Remote Procedure Call

In message passing, the operations `send` and `receive` do not conceal communication at all, which is important to achieve access transparency in distributed system. To tackle this, a proposal was put forward. In a nutshell, the proposal was to allow programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, the execution of the called procedure takes place on B. Information can be transported from the caller to the calleein the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as **Remote Procedure Call**, or often just **RPC**.

The caller and callee executes on different machines, thus different address spaces, so this introduces complications. Parameters and results have to passed, which can be complicated, if the machines are not identical. Finally, either or both can crash and each of the possible failures causes different problems.

Basic RPC operation

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, RPC should be transparent-the calling procedure should not be aware that the procedure is executing on a different machine or vice versa.

Suppose a routine,

`newlist = append(data, dbList)`

Even though `append()` eventually does only a few basic file operations, it is called the usual way, by pushing its parameters onto the stack. The programmer does not know the implementation details of `append()`, and this is, ofcourse, how its supposed to be.

When `append` is actually a remote procedure, a different version of `append`, called a **client stub**, is offered to the calling client. However, unlike the original one, it does not perform an append operation. Instead, it packs the parameters into a message and requests that message to be sent to the server. Following the call to `send`, the client stub calls `receive`, blocking itself until the reply comes back.

`append` is something like:

```
function append(params){
        msg = make_message('append', params);
        send(msg, <server-url>);
        response = receive();
}
```

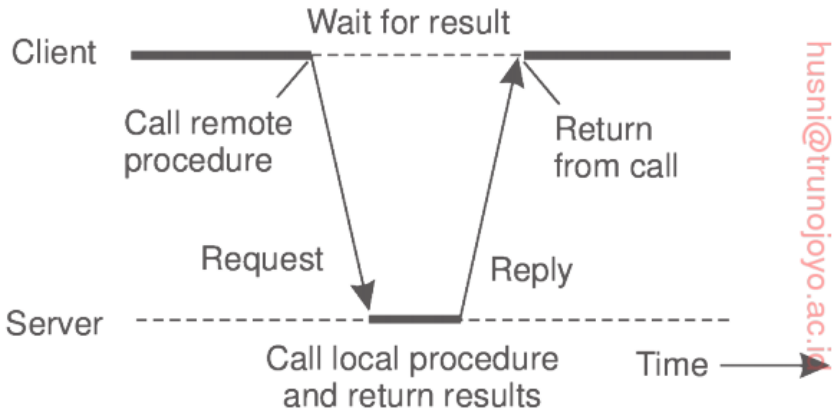

Figure 4.6: The principle of RPC between a client and server program.

When the message arrives at the server, the server's OS passes it to a **server stub**. A server stub is a server-side equivalent of a client stub: it is a piece of code that transforms the request coming over a network into local procedure calls. Typically the server stub will call `receive` and be blocked waiting for incoming messages. The server stub unpacks the parameters from the message and then calls the server procedure the usual way. The server performs its work and returns the result to the caller (in this case the server stub) in the usual way.

When the result arrives at the client's machine, the OS passes it through the `receive` operation, which had been called previously. The client stub inspects the message, unpacks the result, copies it to the caller, and returns it in the usual way. When the caller gets control following the call to `append`, all it knows is that it appended some data to the list. It has no idea that the work was done remotely at another machine.

So, to summarise:

- The client procedure calls the client stub the normal way.
- The client stub builds the message and calls the local OS.
- The client's OS sends the message to the remote OS.
- The remote OS gives the message to the server stub.
- The server stub unpacks the parameter(s) and calls the server.
- The server does the work and returns the result to the server stub.
- The server stub packs the result in a message and calls it local OS.
- The server's OS sends the message to the client's OS.
- The client's OS gives the message to the client stub.
- The stub unpacks the result and returns it to the client.

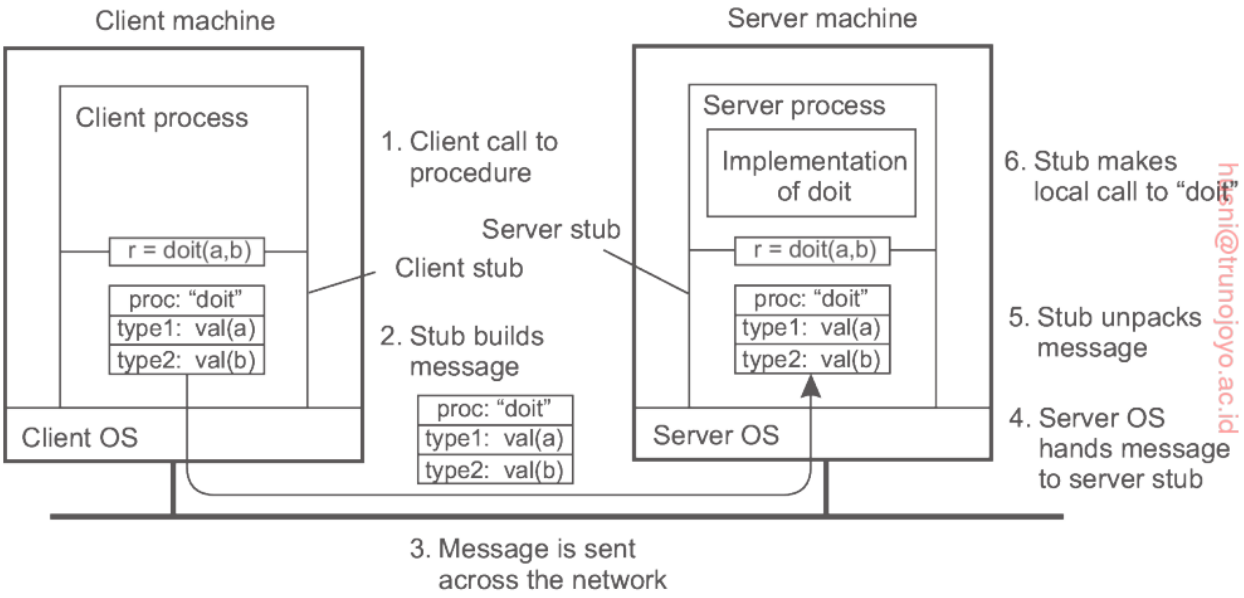

Figure 4.7: The steps involved in calling a remote procedure `doit(a,b)`. The return path for the result is not shown.

Paramter passing

Packing parameters into a message is called **message marshaling**. The thing to realize here is that, in the end, the server will just be seeing a series of bytes coming in that constitute the original message sent by the client. However, no additional information on what those bytes mean is normally provided with the message.

Placement of bytes can differ in machine architectures. Intel Pentium, number their bytes from right to left, this is called **little endian**, and the placement the other way is called **big endian**.

Solution is making sure that both communication parties expect the same *message data type* to be transmitted and transform the data into a machine and network independent format. These can be solved by using programming languages and machine-*dependent* routines repectively.

Difficulty with pointers or references: A pointer is meaningful only within the address space of the process in which it is being used.

Solutions:

1. Forbid pointers and references which is highly undesirable since they are so important. But this is not necessary, we can just copy the entire data structure to which the parameter is referring, effectively replacing the copy by reference with copy by value.
2. Using global references that are meaningful to the calling and the called process.