Remote Procedure Calls

ARCOS Group

Distributed Systems

Bachelor In Informatics Engineering

Universidad Carlos III de Madrid

Message-oriented Protocols

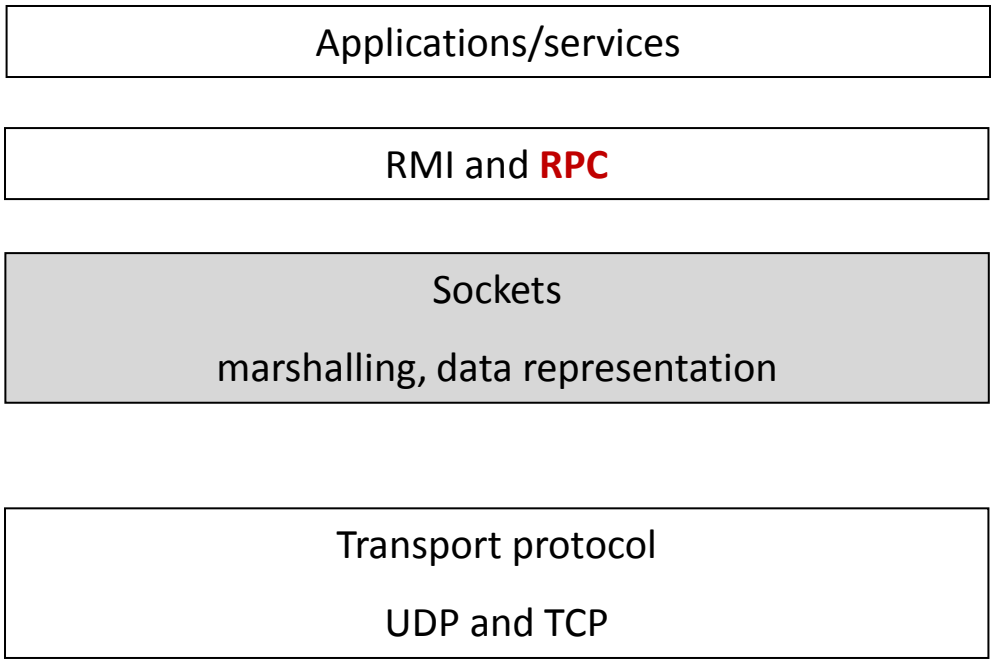
- Many still in widespread use
  - Traditional TCP/IP and Internet protocols
- Difficult to design and implement
  - Especially with more sophisticated middleware
- Many difficult implementation issues for each new implementation
  - Formatting
  - Uniform representation of data
  - Client-server relationships
  - ...

Remote Procedure Call (RPC)

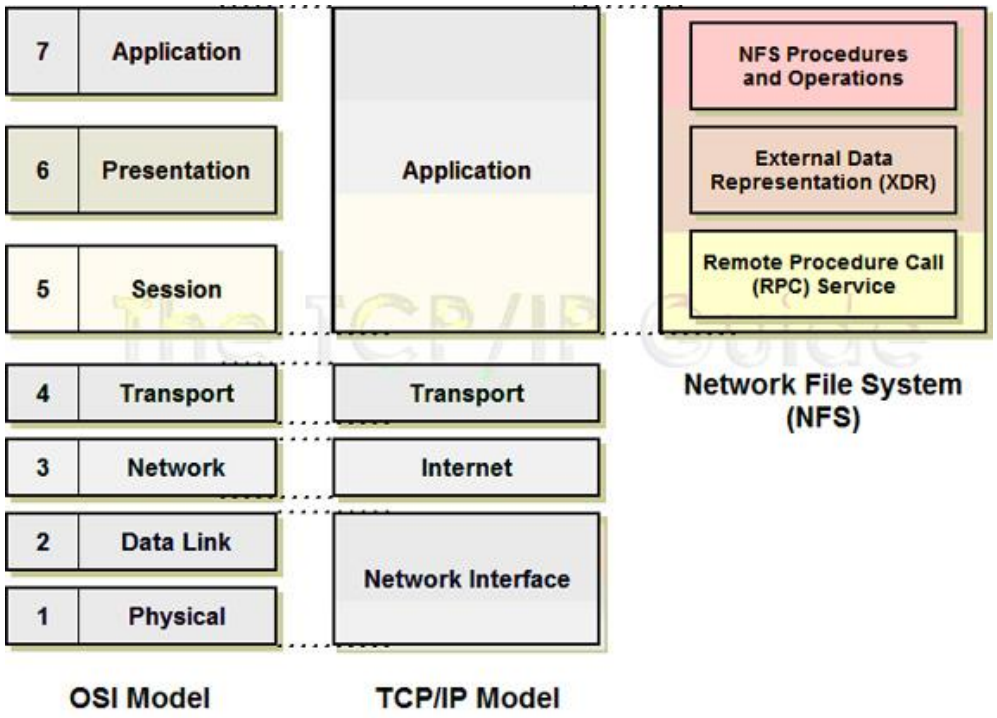
- *The* most common framework for newer protocols and for middleware
- Used both by operating systems and by applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are just RPC systems

RPC

- **RPCs** offer an easy interface to build distributed apps on top of TCP/IP



RPC



RPCs on the protocol stack

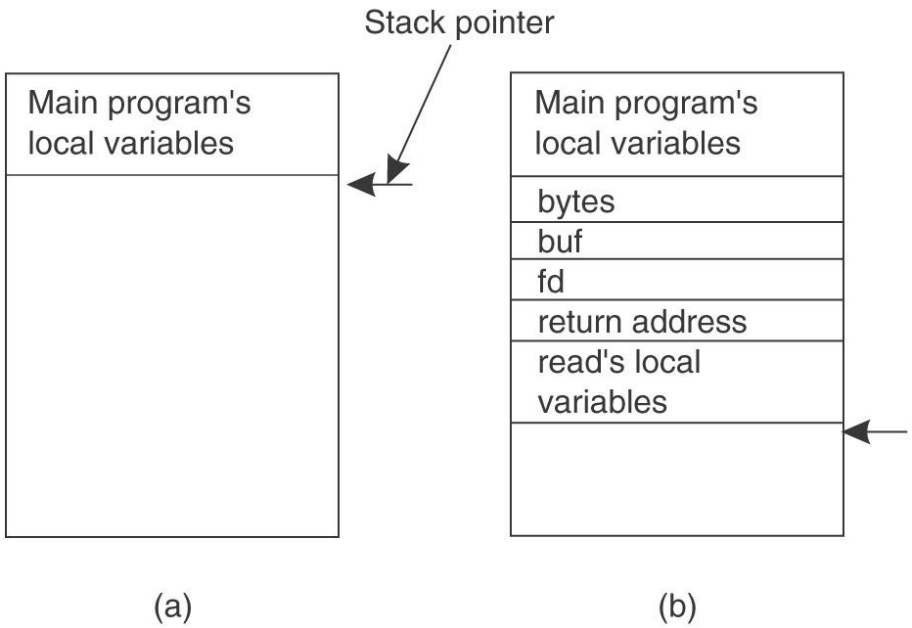
- RPC:
  - Session layer in OSI
  - Application layer in TCP/IP
- Independent of transport protocol
  - TCP
  - UDP
- Reliability not guaranteed
  - Depends on the transport protocol

Remote Procedure Call (RPC)

- Fundamental idea:
  - Server process exports an *interface* of procedures or functions that can be called by client programs
    - similar to library API, class definitions, etc.
- Clients make local procedure/function calls
  - *As if* directly linked with the server process
  - Under the covers, procedure/function call is converted into a message exchange with remote server process

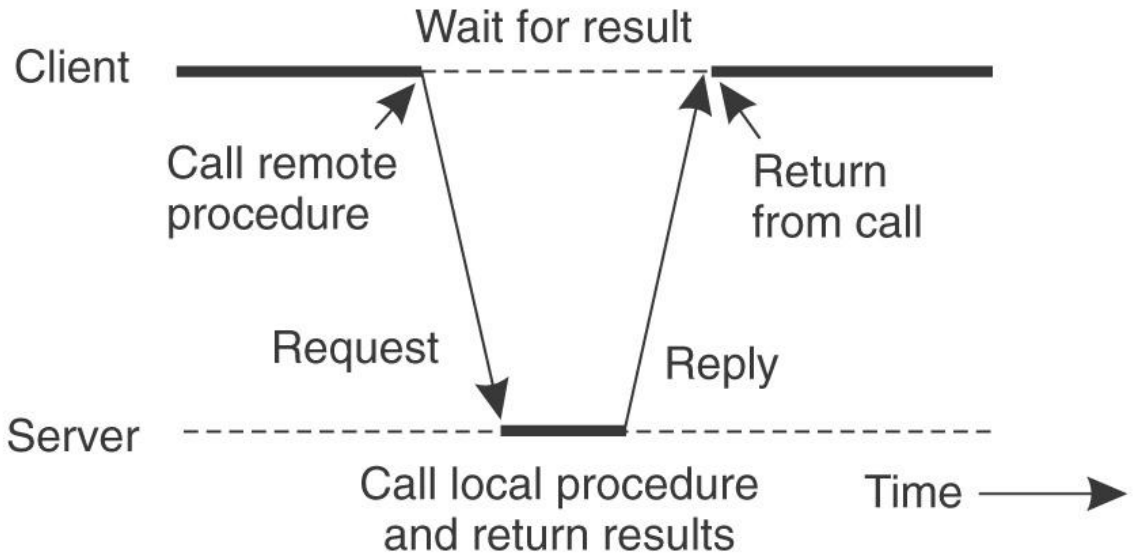
Ordinary procedure/function call

```
count = read(fd, buf, nbytes)
```



Remote Procedure Call

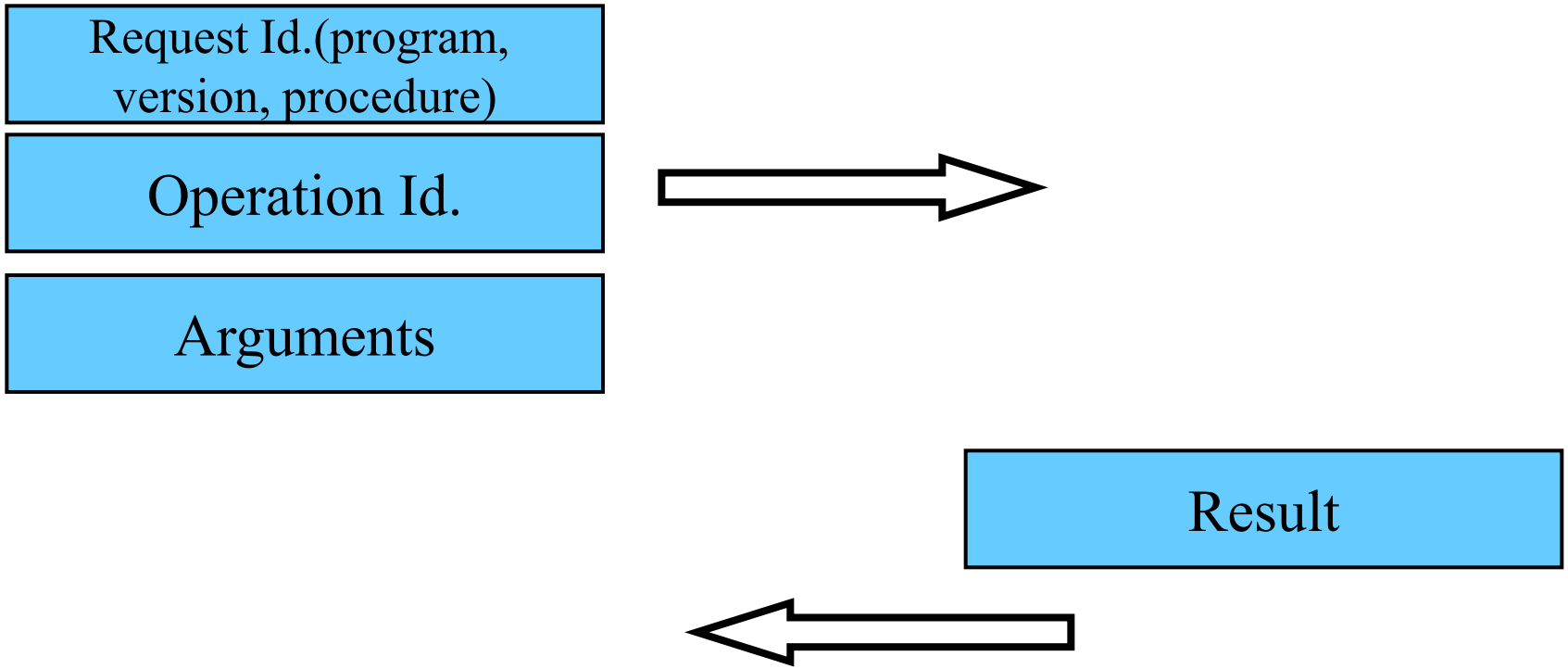
- Would like to do the same if called procedure or function is on a remote server



Solution — a pair of *Stubs*

- Client-side stub
  - Looks like local server function
  - Same interface as local function
  - Bundles arguments into message, sends to server-side stub
  - Waits for reply, un-bundles results
  - returns
- Server-side stub
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

Structure of the request/reply messages



Result

- The hard work of building messages, formatting, uniform representation, etc., is buried in the stubs
    - Where it can be automated!
- Client and server designers can concentrate on the semantics of application
- Programs behave in familiar way

RPC – Issues

- How to make the "remote" part of RPC invisible to the programmer?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- How to handle heterogeneity?
  - OS, language, architecture, …
- How to make it go fast?

RPC Model

- A server defines the service interface using an *interface definition language* (IDL)
  - the IDL specifies the names, parameters, and types for all client-callable server procedures
- A *stub compiler* reads the IDL declarations and produces two *stub functions* for each server function
  - *Server-side* and *client-side*

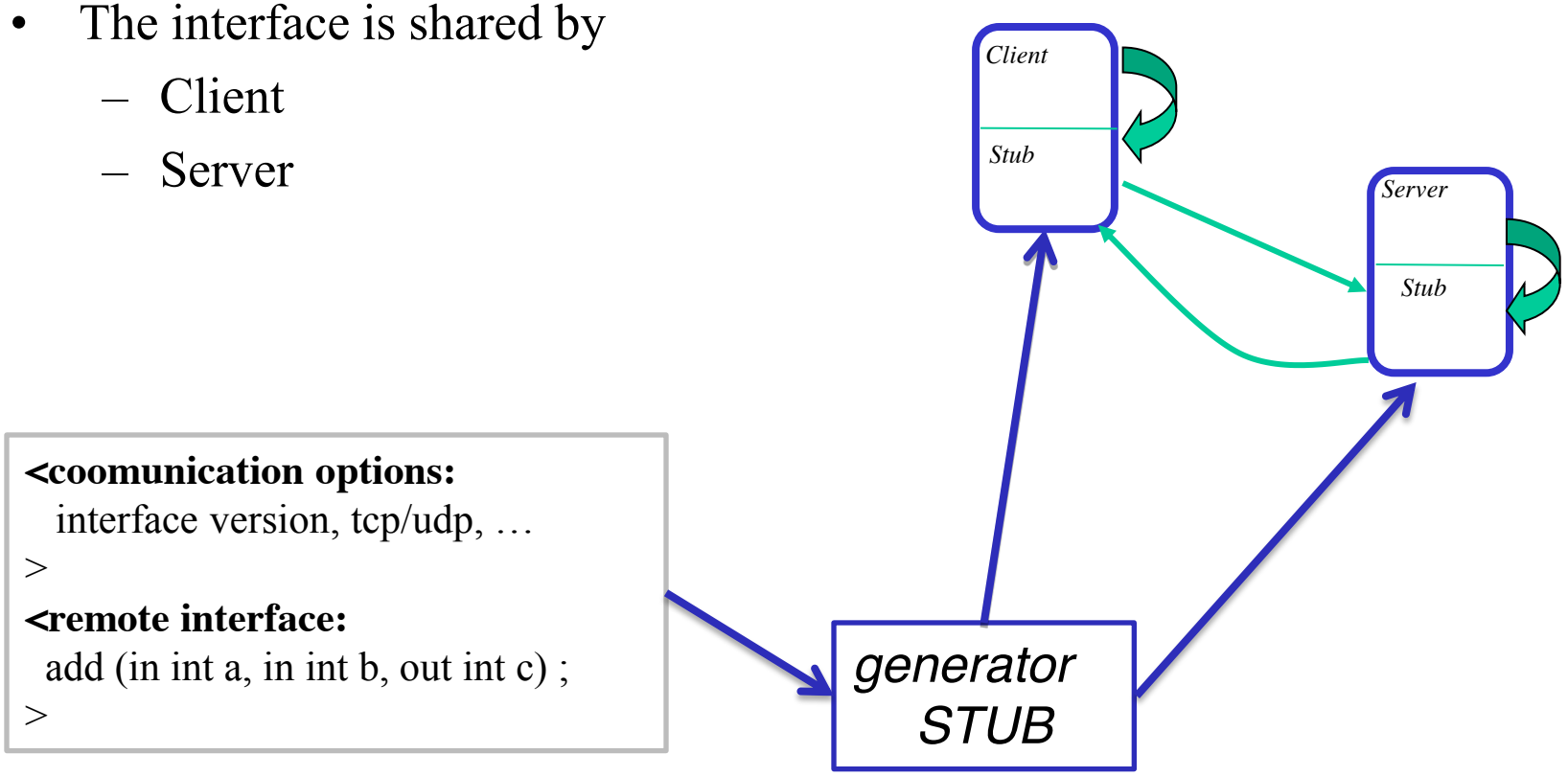
RPC Model (continued)

- Linking:
  - Server programmer implements the service's functions and links with the *server-side* stubs
  - Client programmer implements the client program and links it with *client-side* stubs
- Operation:
  - Stubs manage all of the details of remote communication between client and server

IDL

- IDL allows to specify the RPC format and other communication options

- The interface is shared by
  - Client
  - Server



IDL (cont'd)

- An interface must specify:
  - Service name
  - RP name
  - RP parameters (I, O)
  - Data types of arguments
- Compilers may be designed such that clients and servers use different languages
- IDL types
  - Language-integrated (Cedar, Argus)
  - Interface language such as Sun RPC or DCE RPC

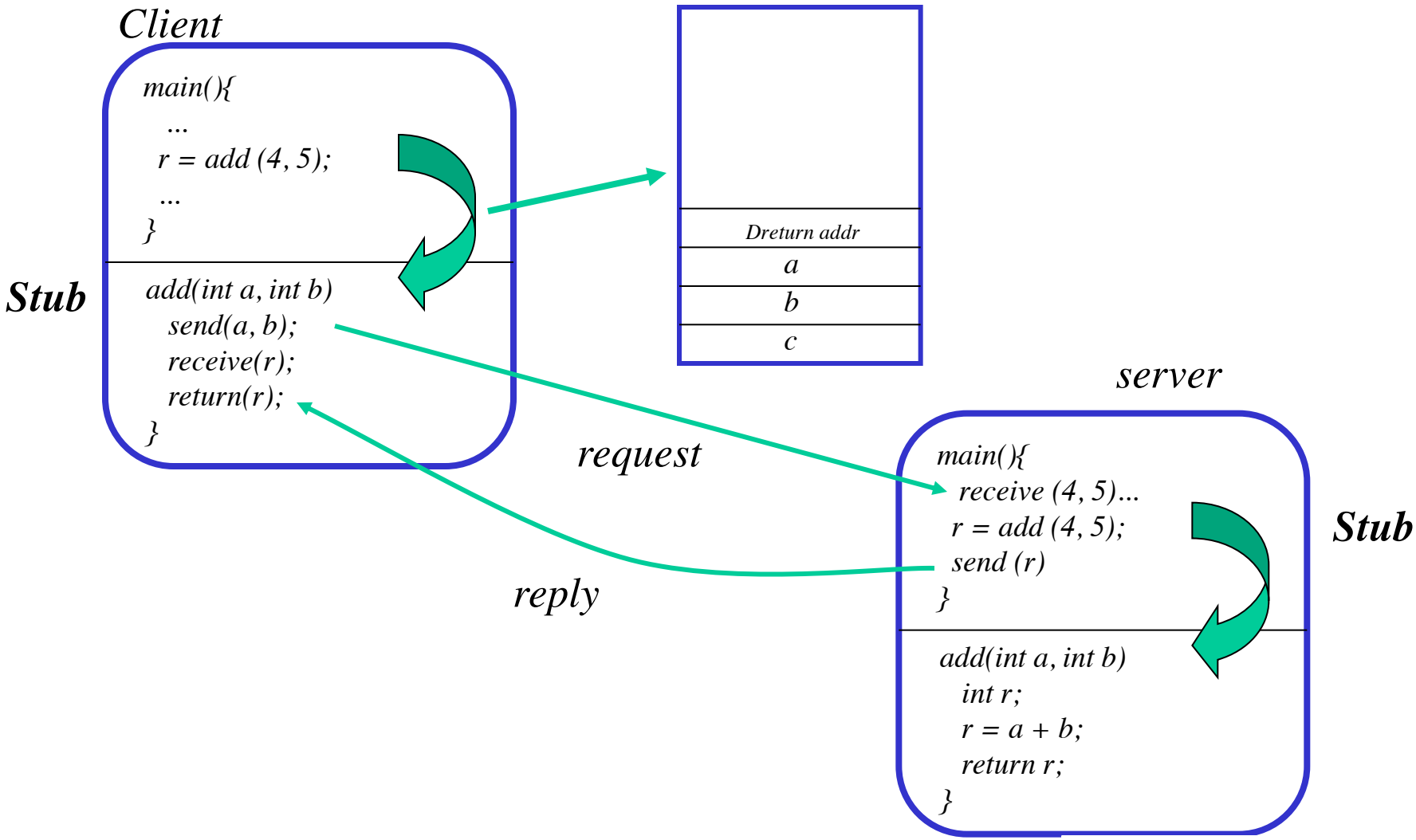
Parameter types

- Input (*in*)
  - From client to server
- Output (*out*)
  - Server to client
- Input/output (*inout*)

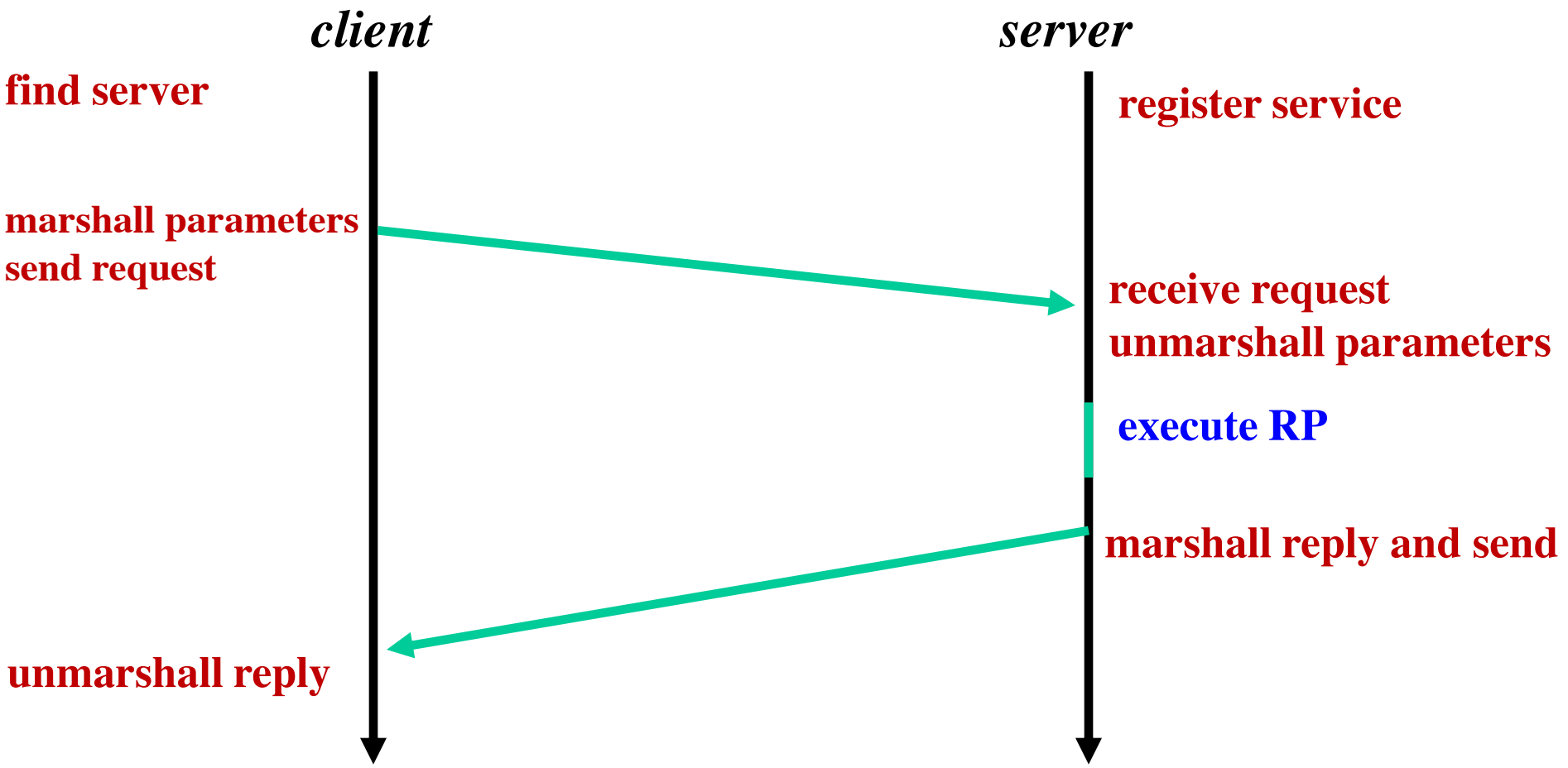
RPC Stubs

- A *client-side stub* is a function that looks to the client as if it were a callable server function
  - I.e., same API as the server's implementation of the function
- A *server-side stub* looks like a caller to the server
  - I.e., like a hunk of code invoking the server function
- The client program thinks it's invoking the server
  - but it's calling into the client-side stub
- The server program thinks it's called by the client
  - but it's really called by the server-side stub
- The stubs send messages to each other to make the RPC happen transparently (almost!)

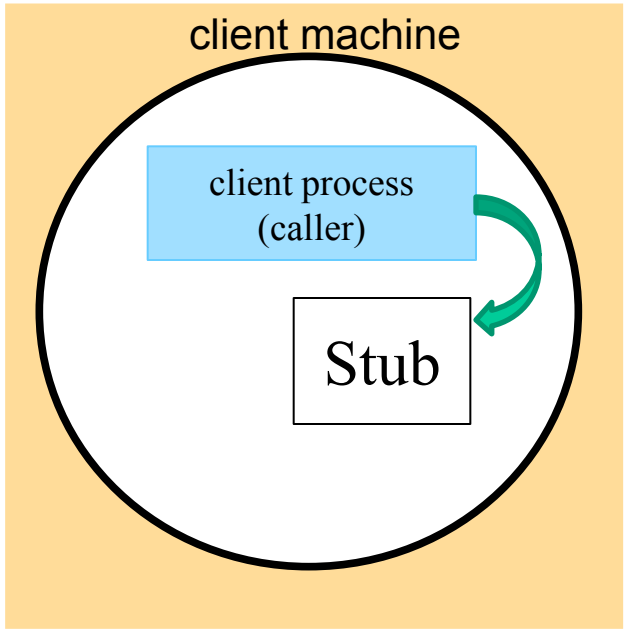
RPC


Client
main(){
...
r = add (4, 5);
...
}
Stub
add(int a, int b)
send(a, b);
receive(r);
return(r);
}
Dreturn addr
a
b
c
request
reply
server
main(){
receive (4, 5)...
r = add (4, 5);
send (r)
}
add(int a, int b)
int r;
r = a + b;
return r;
}
Stub

RPC: basic protocol



RPC: basic protocol



Client process (caller)

- **Connect to server**
- **Invoke a RP**

  Stub:

  – Find server

  – Marshall parameters and build messages

  – Send messages to server

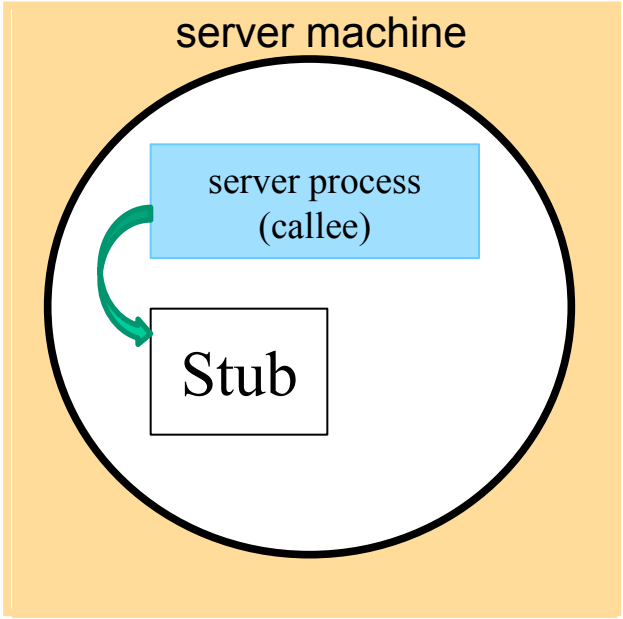
  – Block waiting for reply
- **Get reply**

RPC: basic protocol

Server process (callee)

- **Register** RPCs
- **Implement RPs**

  Stub:
  - Receive client requests
  - Unmarshall parameters
  - Invoke local procedure
- **Get reply and send it to client**

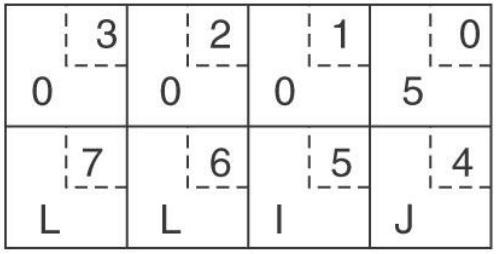


Marshalling Arguments

- *Marshalling* is the packing of function parameters into a message packet
  - the RPC stubs call type-specific functions to marshal or unmarshal the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

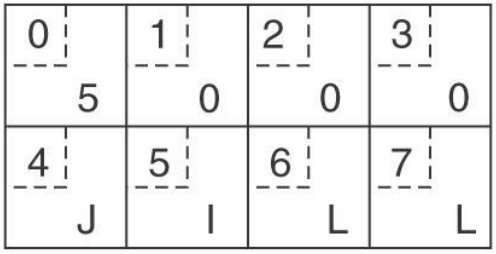
Issue #1 — representation of data

- Big endian *vs*. little endian



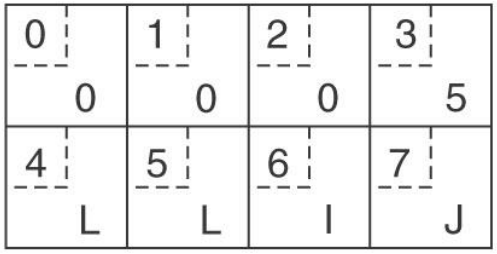
(a)

Sent by Pentium



(b)

Rec'd by SPARC



(c)

After inversion

Representation of Data (continued)

- IDL must also define representation of data on network
  - Multi-byte integers
  - Strings, character codes
  - Floating point, complex, …
  - …
    - example: Sun's XDR (external data representation)
- Each stub converts machine representation to/from network representation
- Clients and servers must *not* try to cast data!

Issue #2 — Pointers and References

```
read(int fd, char* buf, int nbytes)
```

- Pointers are only valid within one address space
- Cannot be interpreted by another process
  - Even on same machine!

- Pointers and references are ubiquitous in C, C++
  - Even in Java implementations!

Pointers and References — Restricted Semantics

- Option: *call by value*
  - Sending stub dereferences pointer, copies result to message
  - Receiving stub conjures up a new pointer
- Option: *call by result*
  - Sending stub provides buffer, called function puts data into it
  - Receiving stub copies data to caller's buffer as specified by pointer

Pointers and References — Restricted Semantics (continued)

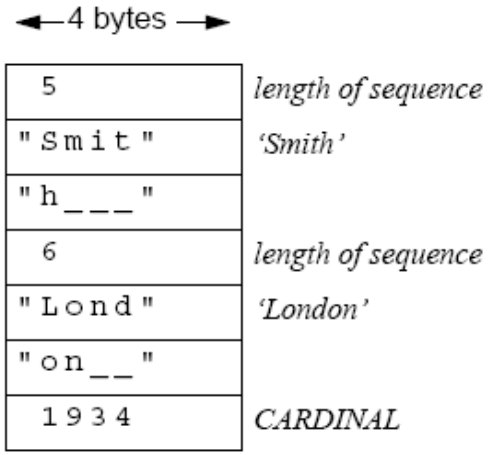
- Option: *call by value-result*
  - Caller's stub copies data to message, then copies result back to client buffer
  - Server stub keeps data in own buffer, server updates it; server sends data back in reply

- Not allowed:–
  - *Call by reference*
  - *Aliased arguments*

Examples of data representations

- Message: ‘Smith’, ‘London’, 1934

XDR



CDR

index in sequence of bytes	4 bytes	notes
0–3	5	length of string
4–7	"Smit"	‘Smith’
8–11	"h___"	
12–15	6	length of string
16–19	"Lond"	‘London’
20-23	"on__"	
24–27	1934	unsigned long

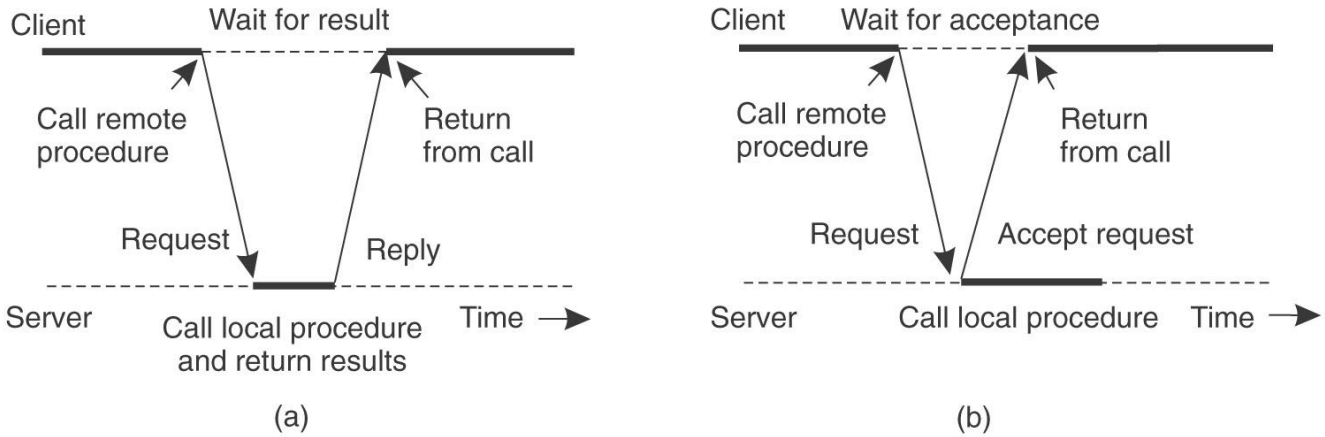
XML

```
<person>
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
</person >
```

Transport of Remote Procedure Call

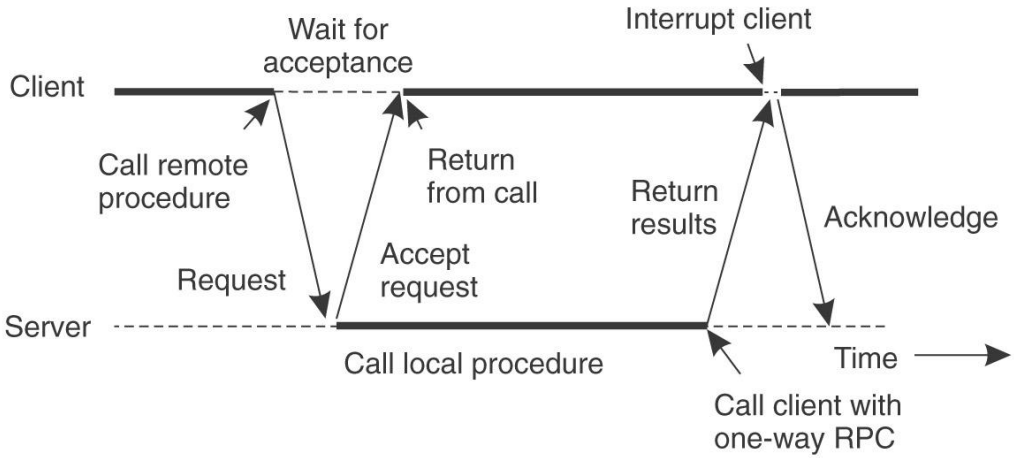
- Option — TCP
  - Connection-based, reliable transmission
  - Useful but heavyweight, less efficient
  - Necessary if repeating a call produces different result
- Alternative — UDP
  - If message fails to arrive within a reasonable time, caller's stub simply sends it again
  - Okay if repeating a call produces same result

Asynchronous RPC



- Analogous to spawning a thread
- Caller must eventually *wait* for result
  - Analogous to *join*

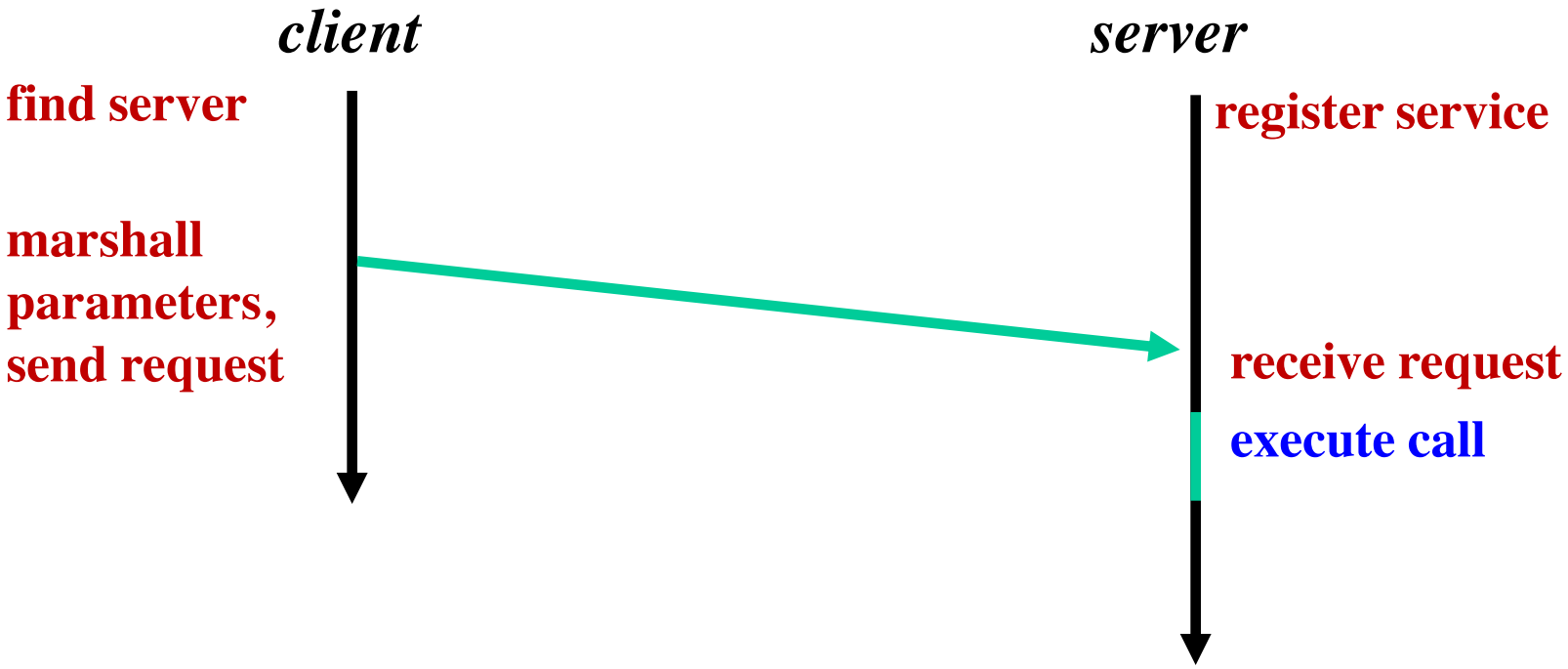
Asynchronous RPC (continued)



- Analogous to spawning a thread
- Caller must eventually *wait* for result
  - Analogous to *join*
  - Or be interrupted (software interrupt)

Asynchronous RPC

- Client does not wait for reply
- Client specifies in request if:
  - He is to be notified via event
  - He is going to poll
  - There will be an asynchronous proc. call
- No output parameters



RPC *Binding*

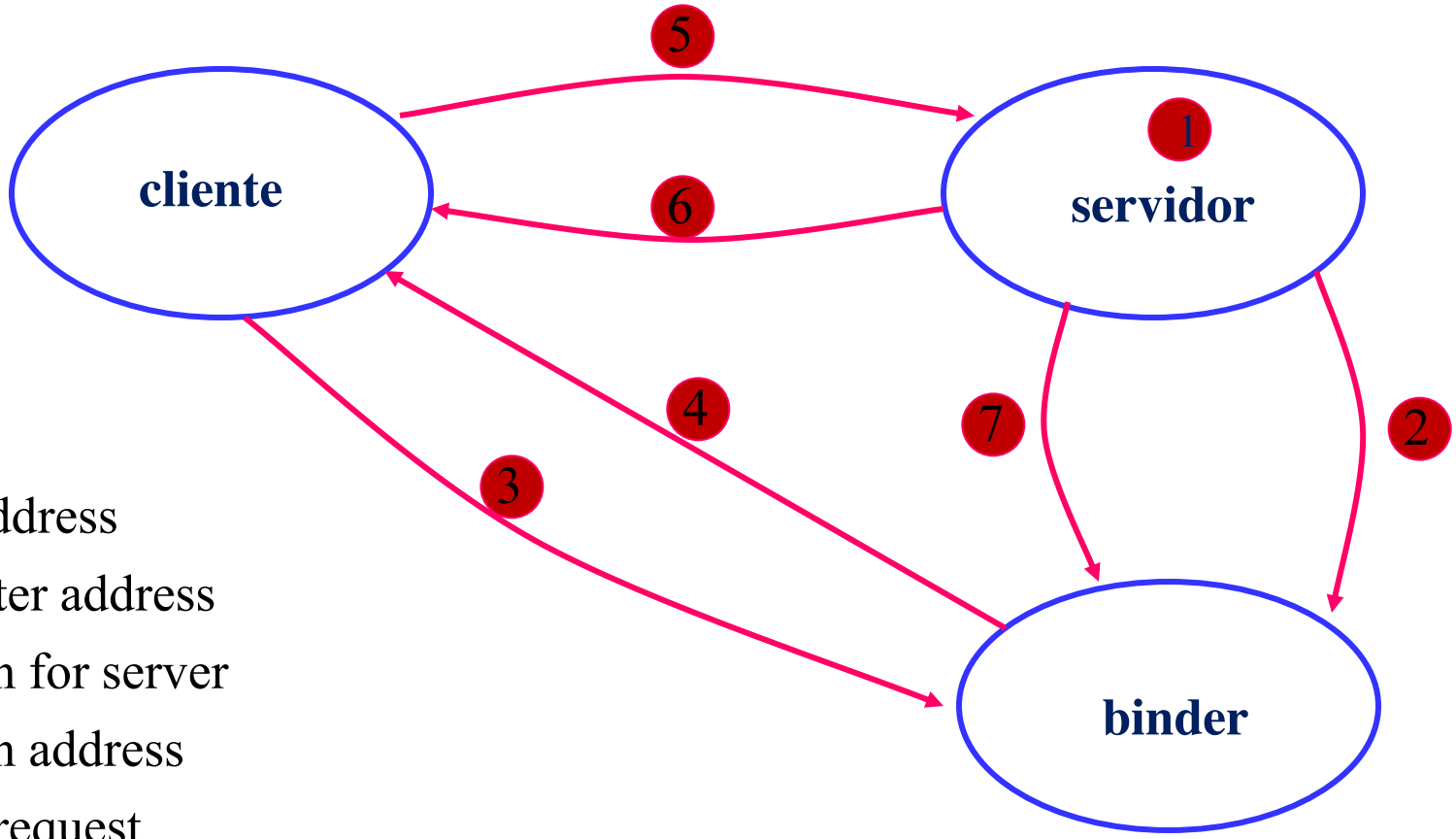
- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Dynamic binding

- Binder: is the service which maintains a translation table between named services and addresses (for servers which register)
- Includes functions for:
  - Registering a named service
  - Eliminate a named service
  - Search for the address corresponding to a named service

- How do we find the binder?
  1. Executes at a well-known address on a well-known computer
  2. The OS is responsible for making the address known
  3. *Broadcast* at process init

Registering/binding schema



1. Get address
2. Register address
3. Search for server
4. Return address
5. Send request
6. Reply
7. Delete address (end of service)

Link types

- Not persistent: connection between client and server re-established at every RPC
  - Fault tolerant
  - Allows service migration
- Persistent: connection maintained after first RPC
  - Useful in app with many reapeated RPCs
  - Problems if servers change address
- Hybrid models

The *portmapper* process

- The binding for Sun RPCs is done by a portmapper process
- Every server executes a portmapper process on a well-known port (111)
- The portmapper stores for each local service:
  - The program number
  - The version number
  - The port number
- Dynamic binding:
  - Nr. of available ports limited but nr. of remote programs may be very large.
  - Only the portmapper executes on a fixed port (111).
  - To get the port nr. where the servers listen the client must ask the portmapper.
- Supports TCP and UDP (/etc/services)
  - `sunrpc 111/tcp portmapper #RPC 4.0 portmapper`
  - `sunrpc 111/udp portmapper`

The *portmapper* process

- Protocol:
  - When a server boots up it registers the previous info on the portmapper
  - When a client needs to invoke a RP it sends to the portmapper of the remote host (must know server IP address) the program name/number and version
- The *portmapper* returns the server port

The *portmapper* process

./server &

rpcinfo –p guernika.lab.inf.uc3m.es

program	vers	proto	port	
100000	2	tcp	111	portmapper
100000	2	udp	111	portmapper
…				
100024	1	udp	32772	status
100024	1	tcp	59338	status
99	1	udp	46936	
99	1	tcp	40427	

Possible faults in the presence of RPCs

- Client cannot find server
- Loss of messages:
  - Request from client to server lost
  - Reply from server to client lost
- Server fails after receiving a request
- Client fails after sending a request

Client cannot find server

- Possible causes:
  - Server down
  - Client may use an old version of server
    - Versioning helps detect access to obsolete copies

- How do we indicate an error to the client?
  - Error code **-1**
    - Not a great solution
      - E.g.: add(a,b) may correctly return -1
  - Raise an exception!
    - Need a language w exceptions

Request from client lost

- Easy to detect
- Timer activated when request sent
- If timer expires (timeout) without a reply retransmit message

Reply message lost

- More dificult to deal with
- In principle one could use timers and retransmissions, BUT it's not clear…
  - Did the request get lost?
  - Did the reply get lost?
  - Is the server slow?

- Idempotent operations (i.e. side effect free operations) return the same result if re-executed
  - A bank transfer is NOT idempotent
  - Adding two number is idempotent

- For non-idempotent operations one should throw away requests
  - Sequence nr in the client
  - A message field which indicates if it's an original request or a retransmission

Problems on the server side

- The server has not executed the operation
  - Could retransmit
- The server DID execute the operation

- Problem: The client cannot distinguish between these!
- What can be done?
  - No guarantees
  - *At-least-once* semantics
    - Retry and guarantee that the RPC goes through at least once
    - Doesn't work for non-idempotent operations
  - *At-most-once* semantics
    - No retry, possible no RPC goes through
  - *Exactly-once* semantics
    - Ideal

Problems on the client side

- No client waits for reply (orphan computation)
  - Wasted CPU cycles
  - Client boot up and RPC re-execute may lead to confusion

Dealing with faults

- When reply is not received
  - Re-send request
- To deal with duplicated requests
  - Filter requests – e.g. sequence numbers
- If need to re-send reply to non-idempotent request (i.e. requests with side effects):
  - Keep history of request / reply to avoid re-executing.

Remote Procedure Call is used …

- Between processes on different machines
  - E.g., client-server model
- Between processes on the same machine
  - More structured than simple message passing
- Between subsystems of an operating system
  - Windows XP (called *Local Procedure Call*)