Distributed System

Remote Procedure Call (RPC)

- Machine A calls a procedure on machine B
- Calling process on machine A is suspended
- Execution of called procedure takes place on machine B
- Information transportation
- Message passing invisible to the programmer
- Problems
  - Calling and called procedures run on different machines, different address spaces
  - Parameter passing in diverse machines
  - Communication overhead
  - Lack of parallelism
  - Lack of flexibility
- RPC should be transparent: Remote procedure call should look just like a local procedure call. Calling procedure should not be aware of the fact that the called procedure is on another machine
- RPC helps in developing modular system

Remote Procedure Call (RPC)

Conventional procedure call

- count=read(fd, buf, nbytes);
- fd: integer, buf: array of characters, nbytes: integer

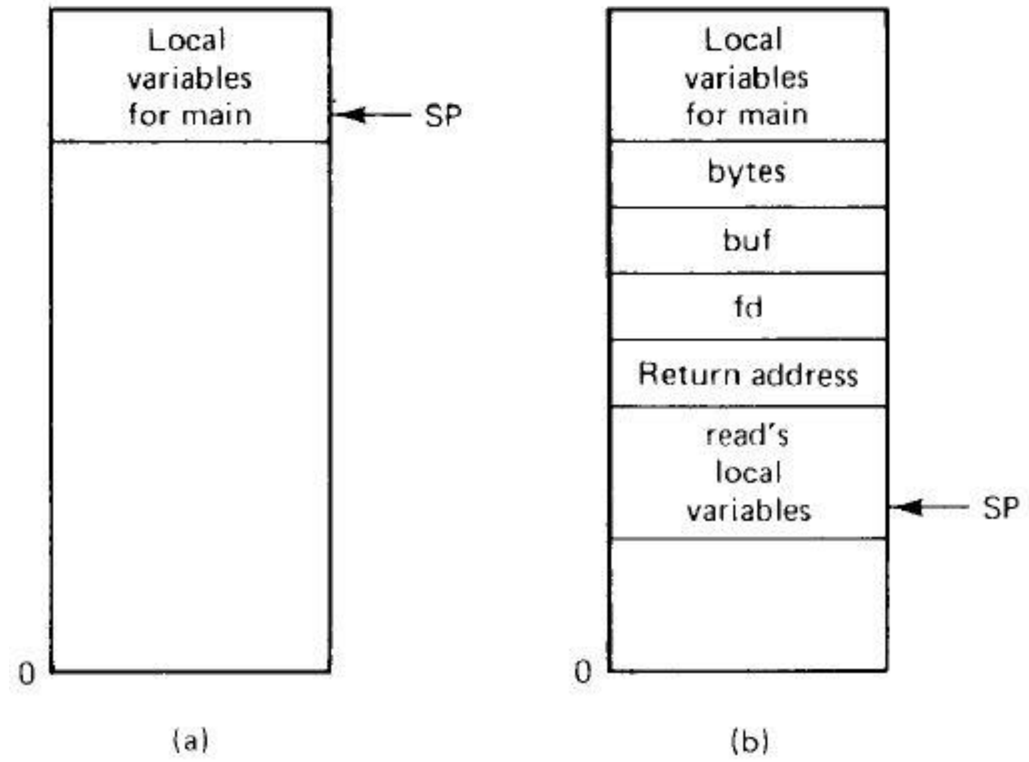


Figure 1: (a) Stack before calling *read*
(b) Stack while calling procedure is active

Remote Procedure Call (RPC)

Steps of RPC

- Client procedure calls the client stub
- Client stub builds a message and traps to the kernel
- Kernel sends the message to remote kernel
- Remote kernel gives the message to server stub
- Server stub unpacks the parameters and calls the server
- Server does the specified work and returns the result to the stub
- Server stub packs it in a message and traps to the kernel
- Remote kernel sends the message to the clients kernel
- Client's kernel gives the message to the client stub
- Client's stub unpacks the result and returns to the client

Remote Procedure Call (RPC)

Steps of RPC

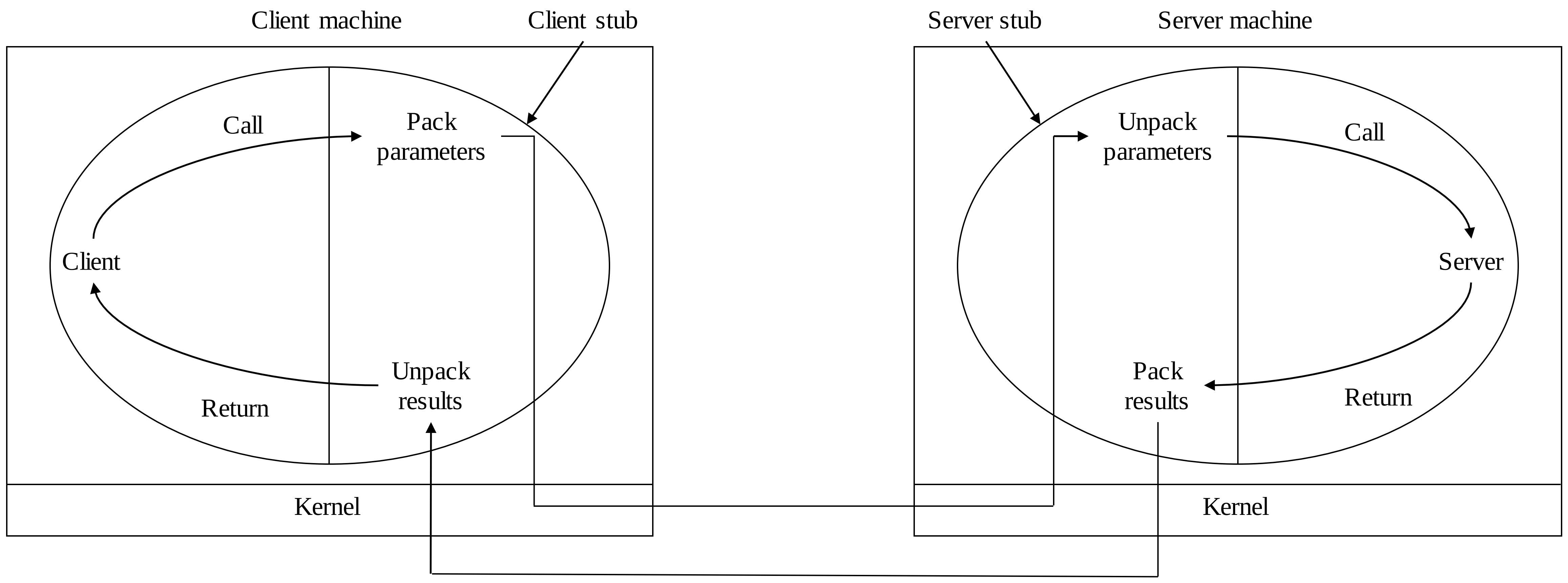


Figure 2: Steps of RPC

Remote Procedure Call (RPC)

Parameter Passing

- Parameter marshalling: Packing parameters into a message
- By taking the parameters from the client, the client stub packs them into a message for sending it to the server stub
- Name or number of the procedure to be called is also included in the message

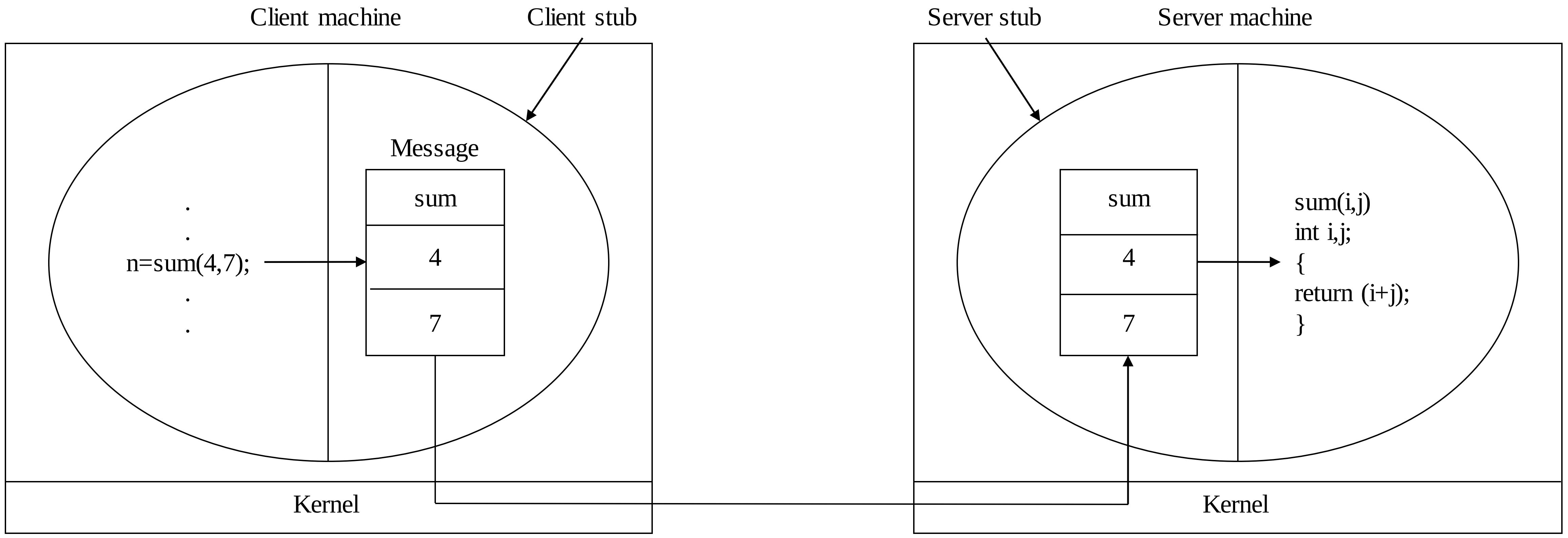
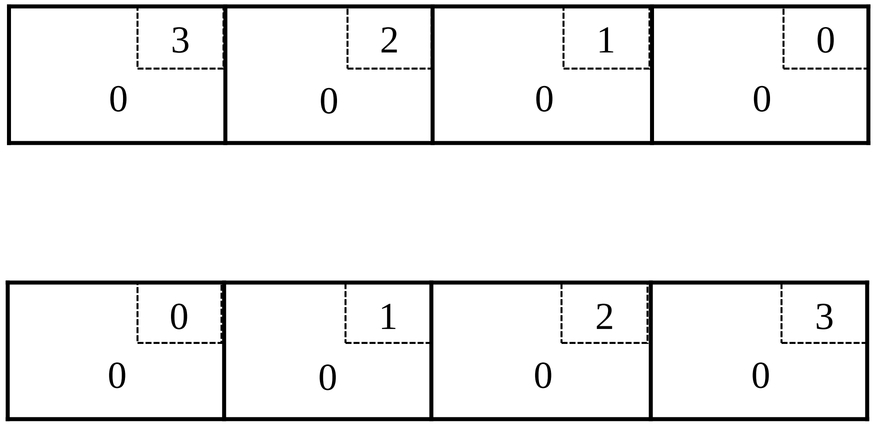


Figure 3: Computing sum(4,7) remotely

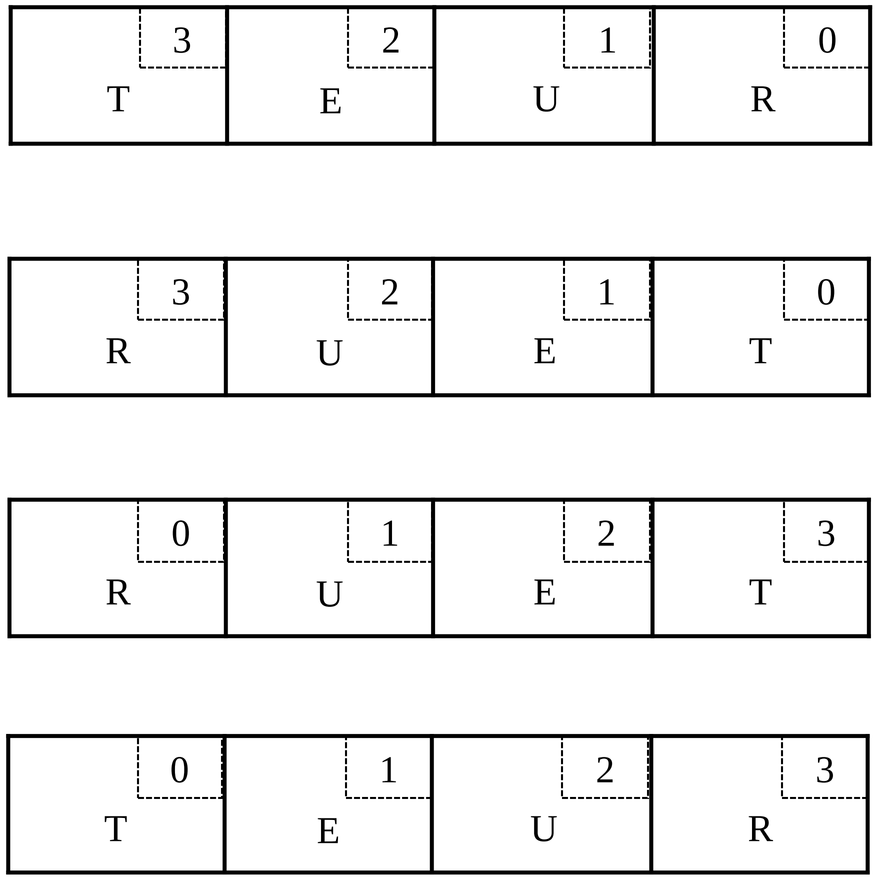
Remote Procedure Call (RPC)

Problems with Different Machines

- Different machines with different representation of data items
- In such case it is not possible to pass data using basic RPC
- How pointers are passed?
- Example: Intel Pentium uses little endian format and Sun SPARC uses big endian format
- Little endian: LSB in smallest address (LSB first)
- Big endian: MSB in smallest address (MSB first)
- String: RUET

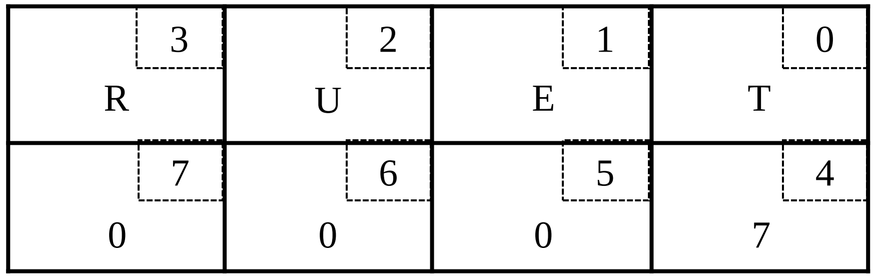


Remote Procedure Call (RPC)

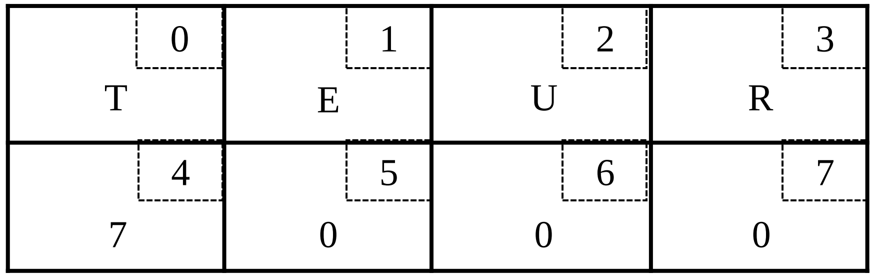


Remote Procedure Call (RPC)

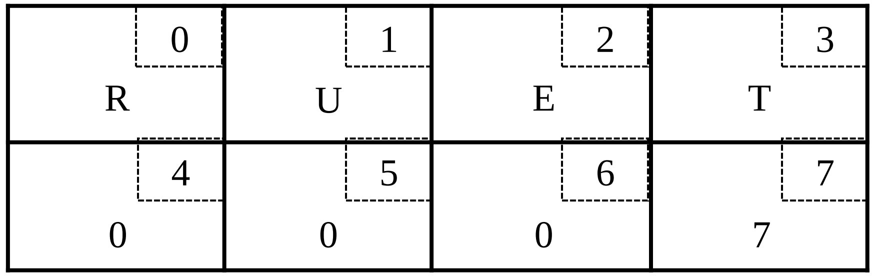
- String: RUET
- Integer: 7



(a)



(b)



(c)

Figure 4: (a) Original message on Pentium
(b) Message after receipt on SPARC
(c) Message after being inverted

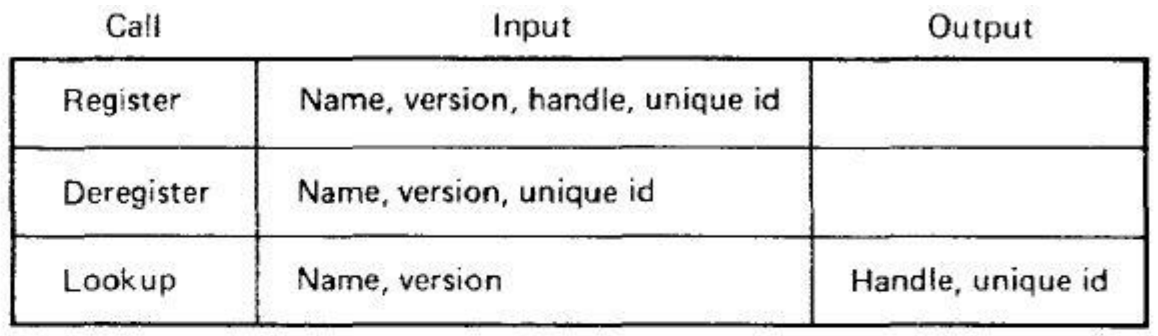
Remote Procedure Call (RPC)

- Connection oriented protocol
- Connectionless protocol
- Packet length
- Extra overhead
- Single large transmission vs multiple small transmissions
- Acknowledgement
    - Stop and wait
    - Cumulative

Remote Procedure Call (RPC)

Dynamic Binding

- How the client knows the location of the server?
- Inform client about the address of the server
- Is it suitable if the server moves or replicates itself?
- Programs would have to be compiled with new address
- Dynamic binding handles the above mentioned issues
- Binder: Server sends a message to binder (a program) to make its existence known
- Registering: The process of server's sending a message to binder
- Server informs the binder about its name, version number, unique id, handle (for locating server) for getting registered
- Handle: IP address, process id, Ethernet address
- Server can also provide authentication related information to binder
- Server can also deregister with the binder



Call	Input	Output
Register	Name, version, handle, unique id	
Deregister	Name, version, unique id	
Lookup	Name, version	Handle, unique id

Figure 5: Binder interface

Remote Procedure Call (RPC)

Dynamic Binding

- How does the client locate the server?
- A message is sent to the binder asking to import any server with any specific name and version
- Binder checks whether there is any server with such name and version
- If there is no such server or no server is willing to support the client then the client’s call is failed
- Or if suitable server exists, binder gives handle and unique id of the server to the client stub
- Fault tolerance: Periodically deregister any inactive server
- Bottleneck problem
- Multiple binders

Remote Procedure Call (RPC)

Asynchronous RPC

- Synchronous communication
- Unnecessary when no result is supposed to return to the client
- Asynchronous RPC: Client continues immediately after issuing RPC
- Upon receiving RPC request, server immediately sends reply to client
- Then the server calls the requested local procedure
- Reply acts as an acknowledgement
- Client continues after receiving acknowledgement from server
- Shorter blocking period

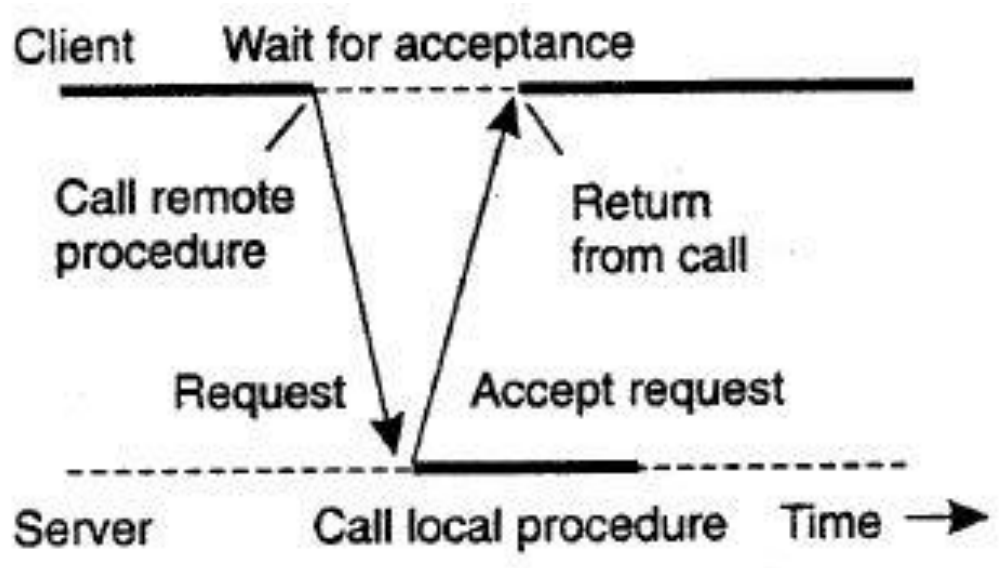


Figure 6: Interaction using asynchronous RPC

Remote Procedure Call (RPC)

One-way RPC

- Variation of asynchronous RPC
- Client continues immediately after sending request to the server
- Client does not wait for acknowledgement (of the request) from the server
- Client can not know whether its request will be processed by the server or not as acknowledgements are not considered

Deferred Synchronous RPC

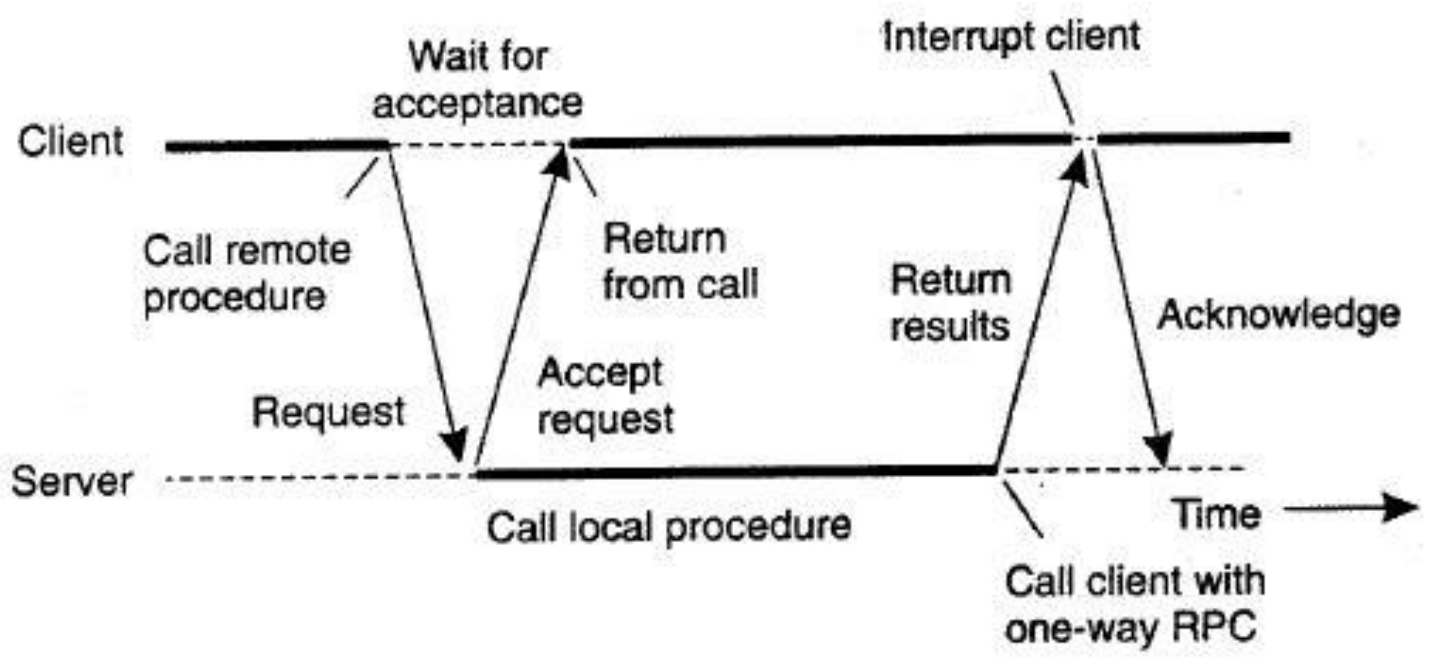


Figure 7: Client server interaction through two asynchronous RPCs

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Client is unable to locate the server
- Request message from the client to the server is lost
- Reply message from the server to the client is lost
- Server crashes after getting a request
- Client crashes after sending a request

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Client is unable to locate the server
  - Server down
  - Client is compiled using a particular version of the client stub. New version of the interface is installed in the server and new stubs are generated. When client runs, binder fails to match it up with a server.
  - If a procedure returns -1 it indicates a failure
  - Global variable errno can indicate failure (UNIX)
  - Inform caller by an error message ‘Cannot locate server’
  - Sometimes -1 can be the desired outcome of a call
  - Exceptions can be used
  - Exceptions are not available in every language
  - Transparency issue

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Request message from the client to the server is lost
  - Why a request message might get lost?
  - Timer
  - Starting a timer at the point of sending request
  - If timer expires before a reply or acknowledgement, then retransmission is done
  - If so many requests are lost kernel gives up by assuming the server is down

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Reply message from the server to the client is lost
  - Using a timer just as lost requests
  - Client’s kernel is not sure about the reason of getting no reply (request, reply, slow server)
  - Idempotent: Request that can be executed several times without any harm
  - Read operations
  - Nonidempotent: Transferring money
  - Sequence number
  - Client’s kernel assigns each request a sequence number
  - Server’s kernel keeps track of most recently received sequence number from each client
  - Server’s kernel can find out a request second time and refuse to perform it
  - A bit in the message header to distinguish initial requests from retransmissions

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Server crashes after getting a request
  - Request arrives, request carried out, reply sent
  - Request arrives, carried out, server crashes before sending reply
  - Request arrives, server crashes before carrying out
  - At least one semantics
  - At most one semantic
  - Guarantee nothing
  - Exactly one semantics

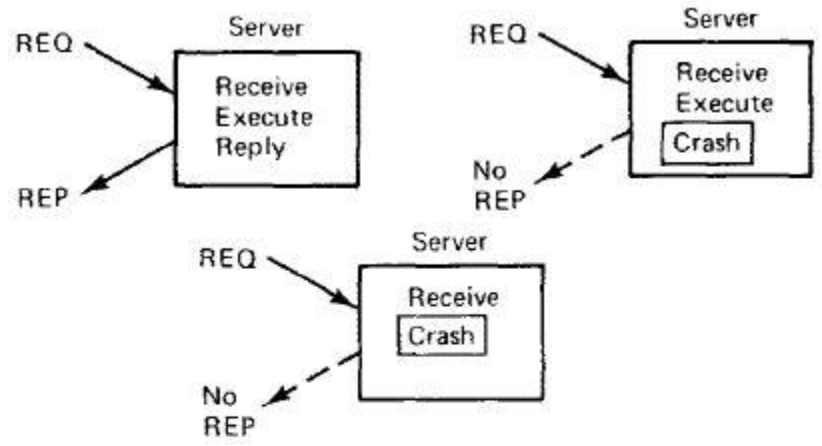


Figure 8: Server crashing scenarios

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Client crashes after sending a request
  - Orphan: Client sends a request, crashes before getting reply. Computation/operation is active but none waiting for it
  - Waste of CPU cycles
  - Tie up valuable resources
  - Client reboots, does RPC, reply from orphan comes and creates confusion (client gets confused between real result and orphan)
  - Extermination: Client stub makes a log entry before RPC regarding what it is about to do. Log is kept on a storage medium that can survive crash. After reboot, log is checked to identify and kill orphan.
  - Writing a disk record for every RPC is expensive
  - Orphans may do RPC and create grandorphans or further descendants which are tough to be tracked
  - Reincarnation: Divide time into sequentially numbered epochs. After a reboot, a client broadcasts a message to all machines to declare the start of a new epoch. Upon receiving such a broadcast message, all remote computations are killed.
  - No need to write a disk record for every RPC
  - Gentle reincarnation: Upon receiving such a broadcast message, each machine checks whether there is any remote computations or not. If there is any such remote computation, it is tried to locate the owner. The computation is killed only if the owner can not be located.

Remote Procedure Call (RPC)

RPC in the Presence of Failures

- Client crashes after sending a request
  - Expiration: RPC is given a standard amount of time to do the task. After a crash, server waits for a T period of time before rebooting. All orphans are sure to be gone within this time.
  - How to choose the value of T?
  - If orphan has obtained locks then killing that orphan might result in infinite locking
  - Killing an orphan might not remove all traces of it

Remote Procedure Call (RPC)

Critical Path

- Sequence of instructions that is executed on every RPC
- Prepare message buffer: Acquiring a buffer where outgoing message can be assembled. It may be a single fixed buffer or one among the partially filled buffers.
- Parameters are converted to appropriate format and inserted into message buffer
- Header files are also inserted into message buffer
- Context switching: CPU registers and memory maps are saved. New memory map is set up for using in kernel mode.
- Kernel and user contexts are disjoint
- Kernel copies the message into its address space, fills in destination address and header files, copies it to network interface.
- Incoming bits are stored in memory or buffer.
- Upon receiving all the bits an interrupt is generated.

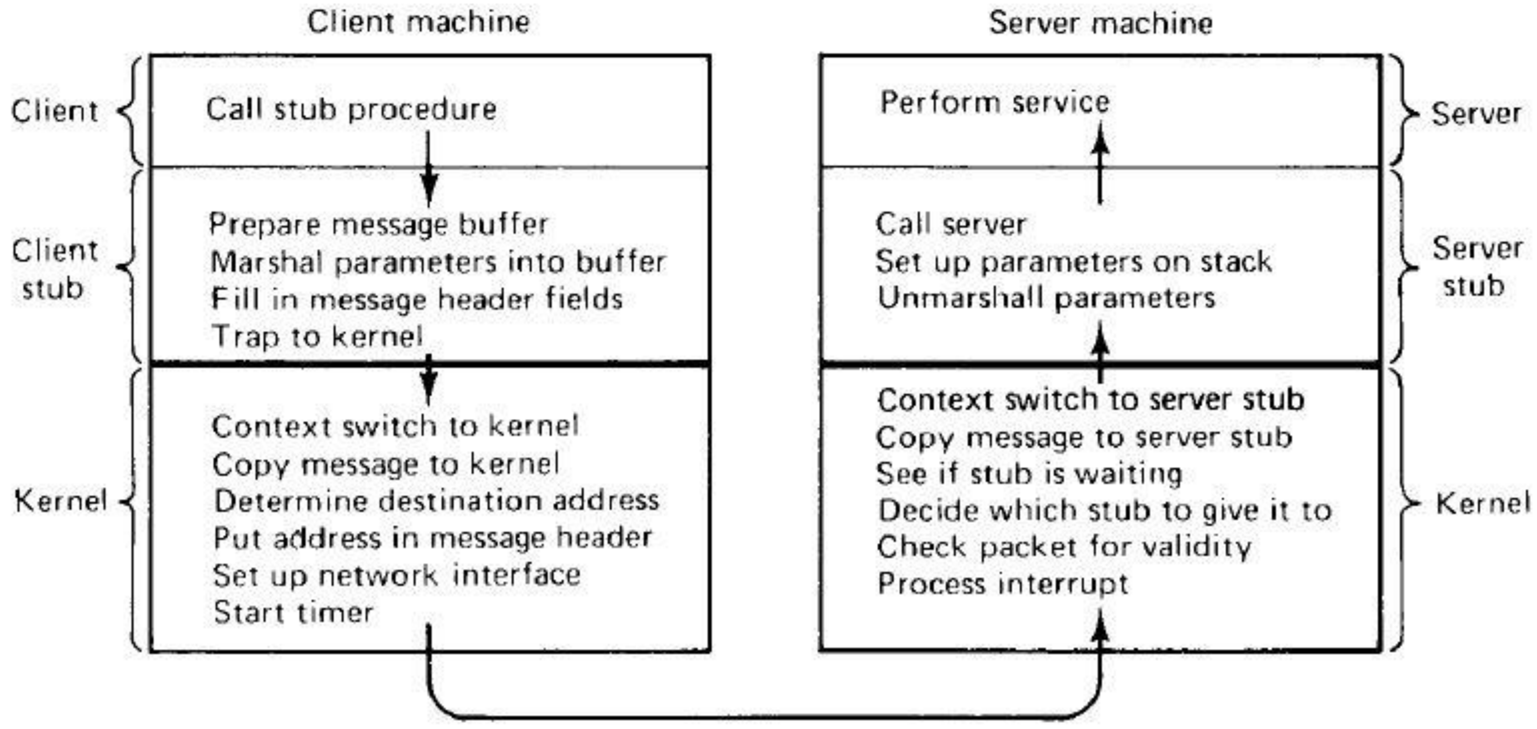


Figure 9: Critical path from client to server

Remote Procedure Call (RPC)

Critical Path

- Interrupt handler checks the validity of incoming packet and determines the recipient stub
- Message copied to the stub
- Context switching
- Unmarshals the parameters
- Calling the server
- Path back to client after server run

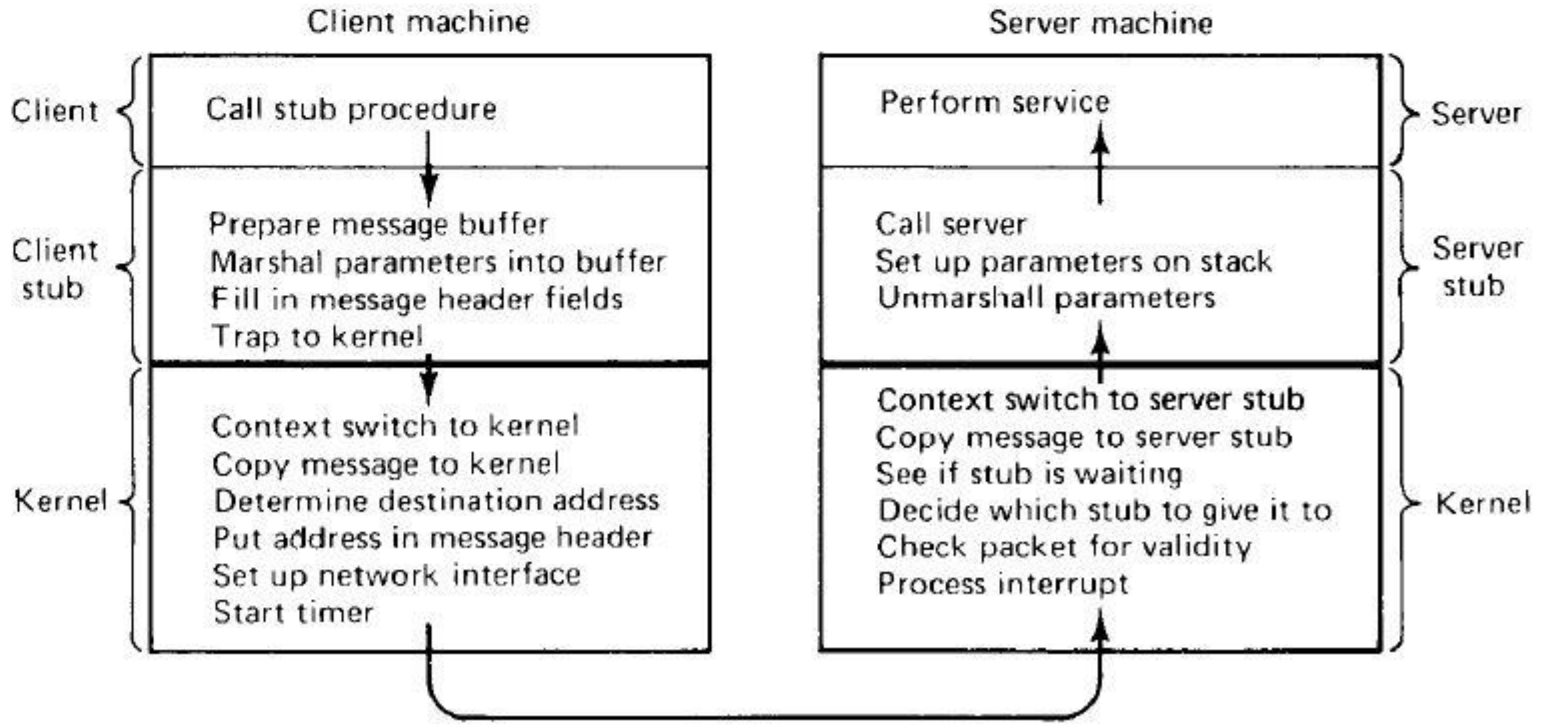
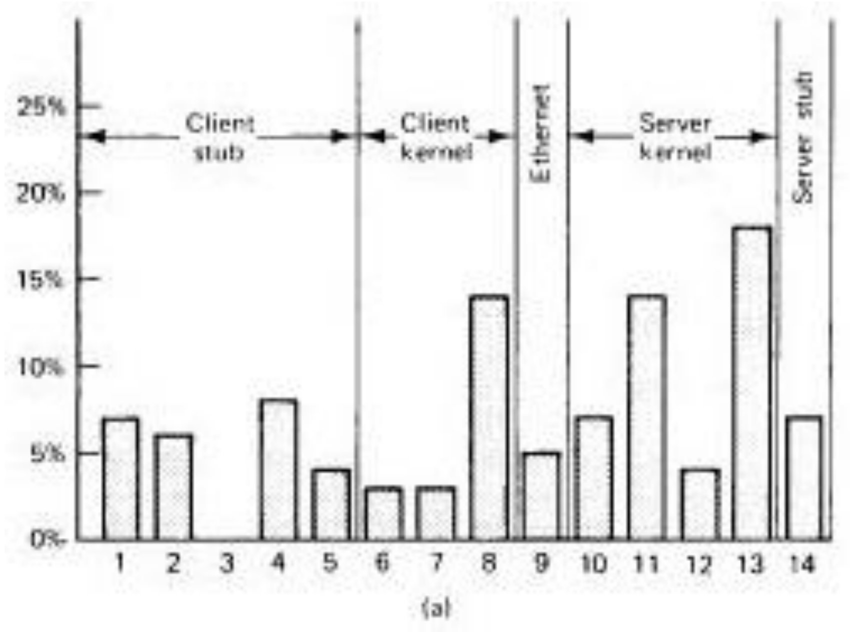
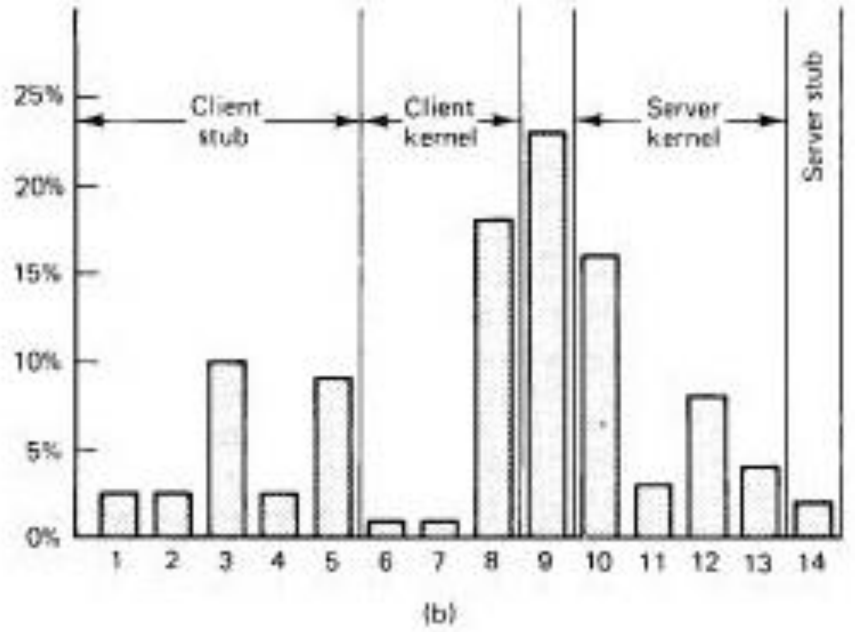


Figure 9: Critical path from client to server

Remote Procedure Call (RPC)

Critical Path





1. Call Stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the QBus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

Figure 10: Breakdown of a RPC critical path
(a) For a null RPC (b) For an RPC with 1440 byte array parameter
(c) Steps of RPC from client to server