Distributed Operating Systems
Communication 1

Ewa Niewiadomska-Szynkiewicz

ens@ia.pw.edu.pl

Institute of Control and Computation Engineering
Warsaw University of Technology

Communication (I)

1. Layered Protocols
2. Remote Procedure Call
3. Remote Object Invocation

Necessary Agreements

Communication in distributed systems is always based on low-level message passing.

Communication through message passing is harder than using primitives based on shared memory.

- √ How many volts should be used to signal a 0-bit, and how many for a 1-bit?
- √ How does the receiver know which is the last bit of the message?
- √ How can it detect if a message has been damaged or lost?
- √ How long are numbers, strings and other data items?
- √ How are they represented?

Protocols – Introduction

To allow a group of computers to communicate over a network, they must all agree on the protocols to be used.

Example protocol as a discussion:

A: Please, retransmit message n,

B: I already retransmitted it,

A: No, you did not,

B: Yes, I did,

A: All right, have it your way, but send it again.

OSI Model

The International Standards Organization (ISO) developed a reference model:

ISO OSI = OSI Model = Open Systems Interconnection Reference Model

The OSI model is designed to allow open systems to communicate.
Open system: a system that is prepared to communicate with any other open system by using standard rules that govern:

- √ format,
- √ contents,
- √ meaning

of the messages sent and received.
These rules are formalized – **protocols**.

OSI model is useful for understanding computer networks.
Protocols that were developed as part of the OSI model were never widely used.

Protocol – Definition

Protocol
A well-known set of rules and formats to be used for communication between processes in order to perform a given task.

Two important parts of the definition:

- √ a specification of the sequence of messages that must be exchanged,
- √ a specification of the format of the data in the messages.

How to create protocols:

On the Design of Application Protocols, RFC 3117,
http://www.rfc-editor.org/rfc/rfc3117.txt

Types of Protocols

A distinction is made between two general types of protocols:

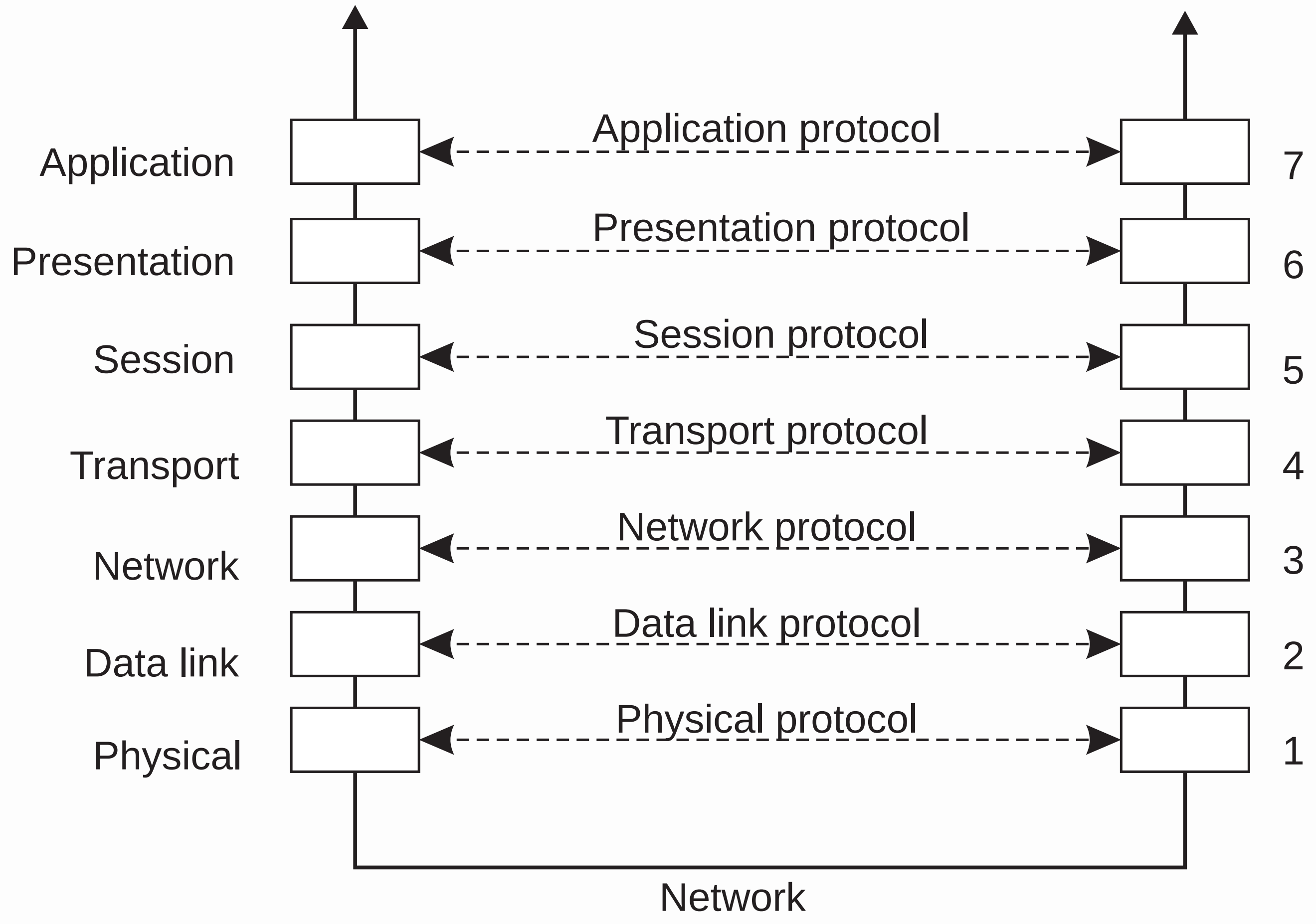
- √ *connection-oriented protocols*: the sender and receiver first explicitly establish a connection before exchanging data,
- √ *connectionless protocols*: the sender transmits the first message when it is ready (no setup between sender and receiver in advance is needed).

Example of connectionless protocols: dropping a letter in a mailbox.

With computers, both connection-oriented and connectionless communication are common.

protocol suite = **protocol stack** = the collection of protocols used in a particular system.

Layered Protocols (1)



Layered Protocols (2)

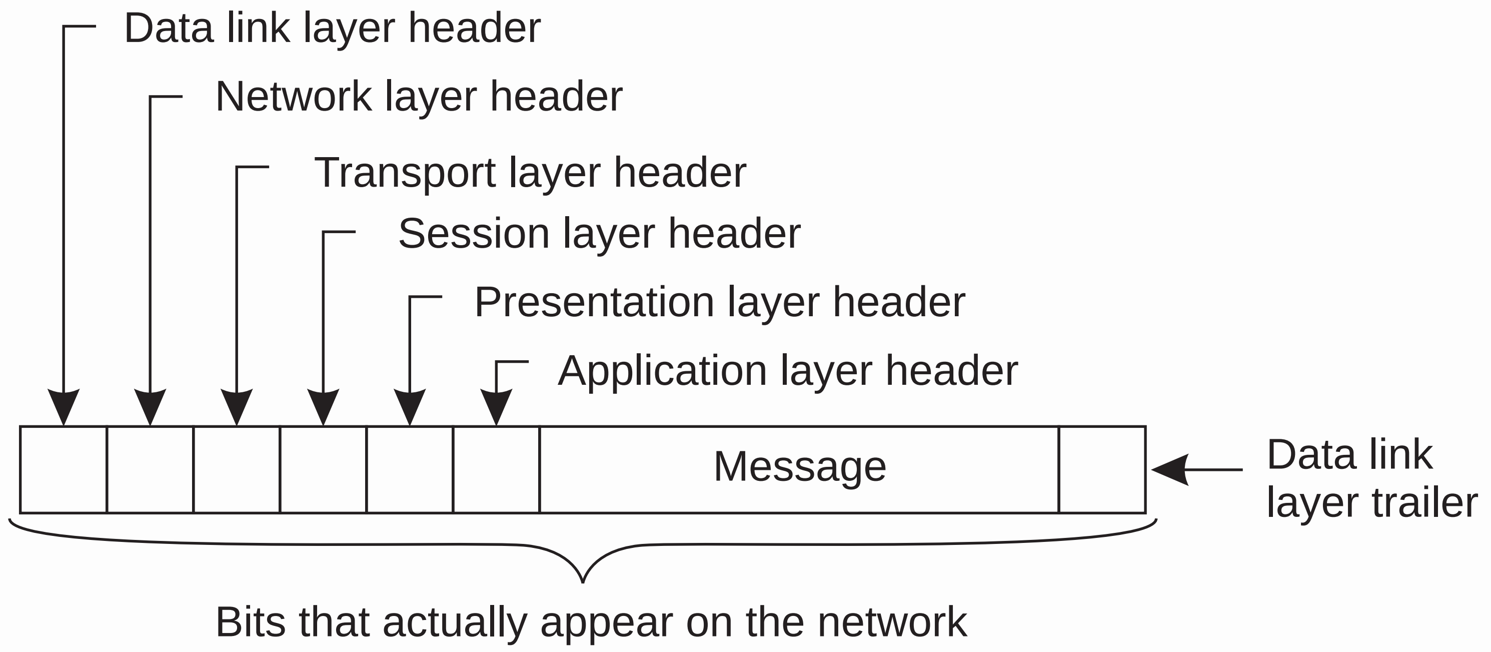
- √ Each Layer deals with one specific aspect of the communication.
- √ Each Layer provides an interface to the one above it.
- √ The interface consists of a set of operations that together define the service the layer is prepared to offer its users.

Process A on machine 1 wants to communicate with process B on machine 2

- √ **process A** builds a message and passes it to the application layer on **machine 1**,
- √ the application layer software adds a **header** to the front of the message,
- √ the application layer passes the message to the presentation layer (across layer 6/7 interface),
- √ the presentation layer adds its own header and passes the message to the session layer, etc...
- √ the physical layer transmits the message.

Layered Protocols Encapsulation

Encapsulation : a protocol at a higher level uses a protocol at a lower level to help accomplish its aims.
Encapsulation is used to provide abstraction of protocols and services.



A typical message as it appears on the network.

Lower-Level Protocols

Physical layer

Contains the specification and implementation of bits, and their transmission between sender and receiver (just sends bits). The physical layer protocol deals with standardizing the electrical, mechanical, and signaling interfaces.

Data link layer

Describes the transmission of a series of bits into a frame to allow error and flow control. It marks each frame (puts a special bit pattern on the start and end of a frame) and computes a checksum by adding up all the bytes in the frame in a certain way.

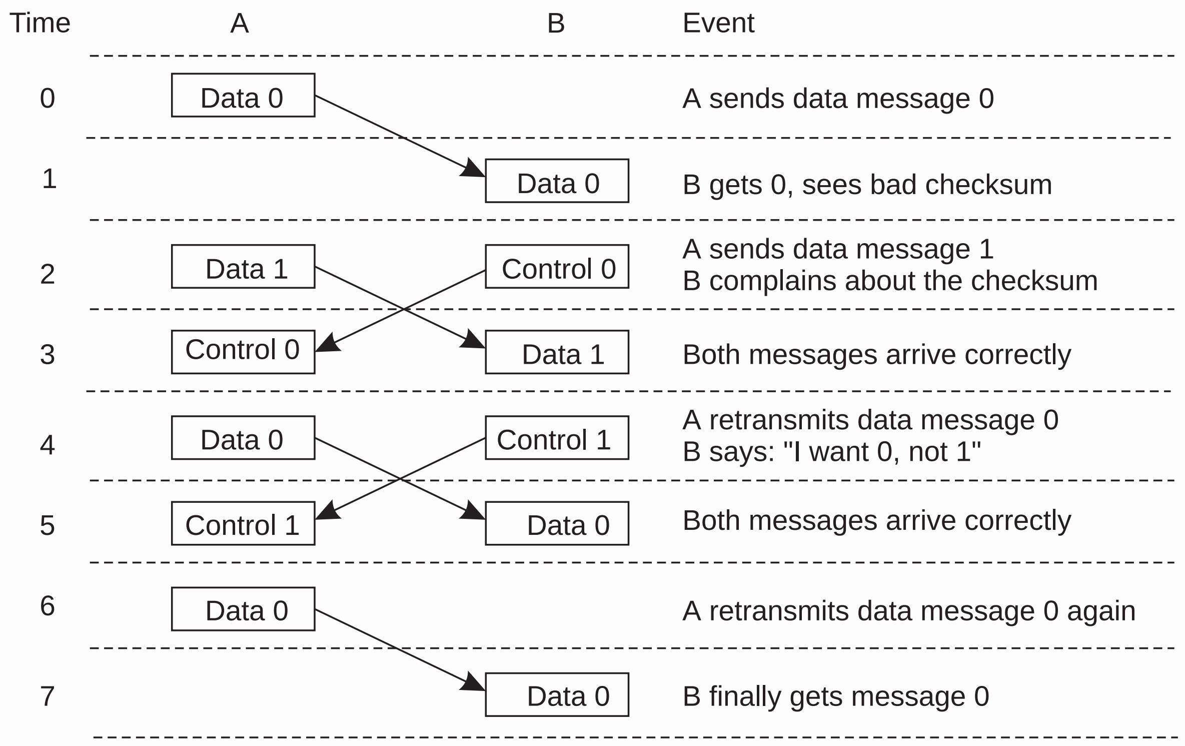
Network layer

Describes how packets in a network of computers are to be routed (different routing algorithms).

Transport Layer

Provides the actual communication facilities for most distributed systems. Provides reliable connection, schedules data (w.r.t. priorities). Upon receiving a messages from the application layer, the transport layer breaks it into pieces small enough to transmission, assigns each one a sequence number, and then sends them all.

Data Link Layer



Time
A
B
Event
0
Data 0
A sends data message 0
1
Data 0
B gets 0, sees bad checksum
2
Data 1
Control 0
A sends data message 1
B complains about the checksum
3
Control 0
Data 1
Both messages arrive correctly
4
Data 0
Control 1
A retransmits data message 0
B says: "I want 0, not 1"
5
Control 1
Data 0
Both messages arrive correctly
6
Data 0
A retransmits data message 0 again
7
Data 0
B finally gets message 0

Discussion between a receiver and a sender in the data link layer.

Network Layer Protocols

- √ IP (*Internet Protocol*): connectionless, part of the Internet protocol suite. IP packet is routed to its destination independent of all others. No internal path is selected and remembered.
- √ ATM virtual channels (*Asynchronous Transfer Mode*): connection-oriented.
  Unidirectional connection from a source to destination, crossing several intermediate ATM switches.
  Virtual path (collection of virtual channels) - predefined routes between pairs of hosts.

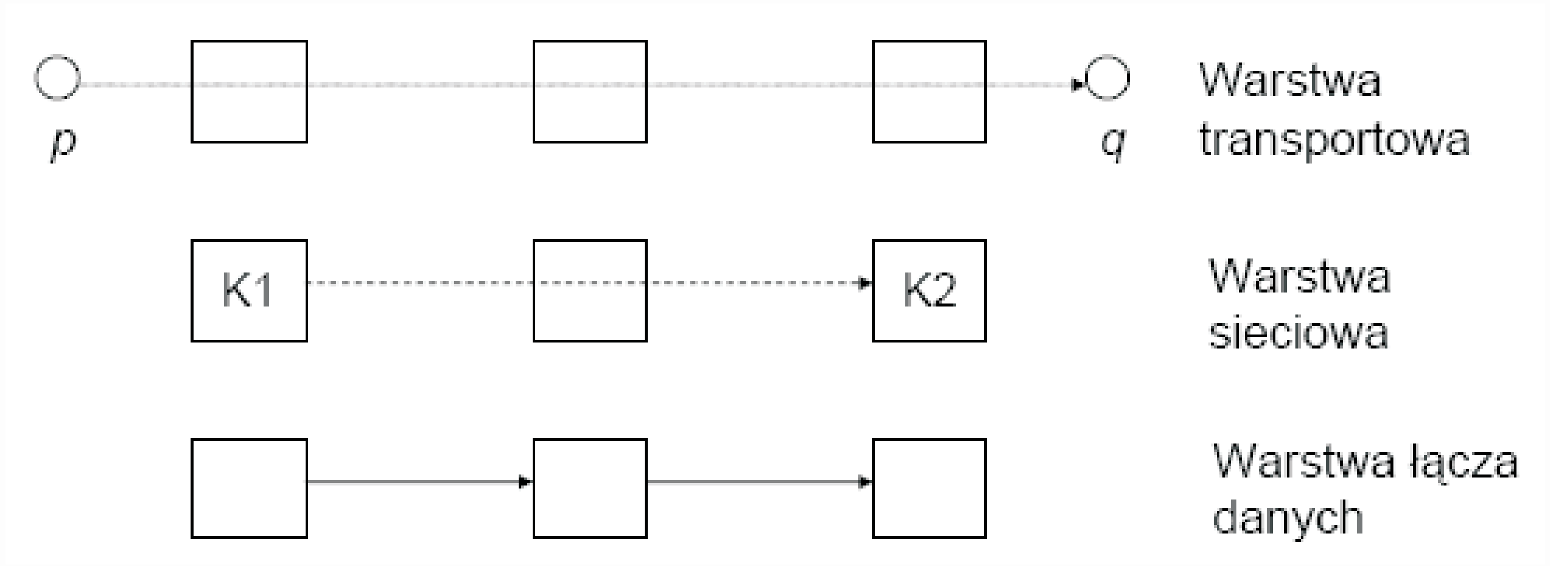
Packet: a unit of data that is routed between an origin and a destination on the Internet or any other packet-switched network. It consists of IP header and variable-length data field.
Datagram: a self-contained, independent entity of data carrying sufficient information to be routed from the source to the destination computer without reliance on earlier exchanges between this source and destination computer and the transporting network.

Transport Layer Protocols

- √ TCP (*Transmission Control Protocol*): connection-oriented, reliable, stream-oriented communication (TCP/IP - standard for network communication).
- √ UDP (*Universal Datagram Protocol*): connectionless, unreliable (best-effort) datagram communication.
- √ RTP (*Real-time Transport Protocol*: specifies packet formats for real-time data without providing the actual mechanisms for guaranteeing data delivery.
- √ TP0 – TP4, the official ISO transport protocols.

Lower-Level Protocols - summary

Layers 2, 3, 4 provide connection of processes to the wide area networks.



Data Link Layer: connects *neighbors* (computers) into the *Local Area Networks* (LANs).
Network Layer: connects local networks into wider structures (example: Internet).
Transport Layer: provides connection for distributed systems.

Layers 2 and 3 **connect devices**, layer 4 **connects processes**.

Higher-Level layers

Session Layer

Provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities.

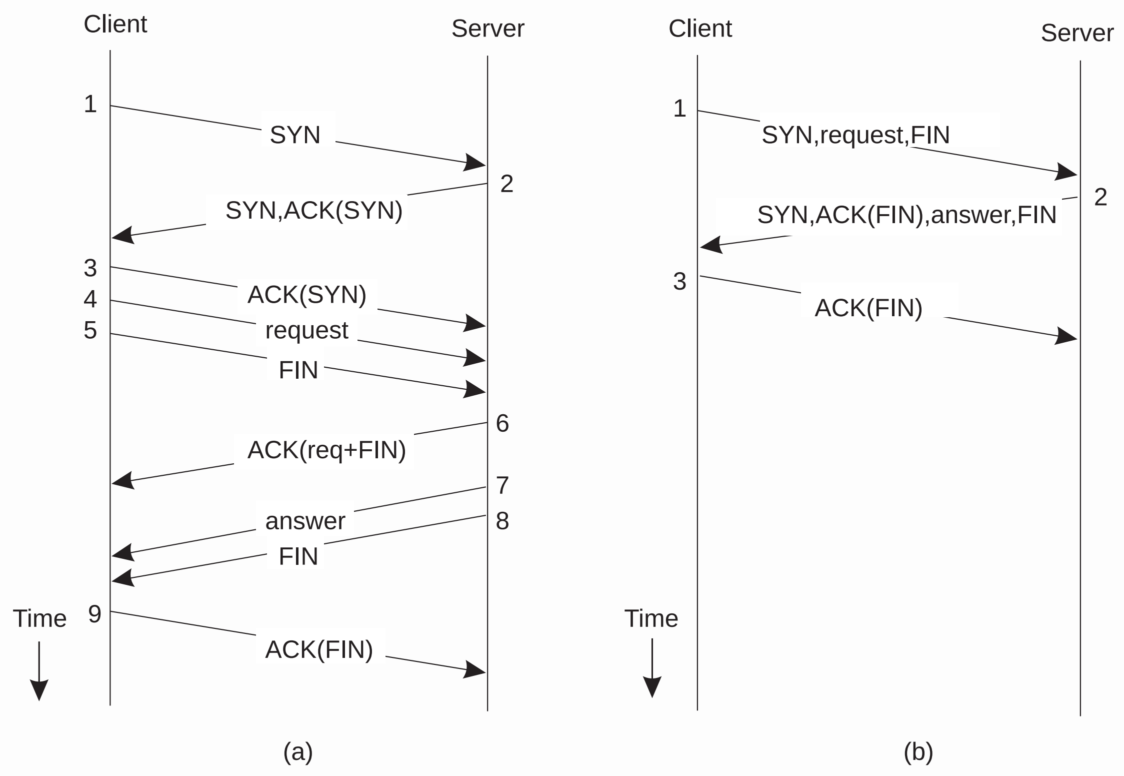
Presentation Layer

Provides facilities allowing machines with different internal representation to communicate.

Application Layer

A container for all applications and protocols that do not fit into one of the underlaying layers (examples: e-mail, WWW, FTP, HTTP, etc.

Client-Server TCP



(a) Normal operation of TCP. (b) Transactional TCP.

Middleware Protocols (1)

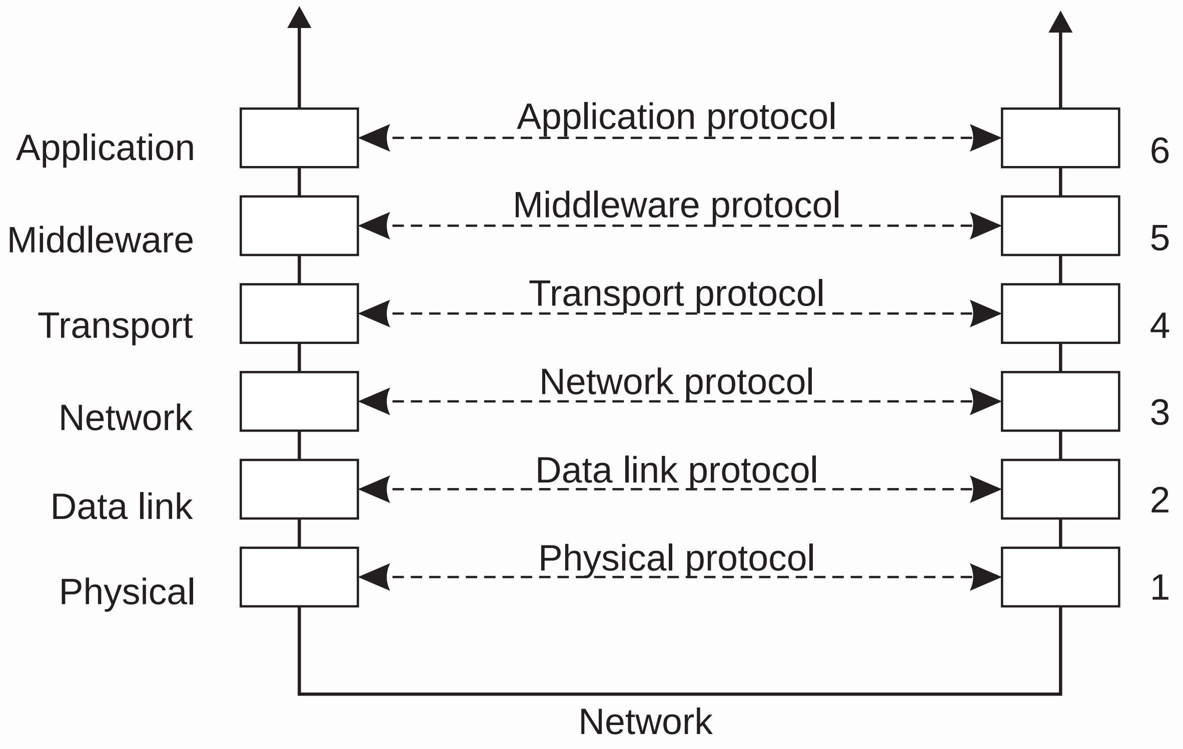
Middleware
An application that logically lives in the application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications.

Middleware invented to provide *common* services and protocols that can be used by many *different* applications:

Example protocols:

- √ open communication protocols,
- √ marshaling and unmarshaling of data, for systems integration,
- √ naming protocols, for resource sharing,
- √ security protocols, distributed authentication and authorization,
- √ scaling mechanisms, support for caching and replication.

Middleware Protocols (2)



An adapted ISO OSI reference model for networked communication.
Instead of session and presentation layers we have middleware layer.

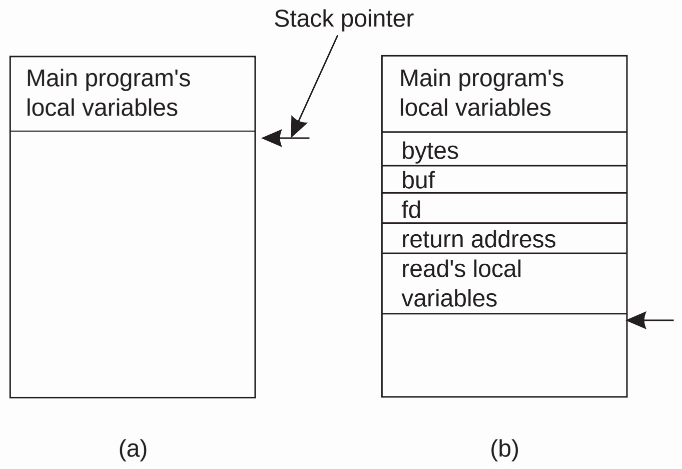
High-level Middleware Communication Services

Some of high-level middleware communication protocol types:

1. remote procedure call,
2. remote object invocation,
3. message queuing services,
4. stream-oriented communication.

Local Procedure Call

count=read(fd, buf, bytes)



Parameter passing:

a. the stack before the call.

b. the stack while the called procedure is active.

√ Application developers familiar with simple procedure model,

√ Procedures as black boxes (isolation),

√ No fundamental reason not to execute procedures on separate machine.

Remote Procedure Call

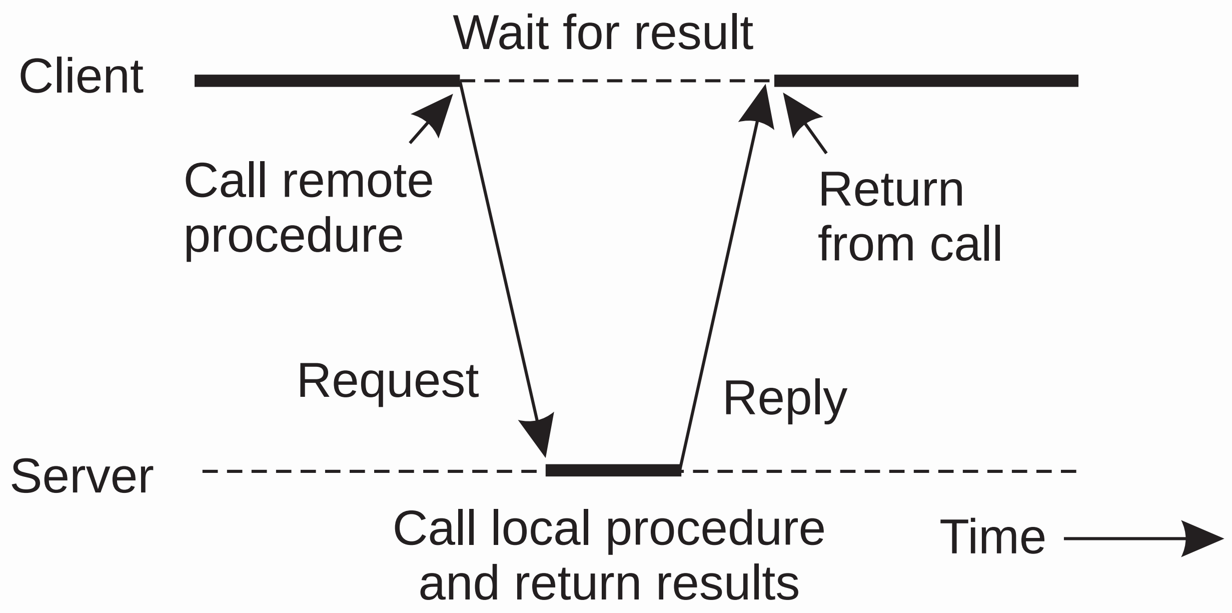
When we try to call procedures located on other machines, some subtle problems exist:

- √ different address spaces,
- √ parameters and results have to be passed,
- √ both machines may crash.

Standard function call parameters types:

- √ call-by-value,
- √ call-by-reference,
- √ call by copy/restore.

Principle of RPC



Principle of RPC between a client and server program.

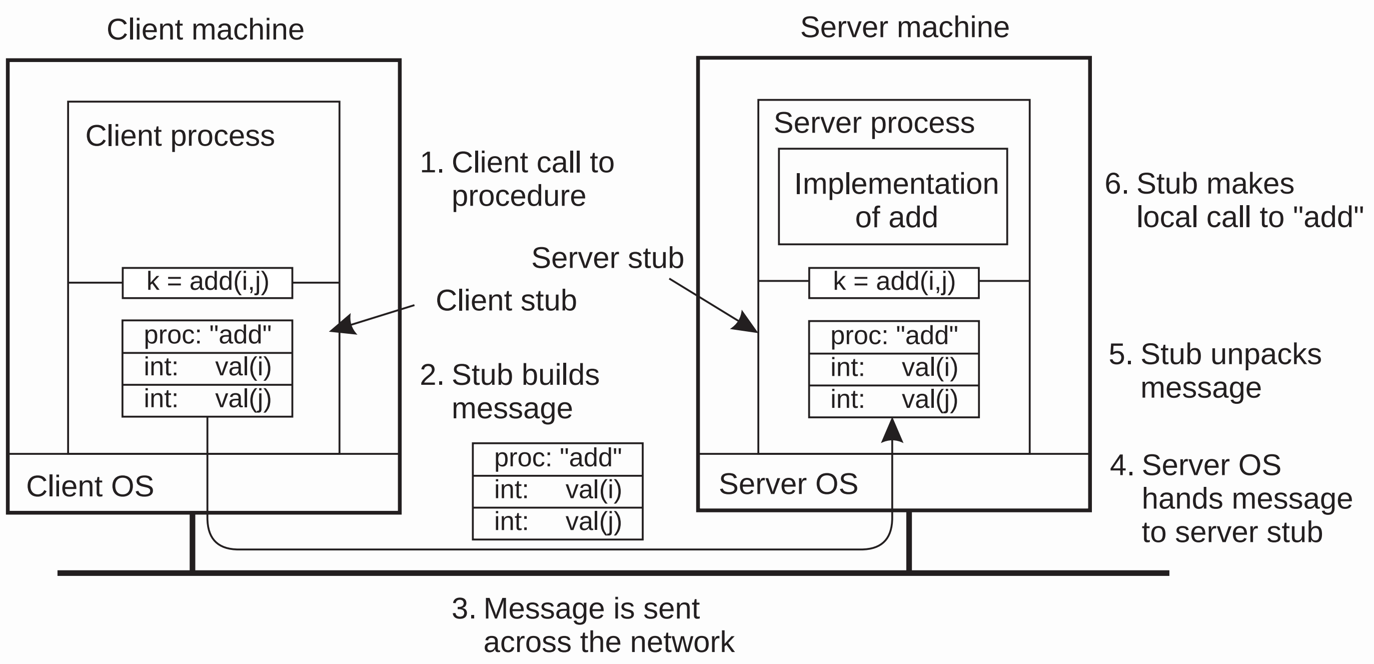
Steps in RPC

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)



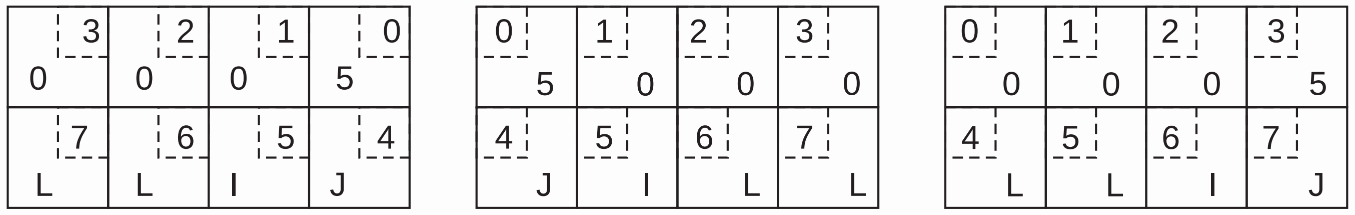
Steps involved in doing remote computation through RPC.

parameter marshaling – packing parameters into a message.

Passing Value Parameters (2)

√ IBM mainframes: **EBCDIC** character code,

√ IBM personal computers: **ASCII** character code.



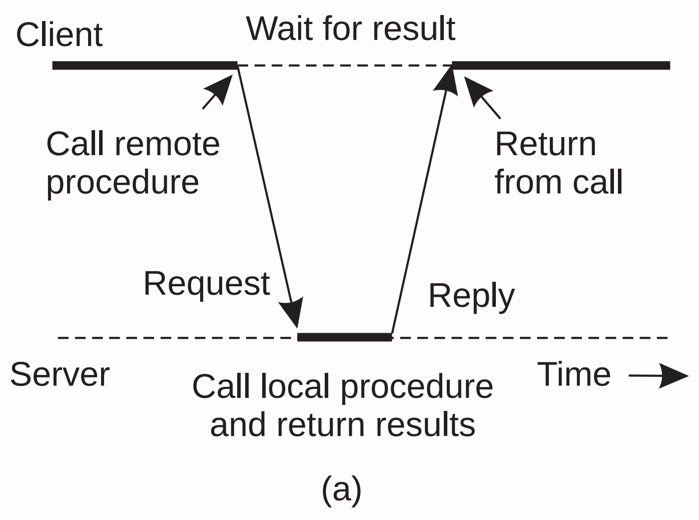
(a) (b) (c)

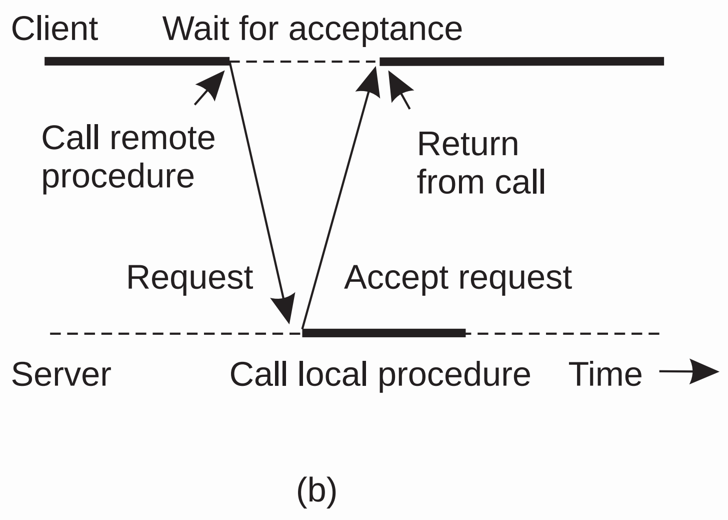
a. Original message on the Pentium

b. The message as being received on the SPARC

c. The message after being inverted. The little numbers in boxes indicate the address of each byte.

Asynchronous RPC (1)

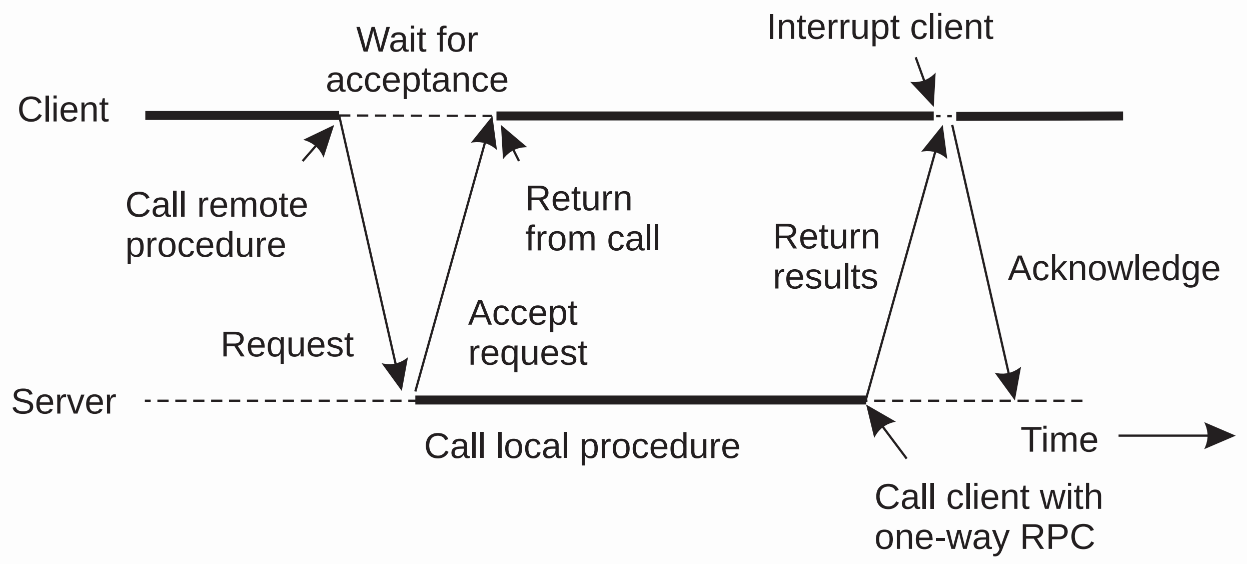




a. The interconnection between client and server in a traditional RPC.

b. The interaction using asynchronous RPC.

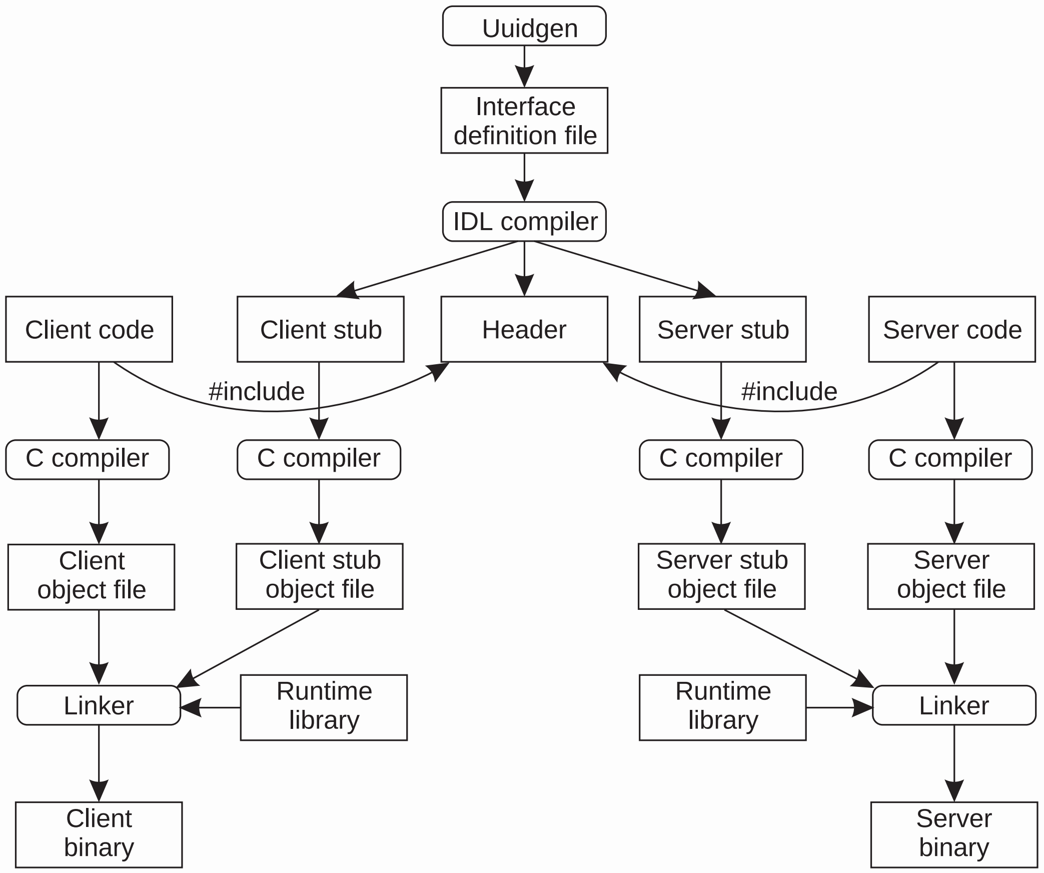
Asynchronous RPC (2)



deferred synchronous RPC – asynchronous RPC with second call done by the server,

one-way RPC – client does not wait for acceptance of the request , problem with reliability.

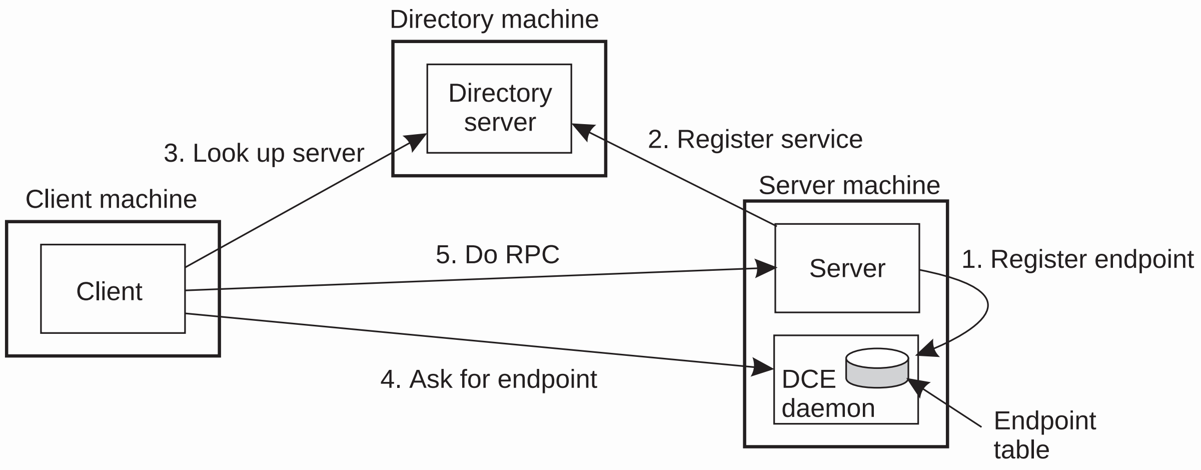
Writing a Client and a Server



Steps in writing a client and a server in DCE RPC. Let the developer concentrate only on the client- and server-specific code. Leave the rest for RPC generators and libraries.

Binding a Client to a Server

Client must locate server machine, and locate the server.



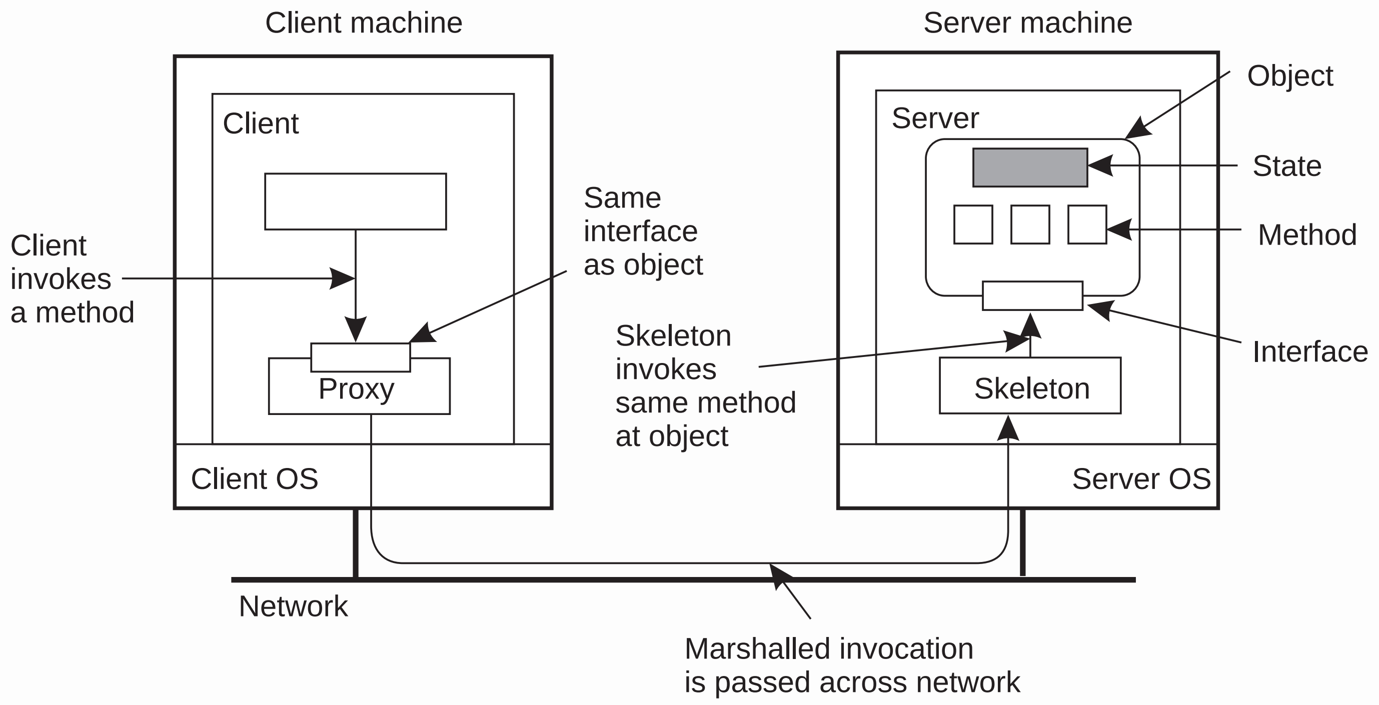
Client-to-server binding in DCE – separate daemon for each server machine.

Remote Distributed Objects (1)

The basic idea of remote objects:

- √ data and operations **encapsulated** in an object,
- √ operations are implemented as **methods**, and are accessible through **interfaces**,
- √ object offers only its **interface** to clients,
- √ **object server** is responsible for a collection of objects,
- √ **client stub (proxy)** implements interface,
- √ **server skeleton** handles (un)marshaling and object invocation.

Remote Distributed Objects (2)



Common organization of a remote object with client-side proxy.

Remote Distributed Objects (3)

Compile-time objects
Language-level objects, from which proxy and skeletons are automatically generated.

Runtime objects
Can be implemented in any language, but require use of an **object adapter** that makes the implementation appear as an object.

- √ **Transient object** lives only by virtue of a server: if the server exits, so will the object.
- √ **Persistent object** lives independently from a server: if a server exits, the object's state and code remain (passively) on disk.

Binding a Client to an Object (1)

Having an object reference allows a client to **bind** to an object:

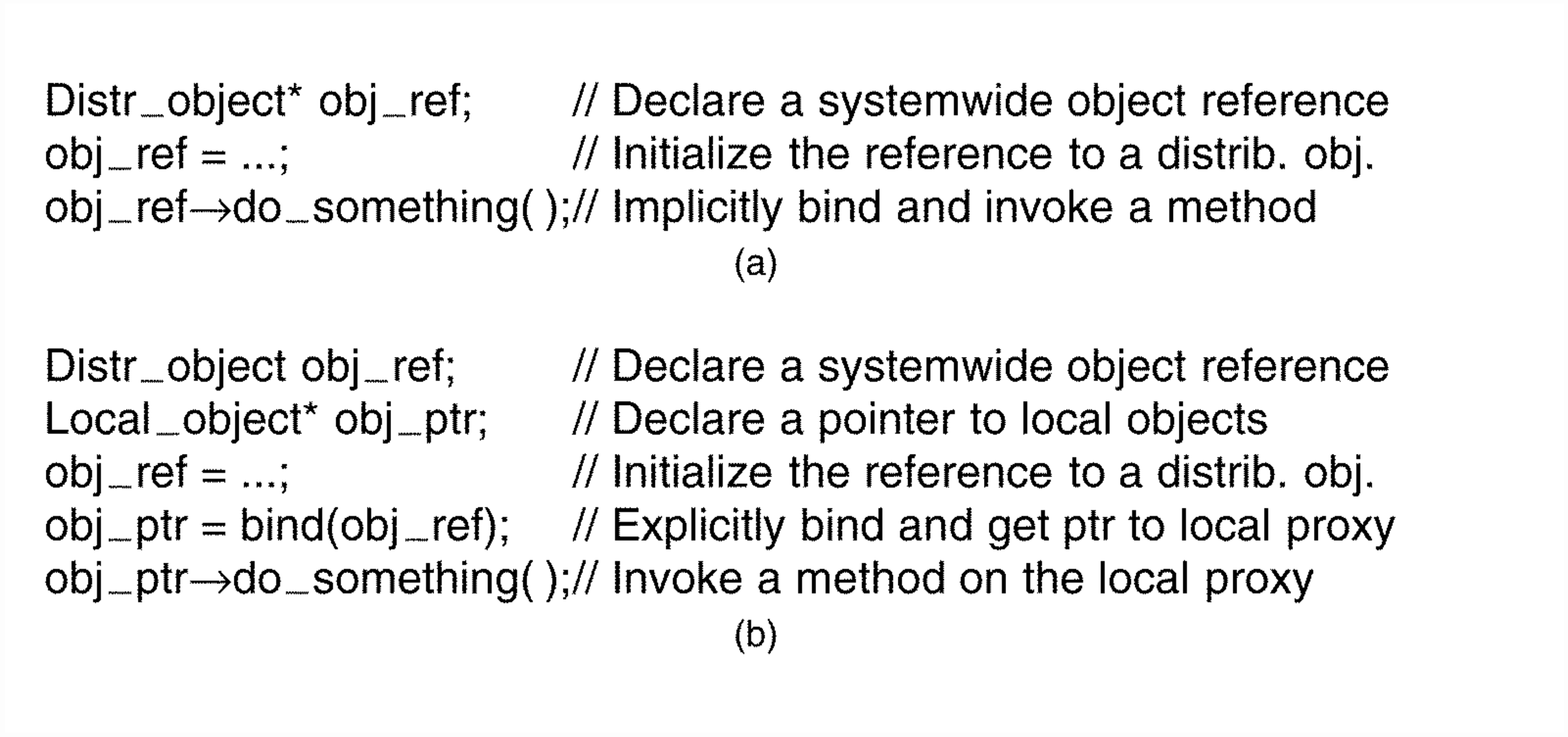
- √ reference denotes server, object, and communication protocol,
- √ client loads associated stub code,
- √ stub is instantiated and initialized for specific object.

Remote-object references enable passing references as parameters, what was hardly possible with ordinary RPCs.

Two ways of binding:

- √ **Implicit:** invoke methods directly on the referenced object.
- √ **Explicit:** client must first explicitly bind to object before invoking it.

Binding a Client to an Object (2)



```
Distr_object* obj_ref;        // Declare a systemwide object reference
obj_ref = ...;                // Initialize the reference to a distrib. obj.
obj_ref→do_something( );// Implicitly bind and invoke a method
```

(a)

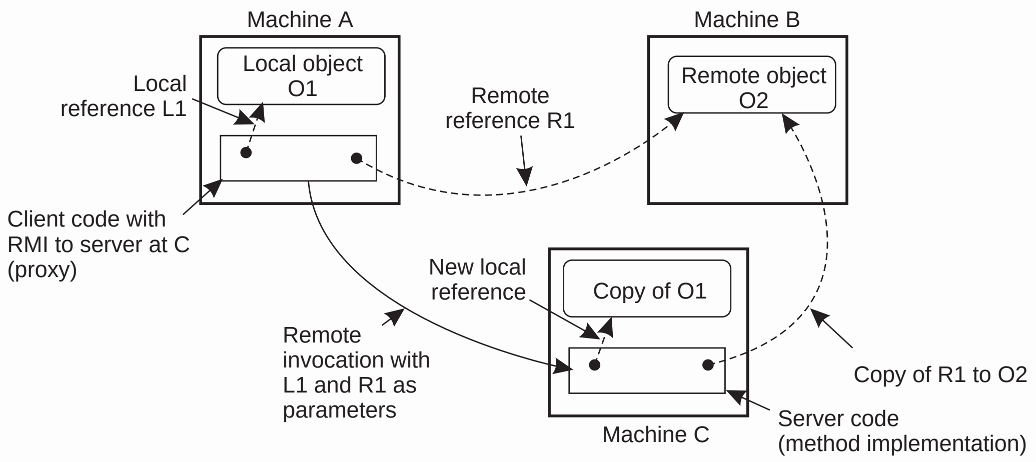
```
Distr_object obj_ref;         // Declare a systemwide object reference
Local_object* obj_ptr;        // Declare a pointer to local objects
obj_ref = ...;                // Initialize the reference to a distrib. obj.
obj_ptr = bind(obj_ref);      // Explicitly bind and get ptr to local proxy
obj_ptr→do_something( );// Invoke a method on the local proxy
```

(b)

a. Example with implicit binding using only global references.

b. Example with explicit binding using global and local references.

RMI - Parameter Passing



Objects sometimes passed by reference, but sometimes by value.

- √ a client running on machine A, a server on machine C,
- √ the client calls the server with two references as parameters, O1 and O2, to local and remote objects,
- √ copying of an object as a possible side effect of invoking a method with an object reference as a parameter (transparency versus efficiency).