Networking

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, Kurose, Ross, E. Sirer, R. Van Renesse]

Application
Transport
Network
Link
Physical

Application Layer

Several figures in this section come from
“Computer Networking: A Top Down Approach”
by Jim Kurose, Keith Ross

Naming

People

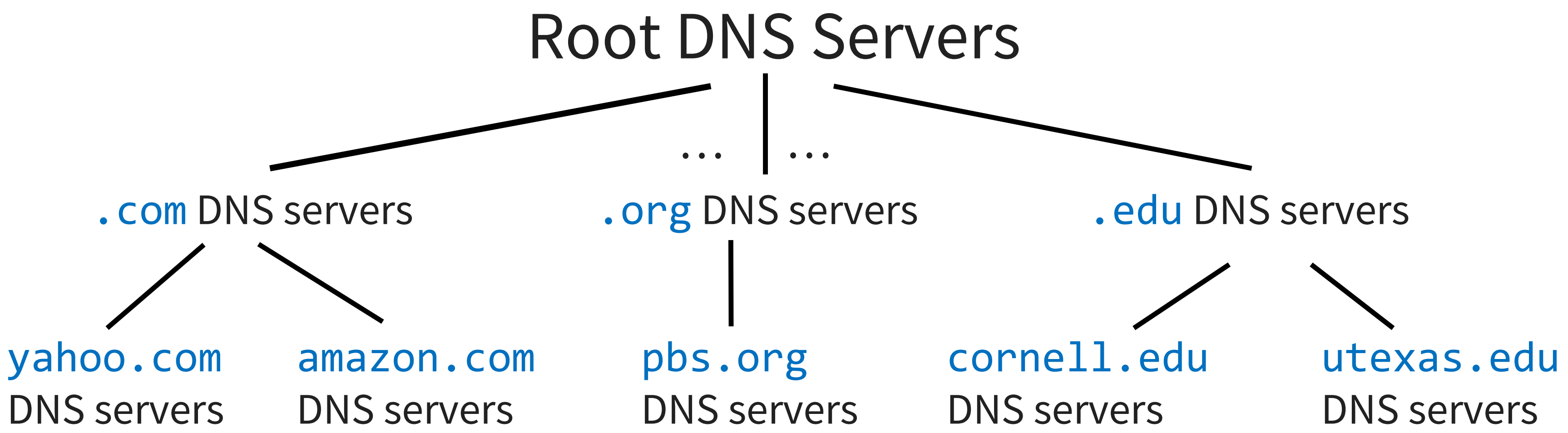
- SSN, NetID, Passport #

Internet Hosts, Routers

1. IP address (32 bit), `151.101.117.67`
   - For now, 32-bit descriptor, like a phone number
   - Longer addresses in the works…
   - Assigned to hosts by their internet service providers
   - **Not physical:** does not identify a single node, can swap machines and reuse the same IP address
   - **Not entirely virtual:** determines how packets get to you, changes when you change your ISP
2. Virtual: "name" `www.cnn.com`
   - Used by humans (no one wants to remember a bunch of #s)

How to convert hostname to IP address?

Domain Name System (DNS)



Distributed, Hierarchical Database

- Application-Layer Protocol: hosts & name servers communicate to resolve names
- Names are separated by dots into components

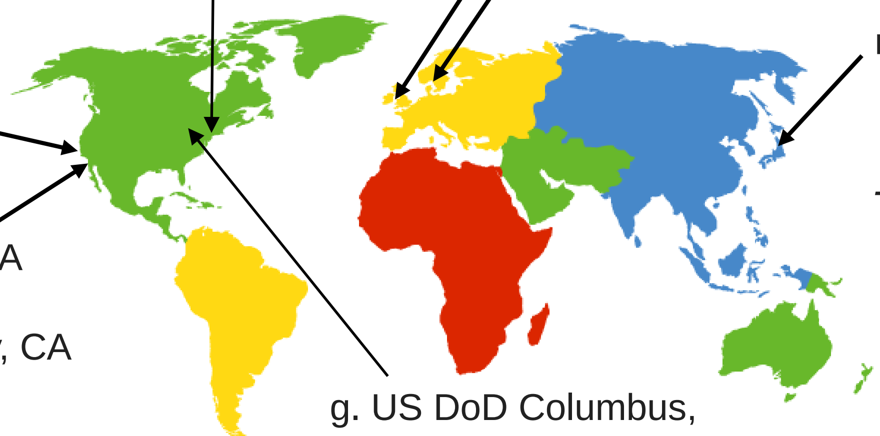
  Not to be confused with dots in IP addresses (in which the order of least significant to most significant is reversed)

- Components resolved from right to left
- All siblings must have unique names
- Lookup occurs from the top down

DNS: root name servers

Contacted by local name server that cannot resolve name

- owned by Internet Corporation for Assigned Names & Numbers (ICANN)
- contacts authoritative name server if name mapping not known
- gets mapping
- returns mapping to local name server



13 root name “servers” worldwide

DNS Lookup

1. the client asks its local nameserver
2. the local nameserver asks one of the *root nameservers*
3. the root nameserver replies with the address of the authoritative nameserver
4. the server then queries that nameserver
5. repeat until host is reached, cache result.

Example: Client wants IP addr of www.amazon.com

1. Queries root server to find com DNS server
2. Queries `.com` DNS server to get `amazon.com` DNS server
3. Queries `amazon.com` DNS server to get IP address for `www.amazon.com`

DNS Services

Simple, hierarchical namespace works well

- Can name anything
- Can alias hosts
- Can cache results
- Can share names (replicate web servers by having 1 name correspond to many IP addresses)

Q: Why not centralize?

- Single point of failure
- Traffic volume
- Distant Centralized Database
- Maintenance

A: Does not scale!

What about security? (don’t ask!)

Application Layer

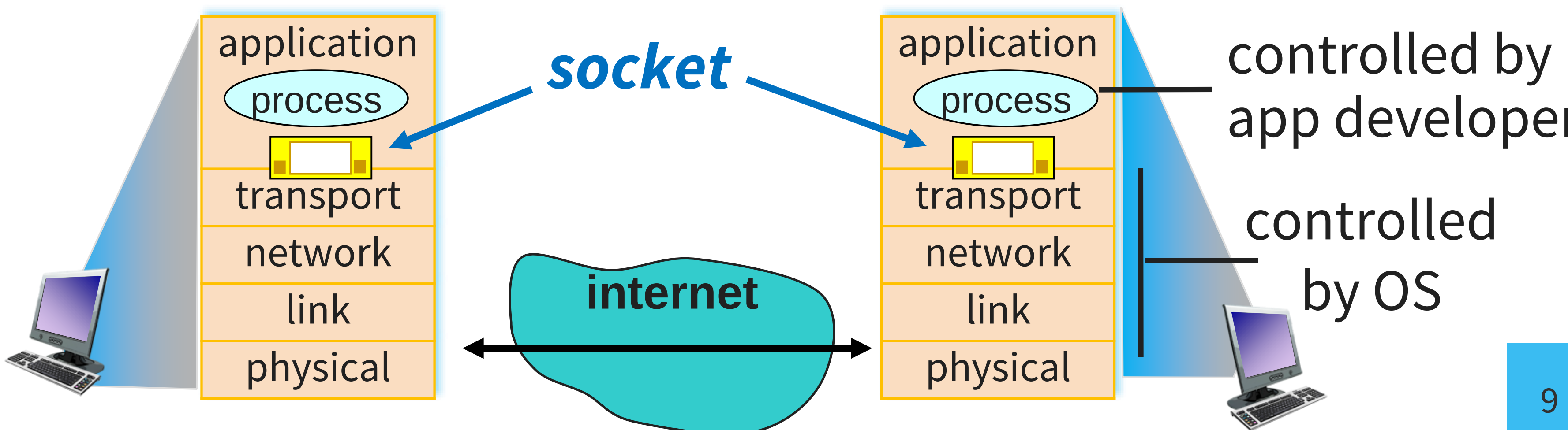
- Network-aware applications
  - Clients & Servers
  - Peer-to-Peer

Sockets

"Door" between application process and end-end-transport protocol

Sending process:

- shoves message out door
- relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Socket programming

Two socket types for two transport services:

- UDP: unreliable datagram
- TCP: reliable, byte stream-oriented

Host could be running many network applications at once. Distinguish them by binding the socket to a **port number**:

- 16 bit unsigned number
- 0-1023 are well-known
  (web server = 80, mail = 25, telnet = 23)
- the rest are up for grabs

Application Example

1. Client reads a line of characters (data) from its keyboard and sends data to server
2. Server receives the data and converts characters to uppercase
3. Server sends modified data to client
4. Client receives modified data and displays line on its screen

Socket programming with UDP

No “connection” between client & server

- no handshaking before sending data
- **Sender:** explicitly attaches destination IP address & port # to each packet
- **Receiver:** extracts sender IP address and port # from received packet

Data may be lost, received out-of-order

Application viewpoint: UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server

Client/server socket interaction: UDP

Server (running on serverIP)

create serversocket, bind to port x

read data (and clientAddr) from serversocket

modify data

send modified data to clientAddr via serversocket

Client

create clientsocket

create message

send message to (serverIP, port x) via clientsocket

receive message (and serverAddr) from clientsocket

close clientsocket

Python UDP Client

```
import socket     #include Python’s socket library
serverName = ‘servername’
serverPort = 12000

#create UPD socket
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#get user input
message = input('Input lowercase sentence: ')

# send with server name + port
clientSocket.sendto(message.encode(), (serverName, serverPort))

# get reply from socket and print it
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())

clientSocket.close()
```

Python UDP Server

```
import socket   #include Python’s socket library
serverPort = 12000

#create UPD socket & bind to local port 12000
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")

while True:
    # Read from serverSocket into message,
    # getting client’s address (client IP and port)
    message, clientAddress = serverSocket.recvfrom(2048)
    print("received message: "+message.decode())
    modifiedMsg = message.decode().upper()
    print("sending back to client")

    # send uppercase string back to client
    serverSocket.sendto(modifiedMsg.encode(), clientAddress)
```

Socket programming w/ TCP

Client must contact server

Server:

- already running
- server already created "welcoming socket"

Client:

- Creates TCP socket w/ IP address, port # of server
- Client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* to communicate with that particular client
  - allows server to talk with multiple clients
  - source port #s used to distinguish clients

Application viewpoint: TCP provides reliable, in-order byte-stream transfer between client & server

Client/server socket interaction: TCP

Server (running on hostID)

create welcoming serversocket,
bind to port x

in response to connection request,
create connectionsocket

read data from connectionsocket

modify data

send modified data to clientAddr
via connectionsocket

close connectionsocket

Client

create clientsocket
connect to (hostID, port x)

create message

send message via clientsocket

receive message from clientsocket

close clientsocket

Python TCP Client

```
import socket     #include Python’s socket library
serverName = ‘servername’
serverPort = 12000

#create TCP socket for server on port 12000
clientSocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
clientSocket.connect((serverName,serverPort))

#get user input
message = input('Input lowercase sentence: ')

# send (no need for server name + port)
clientSocket.send(message.encode())

# get reply from socket and print it
modifiedMessage, serverAddress = clientSocket.recvfrom(1024)
print(modifiedMessage.decode())

clientSocket.close()
```

Python TCP Server

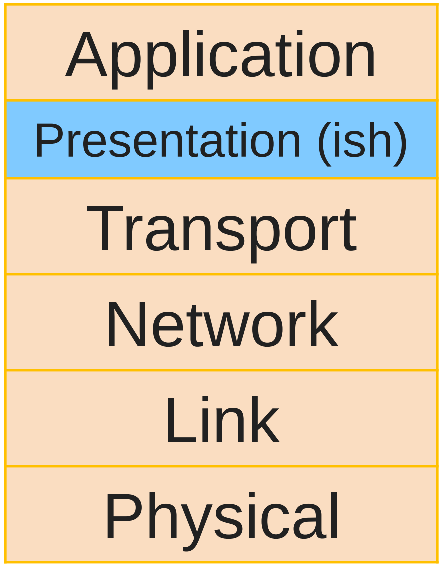
```
import socket   #include Python’s socket library
serverPort = 12000

#create TCP welcoming socket & bind to server port 12000
serverSocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
serverSocket.bind(('', serverPort))
#server begins listening for incoming TCP requests
serverSocket.listen(1)
print("The server is ready to receive")

while True:
    # server waits on accept() for incoming requests
    # new socket created on return
    connectionSocket, addr = serverSocket.accept()
    message = connectionSocket.recv(1024).decode()
    print("received message: "+message)
    modifiedMsg = message.upper()

     # send uppercase string back to client
    connectionSocket.send(modifiedMsg.encode())

    # close connection to this client, but not welcoming socket
    connectionSocket.close()
```



Remote Procedure Call

Several figures in this section come from
“Distributed Systems: Principles and Paradigms”
by Andrew Tanenbaum & Maarten van Steen

Client/Server Paradigm

Common model for structuring distributed computation

- ***Server:*** program (or collection of programs) that provide some *service*, e.g., file service, name service
  - may exist on one or more nodes
- ***Client:*** a program that uses the service

Typical Pattern:

1. Client first *connects* to the server: locates it in the network & establishes a connection
2. Client sends *requests*: messages that indicate which service is desired and the parameters
3. Server returns *response*

Pros and Cons of Messages

+Very flexible communication

- Want a certain message format? *Go for it!*

−Problems with messages:

- programmer must worry about message formats
- must be packed and unpacked
- server must decode to determine request
- may require special error handling functions

Procedure Call

A more natural way to communicate:

- every language supports it
- semantics are well defined and understood
- natural for programmers to use

Idea: Let clients call servers like they do procedures

Remote Procedure Call (RPC)

Goal: design RPC to look like a local PC

- A model for distributed communication
- Uses computer/language support
- 3 components on each side:
  - user program (client or server)
  - set of *stub* procedures
  - RPC runtime support

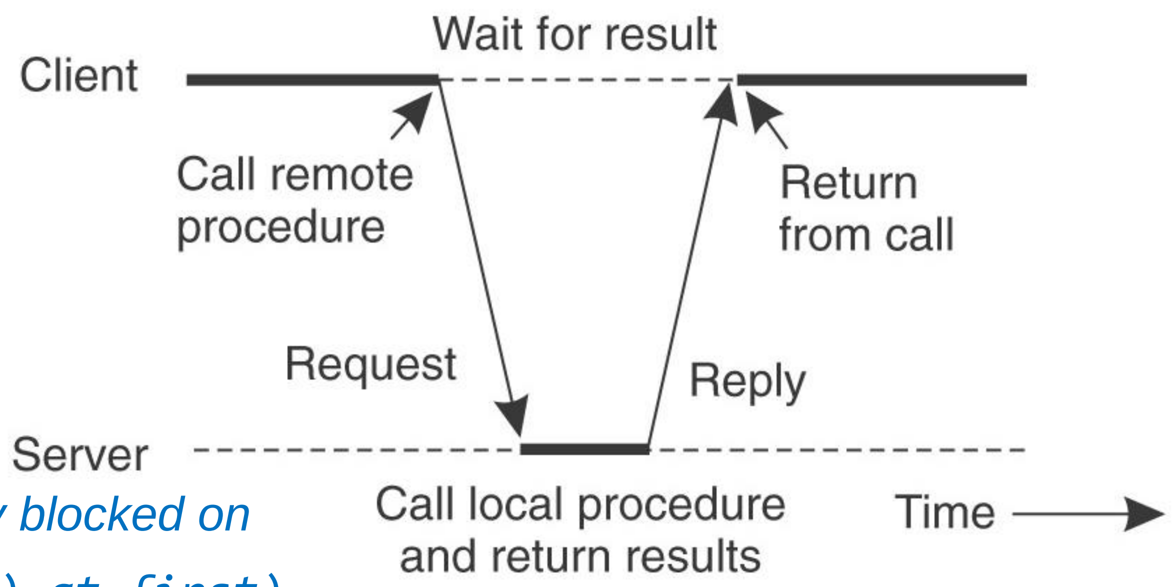
Birrell & Nelson @ Xerox PARC

"Implementing Remote Procedure Calls" (1984)

How does an RPC work?

Basic idea:

- Server *exports* a set of procedures
- Client calls these procedures, as if they were local functions

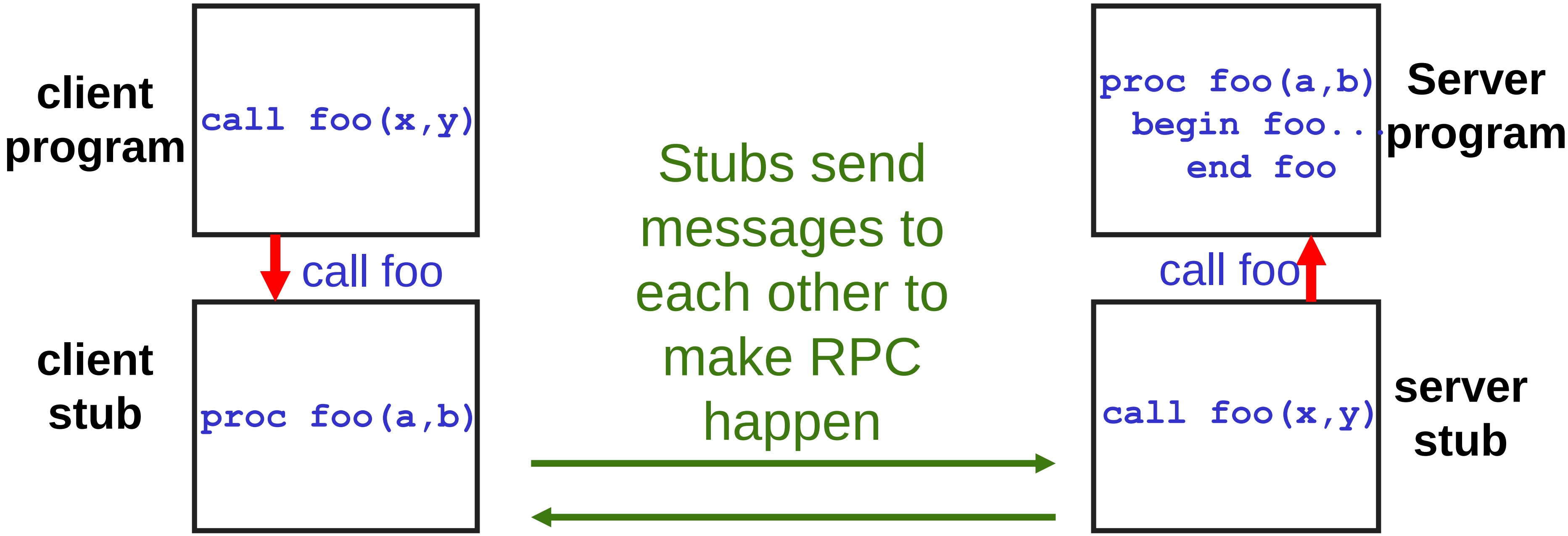


(typically blocked on `receive() at first`)

- Message passing details hidden from client & server (like system call details are hidden in libraries)

[Tanenbaum & van Steen, Fig 4-6]

RPC Stubs



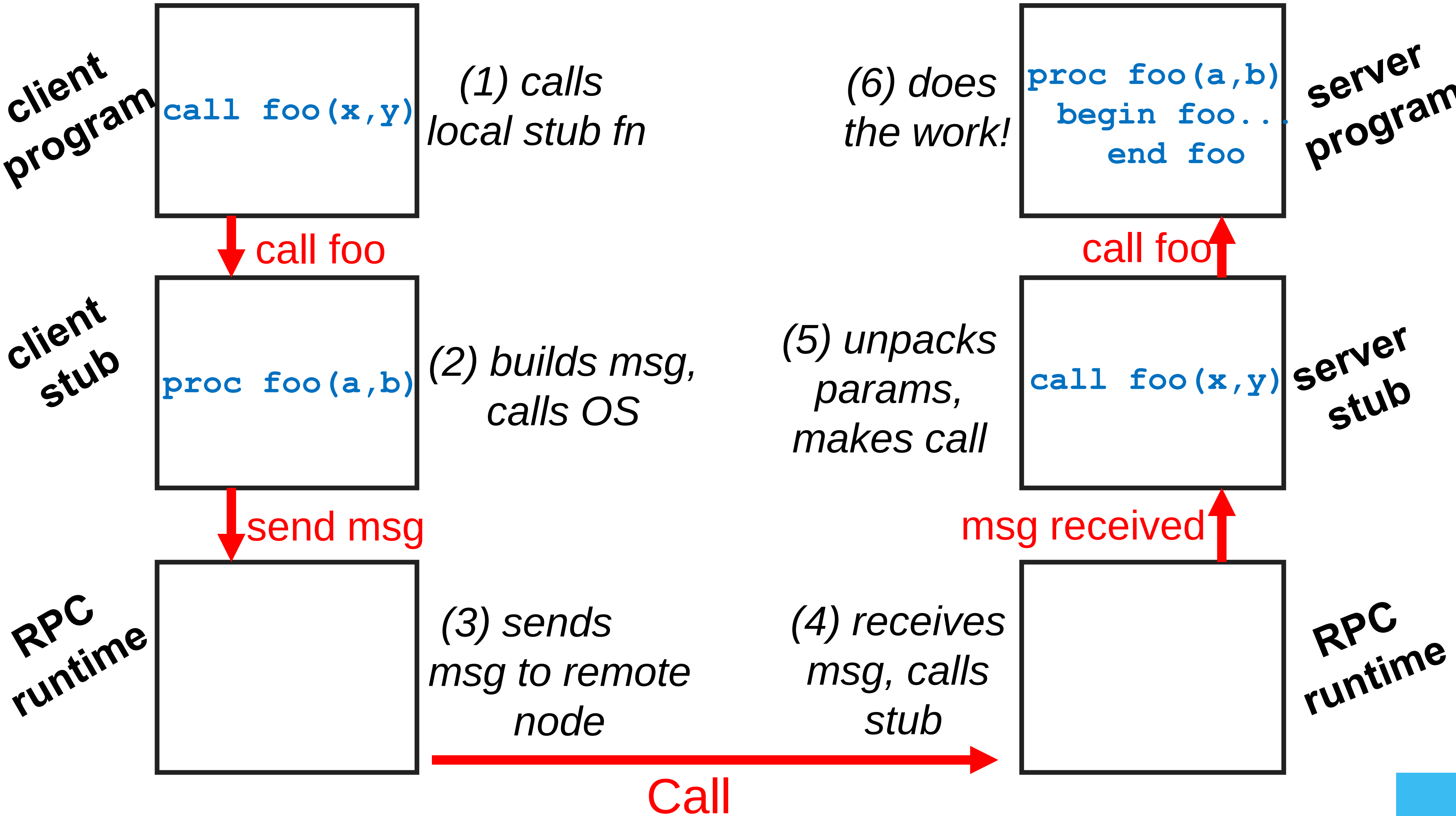
Client-side stub:

- Looks (to the client) like a callable server procedure
- Client program thinks it is calling the server

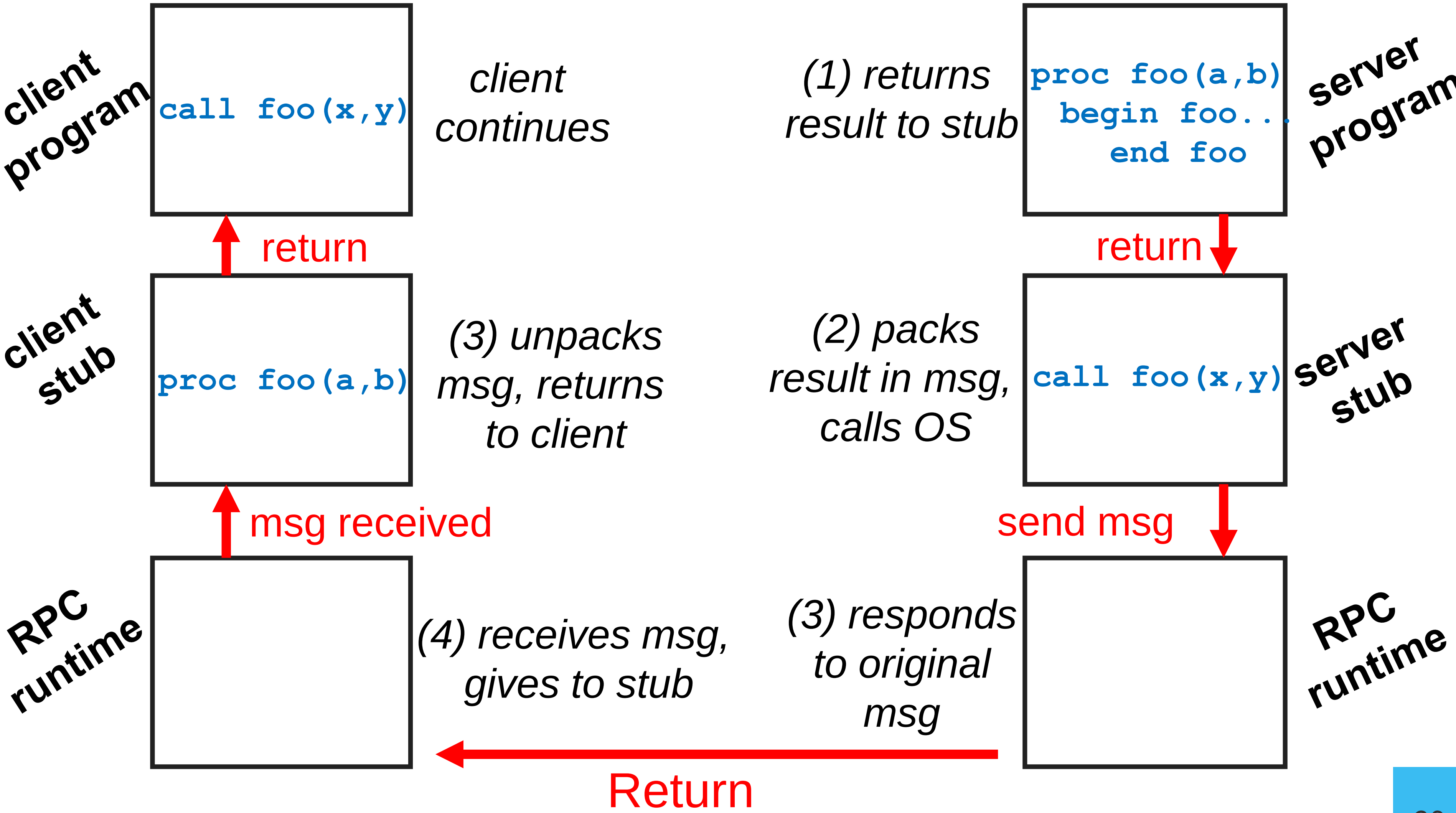
Server-side stub:

- Server program thinks it is called by the client
- `foo` actually called by the server stub

RPC Call Structure



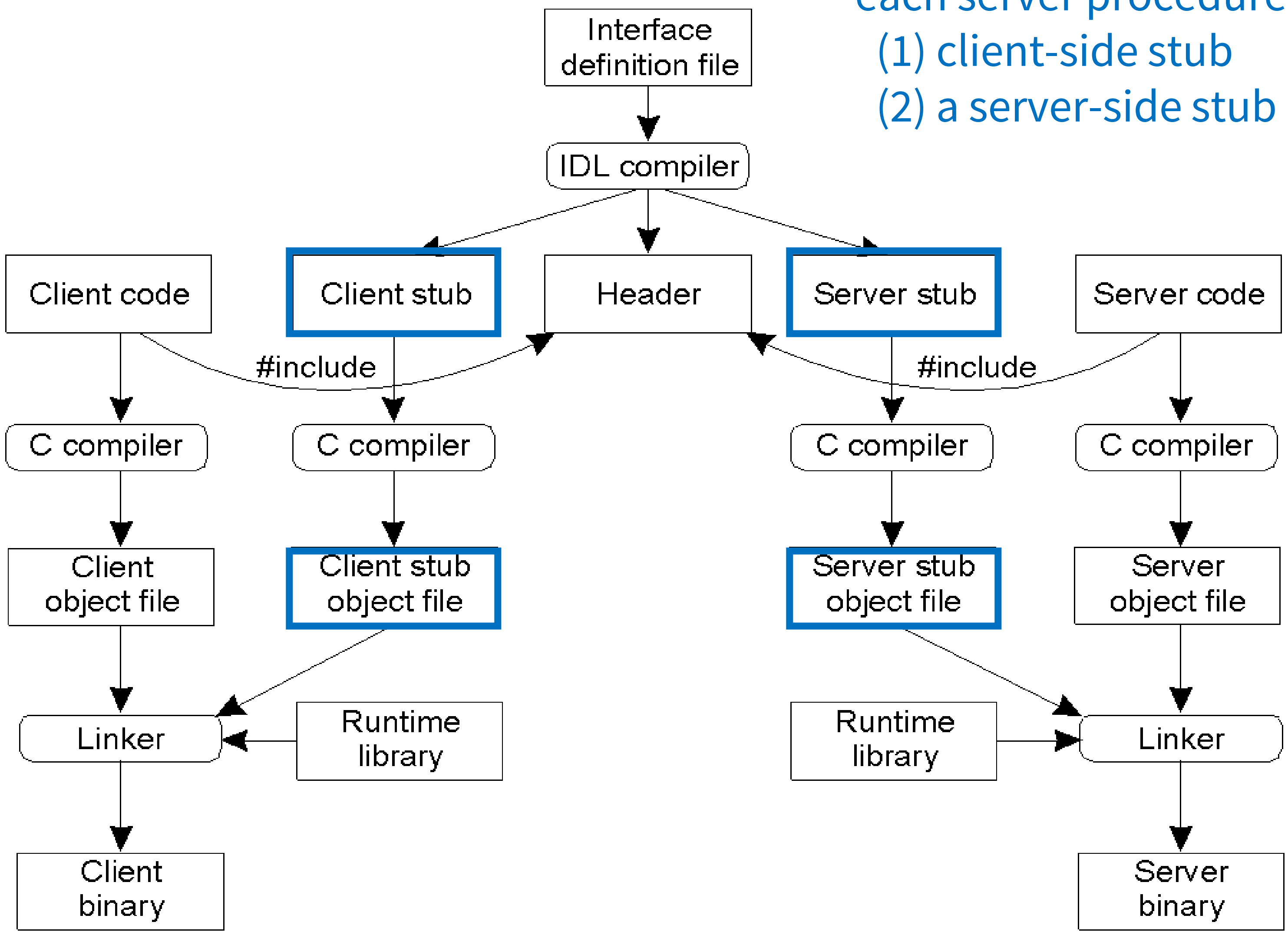
RPC Return Structure



Example RPC system:

Stub compiler

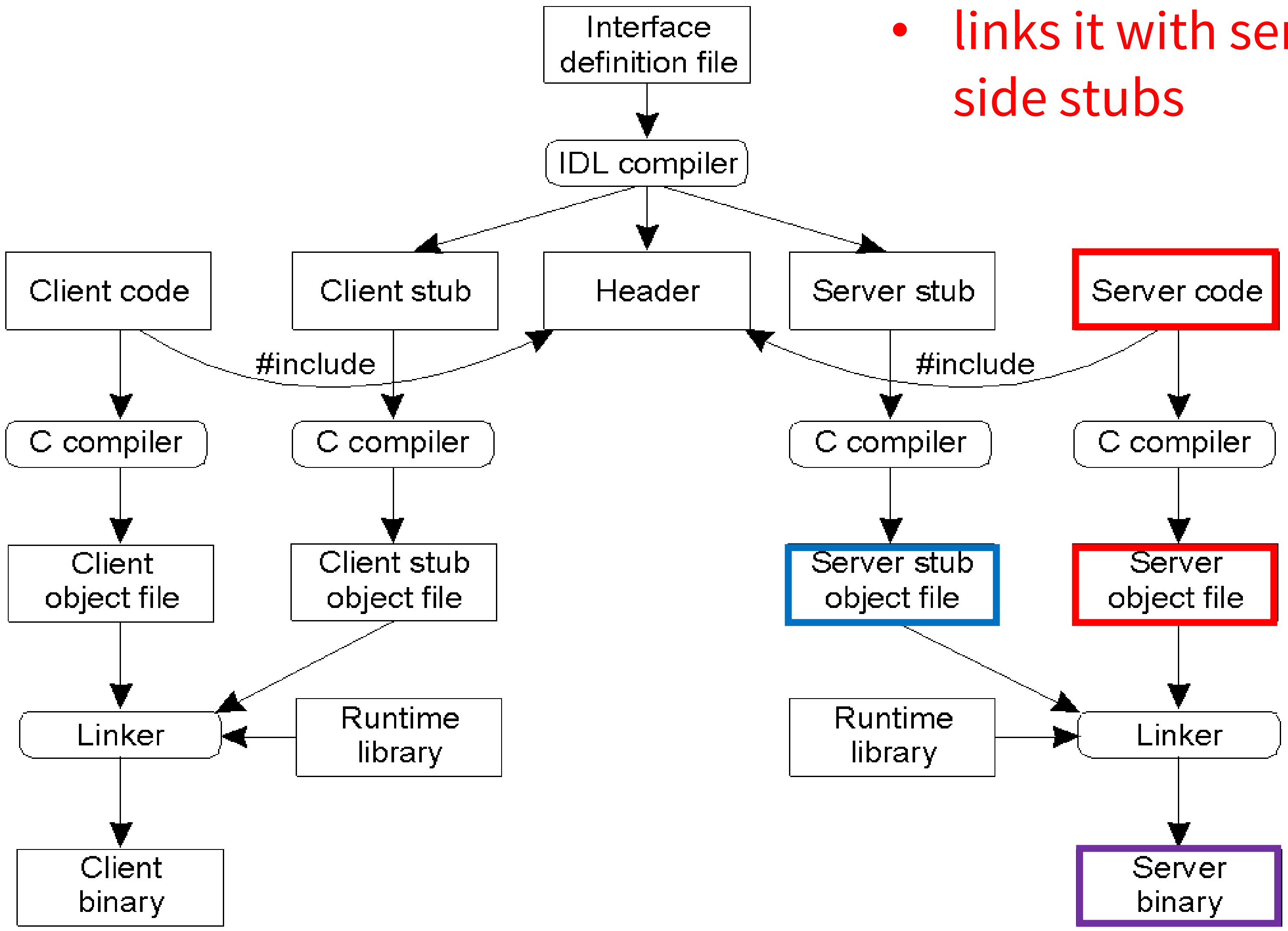
- reads IDL
- produces 2 stub procedures for each server procedure:
  (1) client-side stub
  (2) a server-side stub



Example RPC system:

Server writer:

- writes server
- links it with server-side stubs



Binding: Connecting Client & Server

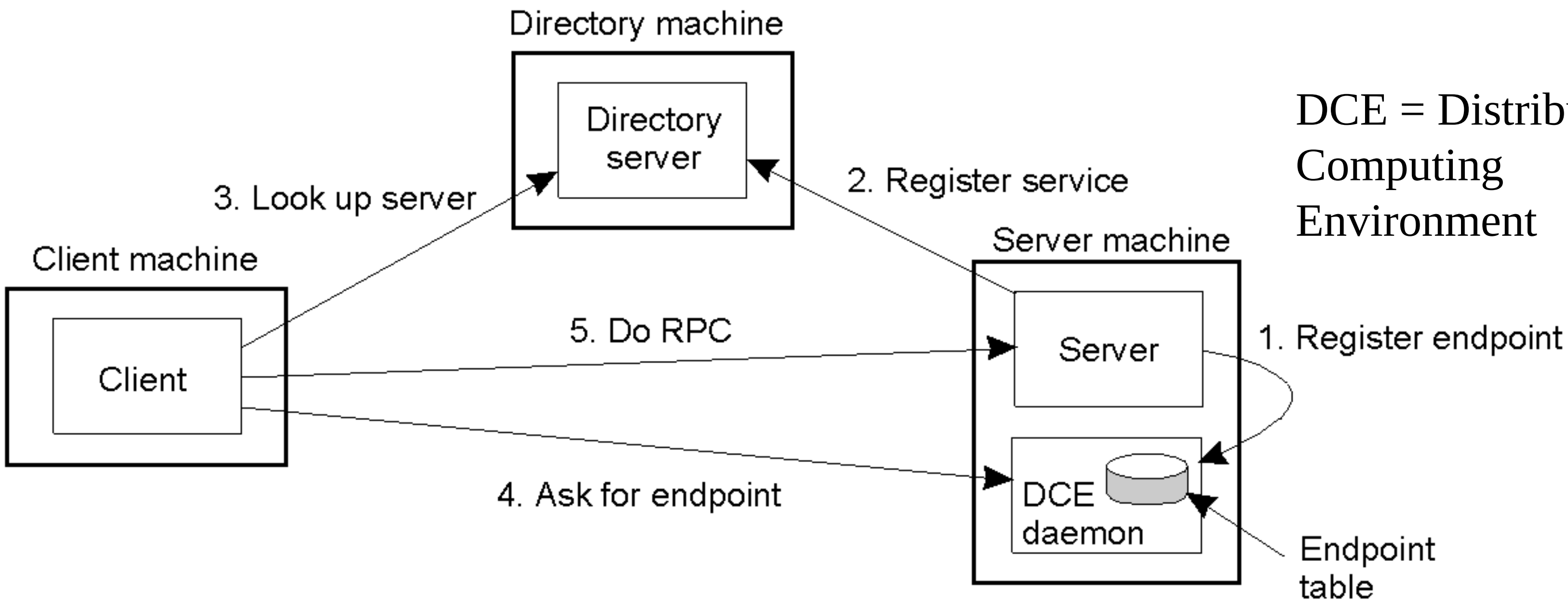
Server *exports* its interface:

- identifying itself to a network name server
- telling the local runtime its dispatcher address

Client *imports* the interface. RPC runtime:

- looks up the server through the name service
- contacts requested server to set up a connection

Import and *export* are explicit calls in the code



DCE = Distributed Computing Environment

RPC Concerns

- Parameter Passing
- Failure Cases
- Performance

Your function call has been secretly replaced with a remote function call. Is this okay?

RPC Marshaling

Packing parameters into a message packet

- RPC stubs call type-specific procedures to marshal (or unmarshal) all of the parameters to the call

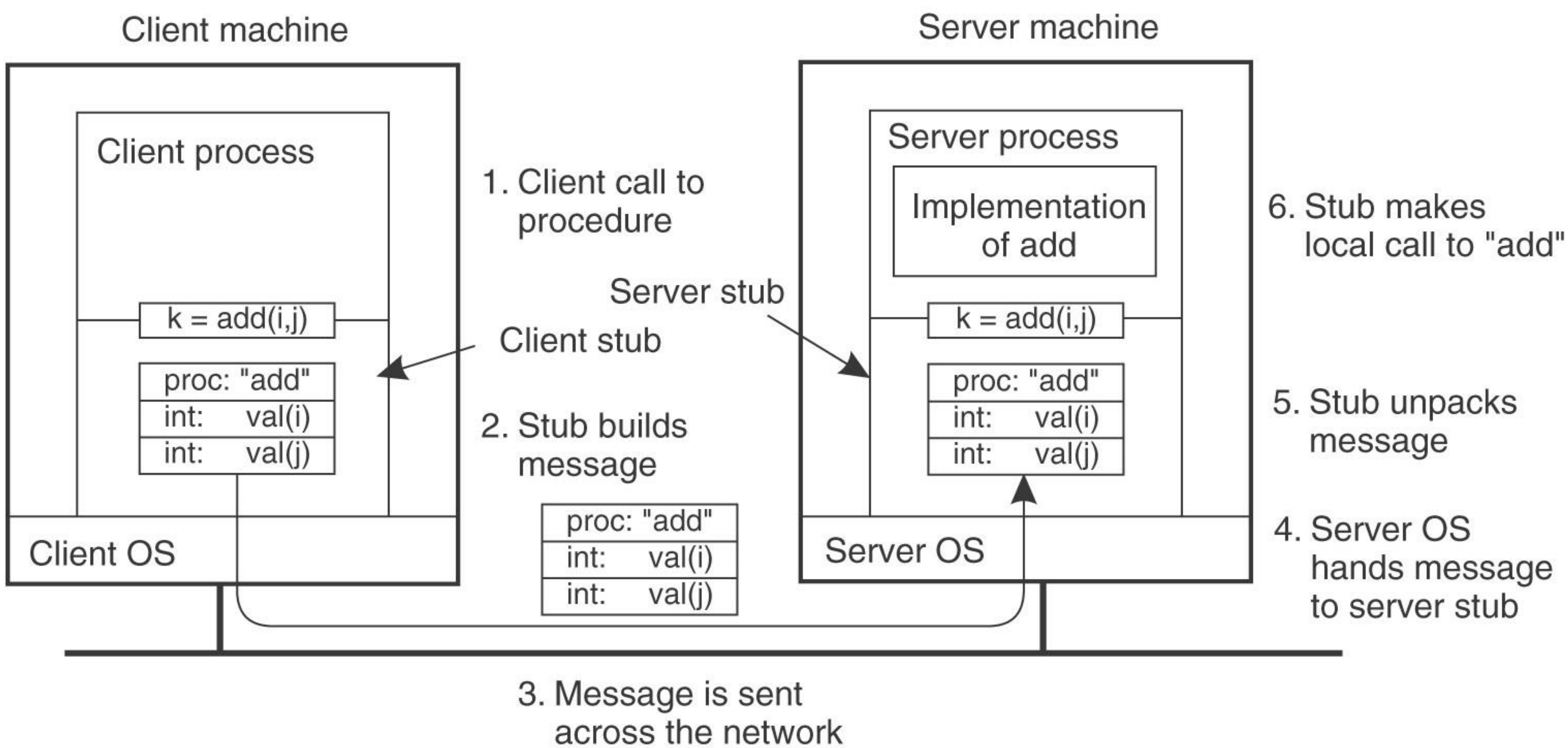
On Call:

- Client stub marshals parameters into the call packet
- Server stub unmarshals parameters to call server's fn

On return:

- Server stub marshals return values into return packet
- Client stub unmarshals return values, returns to client

Parameter Passing



What could go wrong?

[Tanenbaum & van Steen, Fig 4-7]

RPC Concerns

- Parameter Passing
  - Data Representation
  - Passing Pointers
  - Global Variables
- Failure Cases
- Performance

Data Representation

Data representation?

ASCII vs. Unicode, structure alignment, n-bit machines, floating-point representations, endian-ness

→Server program defines interface using an *interface definition language* (IDL)

For all client-callable functions, IDL specifies:

- names
- parameters
- types

Passing Pointers

- Forbid pointers? (breaks transparency)
- Have server call client and ask it to modify when needed (breaks transparency)
- Have stubs replace call-by-reference semantics with Copy/Restore
  - Optimization: if stub knows that a reference is exclusively input/output copy only on call/return
  - Only works for simple arrays & structures
    - Union types? YUCK
    - Multi-linked structures? YUCK
    - Raw pointers? YUCK

RPC Concerns

- Parameter Passing
- Failure Cases
- Performance

RPC Failure Cases

Function call failure cases:

- Called fn crashes → so does the caller

RPC Failure cases:

- server fine, client crashes? (orphans)
- client fine, server crashes?
  - Client just hangs?
  - Stub supports a timeout, error after n tries?
  - Client deals w/failure (breaks transparency)

Aside: Idempotency

Multiple calls yields the same result

What’s idempotent?

- read block 50

What’s not?

- appending to a file

How many times will a function be executed?

A calls B. B never responds… Should A resend or not?

2 Possibilities:

(1) B never got the call:

- Resend → B executes the procedure *once*
- Don't resend → B executes the procedure *zero times*

(2) B performed the call then crashed:

- Resend → B executes the procedure *twice*
- Don't resend → B executes the procedure *once*

Can we even promise transparency?

What semantics will RPC support?

A calls B. B responds… What does A assume about how many times the function was executed?

Exactly once:

- system guarantees local semantics
- at best expensive, at worst, impossible

At-least-once:

\+ easy: no response? A re-sends

− only works for idempotent functions

− server operations must be stateless

At-most-once:

− requires server to detect duplicate packets

\+ works for non-idempotent functions

RPC Concerns

- Parameter Passing
- Failure Cases
- Performance
  - Remote is not cheap
  - Lack of parallelism (on both sides)
  - Lack of streaming (for passing data)

RPC Concluding Remarks

RPC:

- Common model for distributed application communication
- ***language support*** for distributed programming
- relies on a *stub compiler* & IDL server description
- commonly used, *even on a single node*, for communication between applications running in different address spaces (most RPCs *are* intra-node!)

"Distributed objects are different from local objects, and keeping that difference visible will keep the programmer from forgetting the difference and making mistakes."

–Jim Waldo+, "A Note on Distributed Computing" (1994)