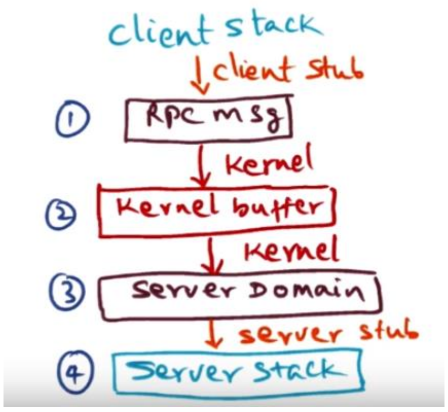
L04d. Lightweight RPC

Remote Procedure Call:

- Remote Procedure Call (RPC) is the mechanism used to manage communication in client-server operations on distributed systems.
- It's also efficient to use RPC even when the client and the server are on the same machine:
  - Safety: We need to make sure that clients and servers are in different memory spaces. This means that RPC calls will go across different protection domains, which will hinder performance.
  - We need to make RPC across protection domains as efficient as a normal procedure call.

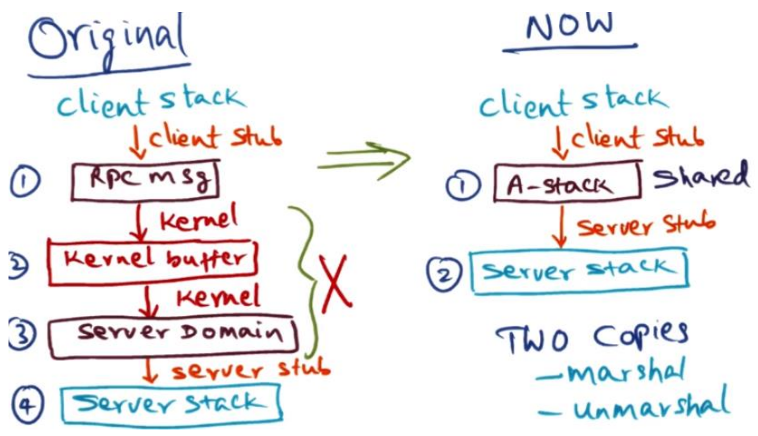
Normal procedure call vs. RPC:

- When a normal procedure call happens
  - The CPU stops the calling process.
  - The CPU executes the called procedure.
  - Return to normal operation.
  - A normal procedure call happens at compile time.
- When a Remote Procedure Call happens:
  - A trap is issued to the kernel.
  - The kernel validates the call and copies the arguments of the call to the kernel buffers.
  - The kernel locates the procedure to be executes and copies the arguments to its address space.
  - The kernel schedules the server to run the particular procedure.
  - When the server executes the requests procedure, it returns to the kernel in the same way (trap & arguments copy).
  - An RPC happens at runtime.
  - Overhead = Two traps + two context switches + one procedure execution.
- The copying overhead that happens with RPCs is a serious concern, since it happens 4 times with every call.
  - 1st copy: A client stub will copy the arguments from the client stack and serialize it into an RPC packet.
  - 2nd copy: The kernel will copy the arguments to its buffers.
  - 3rd copy: The kernel will copy the arguments from its buffers to the server domain.
  - 4th copy: The server stub will de-serialize the RPC packet and copy the arguments to the server stack.
  - These four copies will be executed again to pass the results back from the server to the client.



Making RPC Cheaper:

- How to remove overheads?
  - Binding: Setting up the relationship between the server and the client in the beginning. It’s a one-time cost so it’s OK to leave it as it is.
    1. The client calls the entry point procedure of the server, generating a trap in the kernel.
    2. The kernel checks with the server if this client can make calls to the server. Then the server grants permission.
    3. The kernel sets up a Procedure Descriptor (PD) for each procedure provided by servers, with the entry point address, the arguments stack (A-Stack) size, and the allowed number of simultaneous calls from this specific client.
    4. The kernel allocated a buffer shared between the client and the server with the size of the A-Stack (specified by the server). The client and the server can exchange data using this buffer without any intervention from the kernel.
    5. The kernel provides authentication to the client in a form of a Binding Object (BO). The client can use this BO whenever it needs to make a call to this specific server.
  - Making the actual call:
    1. Passing arguments between the client and the server through the A-Stack can only be by value, not by reference, since the client and server don’t have access to each other’s address space.
    2. The stub procedure copies the arguments from the client memory space into the A-Stack.
    3. The client presented the BO to the kernel (trap) and blocks on the kernel’s response.
    4. At this point, the client will be blocked waiting for the call to be executed. The kernel can use the client thread for executing the procedure on the server’s domain.
    5. The kernel validates the BO and allocates an execution stack (E-Stack) for the server to use it for its own execution.
    6. The server stub copies the arguments from the A-Stack to the E-Stack.
    7. After finishing execution, the server stub will copy the results from the E-Stack to the A-Stack.
    8. The server traps the kernel, and everything will be done reversely to return to the client.
  - Results of using an A-Stack:
    1. Using an A-Stack reduces the number of copies from four to two.
    2. The two copies 3happen in the client or server user space above the kernel.



- Even with this trick, we still have the overhead associated with the context switch itself:
  1. The client trap.
  2. Switching the protection domain.
  3. The server trap.
  4. Loss of locality (implicit).

RPC on SMP:

- On a multiprocessor system, we can reduce context switching overhead by caching domains on idle processors. This keeps the caches warm (no TLB invalidation).
- When a call is made, the kernel checks for a processor idling in the context of the server domain. If one is found, the kernel exchanges the processors of the calling and the idling threads. Then, the called server procedure can execute on that processor without requiring a context switch. Same can be done on return from the call.
- Keeping the caches warm reduces the loss of locality.
- If the same server is serving multiple clients, the kernel can pre-load the same server on multiple CPUs to be able to simultaneously serve multiple clients.