Parallel & Distributed System

Interprocess Communication

Md. Biplob Hosen

Lecturer, IIT-JU

Email: biplob.hosen@juniv.edu

Contents

- Communication - Overview
- Request-Reply Protocols
- Remote Procedure Call
- Remote Method Invocation

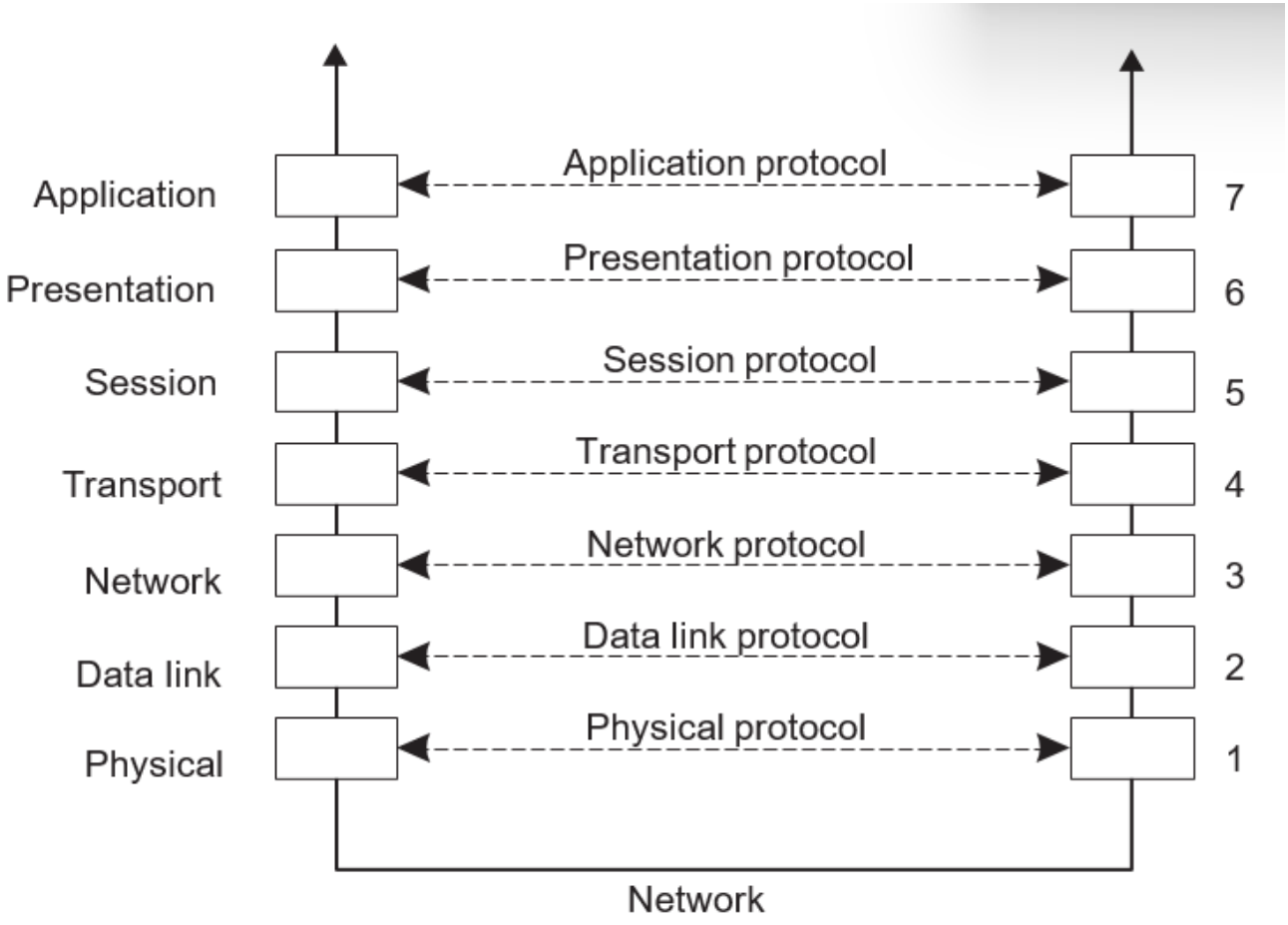
- Reference Books
  - ✓ Distributed Systems: Principles and Paradigms, 3rd Edition by Andrew S. Tanenbaum & Maarten van Steen, Publisher: Pearson Prentice Hall **[CH-04]**.
  - ✓ Distributed Systems: Concepts and Design, 3rd Edition by George Coulouris, Jean Dollimore & Tim Kindberg, Publisher: Addison-Wesley **[CH-05]**.

Overview

- Interprocess communication is at the heart of all distributed systems.
- It makes the ways that processes on different machines can exchange information.
- Communication in distributed systems has traditionally always been based on low-level **message passing** as offered by the underlying network.
- Expressing communication through message passing is harder than using primitives based on **shared memory**, as available for nondistributed platforms.
- Modern distributed systems often consist of thousands or even millions of processes scattered across a network with unreliable communication such as the Internet.
- Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

Layered Protocols

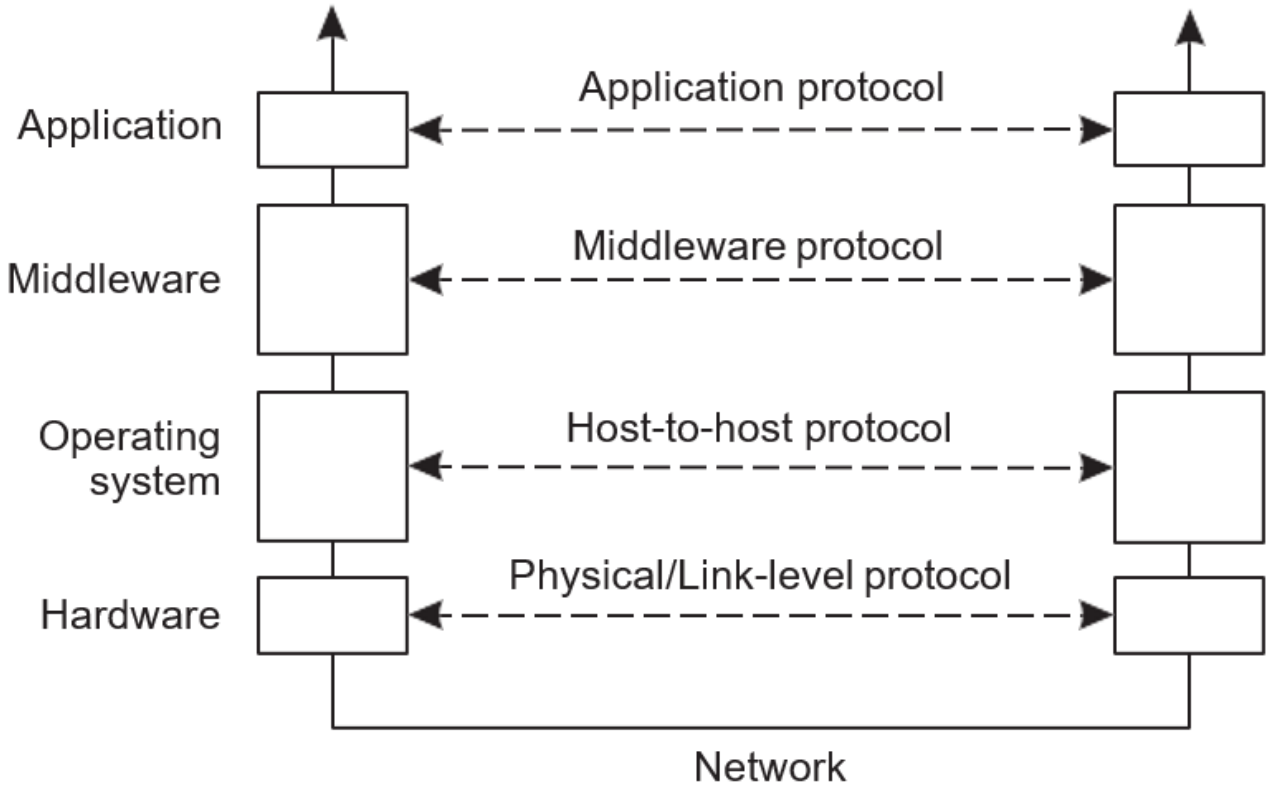
- Due to the absence of shared memory, all communication in distributed systems is based on sending and receiving messages.
- When process P wants to communicate with process Q, it first builds a message in its own address space. Then it executes a system call that causes the operating system to send the message over the network to Q.



Layers, interfaces, and protocols in the OSI model

Middleware Protocols

- Middleware is an application that logically lives in the OSI application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications.
- A rich set of communication protocols:
- **(Un)marshaling of data**, necessary for integrated systems.
- **Naming protocols**, to allow easy sharing of resources.
- **Security protocols** for secure communication.
- **Scaling mechanisms**, such as for replication and caching.
- **Remote procedure call**, to locally call a procedure that is effectively implemented on a remote machine.



Types of Communication

- An electronic mail system is a typical example in which communication is persistent. With **persistent communication**, a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver. In this case, the middleware will store the message at one or several of the storage facilities. As a consequence, it is not necessary for the sending application to continue execution after submitting the message. Likewise, the receiving application need not be executing when the message is submitted.
- In contrast, with **transient communication**, a message is stored by the communication system only as long as the sending and receiving application are executing. More precisely, if the middleware cannot deliver a message due to a transmission interrupt, or because the recipient is currently not active, it will simply be discarded. Typically, all transport-level communication services offer only transient communication. In this case, the communication system consists of traditional store-and-forward routers. If a router cannot deliver a message to the next one or the destination host, it will simply drop the message.

Continue…

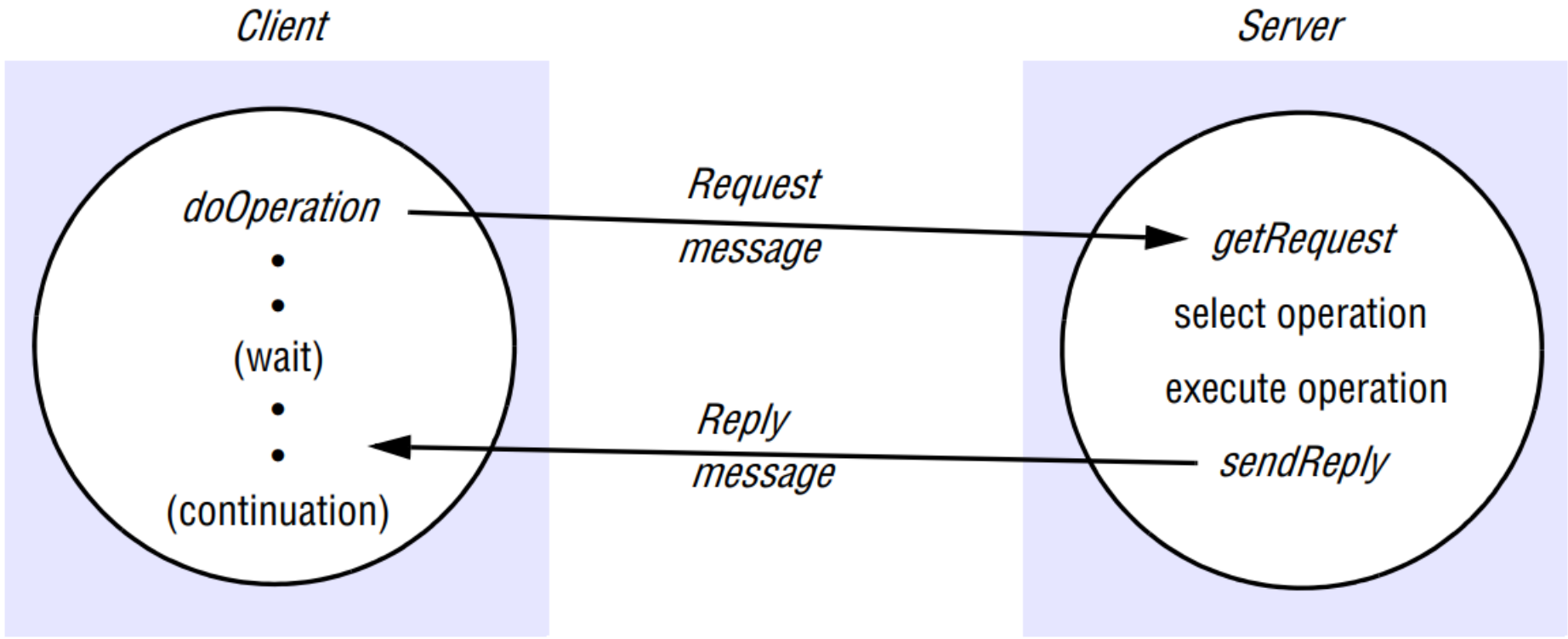
- Communication can also be **asynchronous** or **synchronous**.
- The characteristic feature of asynchronous communication is that a sender continues immediately after it has submitted its message for transmission. This means that the message is temporarily stored immediately by the middleware upon submission.
- With synchronous communication, the sender is blocked until its request is known to be accepted. There are essentially three points where synchronization can take place. **First**, the sender may be blocked until the middleware notifies that it will take over transmission of the request. **Second**, the sender may synchronize until its request has been delivered to the intended recipient. **Third**, synchronization may take place by letting the sender wait until its request has been fully processed, that is, up to the time that the recipient returns a response.

Continue…

- In the case of a **connection-oriented** communication, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate specific parameters of the protocol they will use. When they are done, they release (terminate) the connection. The telephone is a typical connection-oriented communication service.
- With a **connectionless communication**, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of making use of connectionless communication service.
- With computers, both connection-oriented and connectionless communication are common.

Request-Reply Protocols

- This form of communication is designed to support the roles and message exchanges in typical client-server interactions.
- The protocol is based on a trio of communication primitives, **doOperation ()**, **getRequest ()** and **sendReply ()**.



Continue…

- The **doOperation()** method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation.
- The caller of **doOperation()** is blocked until the server performs the requested operation and transmits a reply message to the client process.
- **getRequest()** is used by a server process to acquire service requests.
- When the server has invoked the specified operation, it then uses **sendReply()** to send the reply message to the client.
- When the reply message is received by the client, the original doOperation() is unblocked and execution of the client program continues.

Remote Procedure Call

- When a process on **machine A** calls a process on **machine B**, the calling process A is suspended and the execution of the called procedure takes place on B.
- Information can be transported from the caller to the callee in 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 (RPC).**
- Usually the software that supports RPC is written in C, which does not support objects or classes.
- The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be aware that the called procedure is executing on a different machine.

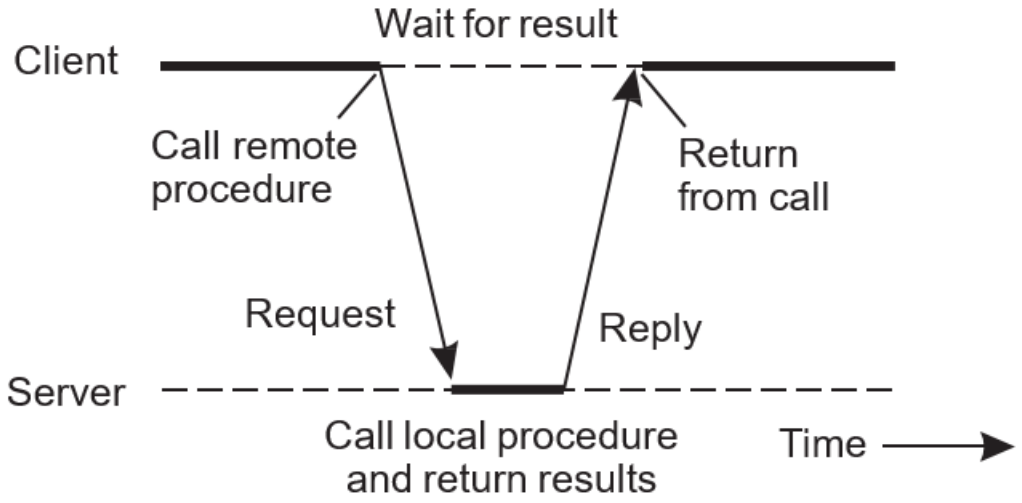
Basic RPC Operation

- Suppose that a program has access to a database that allows it to append data to a stored list, after which it returns a reference to the modified list.
- The operation is made available to a program by means of a routine append:
  - newlist = append(data, dbList)
- In a traditional (single-processor) system, append is extracted from a library by the linker and inserted into the object program.
- In principle, it can be a short procedure, which could be implemented by a few file operations for accessing the database.
- Even though append eventually does only a few basic file operations, it is called in the usual way, by pushing its parameters onto the stack.
- The programmer does not know the implementation details of append, and this is, of course, how it is supposed to be.

Continue…

Client and Server Stub:

- RPC achieves its transparency in an analogous way. When append is actually a remote procedure, a different version of append, called a **client stub**, is offered to the calling client.
- Like the original one, it, too, is called using a normal calling sequence.
- 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 as illustrated.



Continue…

- Following the call to send, the client stub calls receive, blocking itself until the reply comes back.
- When the message arrives at the server, the server's operating system passes it to a server stub.
- A **server stub** is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls. Typically the server stub will have called receive and be blocked waiting for incoming messages.
- The server stub unpacks the parameters from the message and then calls the server procedure in the usual way.
- From the server's point of view, it is as though it is being called directly by the client—the parameters and return address are all on the stack where they belong and nothing seems unusual.
- The server performs its work and then returns the result to the caller (in this case the server stub) in the usual way.

Continue…

- When the server stub gets control back after the call has completed, it packs the result in a message and calls send to return it to the client.
- After that, the server stub usually does a call to receive again, to wait for the next incoming request.
- When the result message arrives at the client's machine, the operating system passes it through the receive operation, which had been called previously, to the client stub, and the client process is subsequently unblocked.
- The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way.
- When the caller gets control following the call to append, all it knows is that it appended some data to a list.
- It has no idea that the work was done remotely at another machine.

Continue…

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Continue…

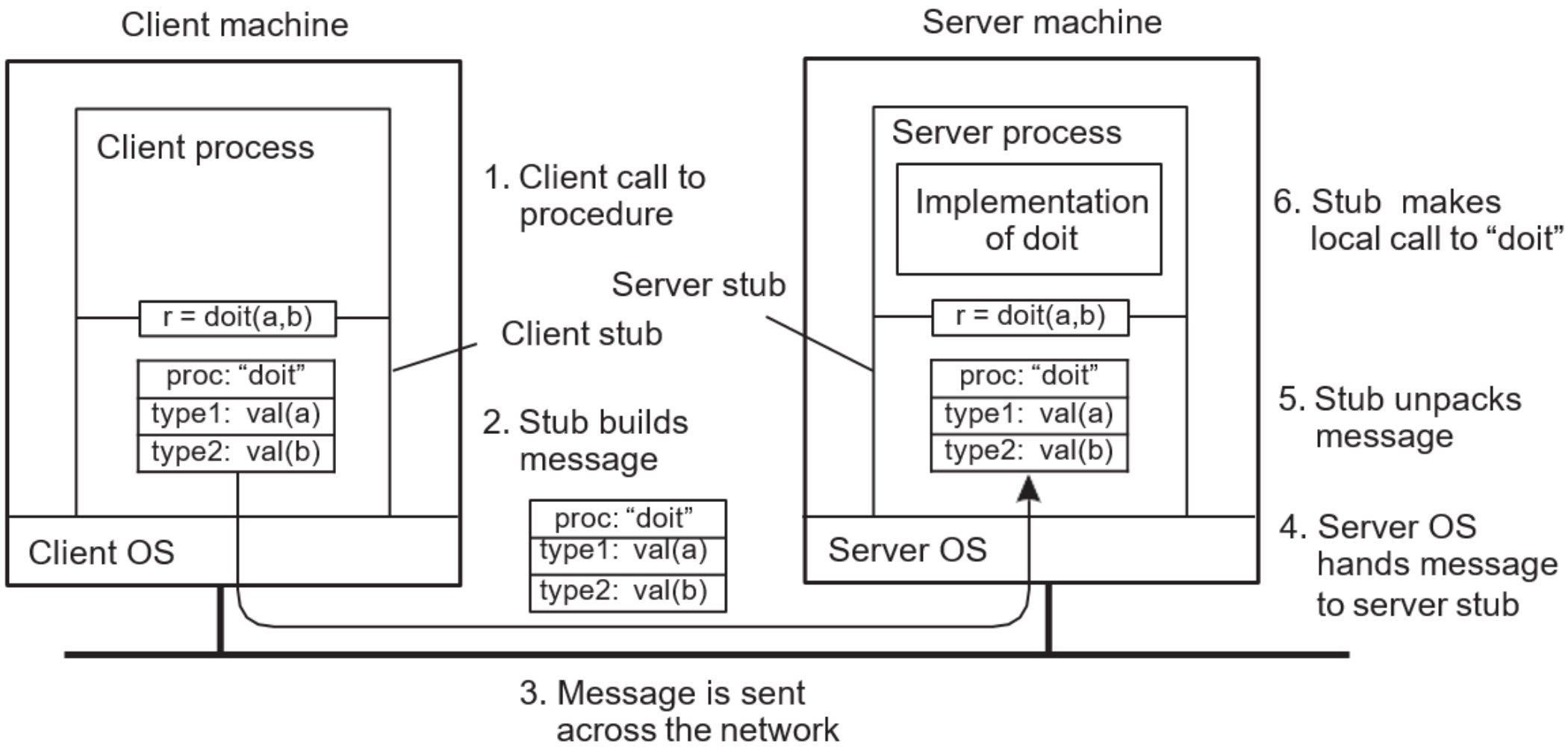
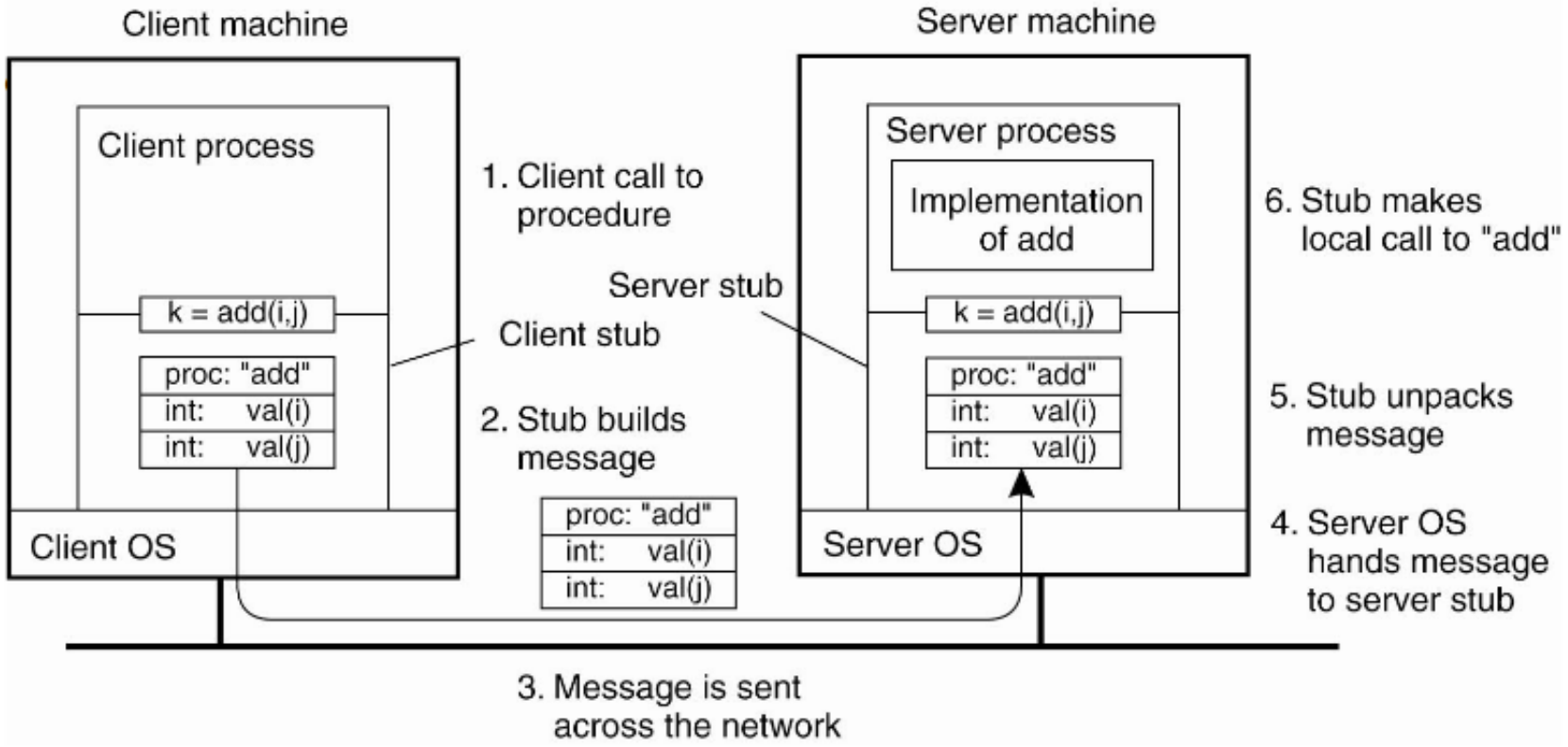


Figure: The steps involved in calling a remote procedure doit(a,b). The return path for the result is not shown.

Parameter Passing

- The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub.
- Packing parameters into a message is called **parameter marshaling**.
- Consider a remote procedure call, add(i,j), that takes two integer parameters i and j and returns their arithmetic sum as a result.

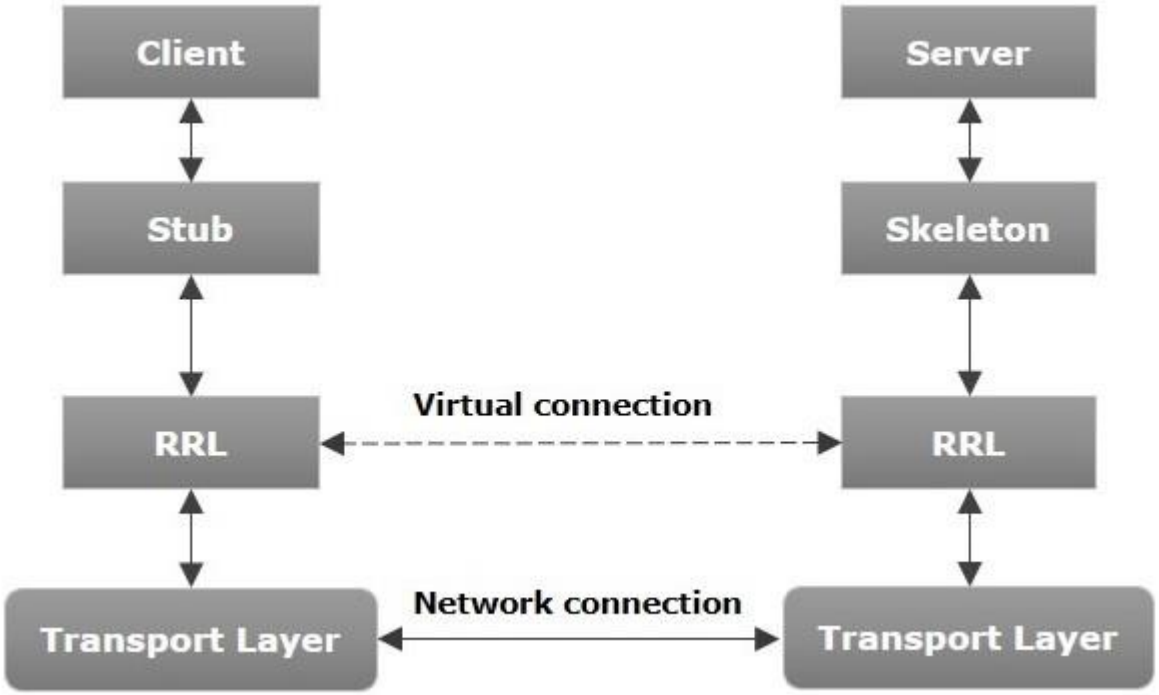


Remote Method Invocation

- Remote Method Invocation (RMI) is an API that allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine.
- Through RMI, an object running in a JVM present on a computer (Client-side) can invoke methods on an object present in another JVM (Server-side).
- RMI creates a public remote server object that enables client and server-side communications through simple method calls on the server object.

Architecture of an RMI Application

- In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).
- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.
- The following diagram shows the architecture of an RMI application.



Continue…

Let us now discuss the components of this architecture.

- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** − This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application

The following points summarize how an RMI application works –

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

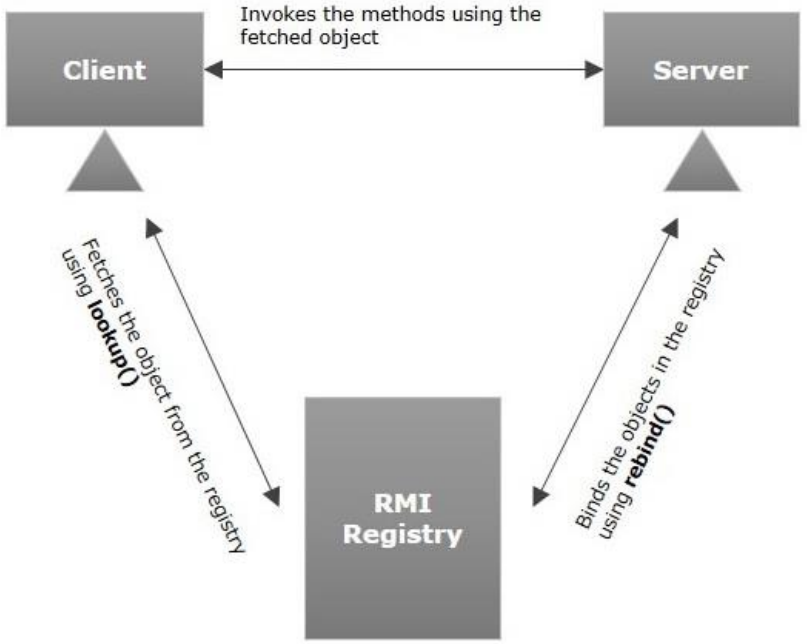
Marshalling and Unmarshalling

- Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. This process is known as marshalling
- At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

Continue…

RMI Registry

- RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using **bind()** method). These are registered using a unique name known as **bind nam**e.
- To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using **lookup()** method).



Thank You ☺