Distributed Systems

5 RPC

May-11-2009

Gerd Liefländer

System Architecture Group

Schedule of Today

- Introduction
- Types of Communication
- RPC Issues
- RPC Stubs
- RPC Semantics & Failures
- Speedup of RPCs
- Appendix
  - Application and RPC Failures
  - Examples
  - Asynchronous RPC
  - Binding
  - Example RPCs

Literature

RPC Onine Tutorialhttp://www.cs.cf.ac.uk/Dave/C/node33.html

ASN.1 Information site. http://asn1.elibel.tm.fr, 2002.

R. G. Herrtwich; G. Hommel: Kooperation und Konkurrenz | Nebenläufige, verteilte und Echtzeitabhängige Programmsysteme. Springer-Verlag, 1989.

H. M. Levy; E. D. Tempero: Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. Software|Practice and Experience, 21(1):77{90, Jan. 1991.

B. H. Liskov: Primitives for Distributed Computing. In Proceedings of the 7. ACM Symposium on Operating System Principles (SOSP), Operating Systems Review, pages 33{42, Pacific Grove, California, USA, Dec. 1979. ACM.

B. H. Liskov ; L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), volume 23 of SIGPLAN Notices, pages 260-267, Atlanta, Georgia, USA, June 1988. ACM.

B. J. Nelson: Remote Procedure Call. Technical Report CMU-81-119, Carnegie-Mellon University, 1982.

R. Srinivasan: XDR: External Data Representation Standard. http://www.faqs.org/rfcs/rfc1832.html, 1995.

Introduction

Motivation

Problems

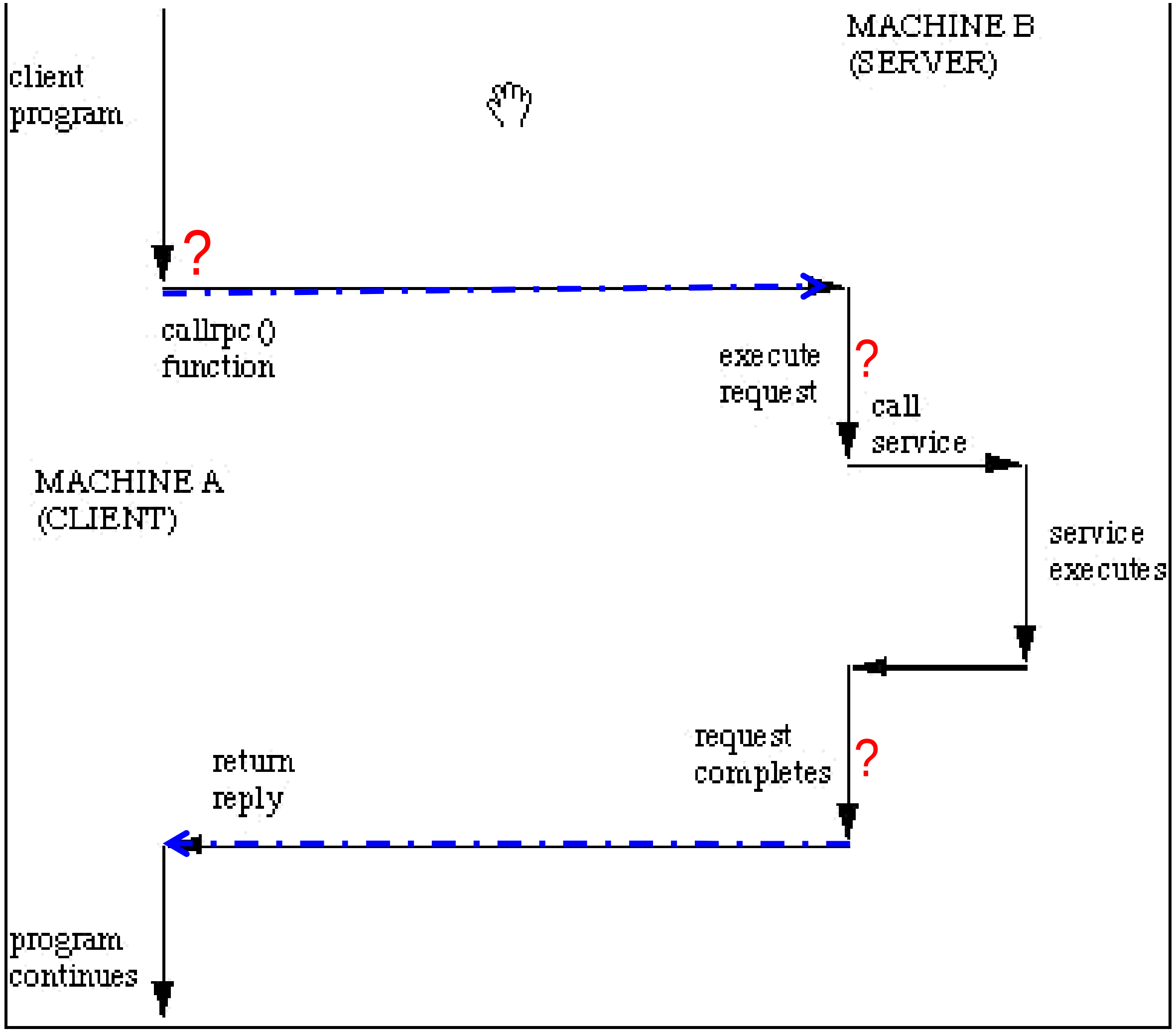
Why Remote "Procedure" Call?

- In the 80ies procedural languages were "en vogue"
- Structured programs at that time consisted of
  - **`main()`** and some **`function()`** ⇒
  - adapt this programming paradigm for DS
- Goal of RPC: mask distributed computing system using a "transparent" abstraction
  - Looks like normal procedure call
  - Hides all aspects of distributed interaction
  - Supports an easy programming model
- Today, RPC is the core of many distributed systems

Remote Procedure Calls

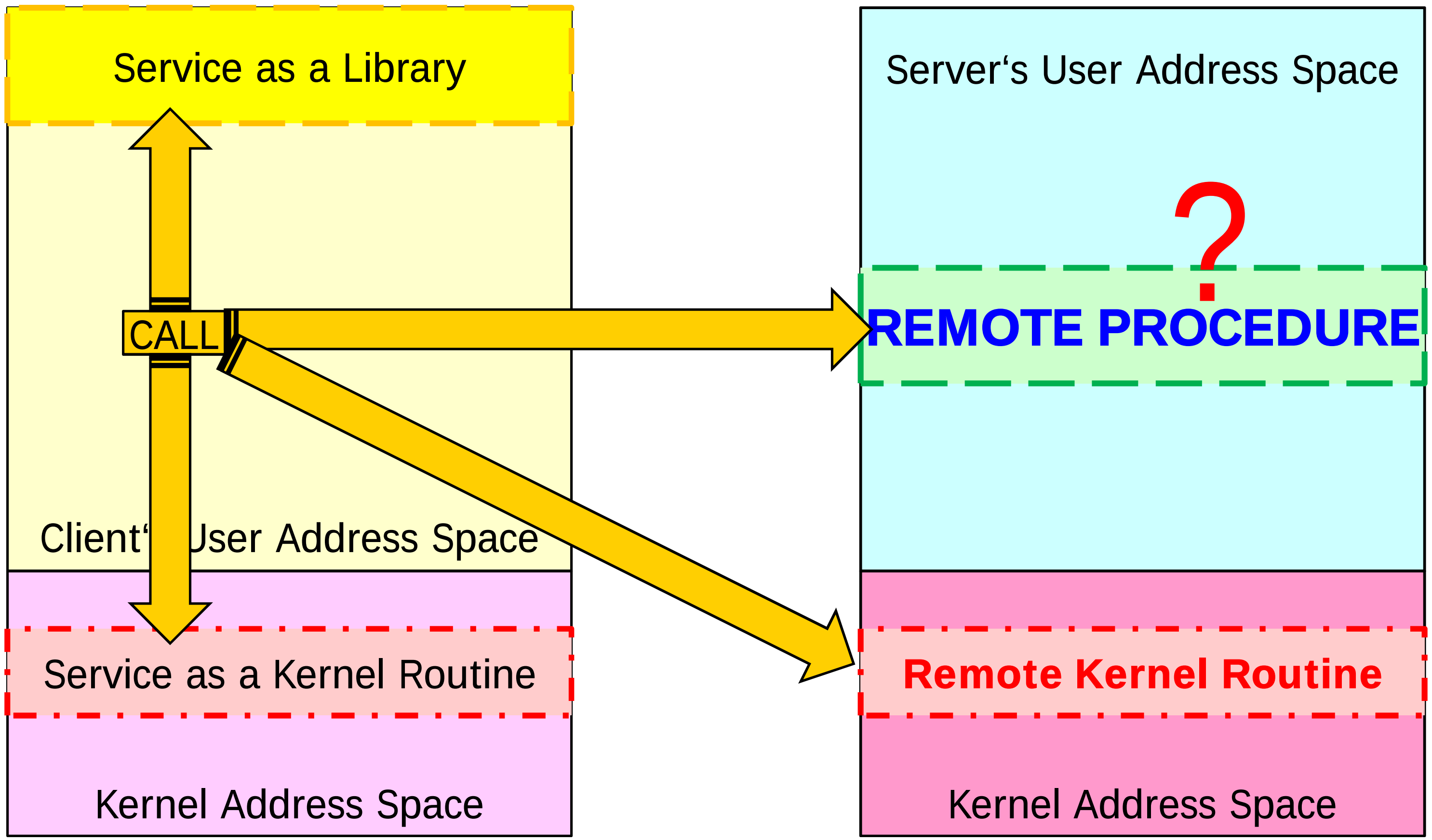
- Early focus was on RPC “environments”
- Culminated in DCE (Distributed Computing Environment), standardizes many aspects of RPC
- Then emphasis shifted to performance, many systems improved by a factor of 10 to 20
- Today, RPC often used from object-oriented systems employing CORBA or COM standards. Reliability issues are more evident than in the past.

How does a RPC work?



Implement a Remote Procedure?

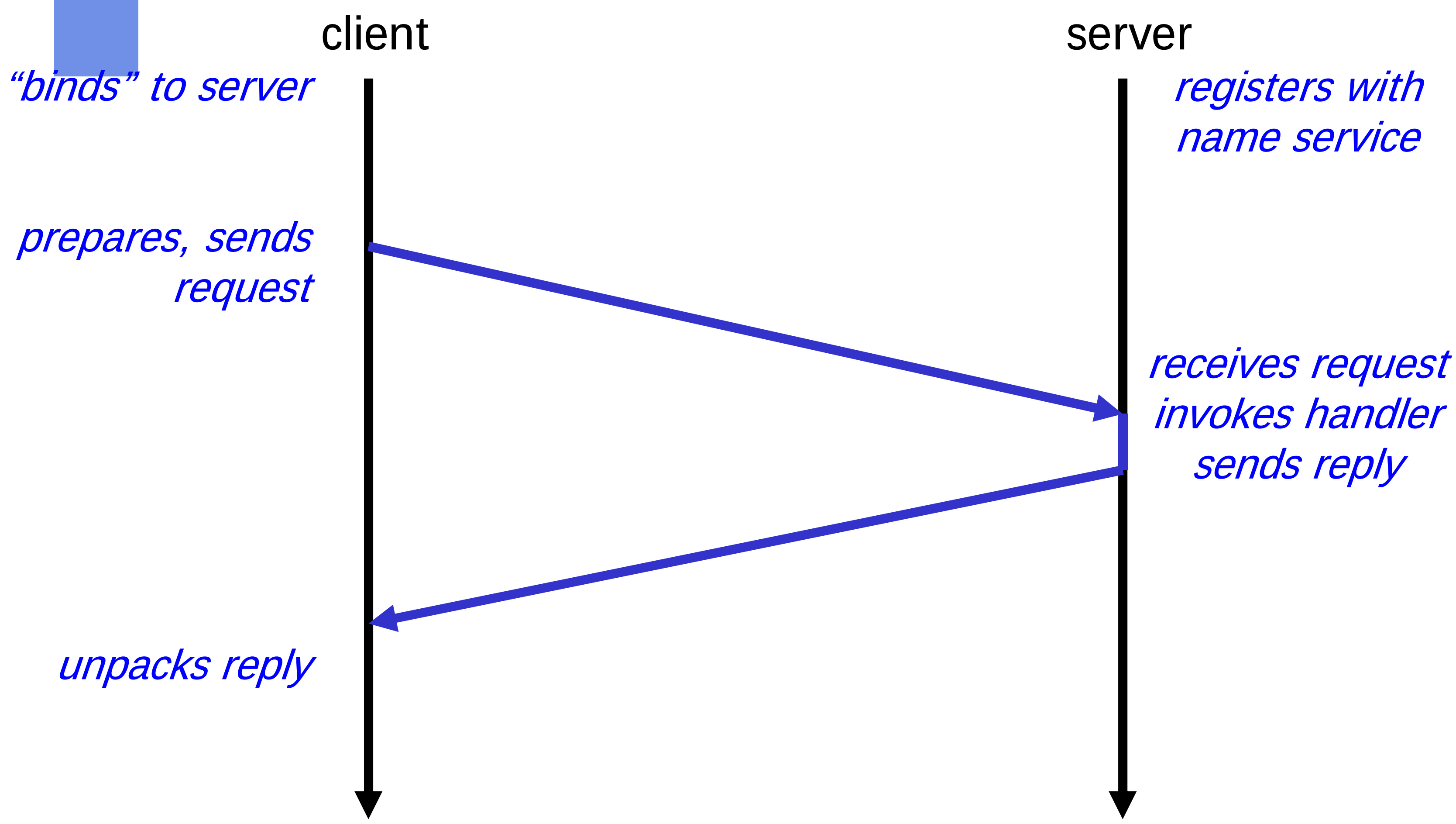
- *RP = what kind of a system entity?*



Fundamental RPC Issues

- Parameter Types
  - *All allowed or efficiently usable?*
- Stubs, acting as substitute instances on both sides
- RPC protocol & Data exchange
  - Binding & Registering
  - Use an intermediate data representation or
  - Add data representation per target machine
- RPC semantics:
  - Exactly once
  - Maybe
  - At least-once
  - At most-once

The Basic RPC Protocol



Compilation Stage

- Server defines and "exports" a header file giving interfaces it supports and arguments expected. Uses "interface definition language" (IDL)
- Client includes this information
- Client invokes server procedures through "stubs"
  - provides interface identical to the server version
  - responsible for building the messages and interpreting the reply messages
  - passes arguments by value (and copy&restore)
  - never use call by reference
  - limit total size of arguments, in bytes

Binding Stage

- Occurs when client and server program first start execution
- Server registers its network address with name directory, perhaps together with other useful information
- Client scans directory to find appropriate or preferred server
- Depending on how the RPC protocol is implemented, client makes a "connection" to the server, but this is not mandatory

RPC Issues

Parameter

Marshalling

Stubs

RPC Semantics

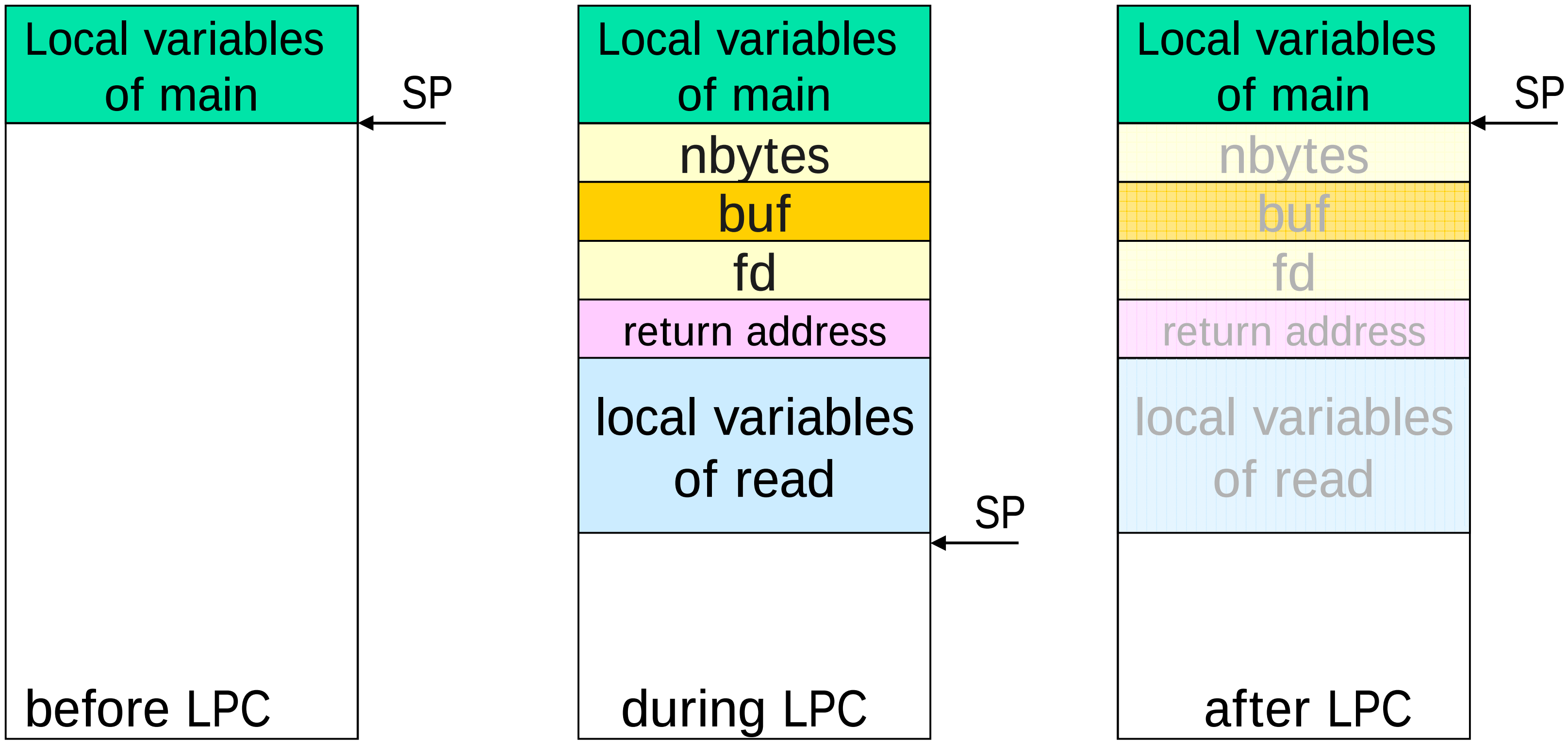
Review: Local Procedure Call*

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

- count → **Convention:** Typically the result of a function is passed via a register, i.e. number of bytes having been read
- read → procedure name
- fd → file-id
- buf → address of a buffer
- nbytes → max. # bytes to be read

*C- Convention

Review: LPC (2)

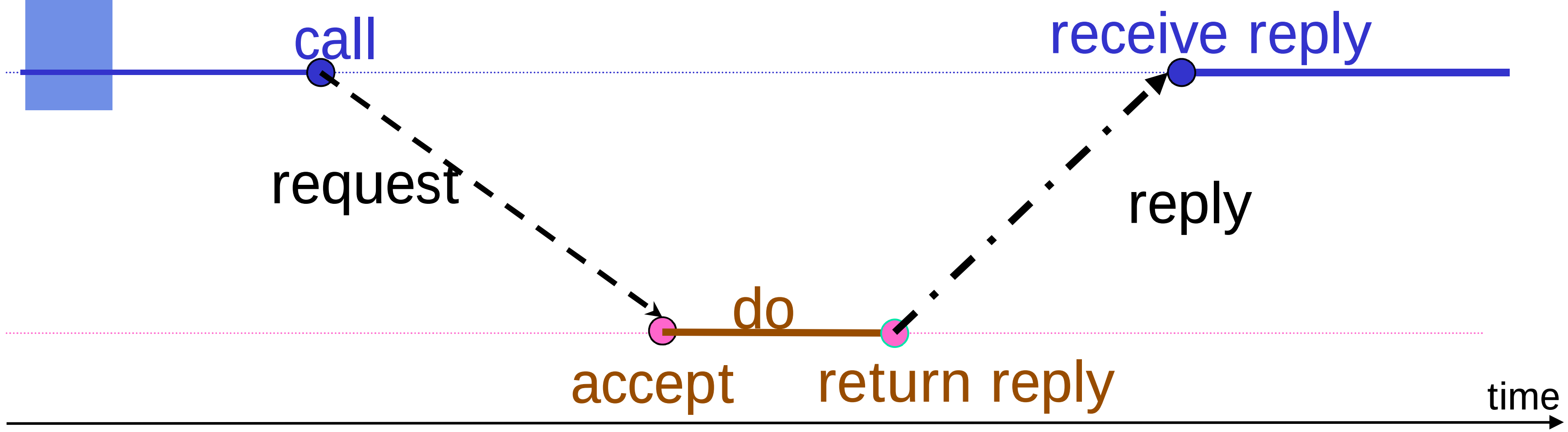


Push parameters and local variables to stack

Pop results via parameters into local or global variables of main

<u>Result:</u> Communication between caller and local callee is handled by copying data from & to the stack

Synchronous RPC



Pro: No explicit communication at application level
Call-by-value/result parameters as usual

Con: No concurrency between client and server

All Parameter Types with RPC?

- Call by value/result
- Call by reference
  - *How to pass reference parameters in a DS?*
  - Usually not supported in a DS (except SASOS), because any reference value has only local meaning
- Can be emulated by copy/restore
  - ∃ some differences
  - ∃ alternatives[1]

No problems at all

Which data type?

[1]In the tutorials Philipp will discuss this topic in more detail.
see also Schröder-Preikschat slides: “Verteilte Systeme” Ch. 5

Reference versus Copy/Restore

```
procedure p(int x, int y) /* x,y inouts */
{
   x = x + 1;
   y = y * y;
}
…
/* somewhere in a caller, e.g. main()     */
i = 2;
p(i, i);
print(i)
…
```

Question: *Output = ?*

x, y via call by reference	9
x, y via call by copy/restore	3 or 4

How to transfer RPC Parameters?

- Somewhere at the client's site there must be a substitute instance, called stub that marshalls all parameter data into an understandable message at the server site
- Example message from client stub to server stub in case of the RPC **`read()`**

`read`
`nbytes`
`length of buf`
`buf[0]`
`...`
`buf[n-1]`
`fd`

In a heterogeneous DS further marshalling info is necessary!

Stubs

Parameter Passing

Marshalling

Stubs

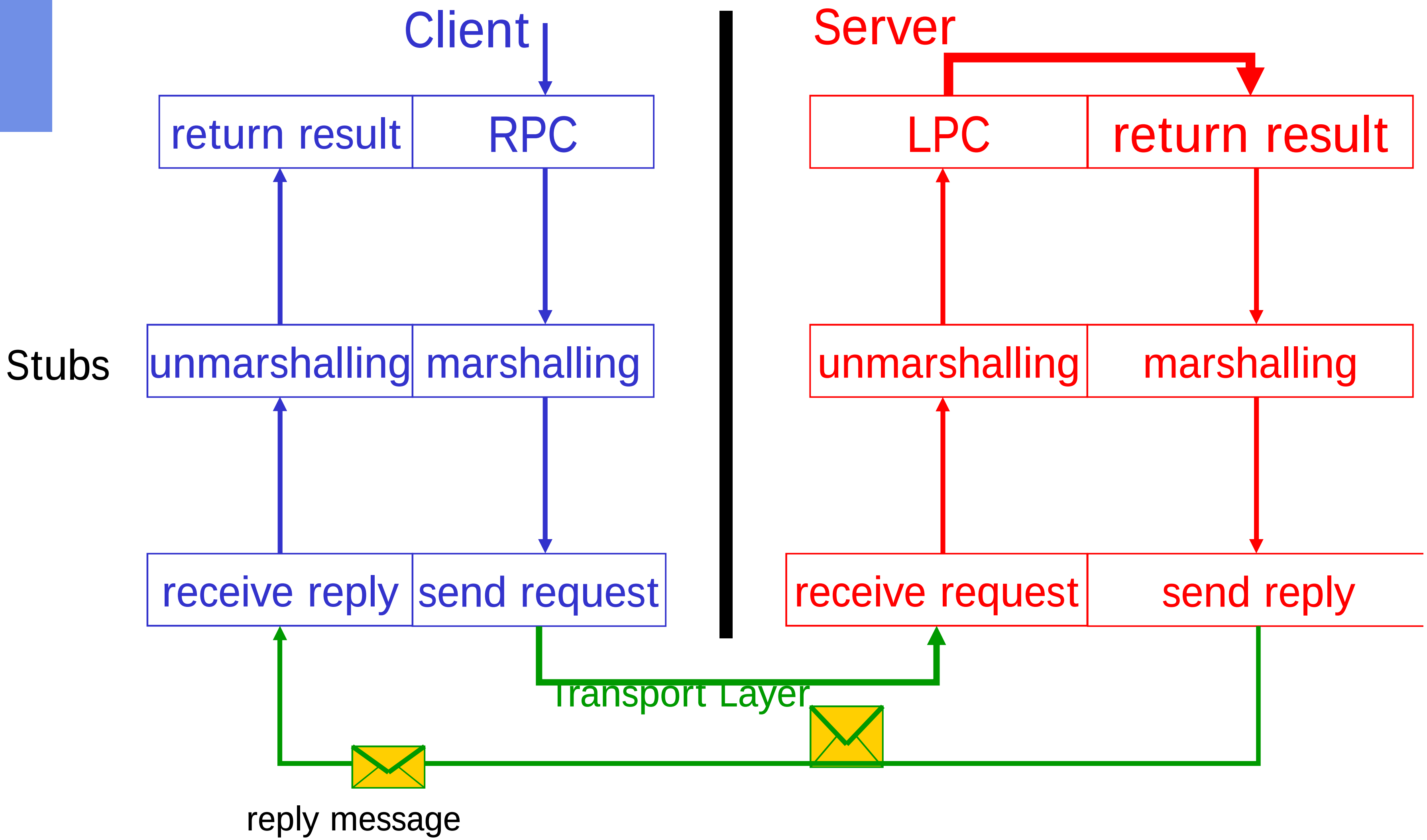
Client Stub:

- Instance that mimics remote procedure in client's environment (on client-node)

Server Stub:

- Instance that mimics a local caller (on behalf of the client) in the server's environment (on server-side)

RPC Protocol



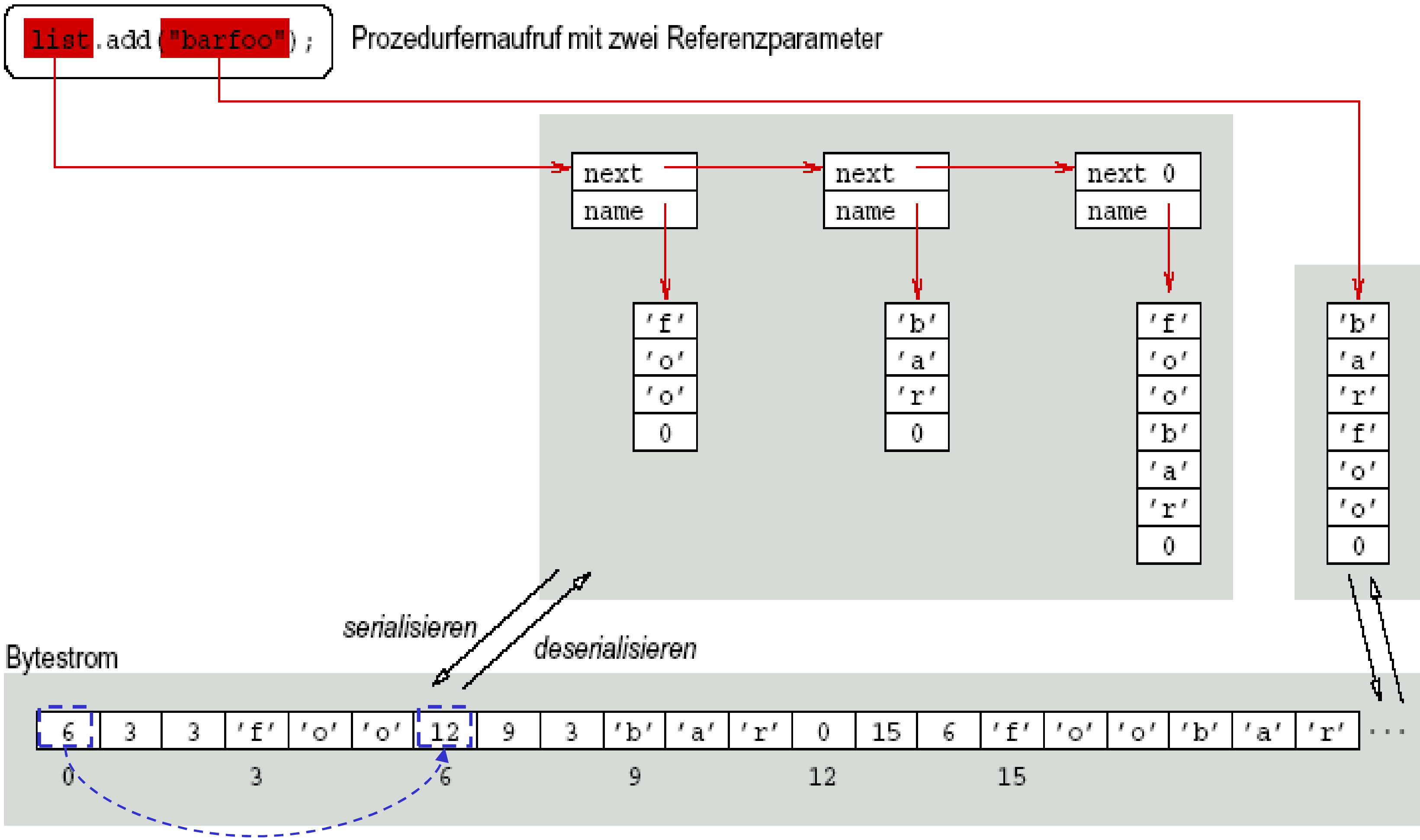
Marshalling Problems

- Composing message with parameters
  - *How to handle complex data structures, e.g. structs, records, arrays, linked lists?*
  - *How to flatten a complex data (structure)*
  - *How to overcome heterogeneity?*
    - Little or big Endian
    - EBCDIC, ASCII, ??

Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a standard representation
  - Example: external data representation (XDR)
- *Problem: how do we pass pointers?*
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- *What about data structures containing pointers?*
  - Prohibit
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream

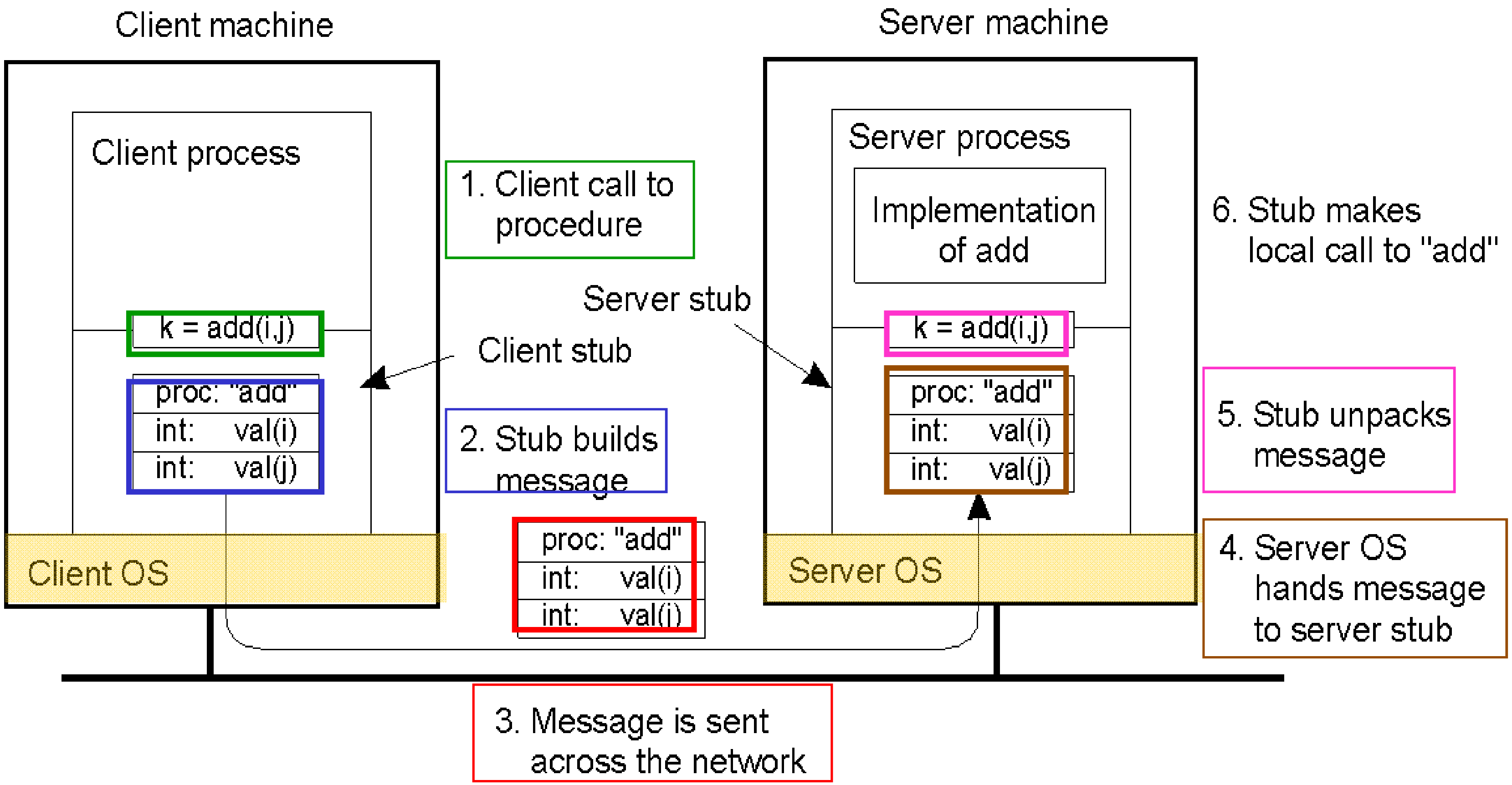
Flattening Data Structures



Methods of Data Conversion

- Client & server must coincide on type of parameter
  - Transformation into a *standard format*
    - e.g. **XDR** (= e**X**ternal **D**ata **R**epresentation)
    - Con: also done if both sides have the same representation
  - Client stub transforms the data into server representation ("Sender makes it right")
    - Con: each node has to know formats of all other nodes
  - Receiver makes it right
    - Pro: if receiver supports same data format as sender, no need for additional transformation
    - Con: see above

Example: Passing Parameters



- Steps involved in doing remote computation via RPC

Marshalling Data Structures

- Via copy/restore complete data structure $\Rightarrow$
  - May send too much data

- Client only sends reference as parameter
  - If server tries to access the data (delivered as reference parameter), it requests this data from the client explicitly $\Rightarrow$
    - increased communication overhead and execution delay
    - Need of preventing anybody from accessing this data as long as the server is using it
      - In case of a client process no problem
      - In case of a multi-threaded client $\rightarrow$ we need additional synchronization

Costs in Basic RPC Protocol

- Allocation and marshalling data into message (can reduce costs if you are certain client, server have identical data representations)
- Two system calls, one to send, one to receive, hence context switching
- Much copying all through the O/S: application to UDP, UDP to IP, IP to Ethernet interface, and back up to application

Typical Optimizations?

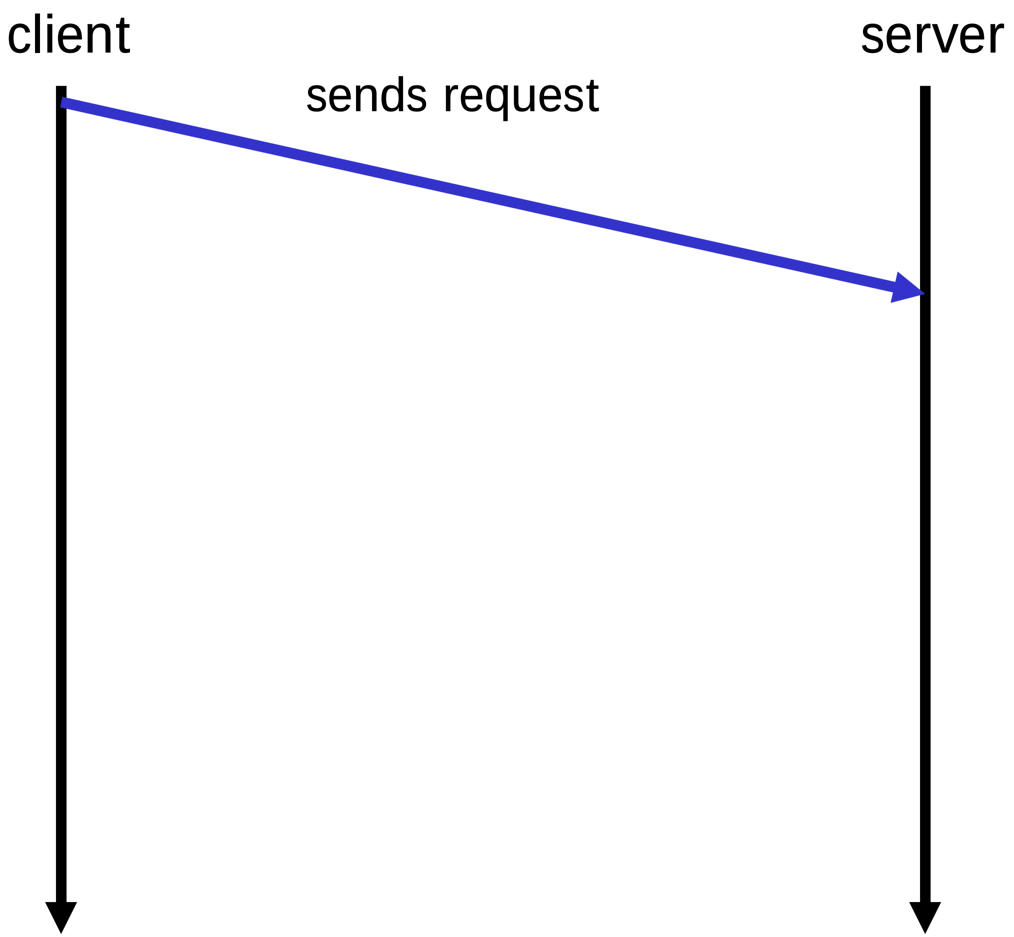
- Compile the stub “inline” to put arguments directly into message
- Two versions of stub; if (at bind time) sender and dest. found to have same data representations, use host-specific rep.
- Use a special “send, then receive” system call (requires O/S extension)
- Optimize the O/S kernel path itself to eliminate copying – treat RPC as the most important task the kernel will do

Fancy Argument Passing

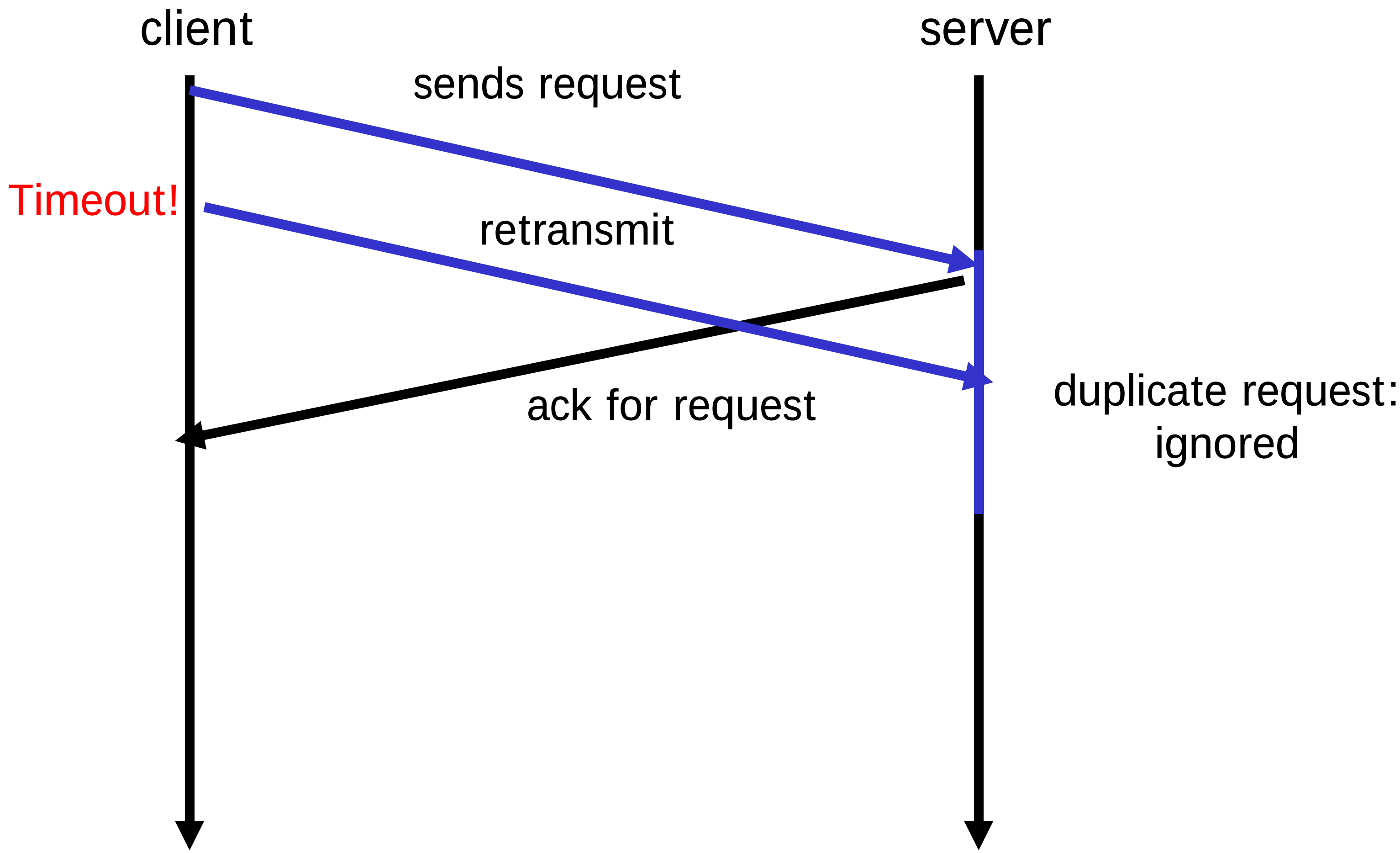
- RPC is transparent for simple calls with a small amount of data passed
  - "Transparent" in the sense that the interface to the procedure is unchanged
  - But exceptions thrown will include new exceptions associated with network
- What about complex structures, pointers, big arrays? These will be very costly, and perhaps impractical to pass as arguments
- Most implementations limit size, types of RPC arguments. Very general systems are less limited but much more costly.

RPC Semantics and Failures

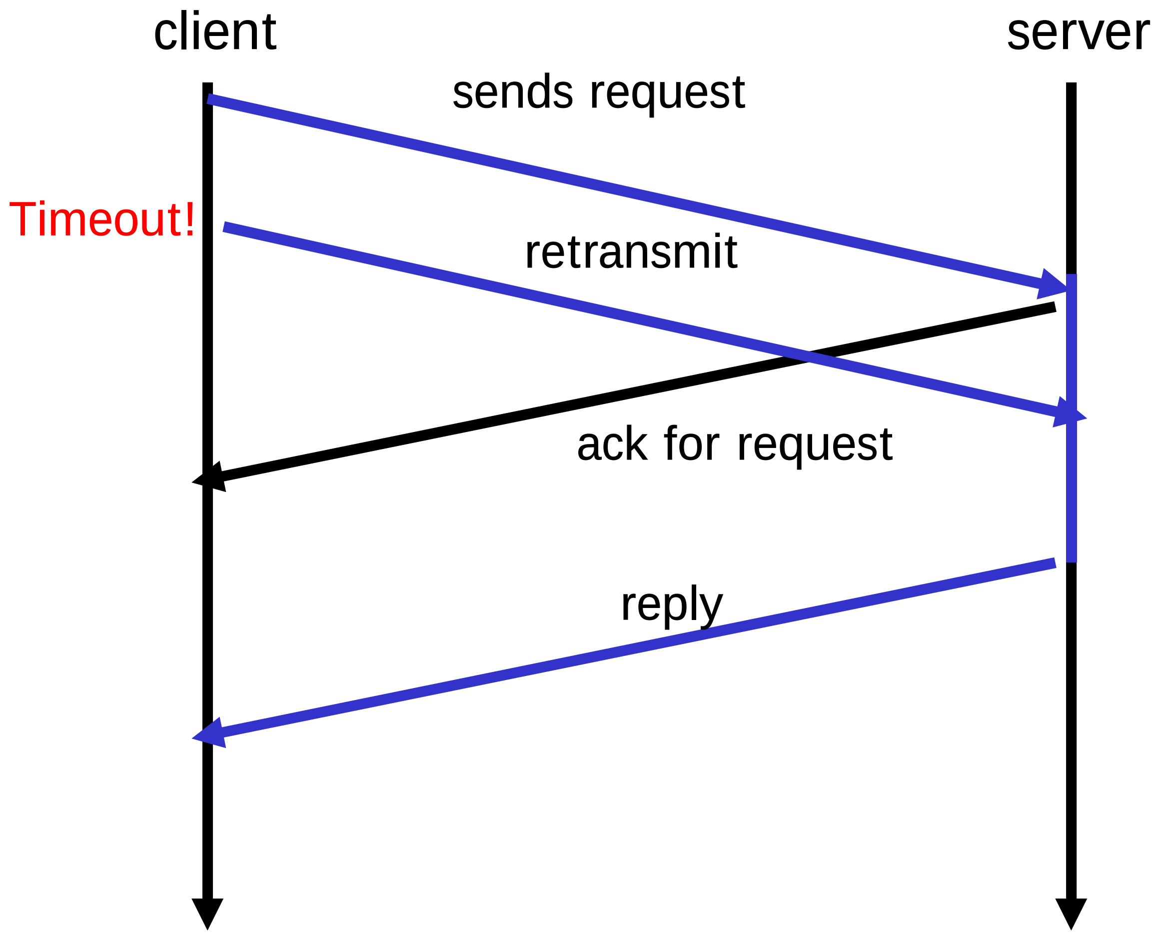
Overcoming lost Packets



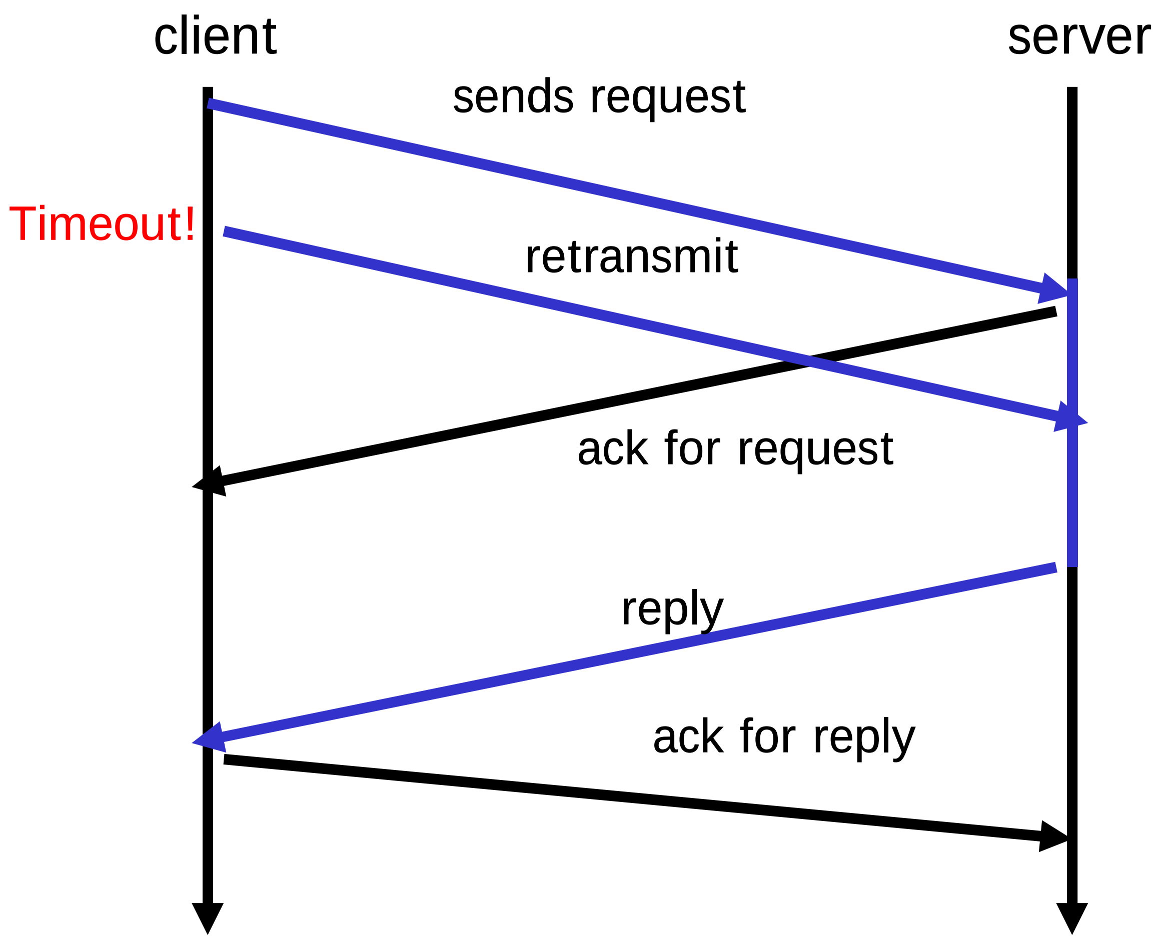
Overcoming lost Packets



Overcoming lost Packets



Overcoming lost Packets



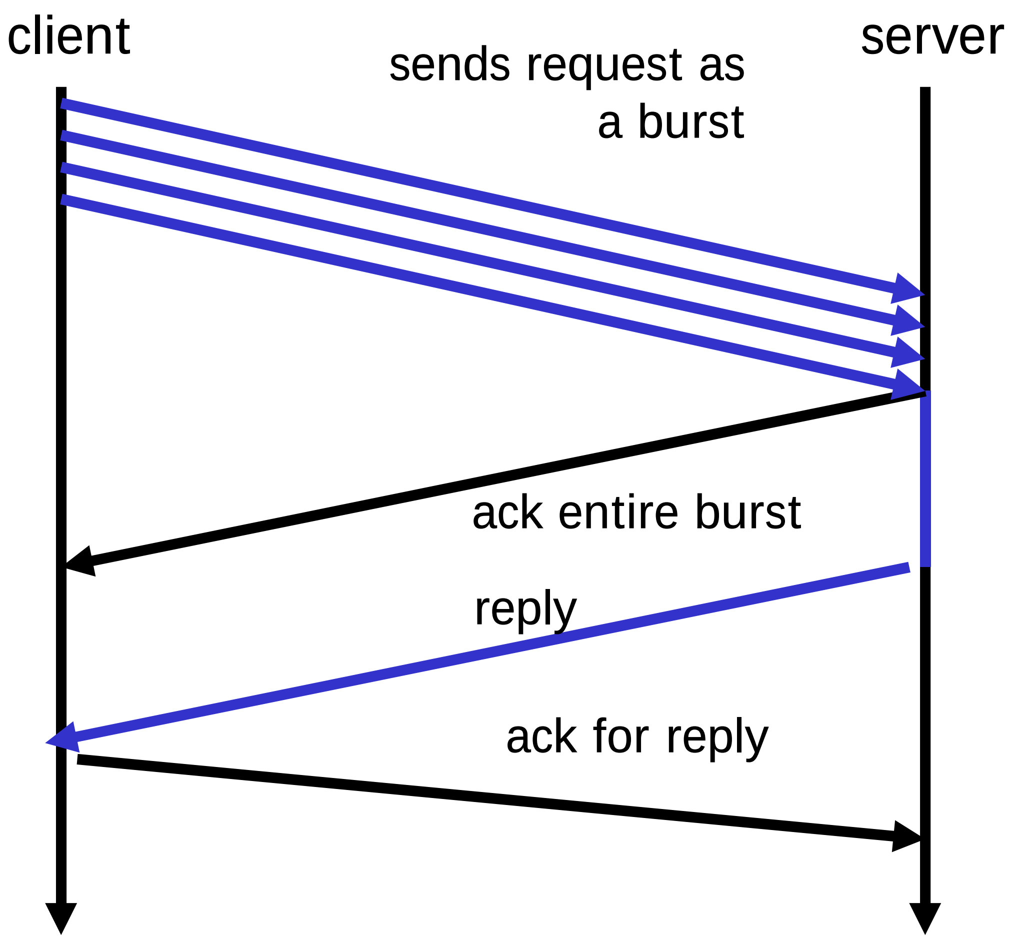
Costs in fault-tolerant Version?

- Acks are expensive. Try and avoid them, e.g. if the reply will be sent quickly, suppress the initial ack

- Retransmission is costly. Try and tune the "accepted delay" to be "optimal"

- For big messages, send packets in bursts and ack a burst at a time, not one by one

Big Packets



client
server
sends request as
a burst
ack entire burst
reply
ack for reply

RPC "Semantics"

- At most once: request is processed 0 or 1 times
- Exactly once: request is always processed 1 time
- At least once: request processed 1 or more times

… but exactly once is impossible because we can't distinguish packet loss from true failures.

In both cases, RPC protocol simply times out.

At Most/Least Once RPC

- Use a timer (clock) value and a unique id, plus sender address
- Server remembers recent id's and replies with same data if a request is repeated
- Also uses id to identify duplicates and reject them
- Very old requests detected and ignored by checking time
  - Assumes that the clocks are working
  - In particular, requires "synchronized" clocks

RPC versus LPC

- Restrictions on argument sizes and types
- New error cases:
  - Bind operation failed
  - Request timed out
  - Argument “too large” can occur if, e.g., a table grows
- Costs may be very high
- … so RPC is actually not very transparent!

Speed Up of RPCs

Cost of Basic RPC

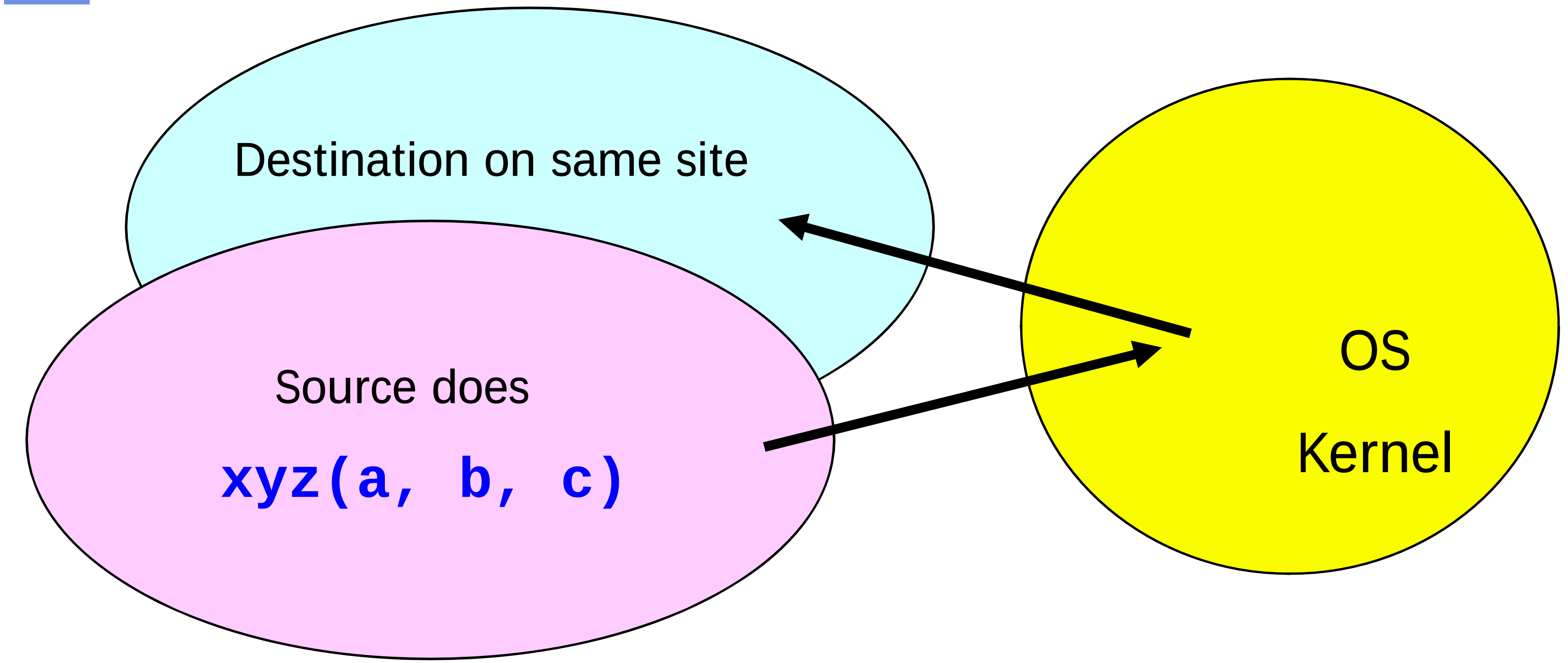
Lightweight RPC

FBUFs

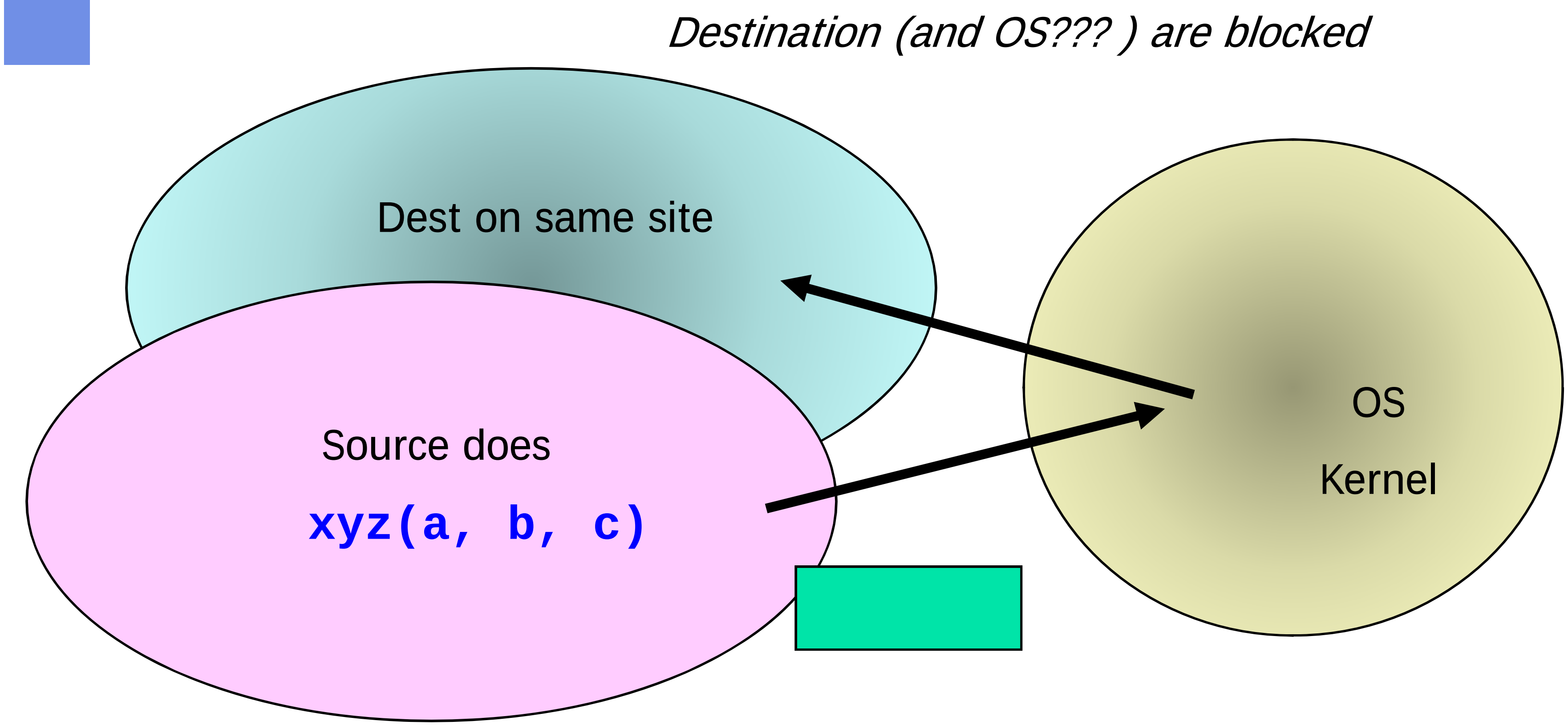
RPC Costs in Case of Local Server

- Sometimes, the server of a client's RPC is right on the caller's machine
  - Caller builds message
  - Issues send system call, blocks, context switch
  - Message copied into kernel, then out to destination
  - Destination is blocked… wake it up, context switch
  - Destination computes result
  - Entire sequence repeated in reverse direction
  - If scheduler is a process, a context switch occurs 6 times

RPC Example

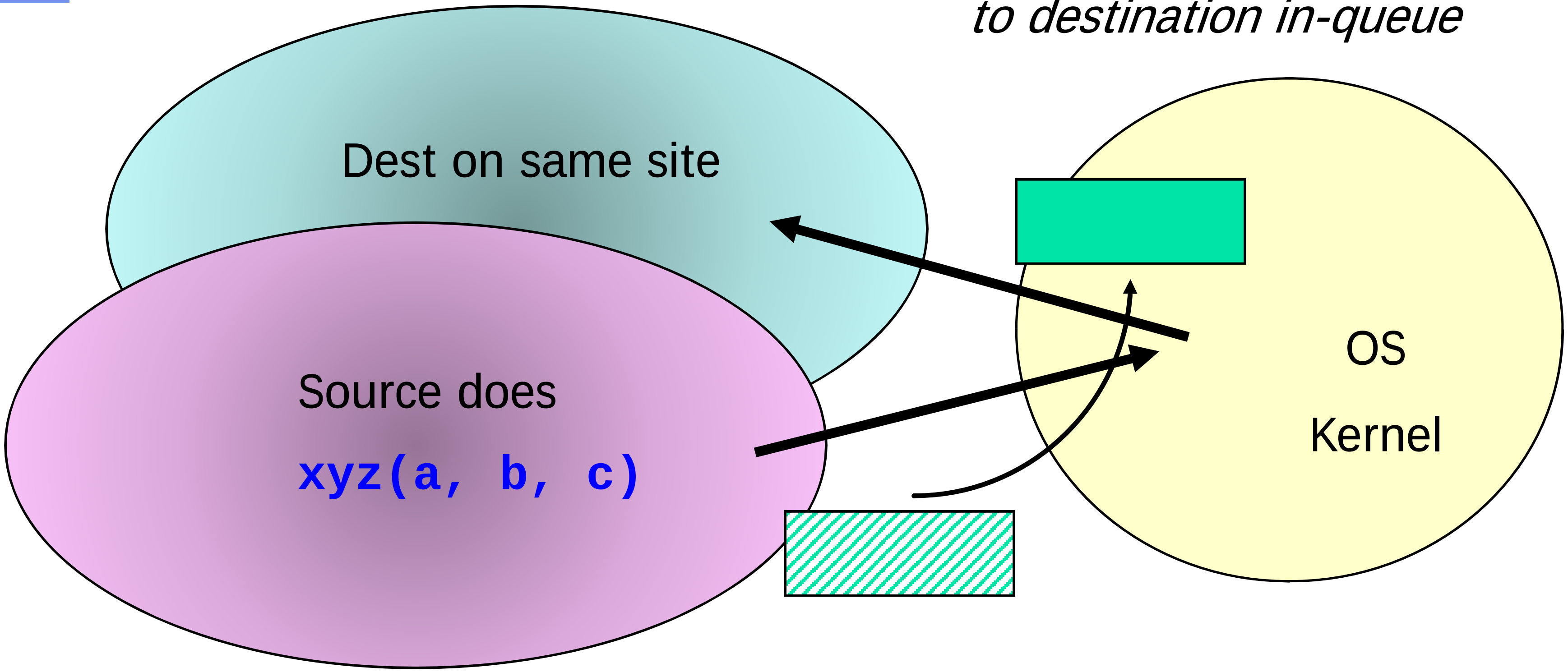


RPC in Normal Case

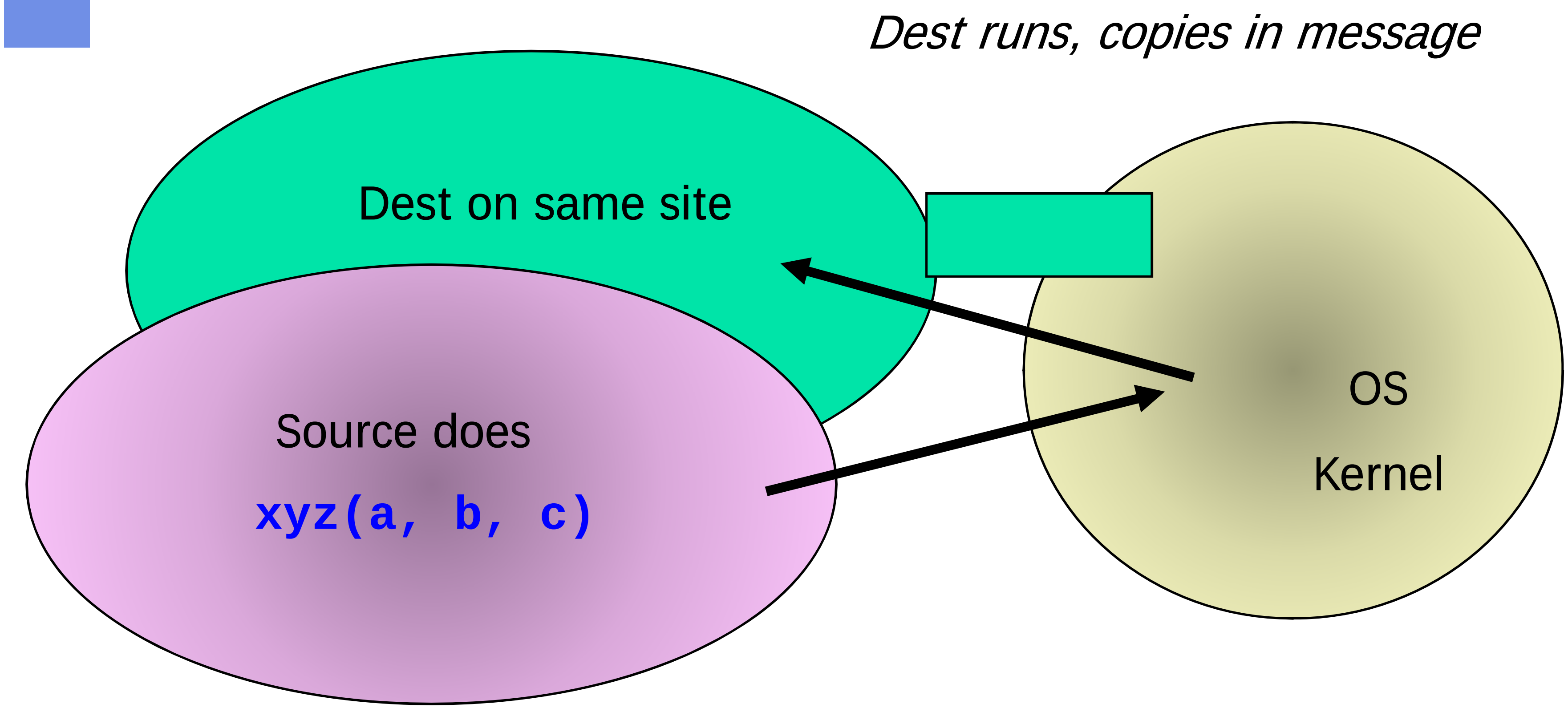


RPC in Normal Case

Both source, destination both block. OS runs its scheduler, copies message from source out-queue to destination in-queue



RPC in Normal Case



Same sequence needed to return results

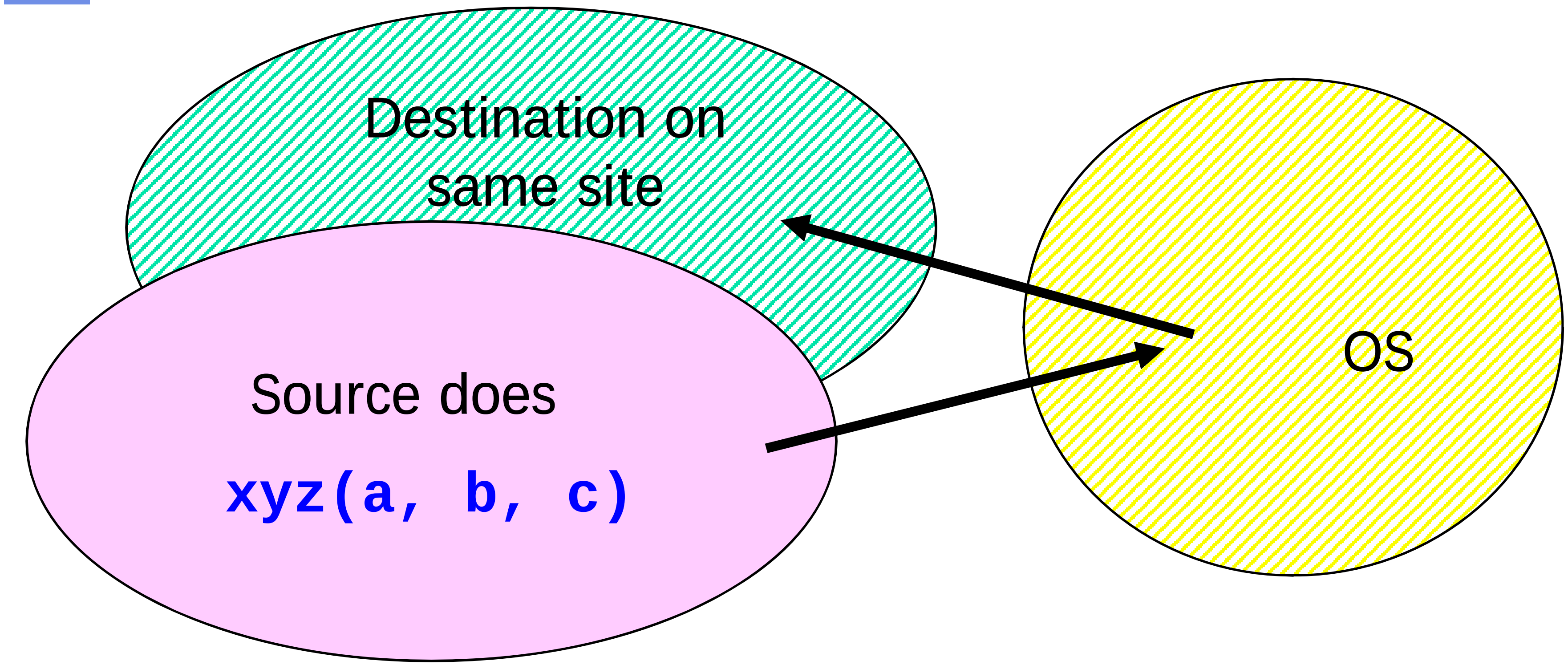
Important Optimizations: LRPC[1]

- Lightweight RPC (LRPC): in case of sender and destination executing on same machine
- Uses memory mapping to pass data
- Reuses same kernel thread to reduce context switching costs (user suspends and server wakes up on same kernel thread or “stack”)
- Single system call: **`send_rcv`** or **`rcv_send`**

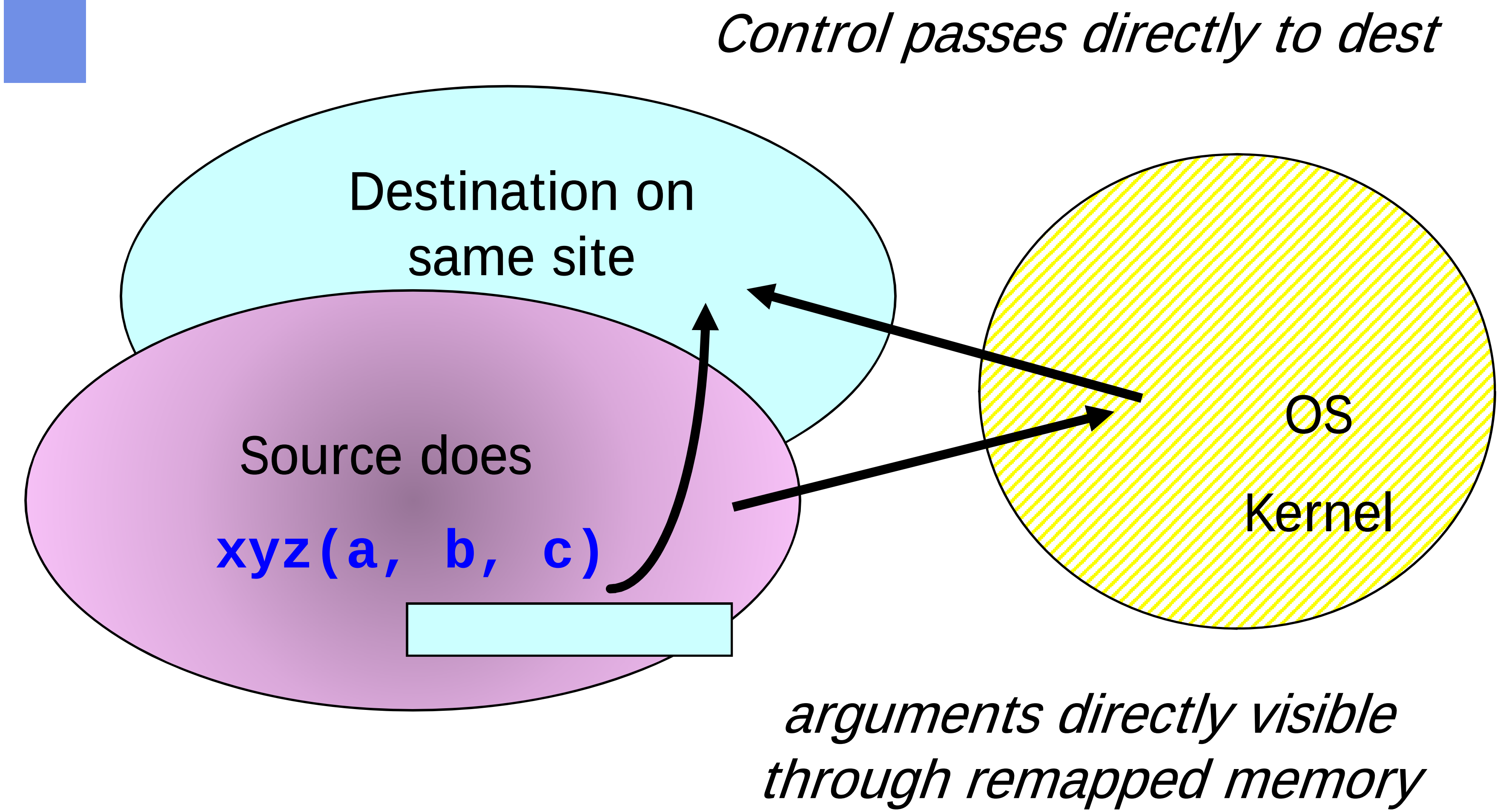
[1]Bershad, Anderson, Lazowska, Levy: „Lightweight Remote Procedure Call“, SOSP 1989

LRPC

OS and destination initially are idle



LRPC


Control passes directly to dest
Destination on
same site
OS
Kernel
Source does
xyz(a, b, c)
arguments directly visible
through remapped memory

LRPC Performance Impact

Measurements have shown:

- On the same OS-platform, LRPC offers about a 10-fold improvement over a hand-optimized RPC implementation
- Does two memory remappings, no context switch
- Runs about 50 times faster than standard RPC by same vendor (at the time of the research)
- Semantics stronger: easy to ensure exactly once

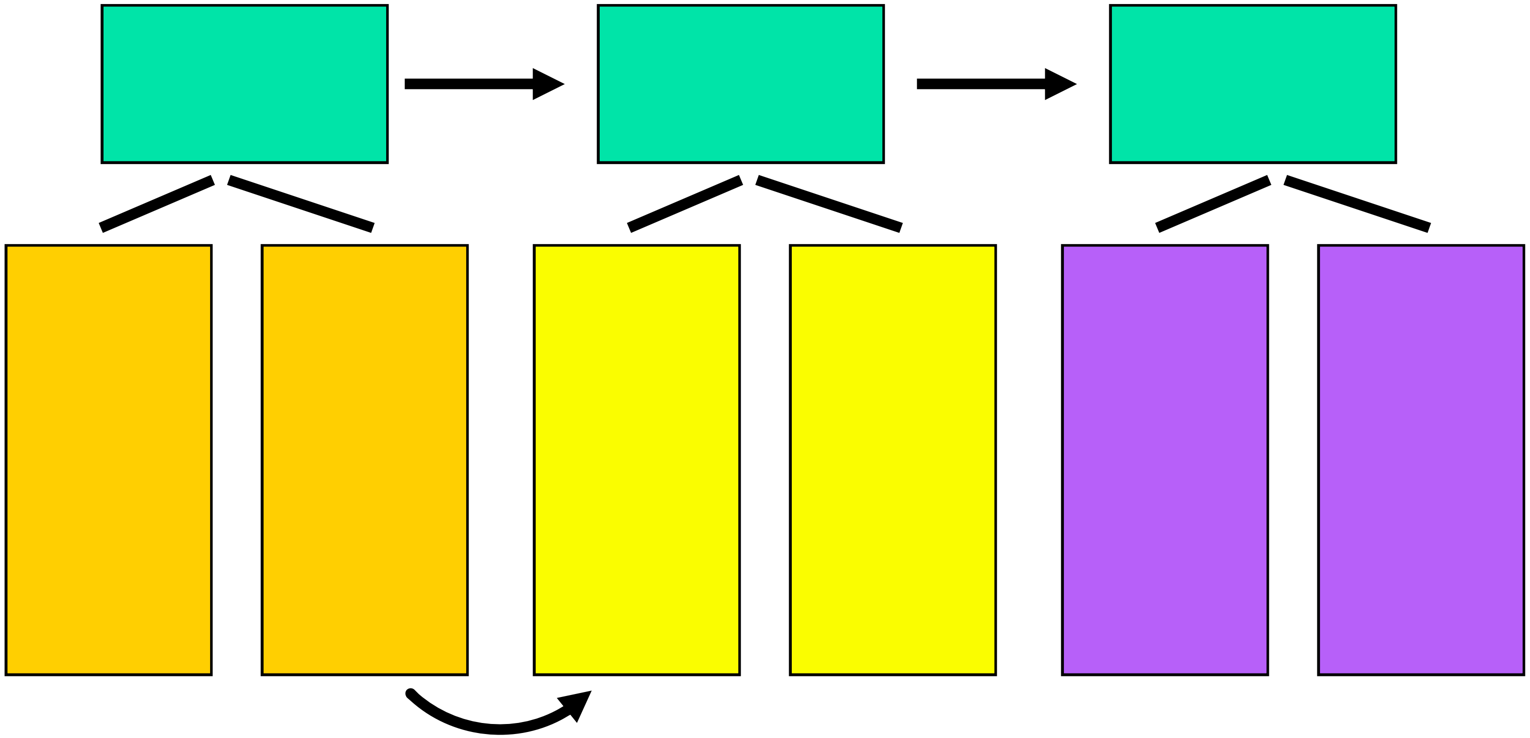
Fast Buffers (Fbuf)[1]

- Tool for speeding up any layered protocol
- Observation: buffer management is a major source of overhead in layered protocols, e.g. ISO style
- Solution: uses memory management, protection to “cache” buffers on frequently used paths
- Stack layers effectively share memory
- Tremendous performance improvement seen

[1]Druschel, Peterson: “Fbufs: a high-bandwidth cross-domain transfer facility”, SIGOPS 1993

Fbufs

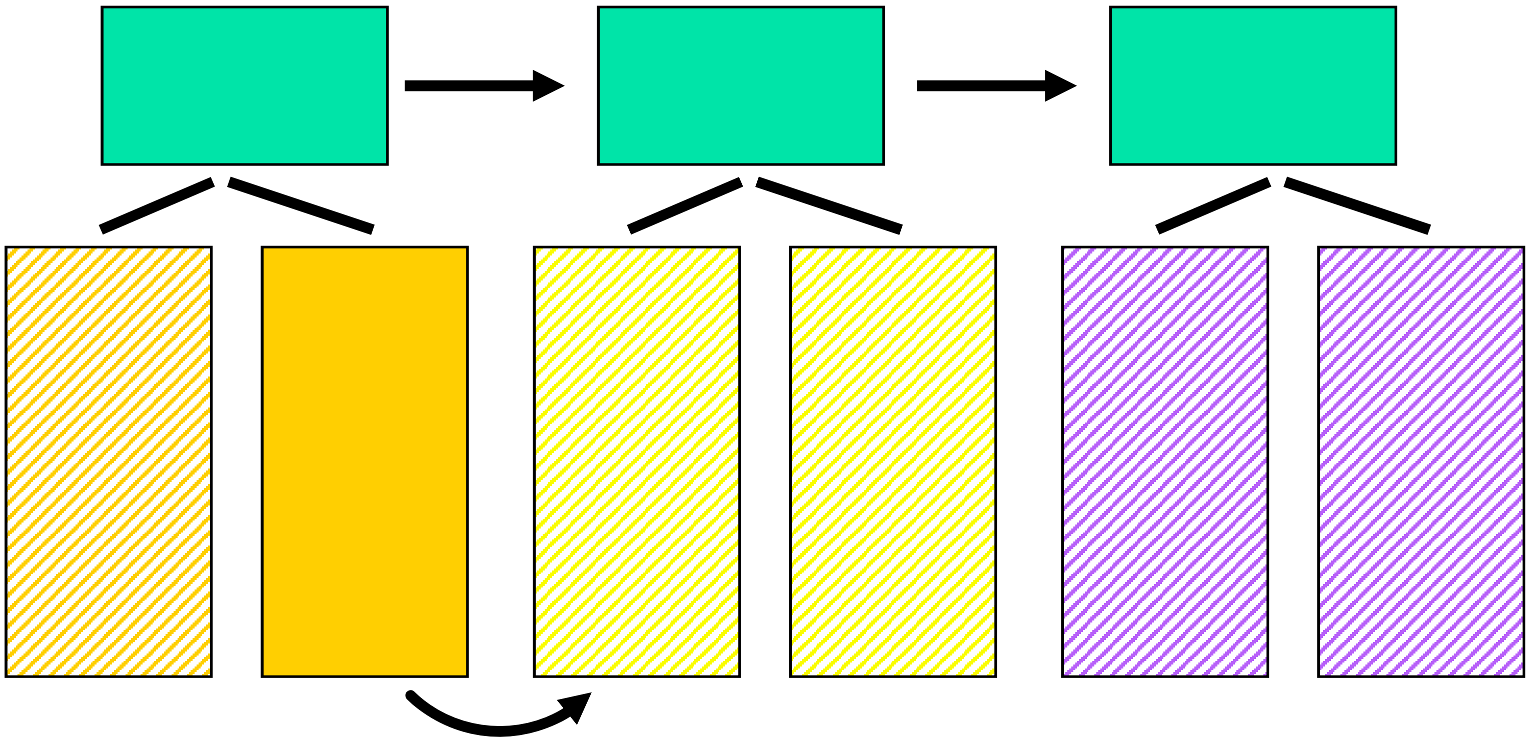
control flows through stack of layers, or pipeline of processes



data copied from "out" buffer to "in" buffer

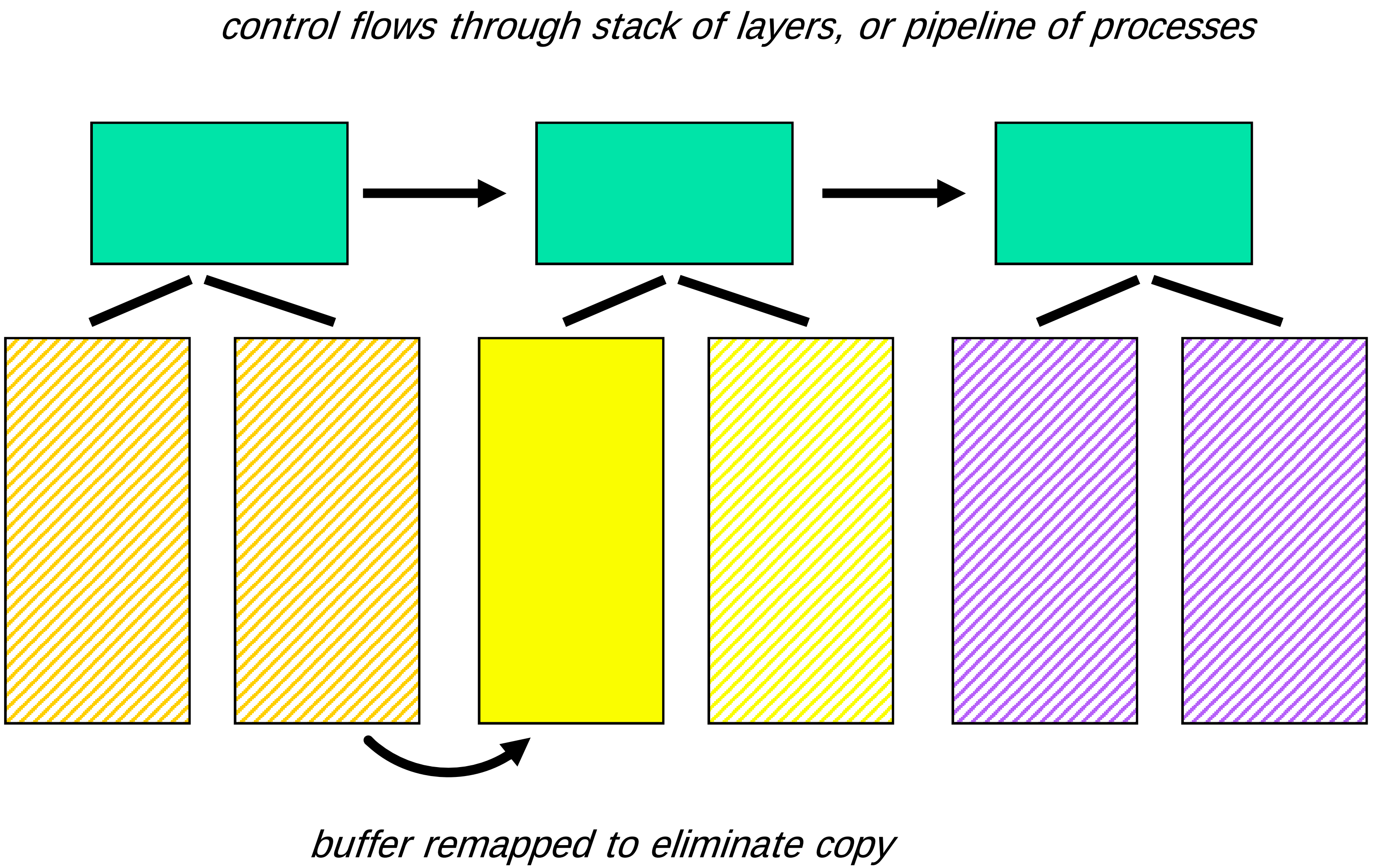
Fbufs

control flows through stack of layers, or pipeline of processes



data placed into “out” buffer, shaded buffers are mapped into address space but protected against access

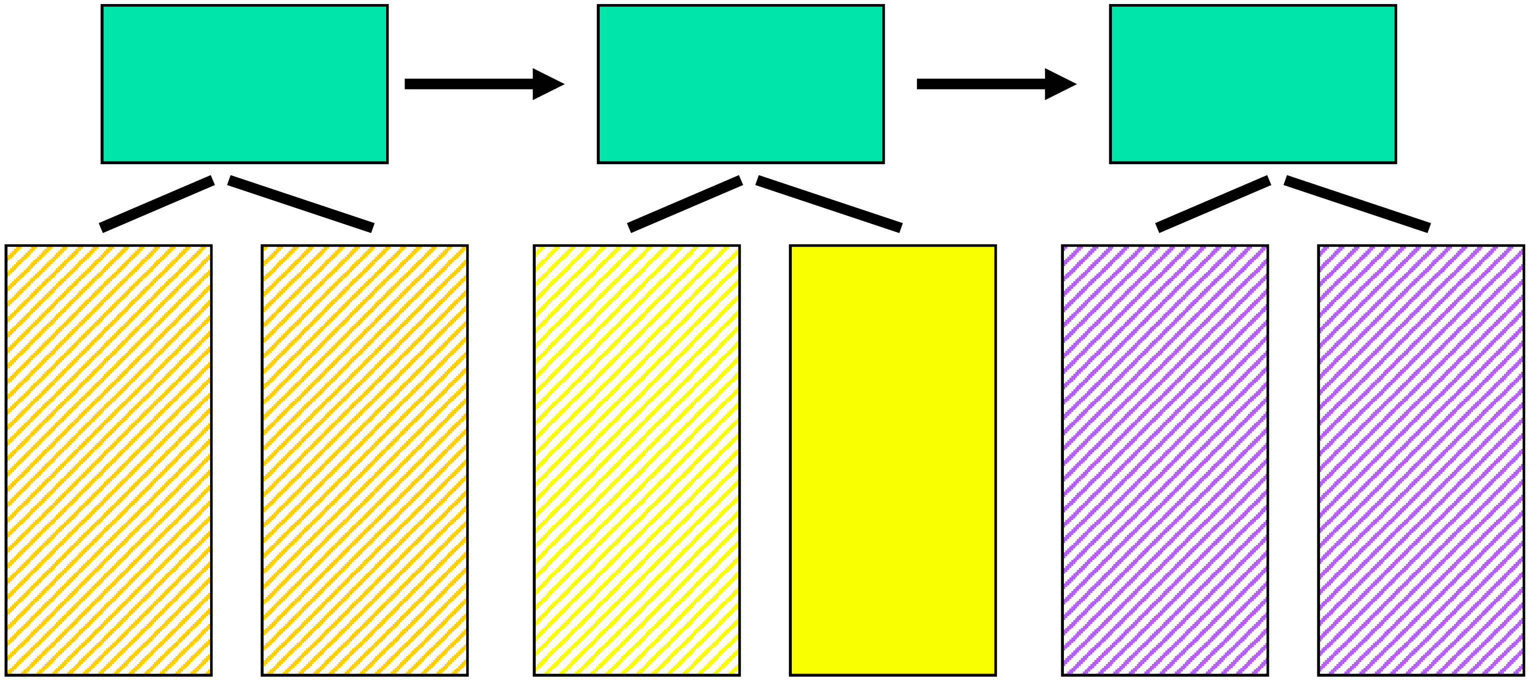
Fbufs



control flows through stack of layers, or pipeline of processes
buffer remapped to eliminate copy

Fbufs

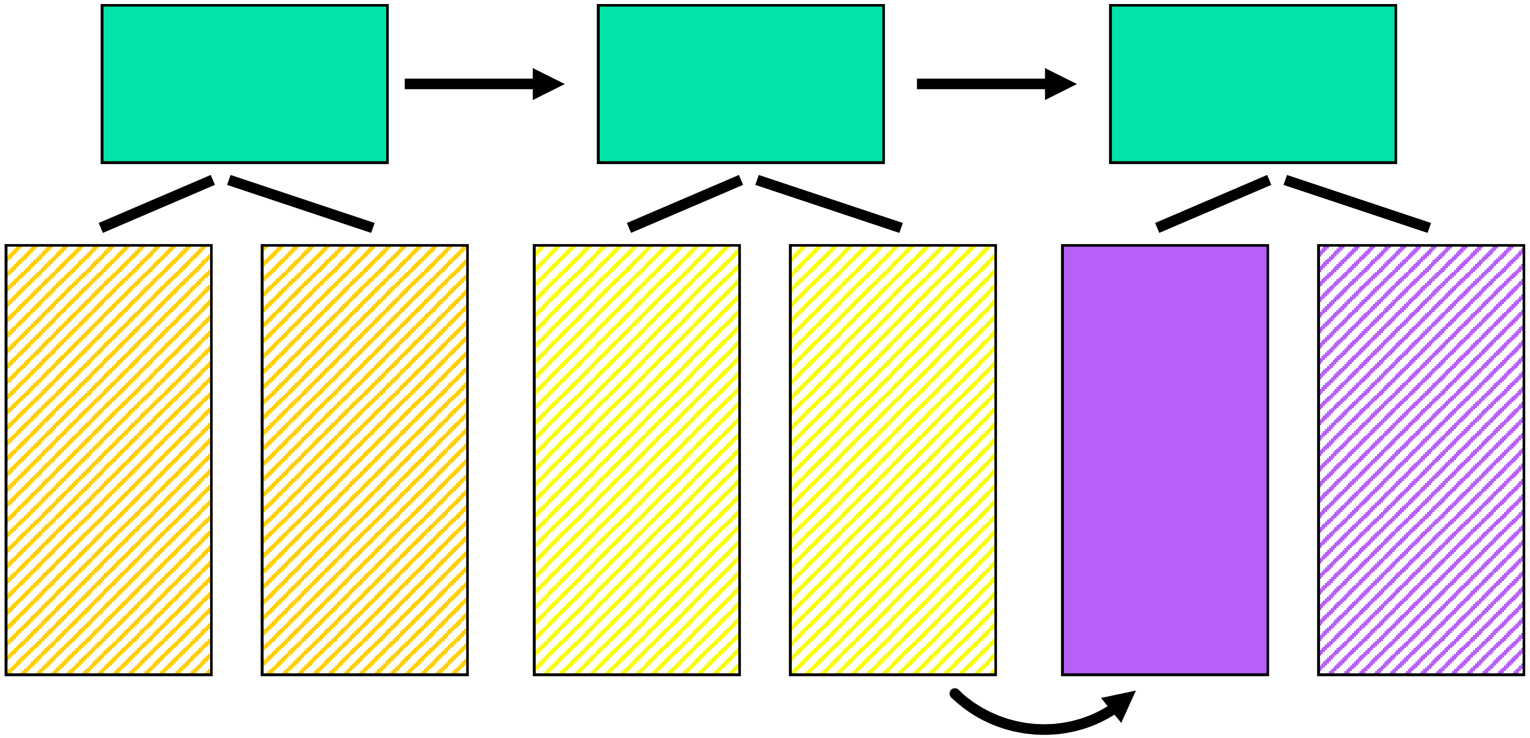
control flows through stack of layers, or pipeline of processes



in buffer reused as out buffer

Fbufs

control flows through stack of layers, or pipeline of processes



buffer remapped to eliminate copy

Where are Fbufs used?

- Although this specific system is not widely used
  - Most kernels use similar ideas to reduce costs of in-kernel layering
  - And many application-layer libraries use the same sorts of tricks to achieve clean structure without excessive overheads from layer crossing

Active Messages[1]

- Concept developed for parallel machines
- Assumes the sender knows all about the destination, including memory layout, data formats
- Message header gives address of handler
- Applications copy directly into and out of the network interface

[1]von Eicken, Culler et aal.: “Active Messages: a Mechanism for Integrated Communication and Computation, *19th International Symp. on Computer Architecture, Gold Coast, Australia, May 1992,* 256-266.

Performance Impact?

- Even with optimizations, standard RPC requires about 1000 instructions to send a null message
- Active messages: as few as 6 instructions! One-way latency as low as 35usecs
- But model works only if “same program” runs on all nodes and if application has direct control over communication hardware

Broad Comments on RPC

- RPC is not very transparent
- Failure handling is not evident at all: if an RPC times out, what should the developer do?
  - Reissuing the request only makes sense if there is another server available
  - Anyhow, what if the request was finished but the reply was lost? Do it twice? Try to duplicate the lost reply?
- Performance work is producing enormous gains: from the old 75ms RPC to RPC over U/Net with a 75usec round-trip time: a factor of 1000!

Contents of an RPC environment

- Standards for data representation
- Stub compilers, IDL databases
- Services to manage server directory, clock synchronization
- Tools for visualizing system state and managing servers and applications

Recent RPC History

- RPC was once touted as the transparent answer to distributed computing
- Today the protocol is very widely used
- ... but it isn't very transparent, and reliability issues can be a major problem
- Today the strongest interest is in Web Services and CORBA, which use RPC as the mechanism to implement object invocation

Appendix

Application and RPC Failures

Examples

Asynchronous RPC

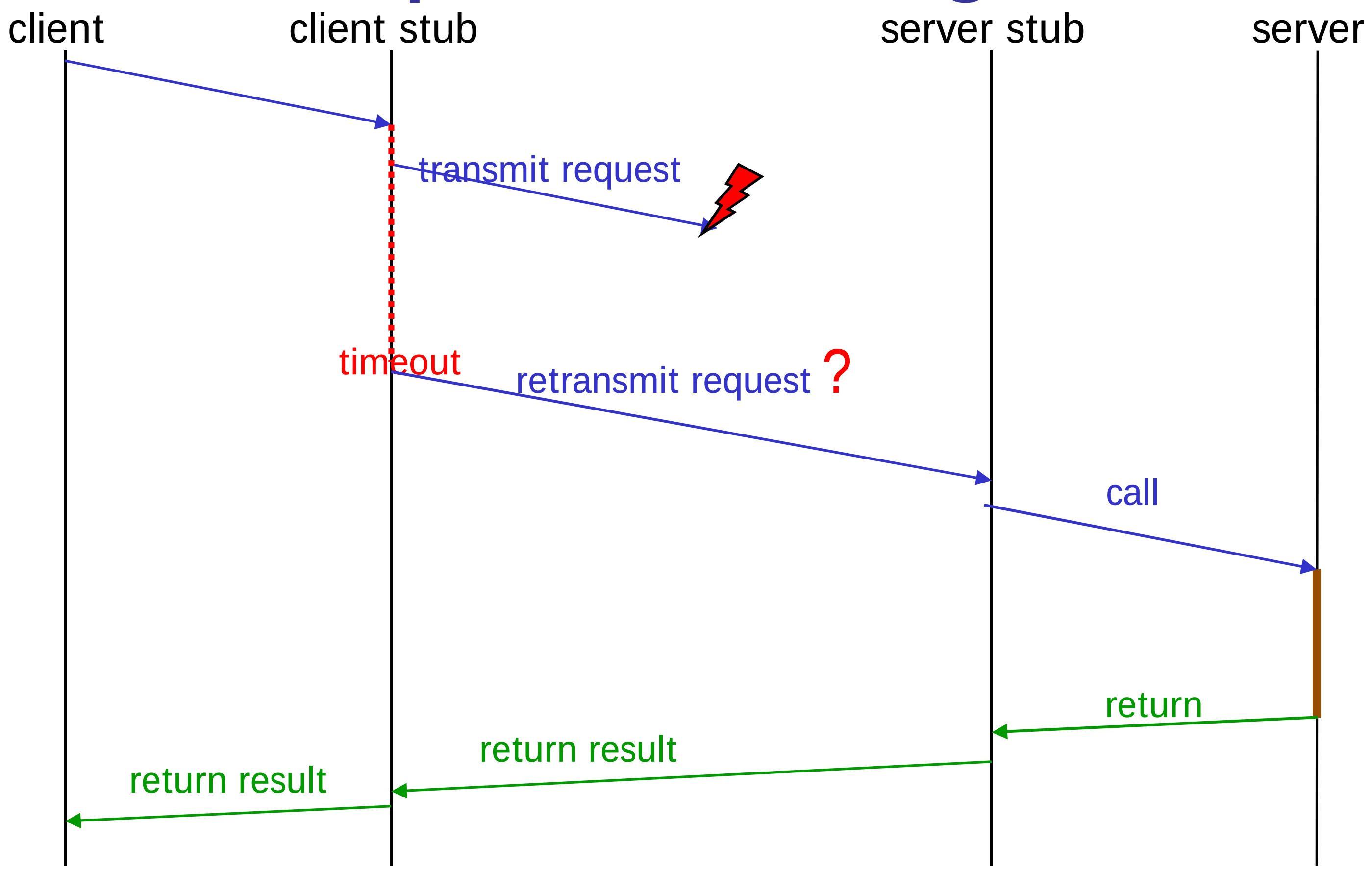
Binding

RPC-Failures with Client/Server

1. Loss of request message
2. Loss of result message
3. Server crash
   1. Before executing the request
   2. After having executed the request
4. Client crash

How to deal with these different cases?

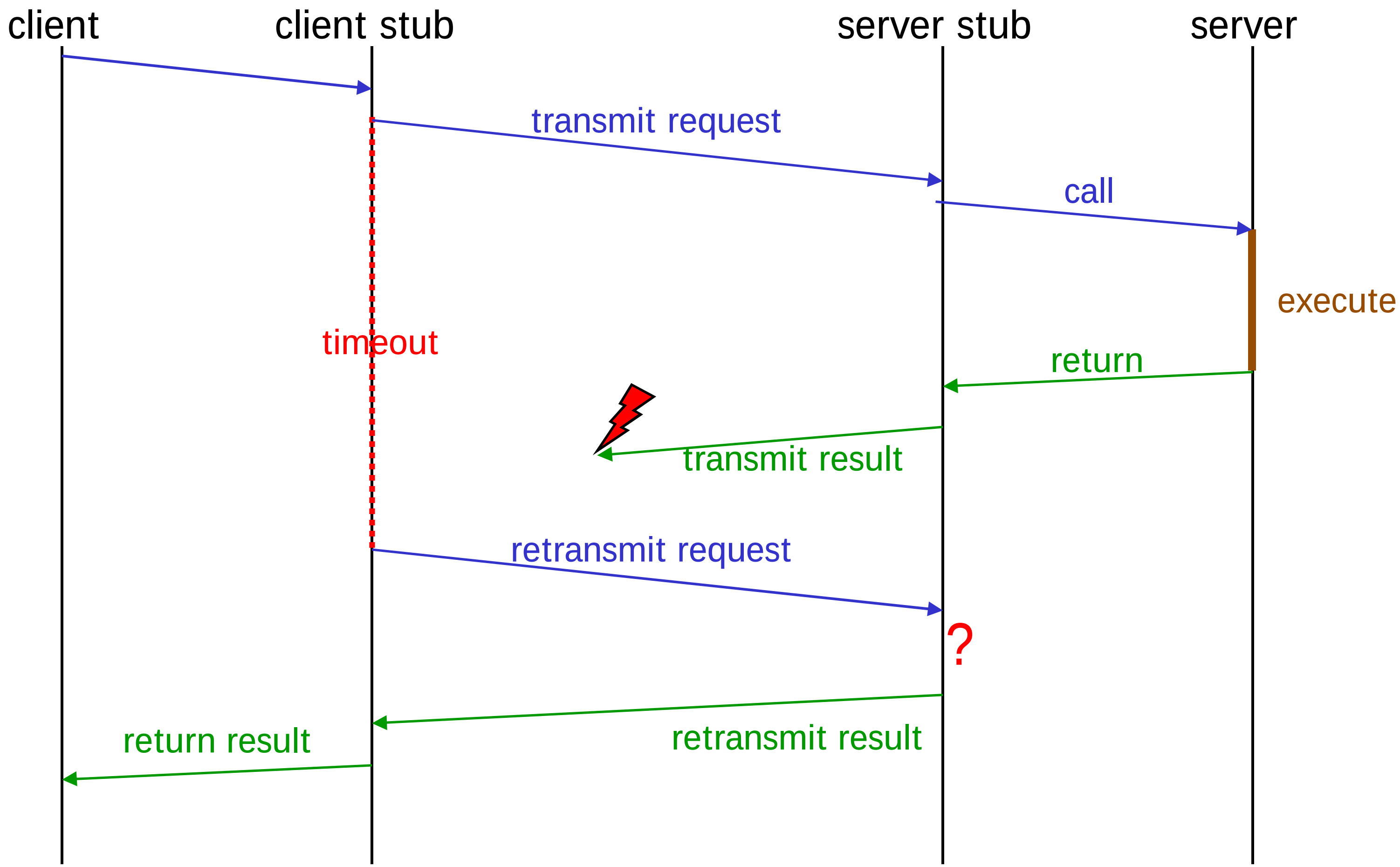
Loss of Request Message



Question: What's not yet specified?

- Optimal value for the timeout
- Sequence number of request *(why still inconvenient?)*

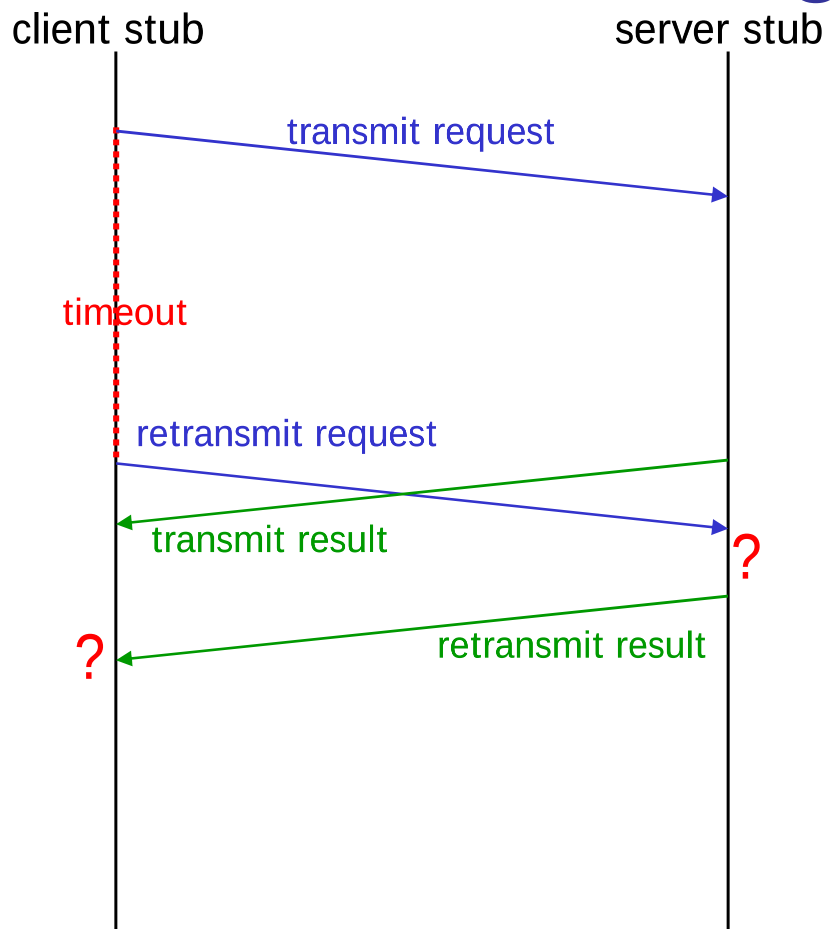
Loss of Result Message



Protocol requires that server stub knows about its previous executions

Question: How long will server stub keep this information?

Lateness of Result Message



Server Crash

Three semantics

- At least once
  - Keep trying until server responds
  - Works OK for idempotent requests
  - RPC is executed once or many times
- At most once
  - Always report error on (assumed) failure
  - RPC might be executed up to one time
- Exactly once
  - RPC is always executed once
  - Not computable

Client Crash

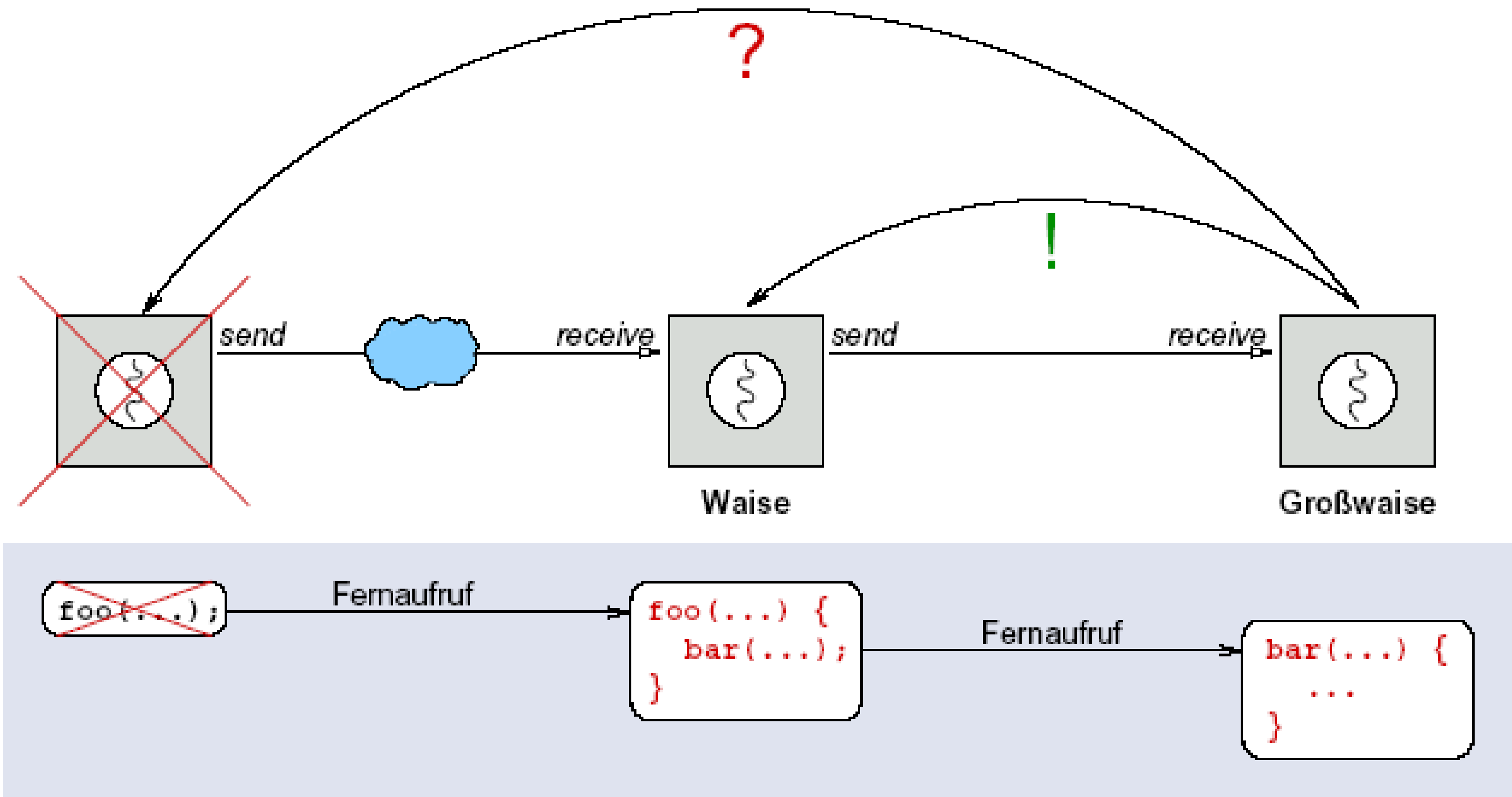
- Client sends a request to the server, then crashes
- Executing process is called an *orphan*
- Ties up server resources

⇒Countermeasures:

1. Additional timeout in server, value might depend on the specific client
2. Manage a crash counter per client
3. Install direct alive-messages from a server to its clients

- *What if client reboots and immediately gets a reply?*
- *Additional difficulties with chains of RPCs*

Orphaned RPC



Client Crash

- Extermination
  - Client keeps a log, kills orphans on reboot
- Reincarnation
  - Client broadcasts the beginning of a new epoch when it reboots
  - All remote processes are tagged with their epoch
- Gentle reincarnation
  - Server kills process at the start of a new epoch
  - Expiration
  - Give each RPC process a quantum T
  - When quantum T expires, the client must be contacted

Summary: RPC & Client/Server

- Client/Server oriented interaction
- Synchronous communication

⇒ some inconveniences:

  - All machines have to be online at the same time
  - No parallel execution
  - Connection overhead
  - Higher probability of failures

- SOAP[1] specification with RPC in mind

⇒ need for a more flexible protocol, e.g. async. RPC

SOAP[1] = simple object access protocol

Examples

DCE RPC

SUN RPC

Example: DCE[1] RPC

- DCE = middleware packkage
  - Intermediate software layer between network-OSes and distributed applications
  - Developed for Unix environments
  - Adopted to other commodity desktop OSes
    - MS Windows
    - DEC VMS
- DCE implemented as a client-server model
  - Services implemented in DCE or at application level
  - All communication between client application and server is done via DCE RPC

[1]DCE Distributed Computing Environment

DCE Services

- Distributed File System
  - Transparent use of files
  - Either mapped to host's own File System
  - Or used instead of
- Distributed Directory Service
  - Get location of all resources in the DS, e.g.
    - Machines
    - Printer
    - Server
    - ...
- Security Service
  - Protects access to resources
- Time Service
  - Synchronize the clocks of all nodes

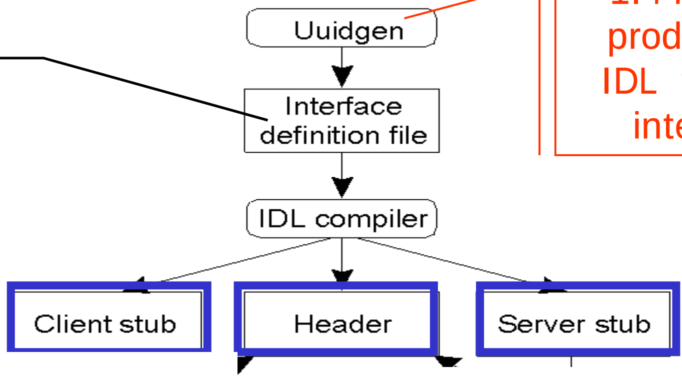
DCE Programming

- DCE RPC system can automatically locate the correct server and set up the communication between client & server (binding)
- DCE uses IDL to support that clients or servers are coded in different languages (e.g. C, C++, Java)
- IDL allows procedure declaration ~ ANSI C
- IDL files contain all necessary information to allow marshalling, flattening etc. needed to install stubs

Writing DCE Client and Server

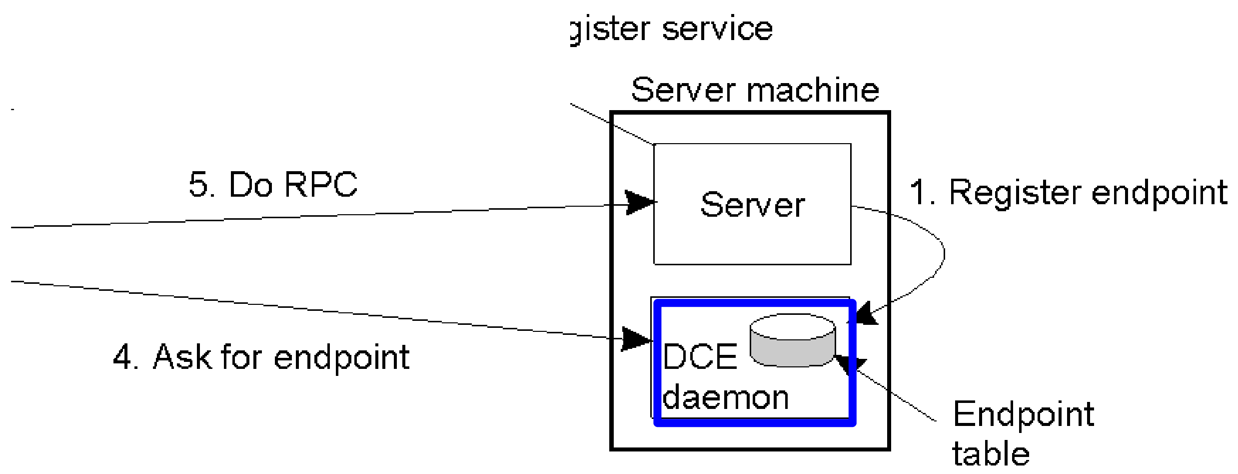
2. Editing the IDL file, i.e. specify names and parameters of all remote procedures

1. Program **uuidgen** produces a prototype IDL file with a unique interface identifier



Steps in writing a client and a server in DCE RPC

Binding Client to Server in DCE



- Client-to-server binding in DCE
- Per server (machine) a DCE daemon

DCE RPC Semantics

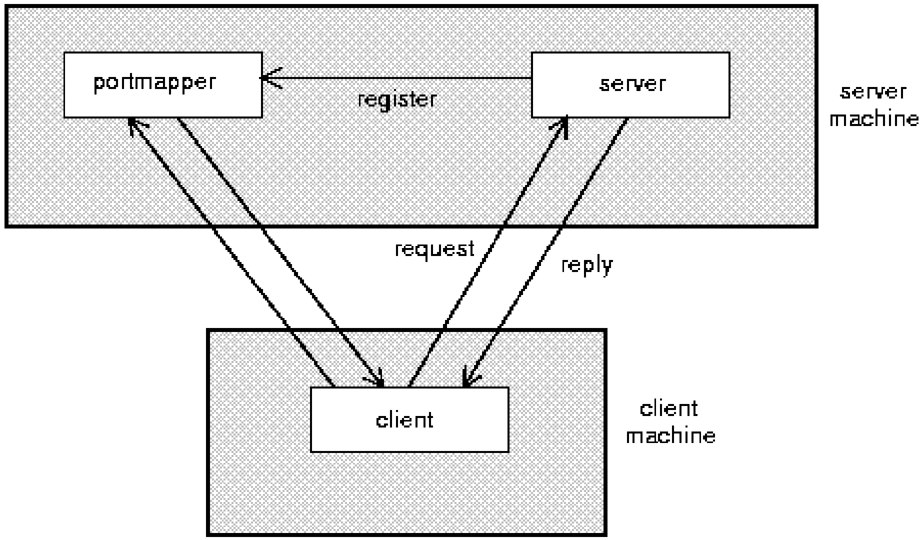
- Default: at-most once, i.e. no call is carried out more than once, even with of system crashes
  - In practice, that means in case of a server crash with quick recovery, the client does not repet the request, for fear it might have been done already
- Alternatively, is remote procedure is marked idempotent (in the IDL file), the request will be repeated multiple times if time out take place
- Alternatively, broadcasting RPC to all machines on the LAN can be used

SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures

Binder: Port Mapper

- Server start-up: create port
- Server stub calls *svc_register* to register prog. #, version # with local port mapper
- Port mapper stores prog #, version #, and port
- Client start-up: call *clnt_create* to locate server port
- Upon return, client can call procedures at the server



RPC Variants

Synchronization

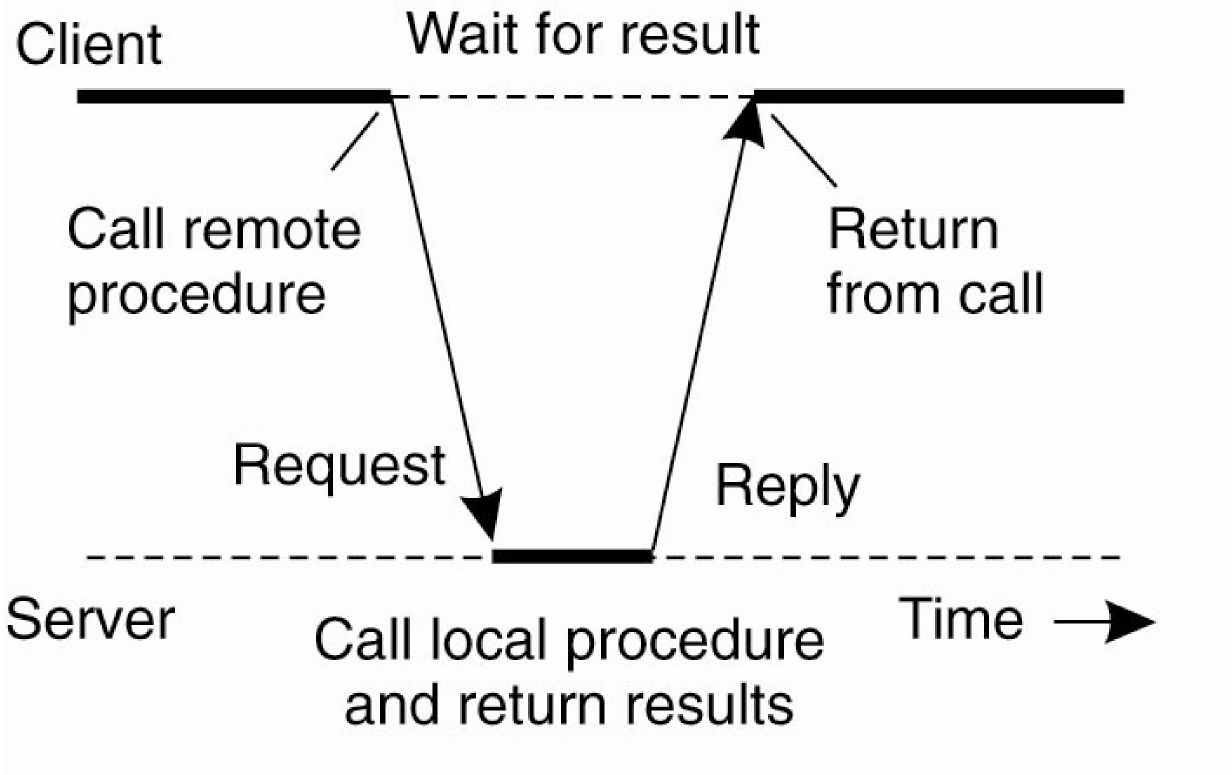
Synchronous RPCs

- `remote-invocation` supports the typical ***request semantics***, i.e.
  - The call via a `send` delivers the request, blocks the caller, and deblocks the callee
  - `receive` in callee accepts the request
  - `reply` delivers result, deblocks caller
- `remote-notification` supports all RPC ***without result***
  - The call via a `send` (see above)
  - `receive` accepts request & deblocks caller

Asynchronous RPC (Promise)

- Asynchronous RPC returns immediately after having sent the request, *promising* to *accept* the result *later*, but without determining when to do
  - **`promise`** object to hold the result
  - State of object **`promise`** is either
    - blocked (result is still missing)
    - ready (result is stored)
  - 2 interface functions to manage object **`promise`**:
    - **`ready()`** delivers state of **`promise`**
    - **`claim()`** blocks a caller as long as promise is blocked;
      when promise will be filled with a result, it deblocks the waiting caller

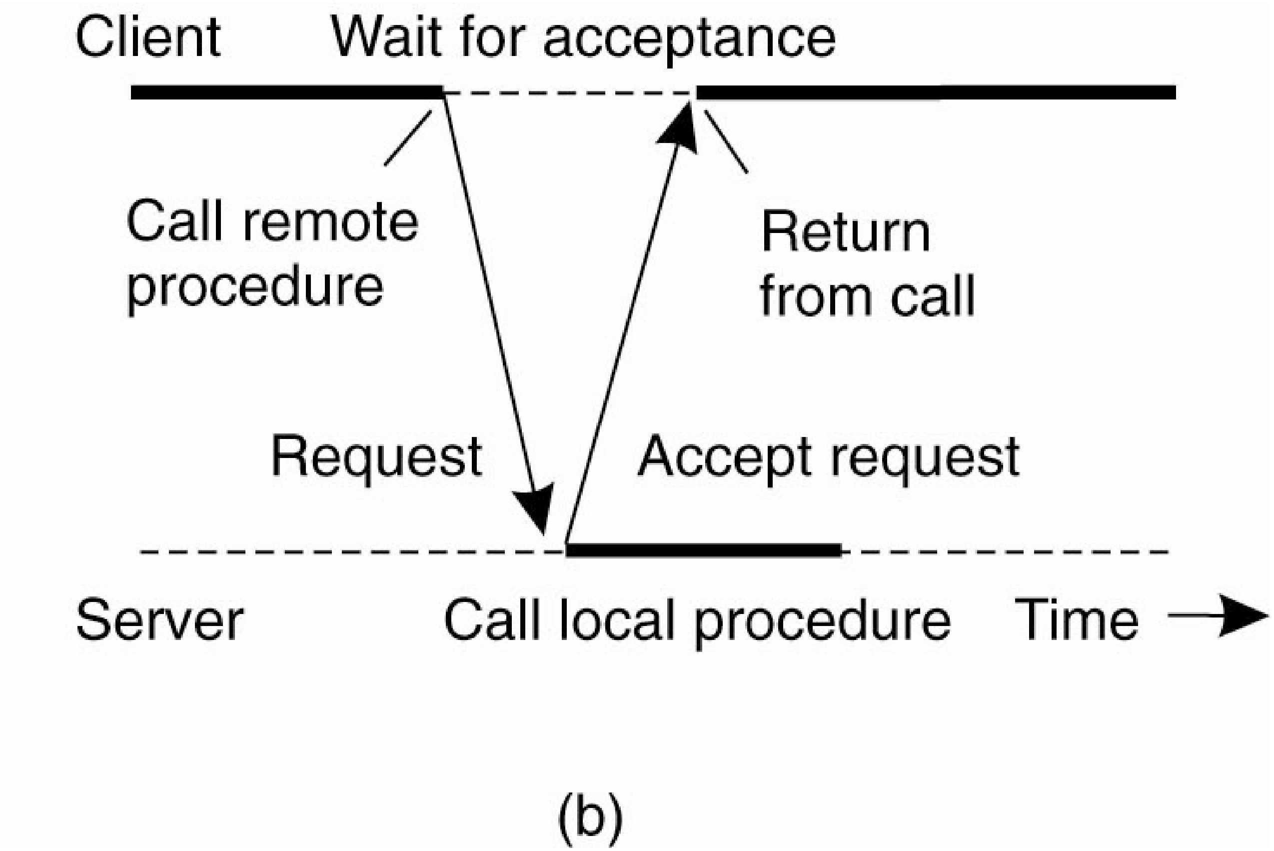
Asynchronous RPC (1)



(a)

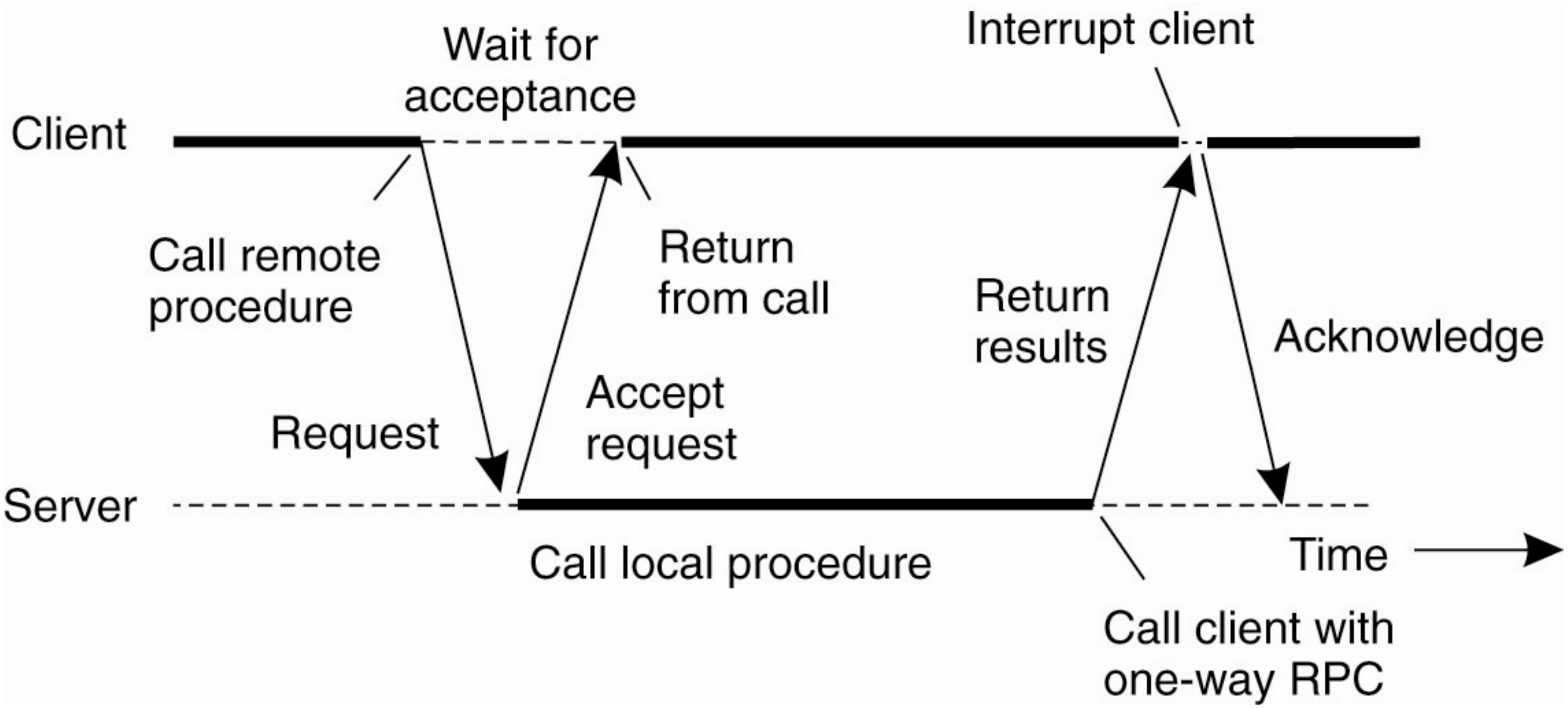
- The interaction between client and server in a traditional RPC

Asynchronous RPC (2)



- The interaction using asynchronous RPC

Asynchronous RPC (3)



- Client & server interacting with 2 asynchronous RPCs

Binding

Static versus Dynamic Bindin

How does Client locate Server?

- Hardwire the server's address into the client
  - Fast but inflexible
- Alternative: Dynamic binding
  - When a server starts executing, it sends a message to a *binder* to make its existence known. This process is referred to as *registering.* To register, the server gives the binder its name, version number, a unique identifier (32-bits), and a *handle* used to locate it
  - The handle is system dependent (e.g. Ethernet address, IP address, X 500 address, ...)

binder interface

call	input	output
register	name, version, handle, unique id	
deregister	name, version, unique id	

Dynamic Binding

- When client calls RPC (e.g. **`read`**) for the first time ⇒
  - Client stub sees that it is not yet bound to the server, so it sends a message to the binder asking to import version xyz of server's interface
  - Binder checks if a server has already exported an interface with the name and the version number
    - If no server will support this interface, read RPC fails
    - If a corresponding server is available, binder gives server's handle and unique identifier to the client stub
  - Client stub uses handle as the address to send its request message to.

binder interface	call	input	output
	lookup	name, version	Handle, unique id

Dynamic Binding

- Advantages
  - Increased flexibility
  - Can support *multiple servers* with same interface, e.g.
    - Binder spreads clients randomly over all servers to even load
    - Binder can poll servers periodically, automatically deregistering servers that are not responding
    - Binder assists in authentication, e.g. a server specifies a list of users, binder will refuse to bind other users to this server
  - Binder can verify that both client and server use the same version of the interface
  - Binder can support load balancing
- Disadvantages
  - Extra overhead: exporting/importing interfaces costs time
  - Binder might become a bottleneck in large DS