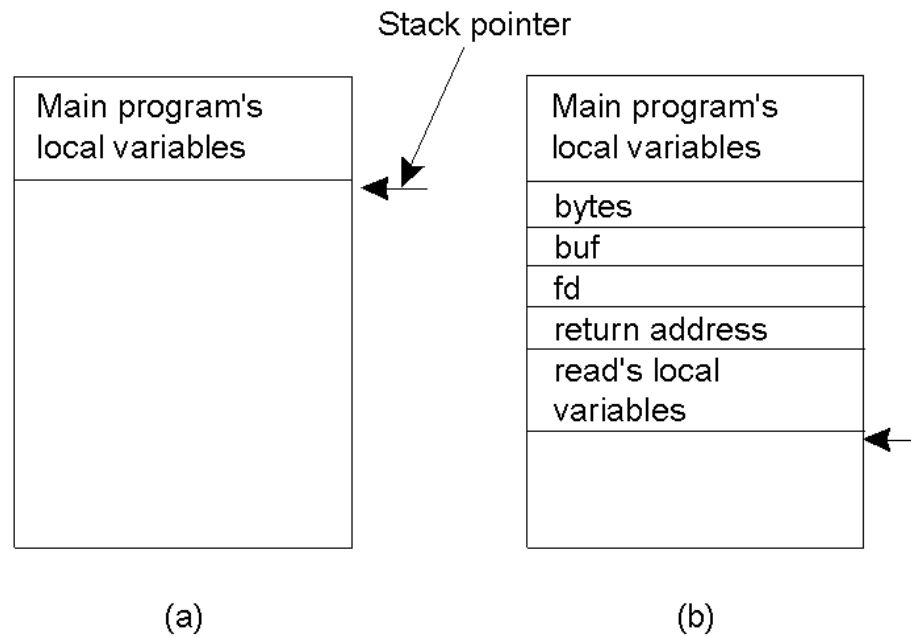


The Remote Procedure Call model

- Distributed systems -> processes interact via messages
 - Remote services – access to a file
- However the communication is implicit
 - RPC looks and smells like a local procedure call
 - Conceptually simple to implement, the devil is in the details
- A RPC has multiple functions
 - Data transport
 - Synchronization of executions of processes

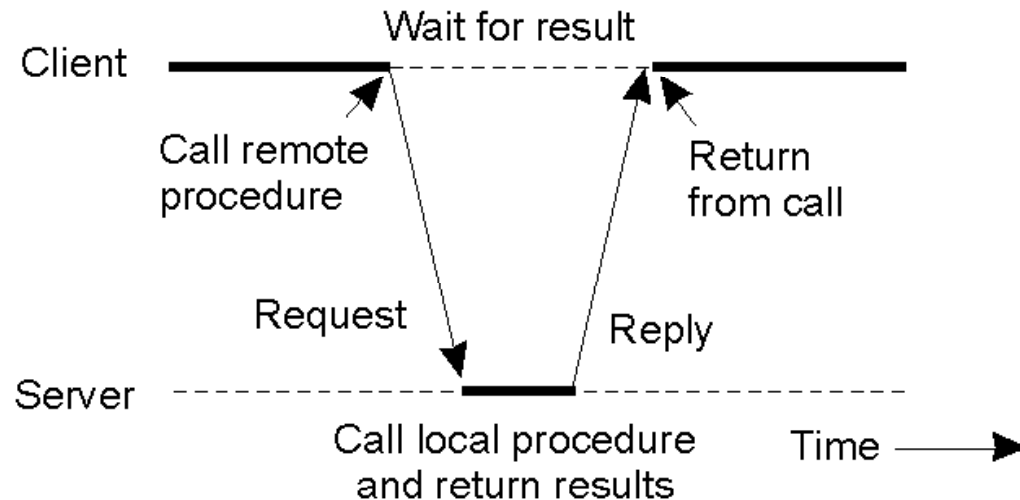
Conventional Procedure Call

- a) Parameter passing in a local procedure call: the stack before the call to read
- b) The stack while the called procedure is active



Client and Server Stubs

- Principle of RPC between a client and server program.
- Client and server stubs (intermediate auxiliary functions) used to build the illusion of a local procedure call
- Main task of stub: packing and unpacking of parameters and results in messages

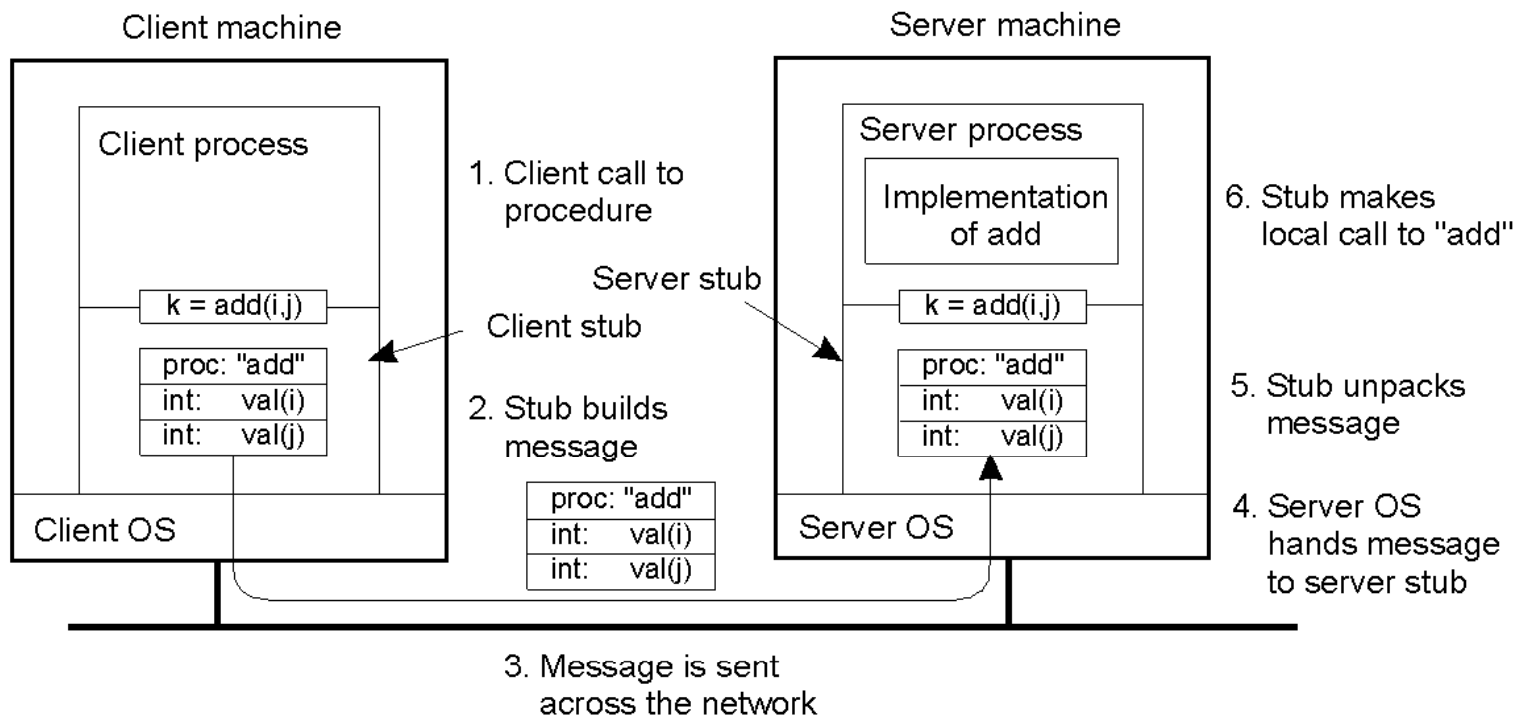


Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Passing Value Parameters (1)

- Steps involved in doing remote computation through RPC



Passing Value Parameters (2)

A major complication is the different byte ordering adopted in different processor architectures

- Original message on the Pentium
- The message after receipt on the SPARC
- The message after being inverted. The little numbers in boxes indicate the address of each byte

3	2	1	0
0	0	0	5
7	6	5	4
L	L	I	J

(a)

0	1	2	3
5	0	0	0
4	5	6	7
J	I	L	L

(b)

0	1	2	3
0	0	0	5
4	5	6	7
L	L	I	J

(c)

Parameter Specification and Stub Generation

- a) A procedure
- b) The corresponding message.

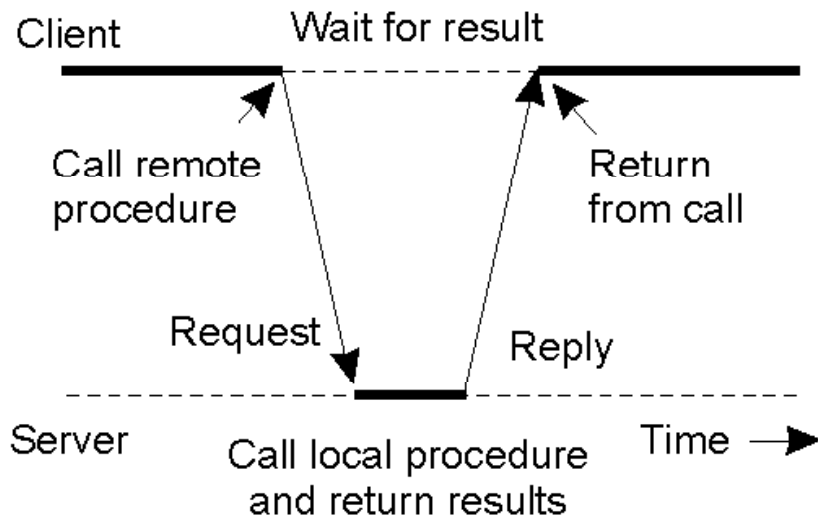
```
foobar( char x; float y; int z[5] )  
{  
  ....  
}
```

(a)

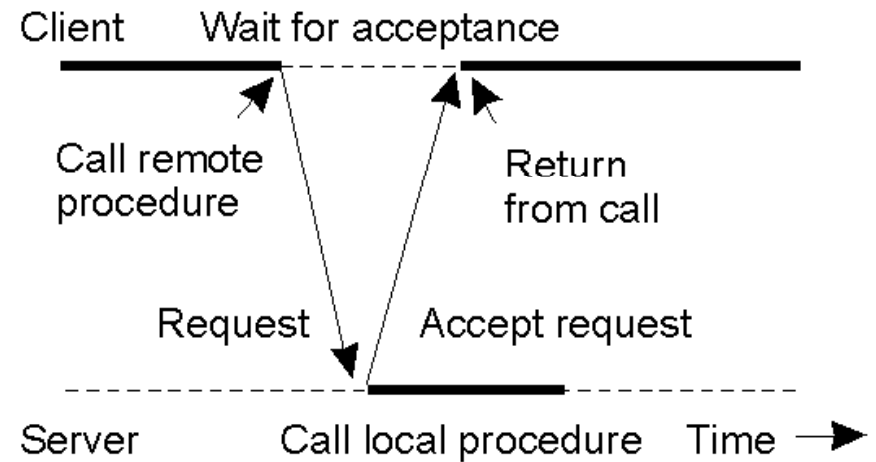
foobar's local variables	
	x
y	
5	
z[0]	
z[1]	
z[2]	
z[3]	
z[4]	

(b)

Asynchronous RPC (1)



(a)

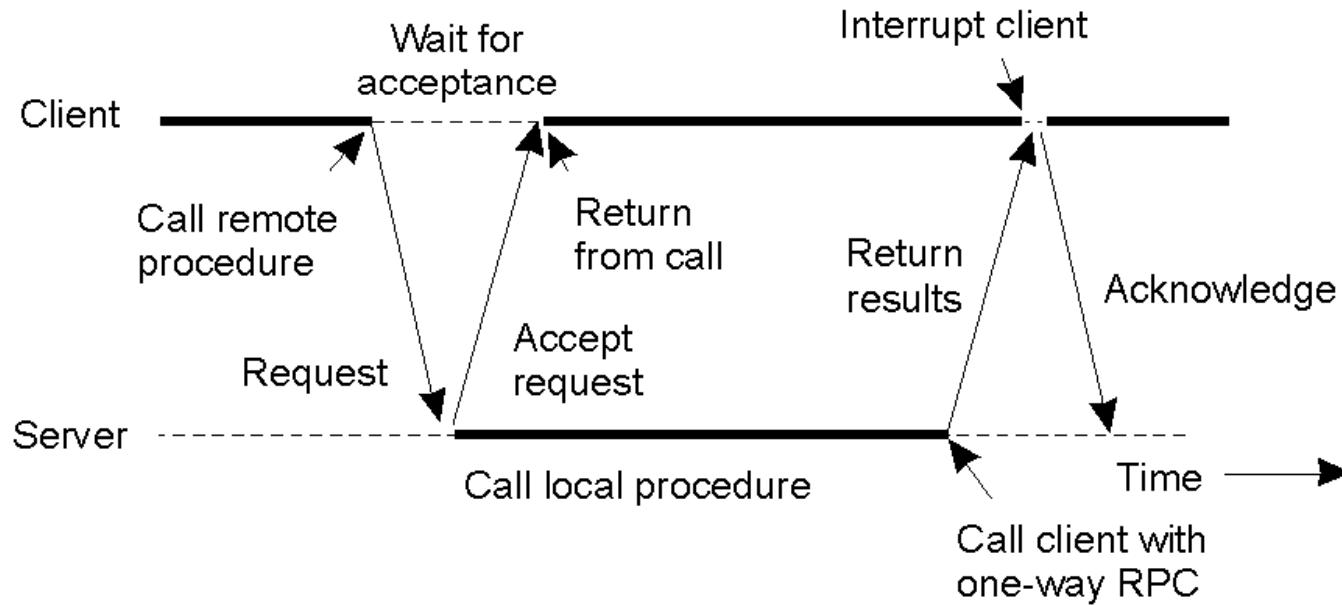


(b)

- a) The interconnection between client and server in a traditional RPC
- b) The interaction using asynchronous RPC

Asynchronous RPC (2)

- A client and server interacting through two asynchronous RPCs



Mutual exclusion in distributed systems

- All the solutions to the mutual exclusion problem studied assume presence of shared memory
 - Ex. Semaphores, monitors, etc. all rely on shared variables
- The mutual exclusion problem is complicated in distributed system by
 - lack of shared memory
 - lack of a common physical clock
 - unpredictable communication delays
- Several algorithms have been proposed to solve this problem with different performance trade-offs

Simple algorithm

- A trivial solution to the distributed mutual exclusion problem:
 - a single *control site* in charge of granting permissions to access the resource
- This solution has several drawbacks:
 - existence of a single point of failure
 - control site is a bottleneck
 - time to grant a new permission is $2T$ (T = average message delay)

Lamport's Algorithm

- Assumption: message delivered in FIFO order
- Requesting the CS
 - P_i sends message **REQUEST**(t_i, i) to other processes, then enqueues the request in its own *request_queue_i*
 - when P_j receives a request from P_i , it returns a timestamped **REPLY** to P_i and places the request in *request_queue_j*
- A process P_i executes the CS only when:
 - P_i has received a message with timestamp larger than t_i from all other processes
 - its own request is the first of the *request_queue_i*

Lamport's Algorithm (2)

- Releasing the critical section:
 - when done, a process remove its request from the queue and sends a timestamped **RELEASE** message to all
 - upon receiving a **RELEASE** message from P_i , a process removes P_i 's request from the request queue

Lamport's Algorithm Example

