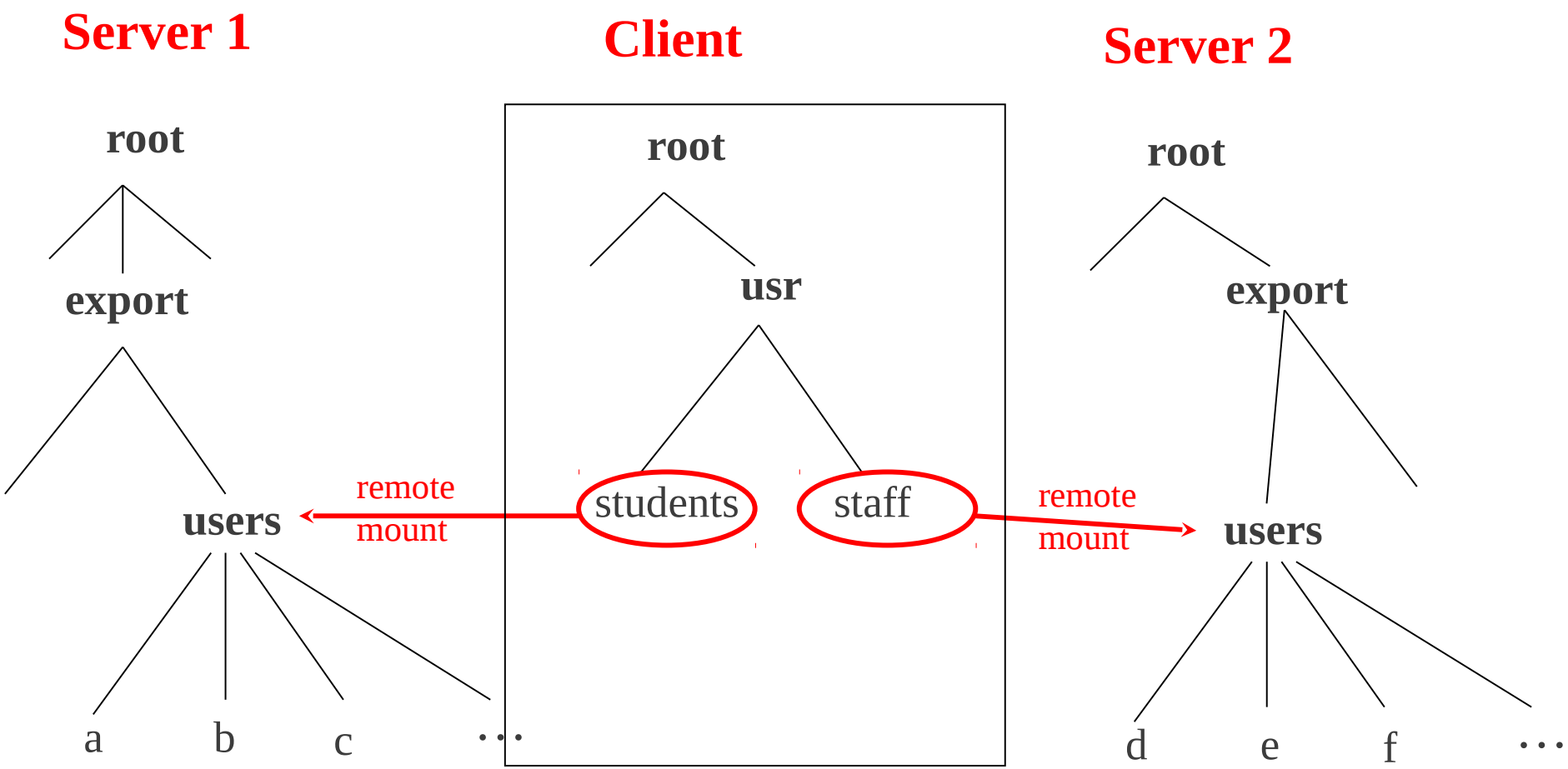
NFS: Design

- Location Transparency:
  - Each client adds the remote file system to its local name space
  - File systems are *exported* by the server and *remote-mounted* by a client before they can be accessed by client processes

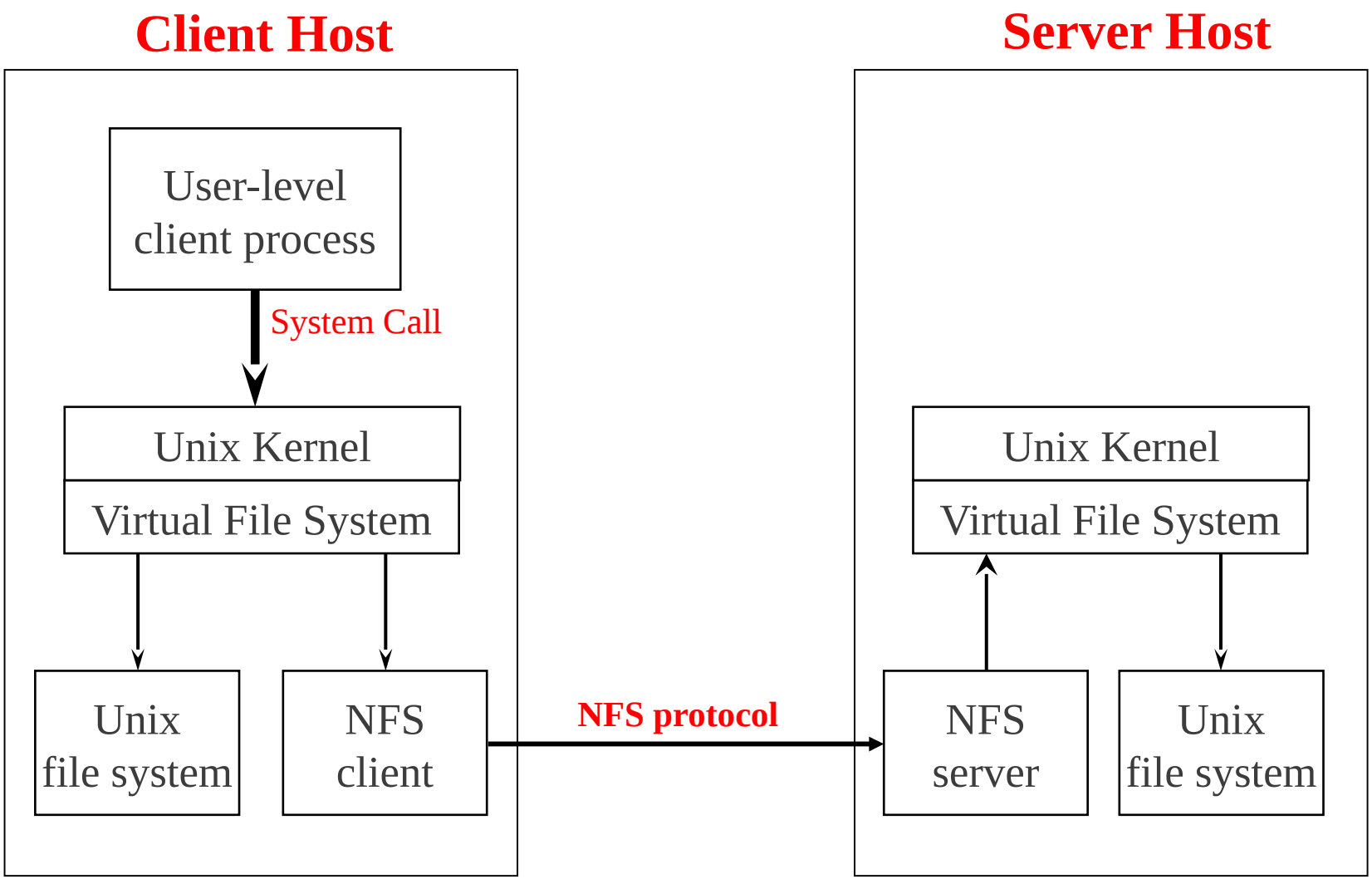
NFS: Design

- Failure Transparency:
  - NFS service is *stateless* and the file access operations are *idempotent*
  - UNIX file operations are translated into NFS calls by client module
- Performance Transparency:
  - Server side caching using the Unix disk block caching mechanism
  - Client side caching by maintaining local cache of blocks from remote files

NFS application view



NFS Architecture



NFS (Unix domain)

- Important files:
  - /etc/exports: access control list for exporting
  - /etc/fstab: contains entries for remote directories and mount points for them
- Important programs:
  - portmap: facilitates initial connection
  - rpc.mountd
  - rpc.nfsd

NFS implementation

- rpc.mountd:
  - answers NFS access requests
  - answers file system mount requests
  - reads the /etc/exports file to determine which file systems to mount
- rpc.nfsd:
  - handles client file system requests
  - allows the clients read-only and read-write access to file hierarchy of the server machine

Topics NOT covered

- RPC programming without using rpcgen
- Details of NFS
- NIS (Network Information Service)
- Other RPC based applications