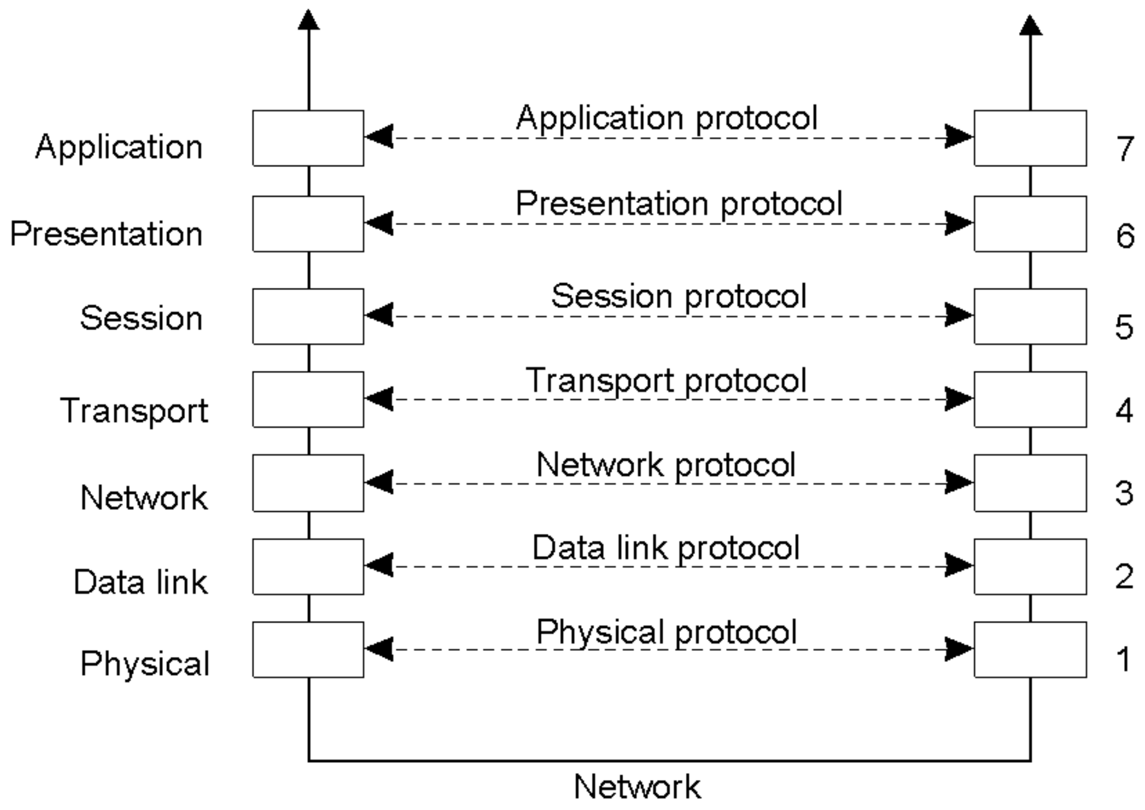
ECE151 – Lecture 2

Chapter 2

Communication

Layered Protocols (1)



Layers, interfaces, and protocols in the OSI model.

Low-level layers

Physical layer: contains the specification and implementation of bits, and their transmission between sender and receiver

Data link layer: prescribes the transmission of a series of bits into a frame to allow for error and flow control

Network layer: describes how packets in a network of computers are to be *routed*.

Observation: for many distributed systems, the lowest level interface is that of the network layer – IP.

Note: IP multicasting is generally considered a standard available service.

Transport Layer

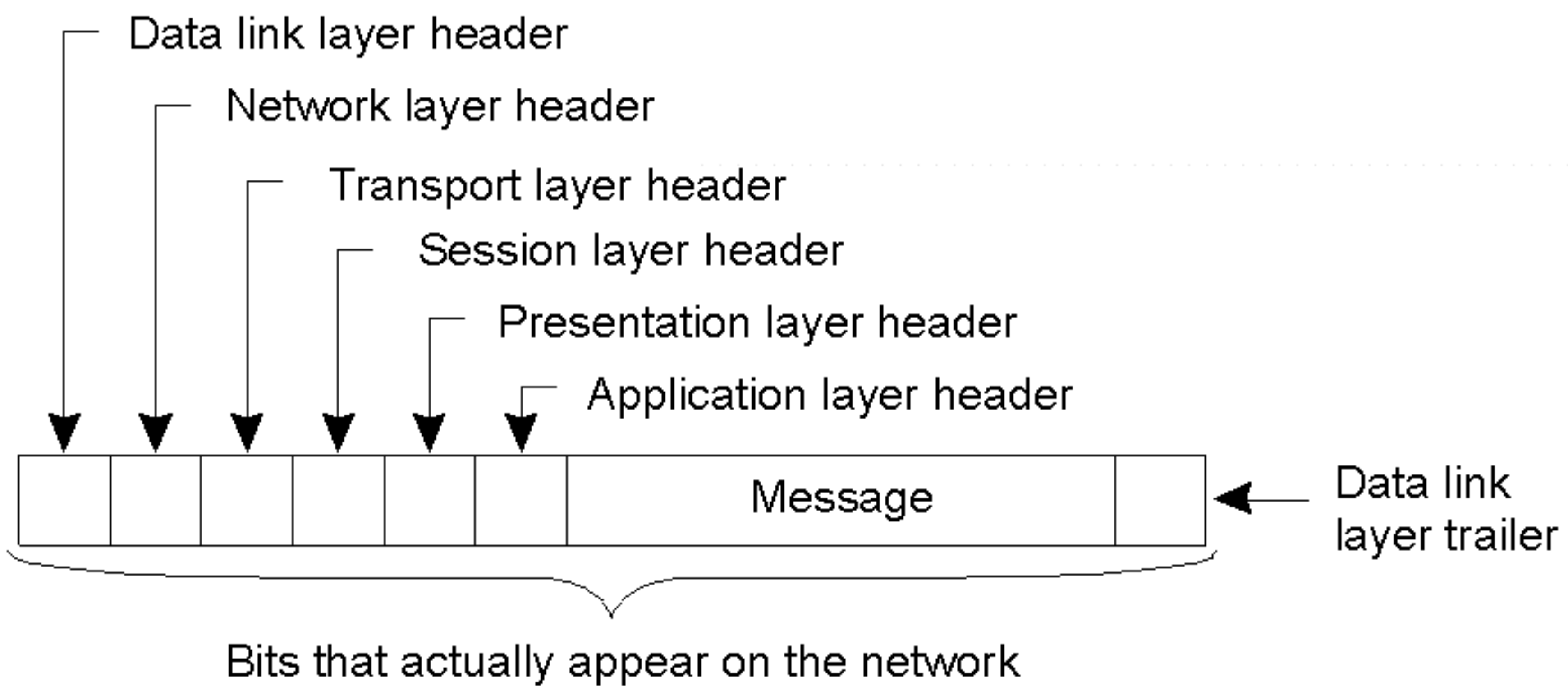
Important: The transport layer provides the actual communication facilities for most distributed systems.

Standard Internet protocols:

TCP: connection-oriented, reliable, stream-oriented communication

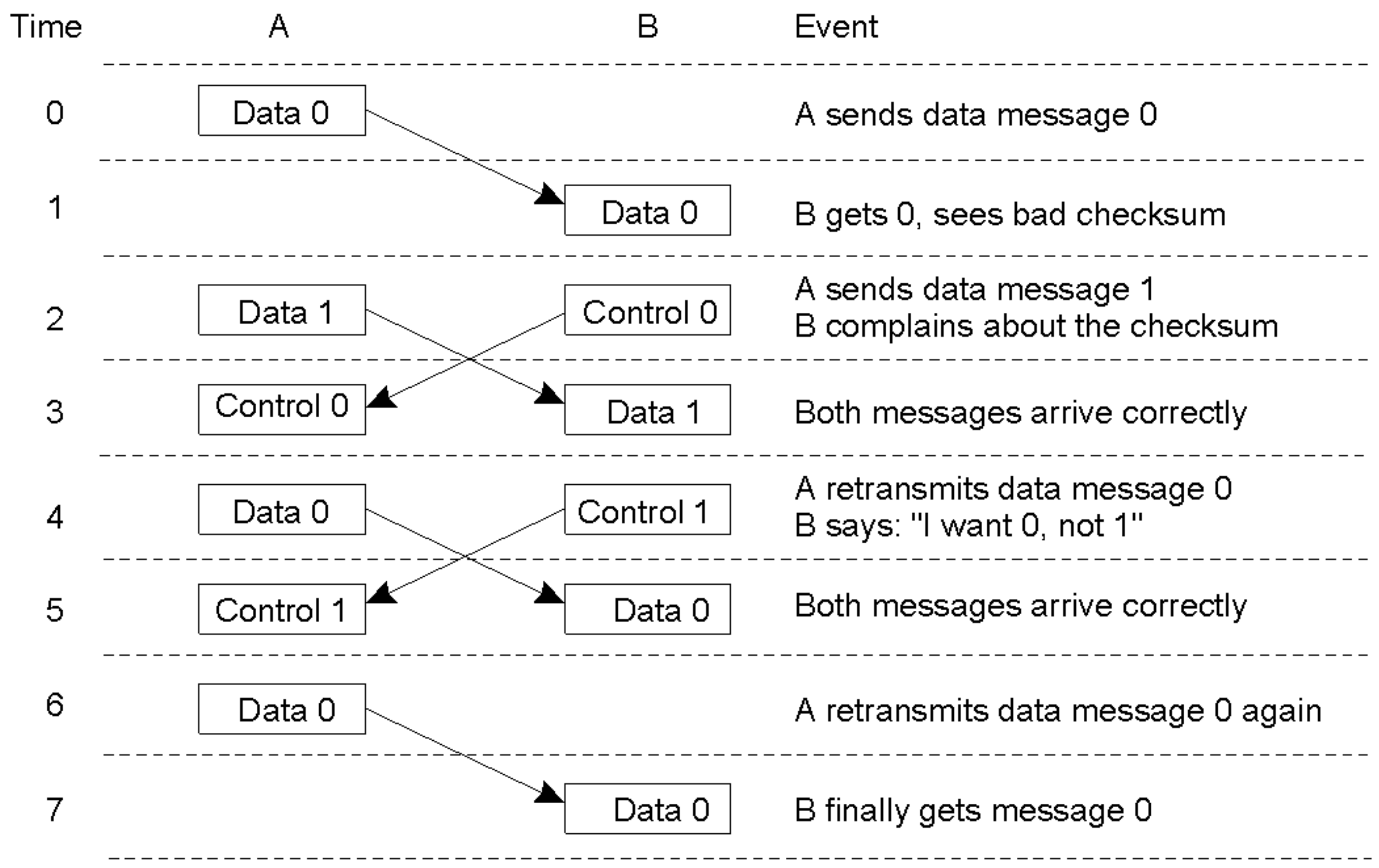
UDP: unreliable (best-effort) datagram communication

Layered Protocols (2)



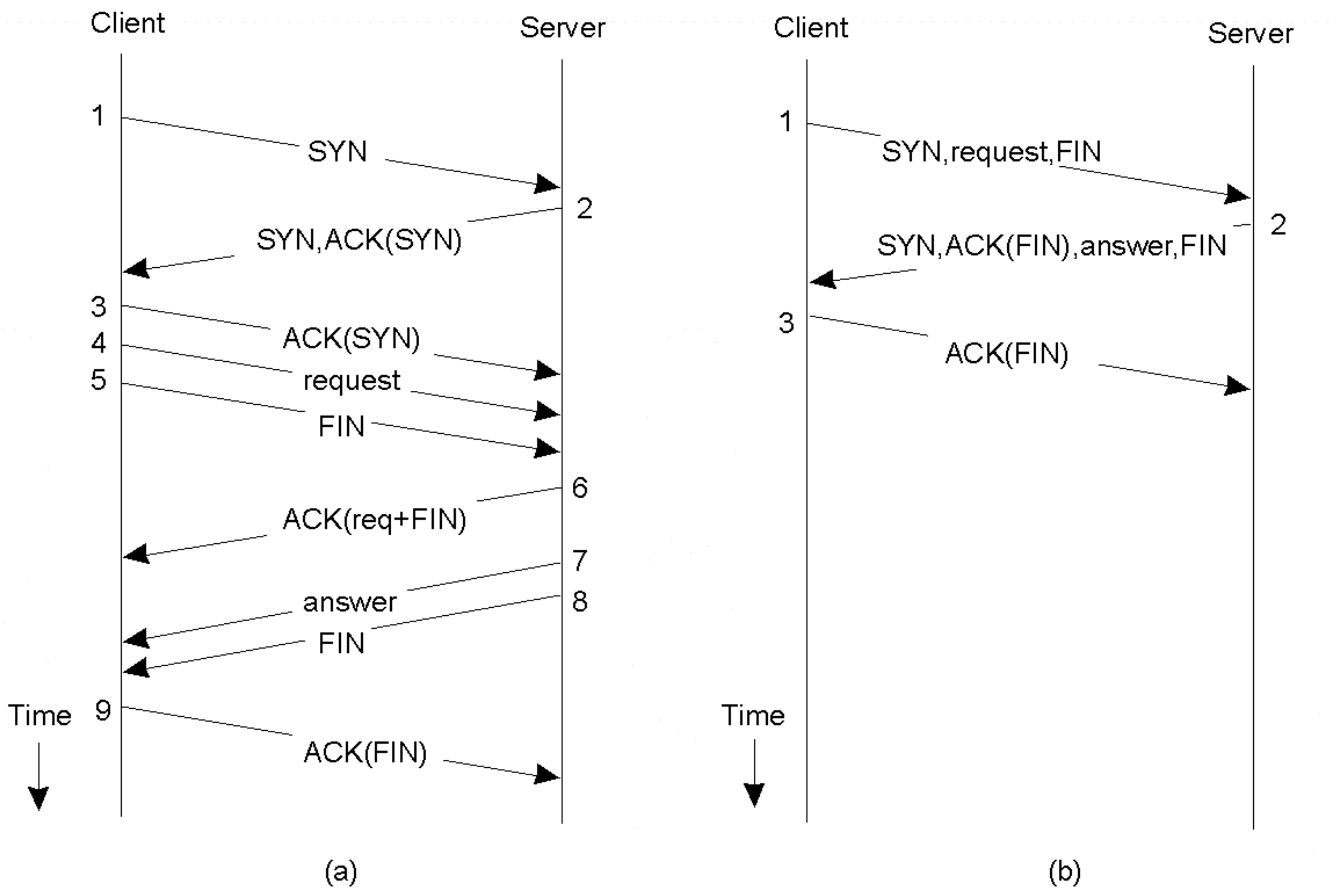
A typical message as it appears on the network.

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.

Client-Server TCP



a) Normal operation of TCP.

b) Transactional TCP.

Application Layer

Observation: Many application protocols are directly implemented on top of transport protocols that do a lot of application-independent work.

	News	**FTP**	**WWW**
Transfer	NNTP	FTP	HTTP
Encoding	7-bit + MIME	7-bit text + 8-bit binary (user has to guess)	8-bit + content type
Naming	Newsgroup	Host + path	URL
Distribution	Push	Pull	Pull
Replication	Flooding	Caching + DNS tricks	Caching + DNS tricks
Security	None (PGP)	Username + Password	Username + Password

Middleware Layer

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

A rich set of communication protocols, but which allow different applications to communicate

Marshaling and unmarshaling of data, necessary for integrated systems

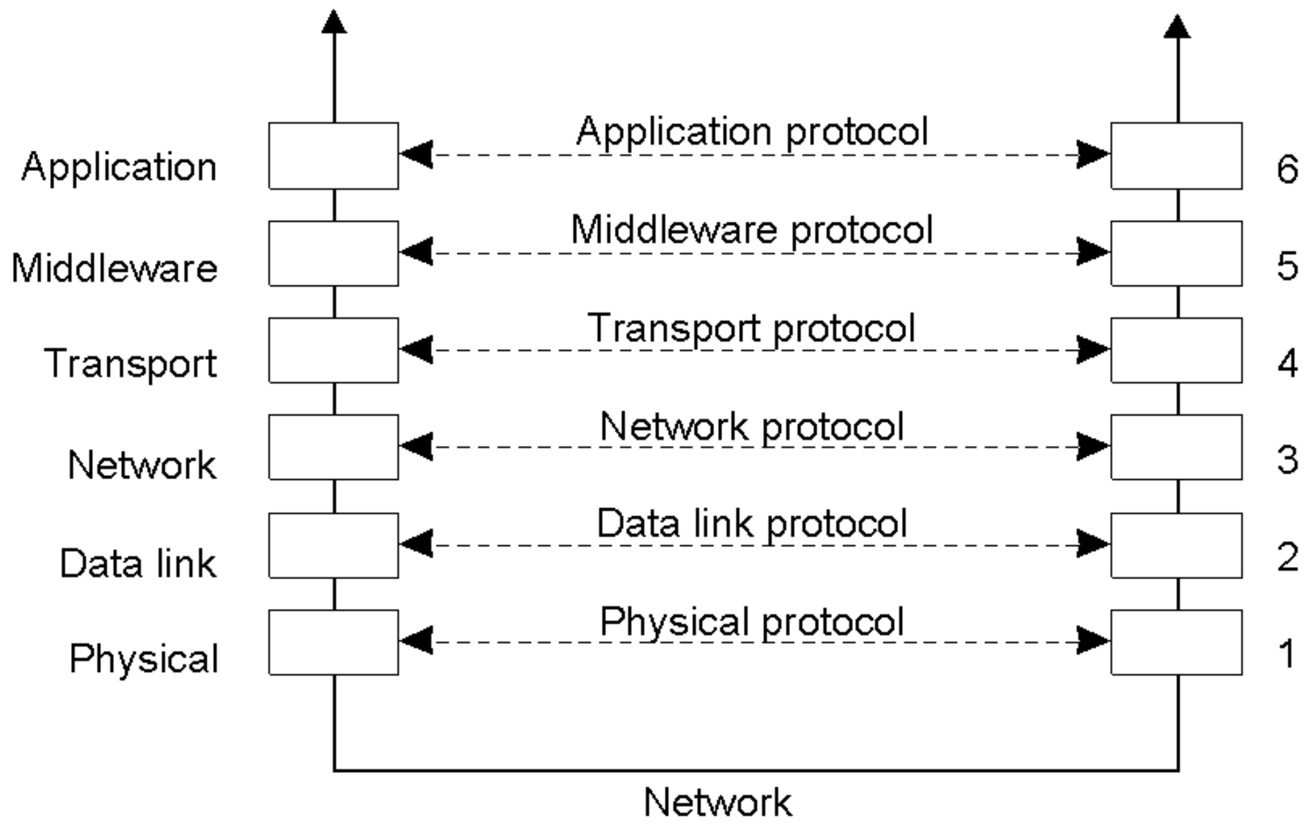
Naming protocols, so that different applications can easily share resources

Security protocols, to allow different applications to communicate in a secure way

Scaling mechanisms, such as support for replication and caching

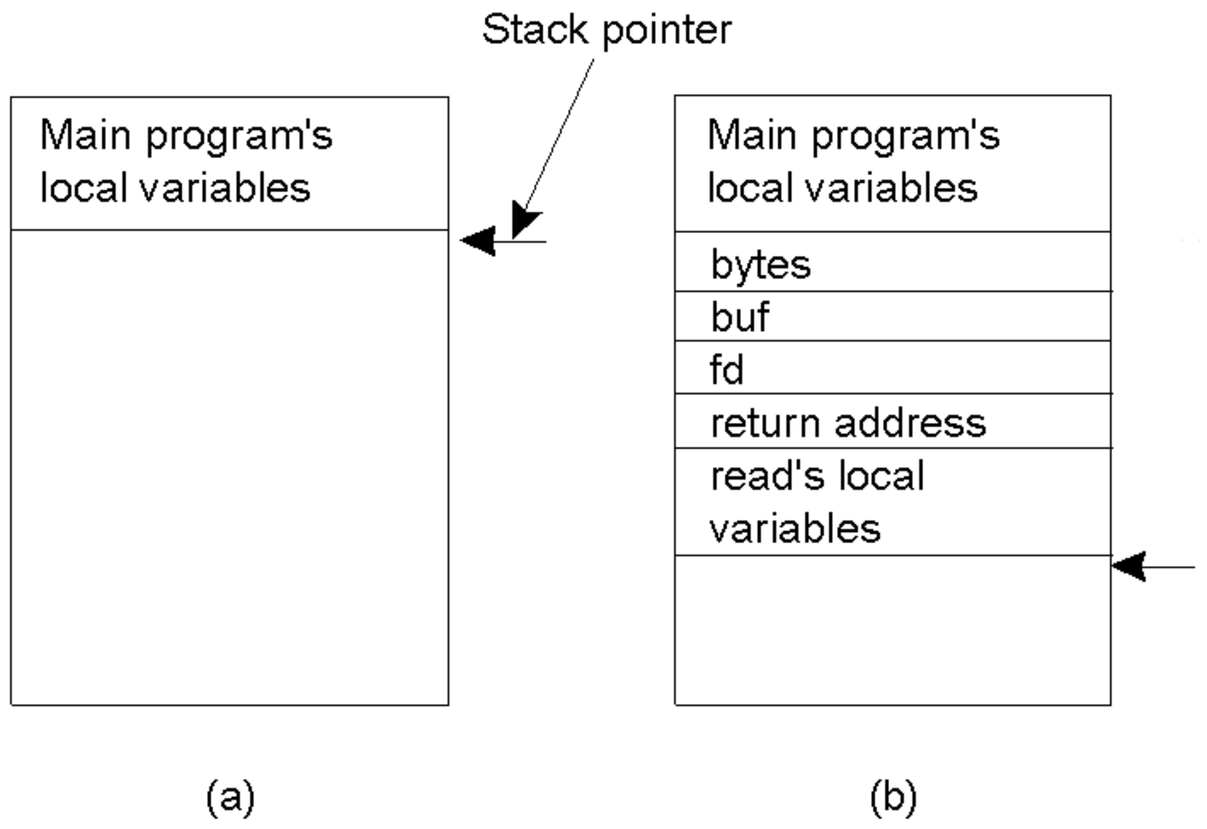
Note: what remains are truly *application-specific* protocols

Middleware Protocols



An adapted reference model for networked communication.

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

Basic RPC Operation

Observations:

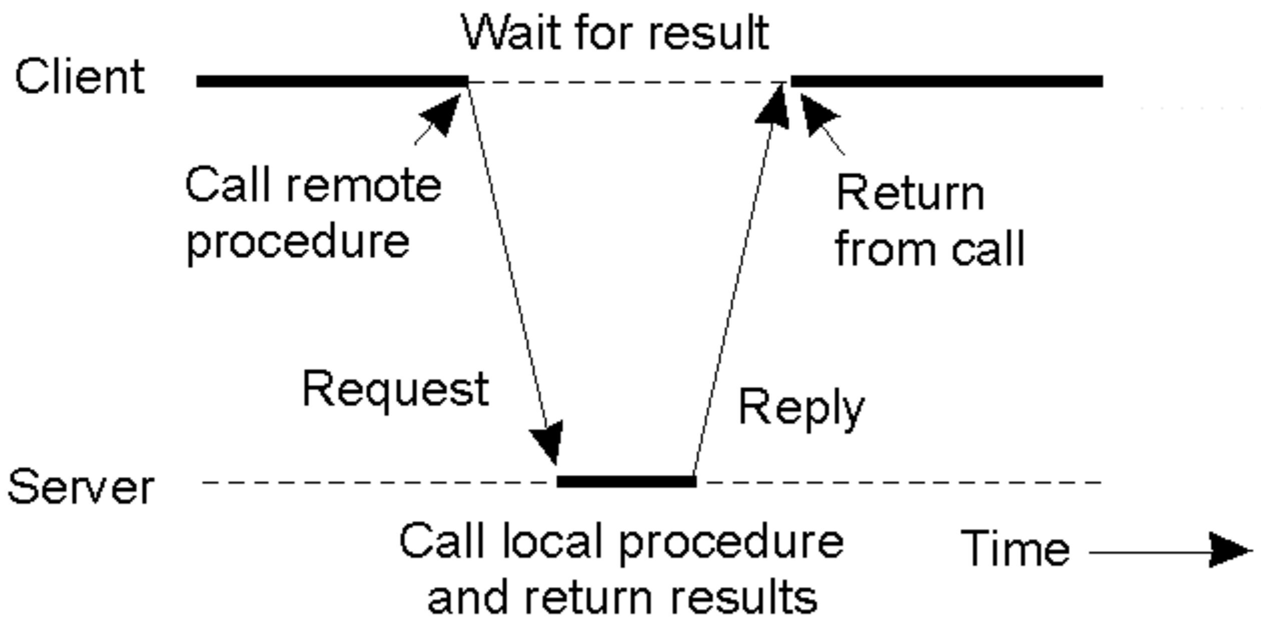
Application developers are familiar with simple procedure model

Well-engineered procedures operate in isolation (black box)

There is no fundamental reason not to execute procedures on separate machine

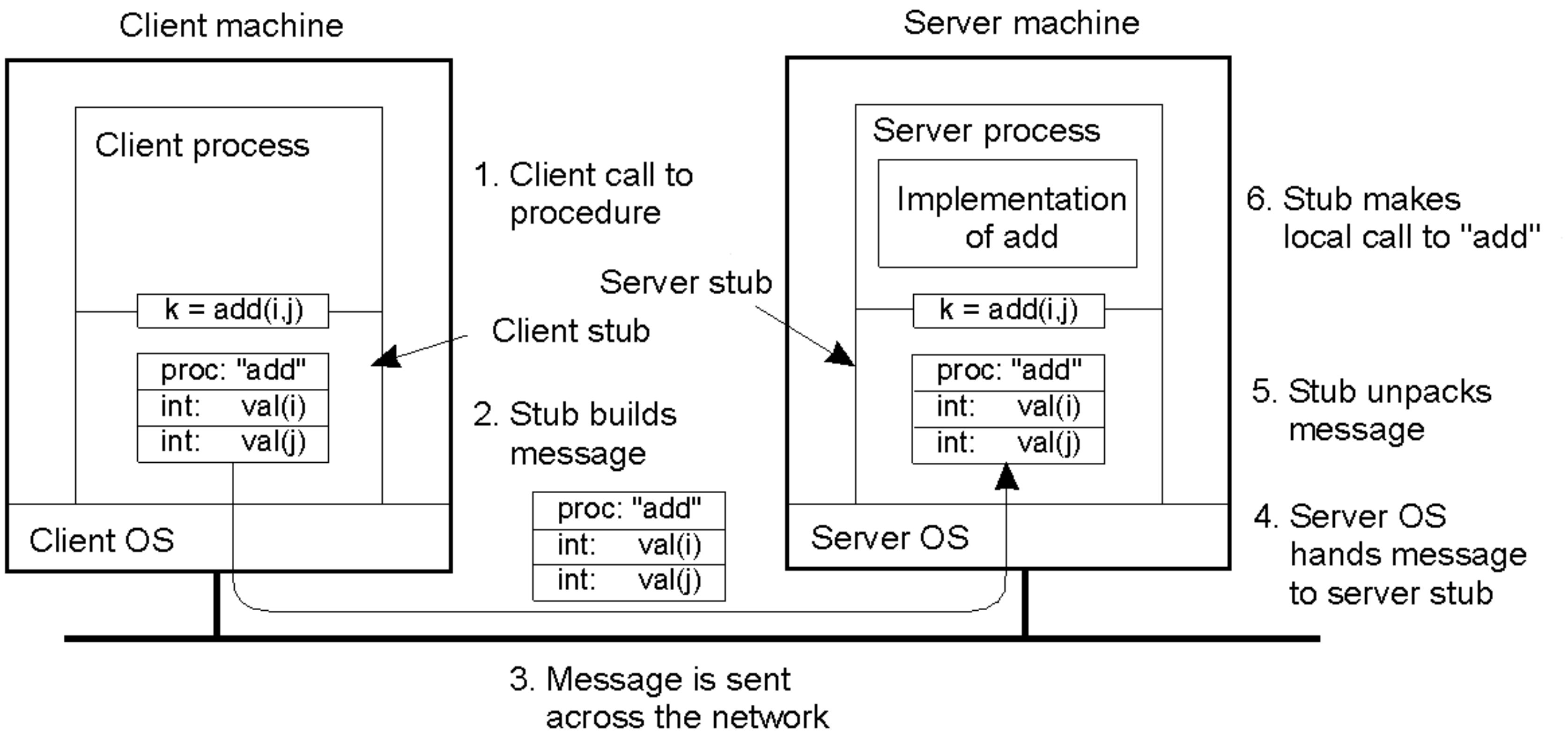
Conclusion: communication between caller & callee can be hidden by using procedure-call mechanism.

Client and Server Stubs



Principle of RPC between a client and server program.

Steps of a Remote Procedure Call



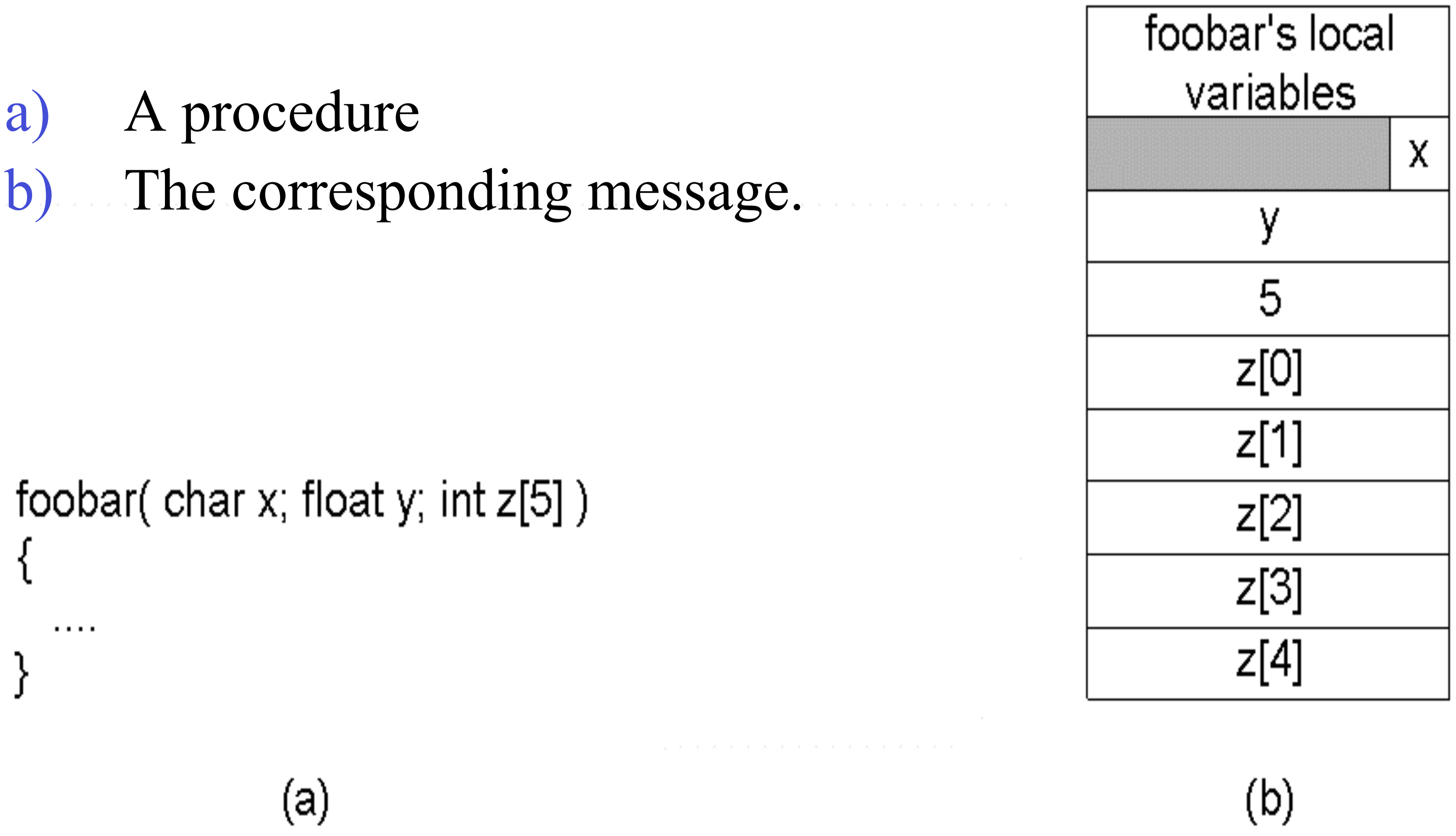
Steps involved in doing remote computation through RPC

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

Parameter Specification and Stub Generation

a) A procedure
b) The corresponding message.



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

RPC: Parameter Passing

Parameter marshaling: There's more than just wrapping parameters into a message:

Client and server *machines* may have different data representations (think of byte ordering)

Wrapping a parameter means transforming a value into a sequence of bytes

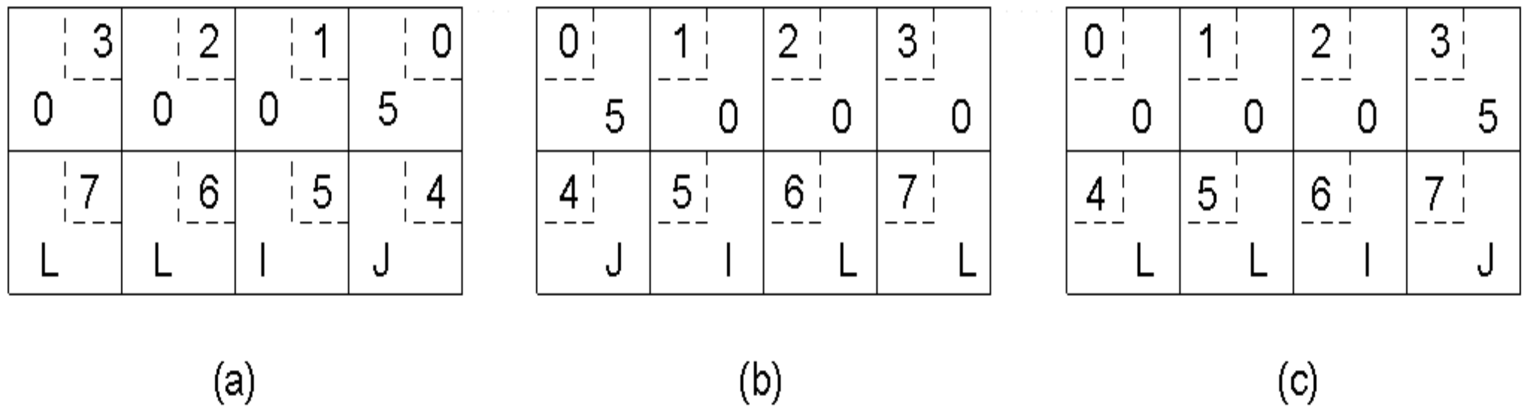
Client and server have to agree on the same encoding:

– How are basic data values represented (integers, floats, characters)

– How are complex data values represented (arrays, unions)

Client and server need to properly interpret messages, transforming them into machine-dependent representations.

Passing Value Parameters (2)



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

RPC: Parameter Passing

RPC parameter passing:

RPC assumes **copy in/copy out** semantics:
while procedure is executed, nothing can be assumed about parameter values (only Ada supports this model).

RPC assumes *all* data that is to be operated on is passed by parameters. Excludes passing **references** to (global) data.

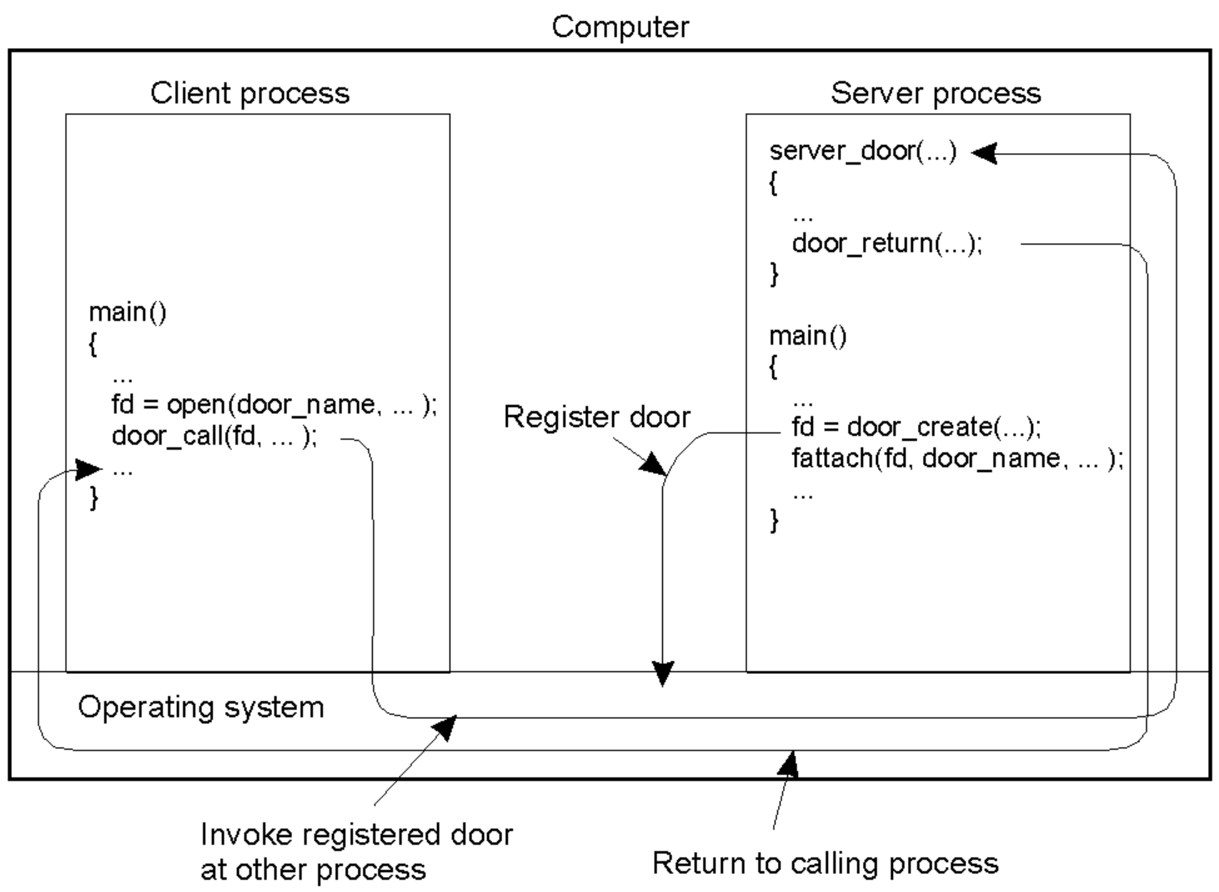
Conclusion: full access transparency cannot be realized.

Observation: If we introduce a **remote reference** mechanism, access transparency can be enhanced:

Remote reference offers unified access to remote data

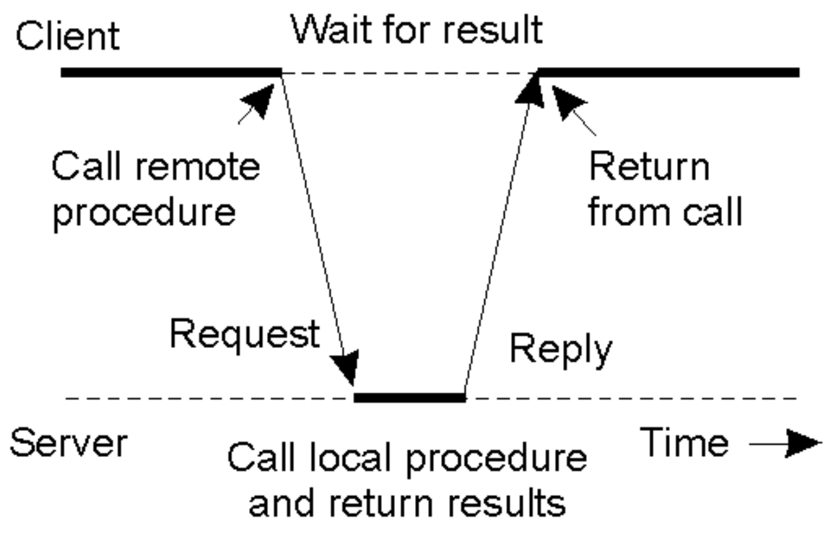
Remote references can be passed as parameter in RPCs

Doors

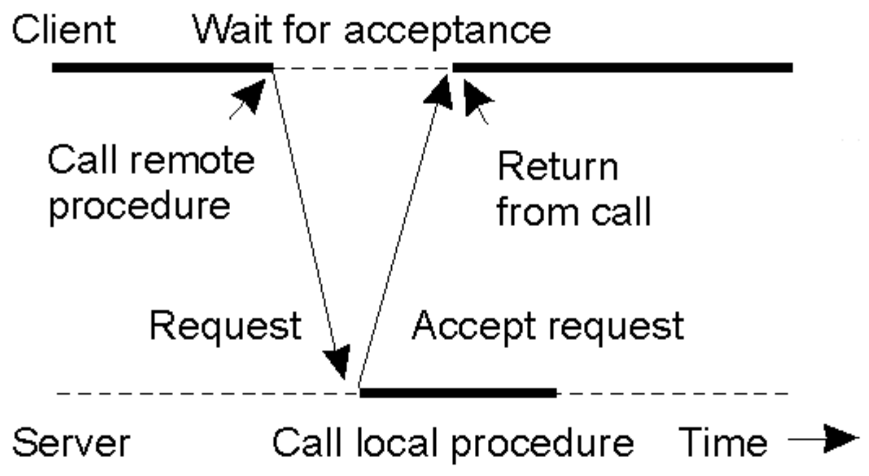


The principle of using Doors as IPC mechanism, even on the same machine.

Asynchronous RPC (1)



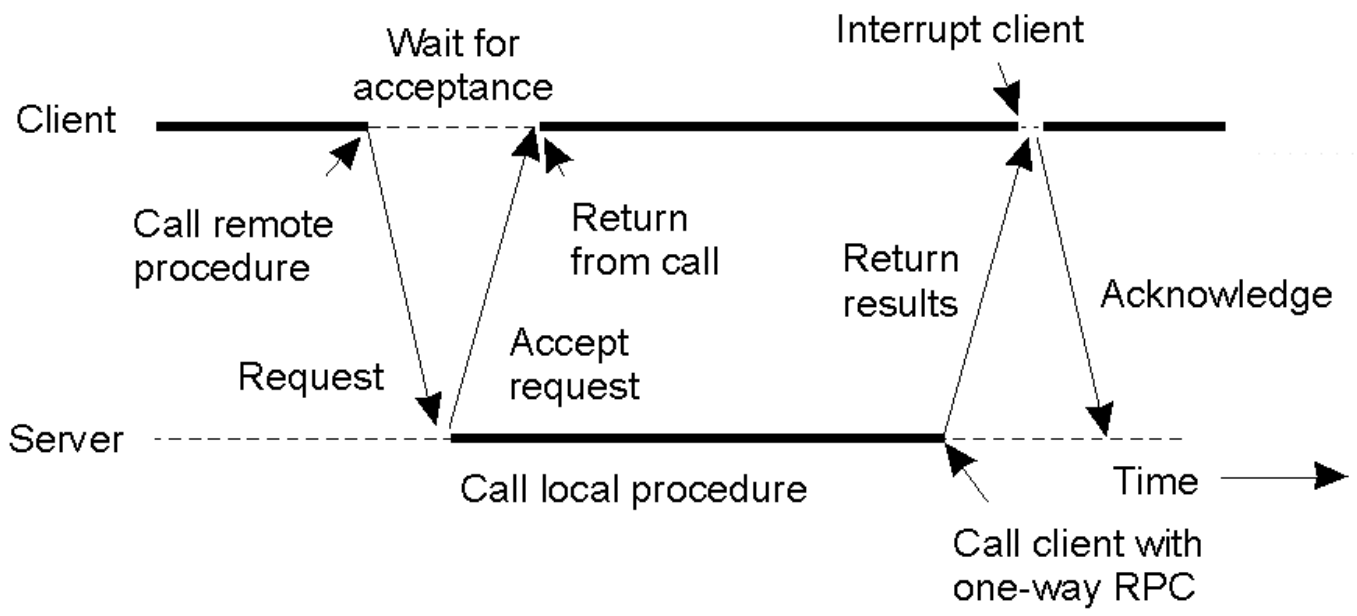
(a)



(b)

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

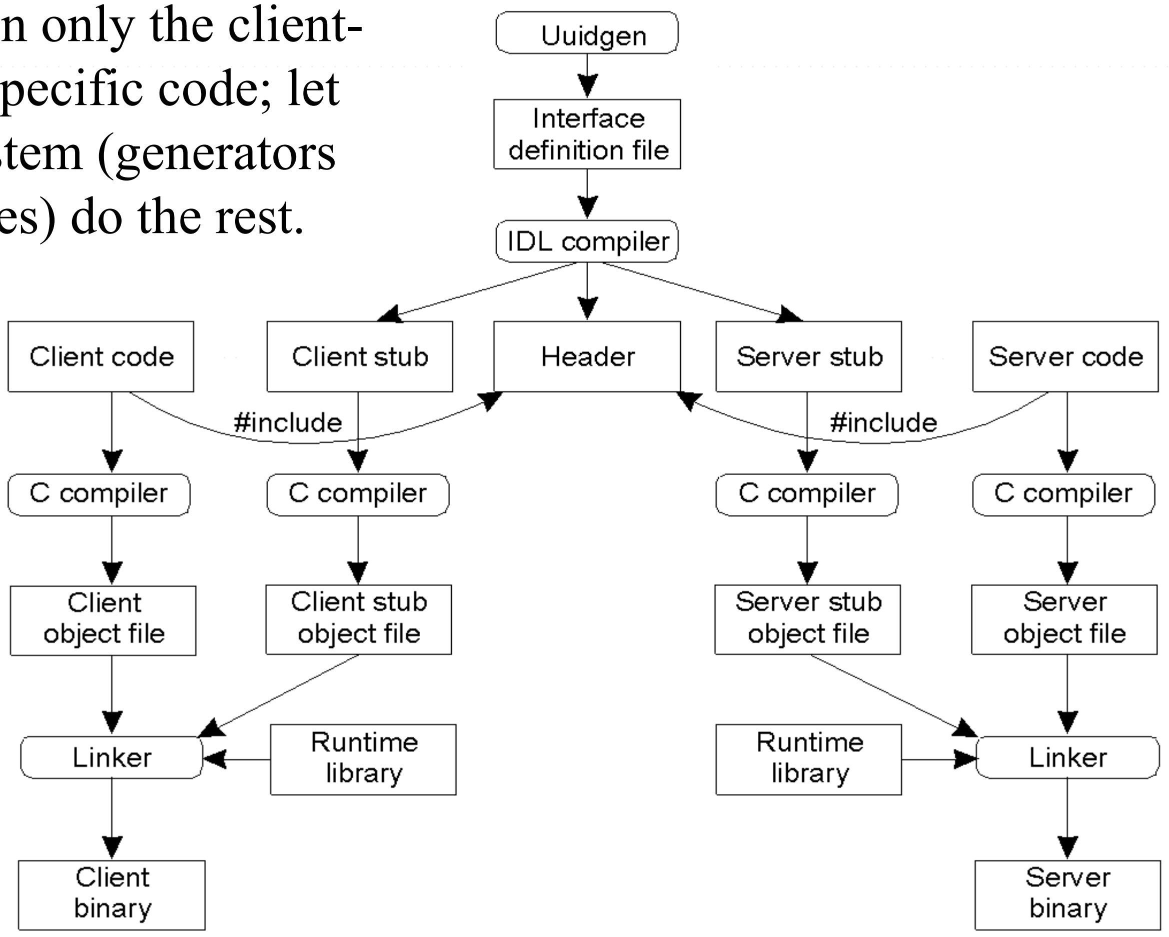
Asynchronous RPC (2)



A client and server interacting through two asynchronous RPCs

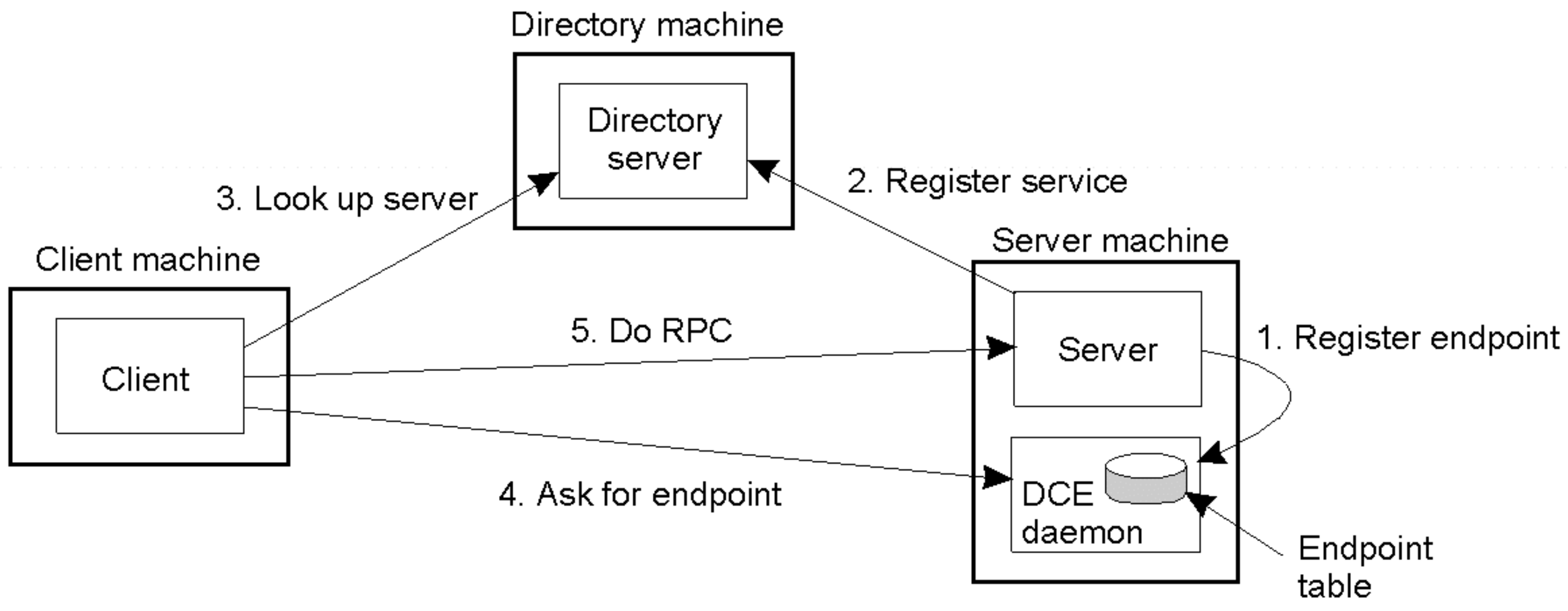
Writing a Client and a Server

Essence: Let the developer concentrate on only the client- and server-specific code; let the RPC system (generators and libraries) do the rest.



The steps in writing a client and a server in DCE RPC.

Binding a Client to a Server



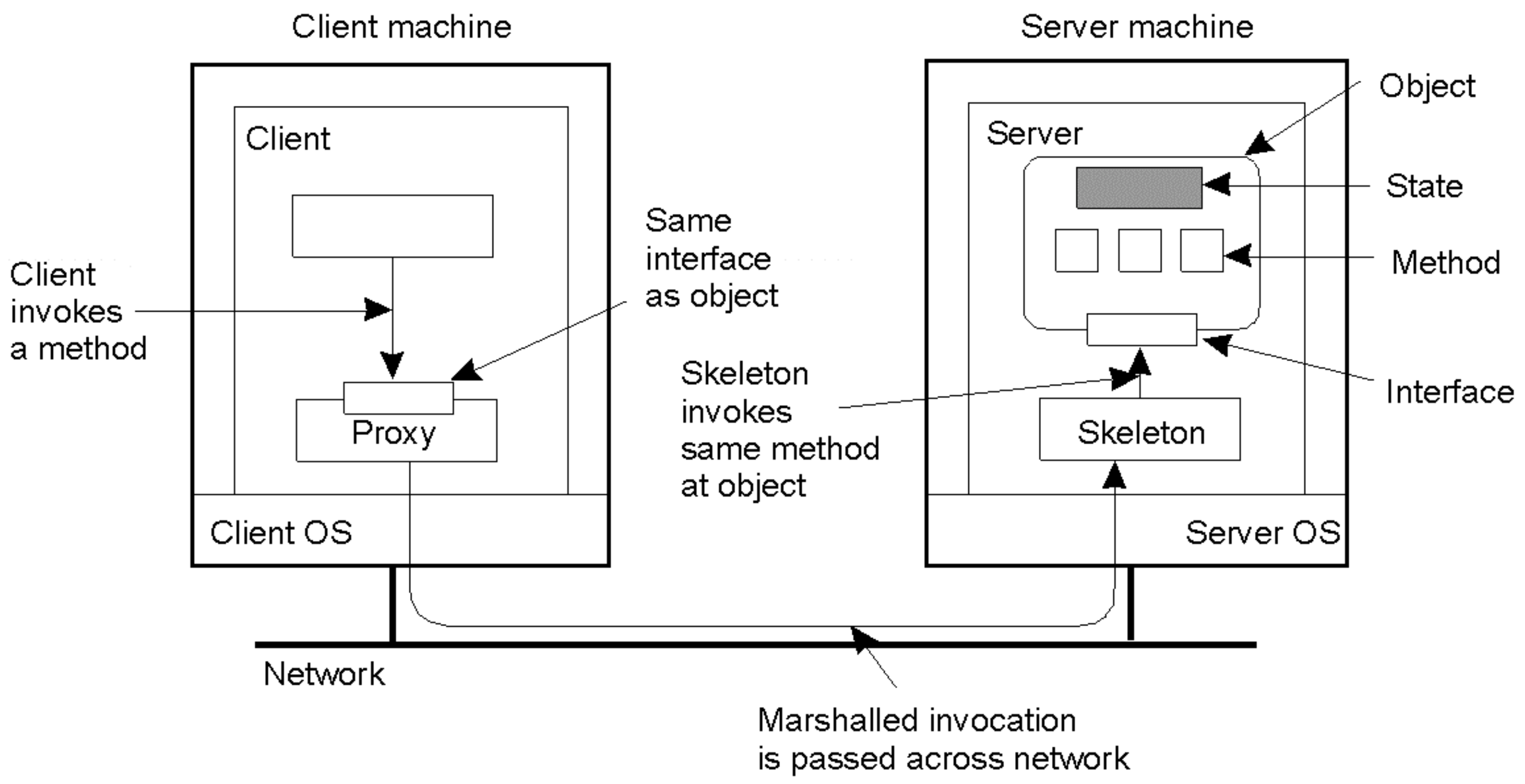
Client-to-server binding in DCE.

Remote Distributed 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

Distributed Objects



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

Remote Distributed Objects

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 objects: live only by virtue of a server: if the server exits, so will the object.

Persistent objects: live independently from a server: if a server exits, the object's state and code remain (passively) on disk.

Client-to-Object Binding

Object reference: 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

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

```
Distr_object* obj_ref;          //Declare a systemwide object reference
obj_ref = …;                    // Initialize the reference to a distributed object
obj_ref-> do_something();       // Implicitly bind and invoke a method
```

(a)

```
Distr_object objPref;           //Declare a systemwide object reference
Local_object* obj_ptr;          //Declare a pointer to local objects
obj_ref = …;                    //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);        //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();      //Invoke a method on the local proxy
```

(b)

a) An example with implicit binding using only global references
b) An example with explicit binding using global and local references

Binding a Client to an Object

Some remarks:

Reference may contain a URL pointing to an implementation file

(Server,object) pair is enough to locate target object

We need only a standard protocol for loading and instantiating code

Observation: Remote-object references allows us to pass references as parameters. This was difficult with ordinary RPCs.

Remote Method Invocation

Basics: (Assume client stub and server skeleton are in place)

Client invokes method at stub

Stub marshals request and sends it to server

Server ensures referenced object is active:

- Create separate process to hold object
- Load the object into server process
- ...

Request is unmarshaled by object's skeleton, and referenced method is invoked

If request contained an object reference, invocation is applied recursively (i.e., server acts as client)

Result is marshaled and passed back to client

Client stub unmarshals reply and passes result to client application

RMI: Parameter Passing

Object reference: Much easier than in the case of RPC:

Server can simply bind to referenced object, and invoke methods

Unbind when referenced object is no longer needed

Object-by-value: A client may also pass a complete object as parameter value:

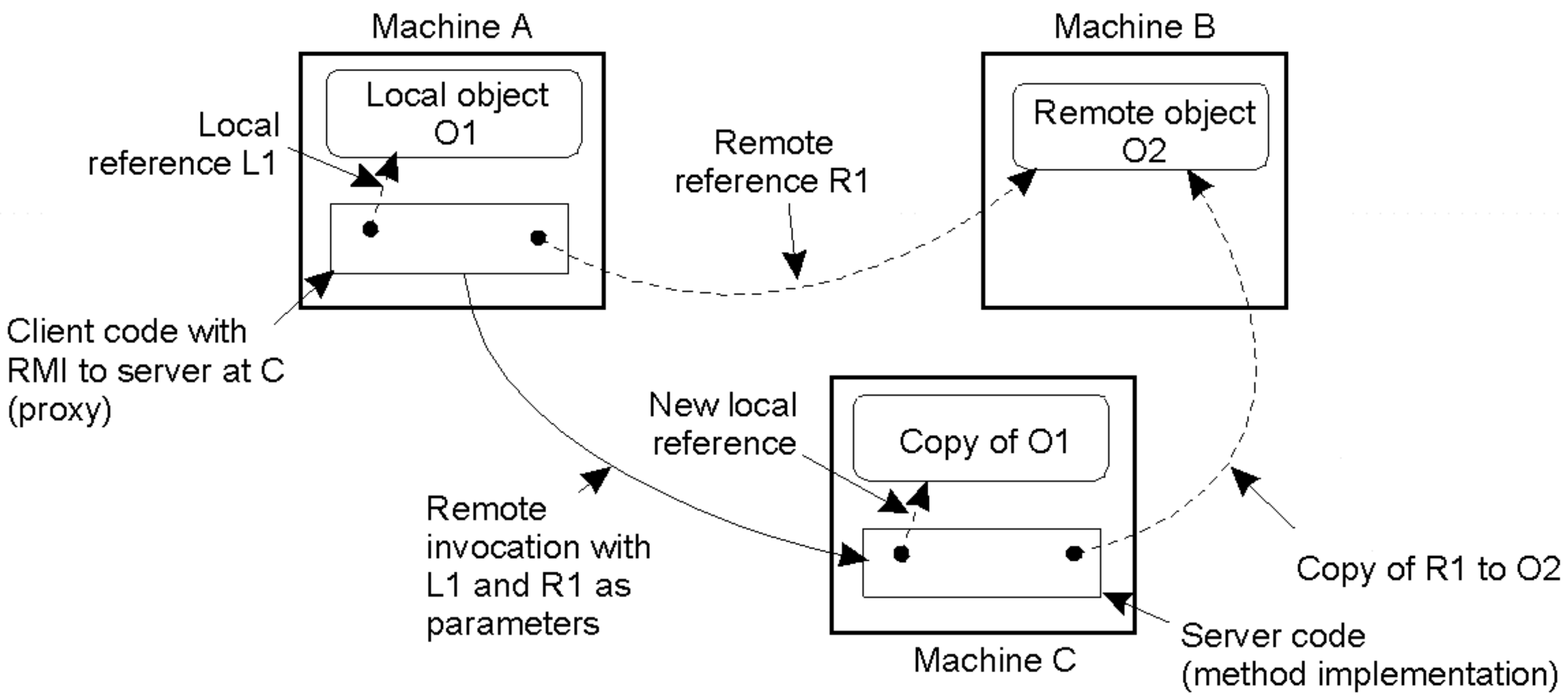
An object has to be marshaled:

- – Marshall its state
- – Marshall its methods, or give a reference to where an implementation can be found

Server unmarshals object. Note that we have now created a *copy* of the original object.

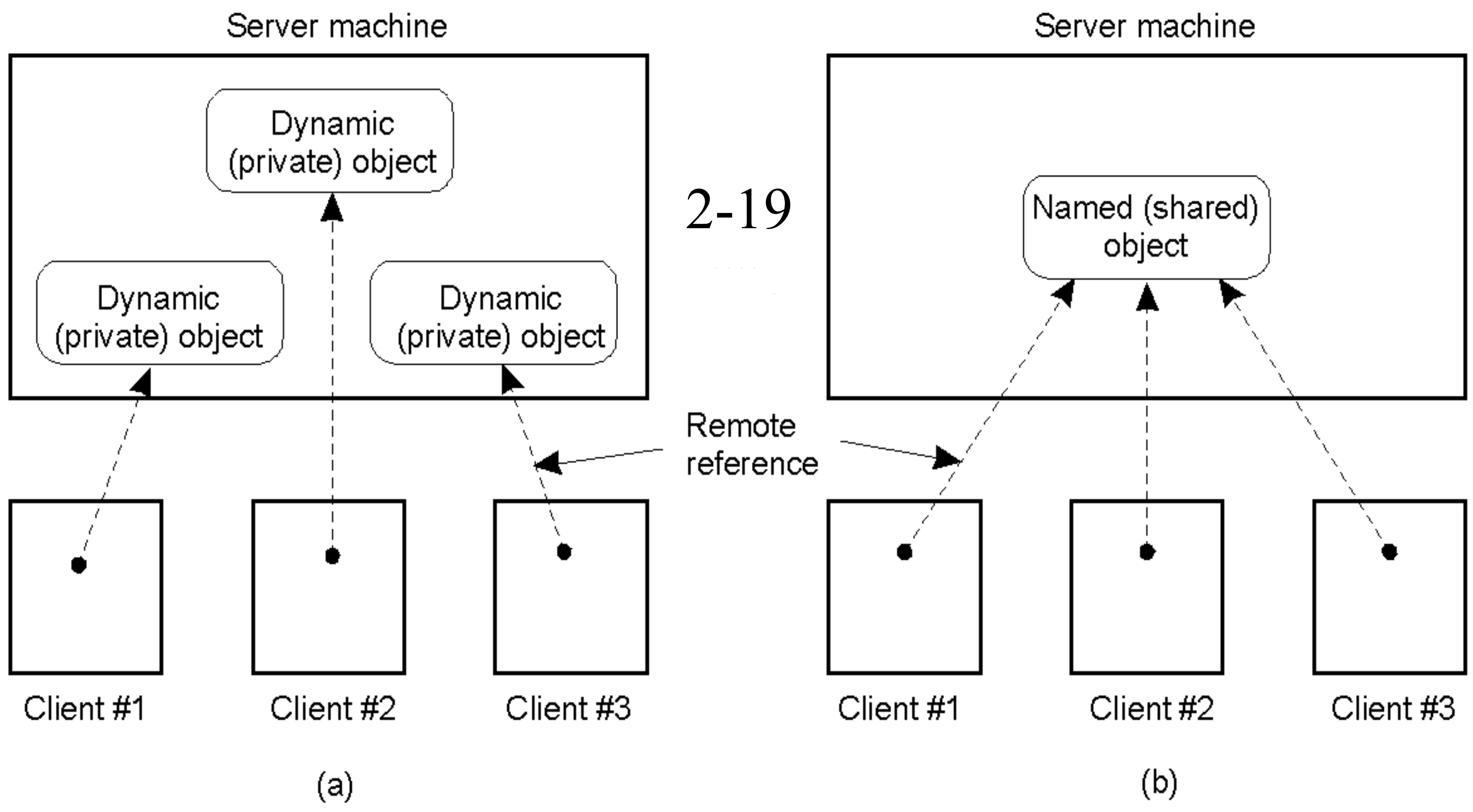
Object-by-value passing tends to introduce nasty problems

Parameter Passing



The situation when passing an object by reference or by value.

The DCE Distributed-Object Model



a) Distributed dynamic objects in DCE.
b) Distributed named objects

Message-Oriented Communication

Synchronous versus asynchronous communications

Message-Queuing System

Message Brokers

Example: IBM MQSeries

Synchronous Communication

Some observations: Client/Server computing is generally based on a model of **synchronous communication**:

Client and server have to be active at the time of communication

Client issues request and blocks until it receives reply

Server essentially waits only for incoming requests, and subsequently processes them

Drawbacks synchronous communication:

Client cannot do any other work while waiting for reply

Failures have to be dealt with immediately (the client is waiting)

In many cases the model is simply not appropriate (mail, news)

Asynchronous Communication

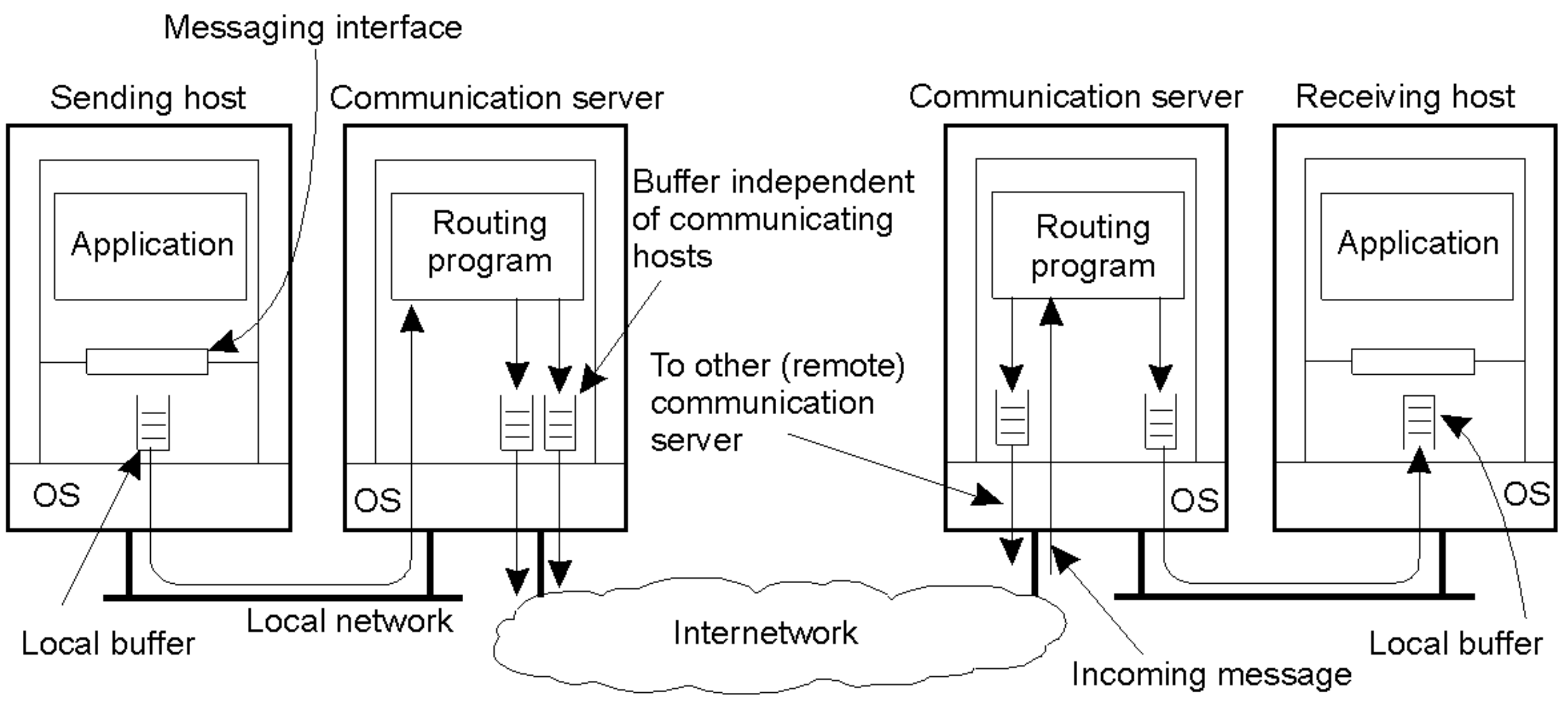
Message-oriented middleware: Aims at high-level **asynchronous** communication:

Processes send each other messages, which are queued

Sender need not wait for immediate reply, but can do other things

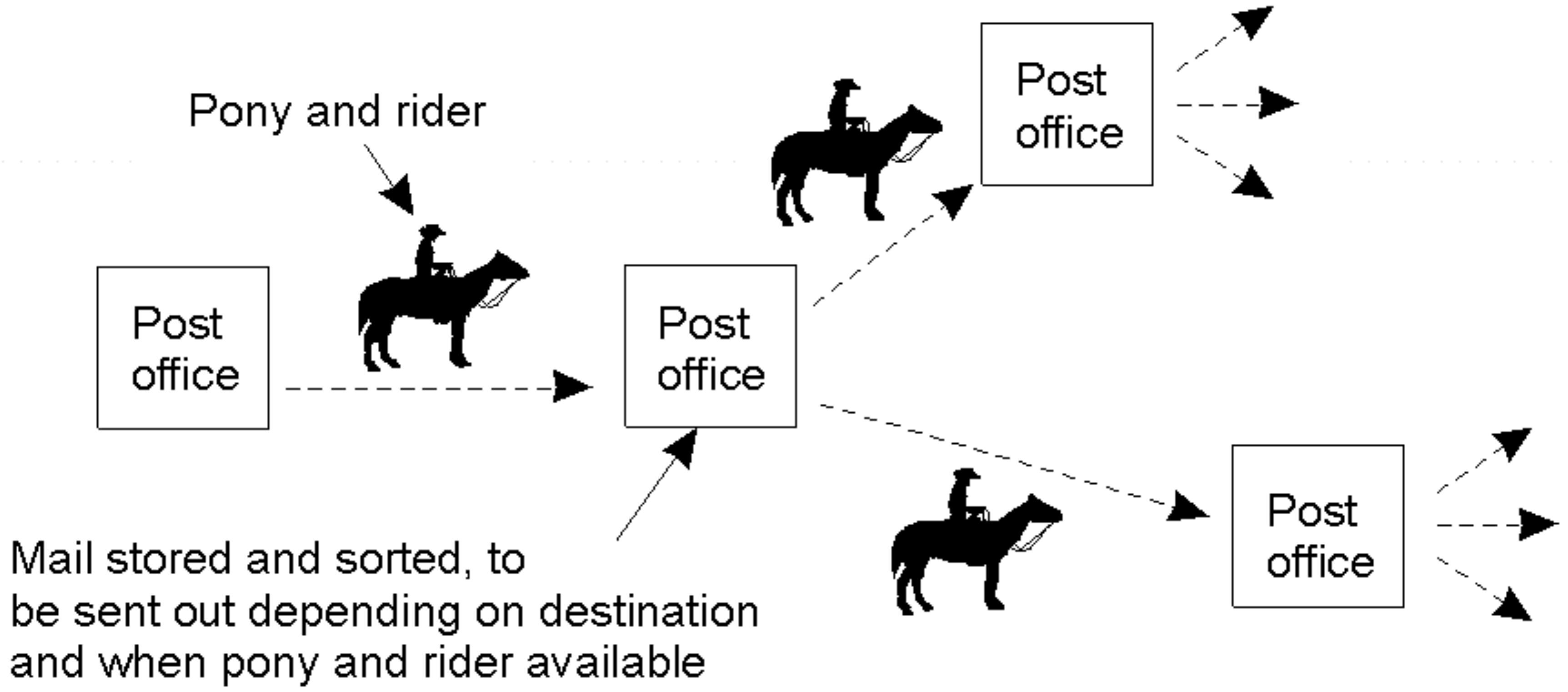
Middleware often facilitates fault tolerance

Persistence and Synchronicity in Communication (1)



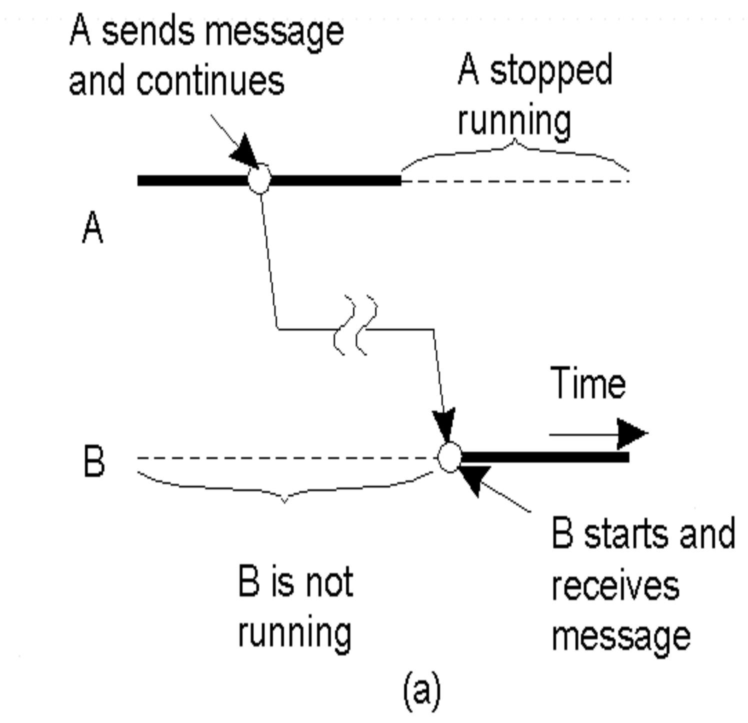
General organization of a communication system in which hosts are connected through a network

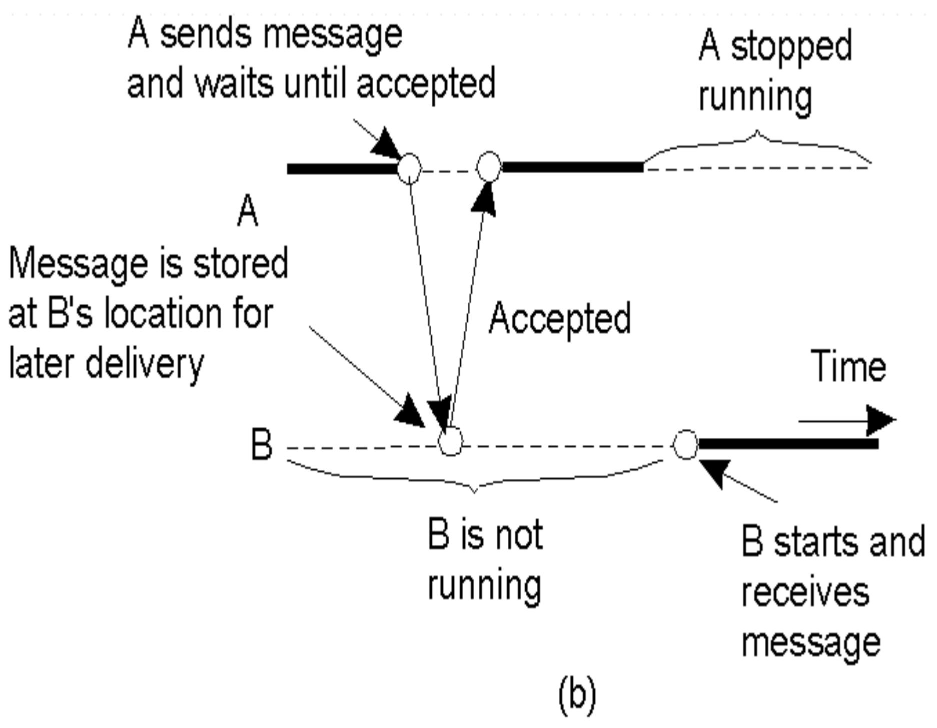
Persistence and Synchronicity in Communication (2)


Pony and rider
Post office
Post office
Post office
Post office
Mail stored and sorted, to be sent out depending on destination and when pony and rider available

Persistent communication of letters back in the days of the Pony Express.

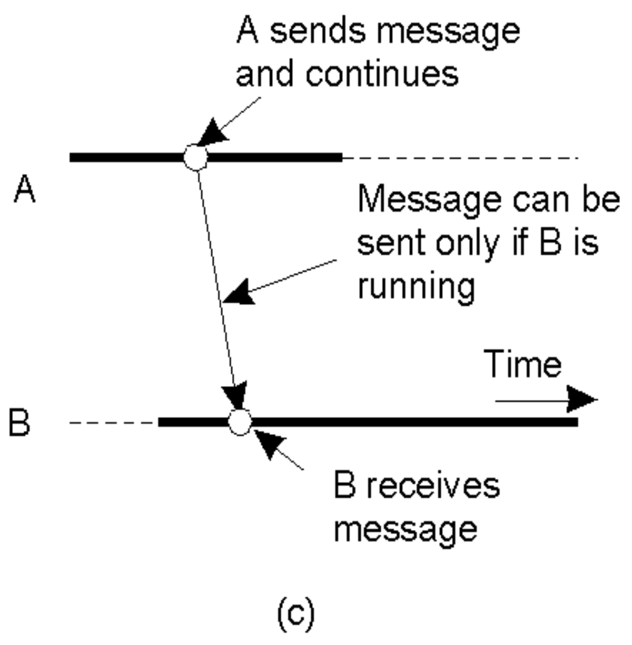
Persistence and Synchronicity in Communication (3)

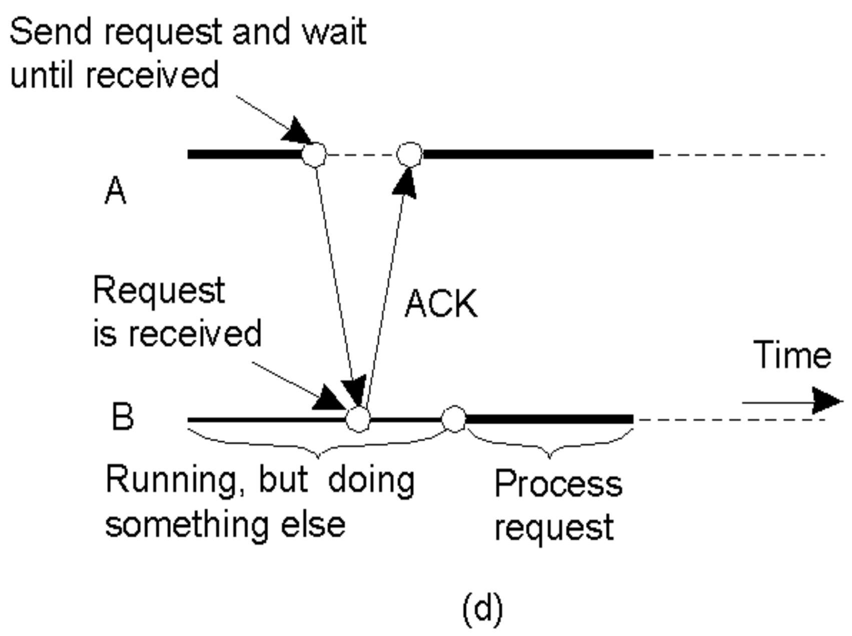



A sends message and waits until accepted
A stopped running
A
Message is stored at B's location for later delivery
Accepted
Time
B
B is not running
B starts and receives message
(b)

a) Persistent asynchronous communication
b) Persistent synchronous communication

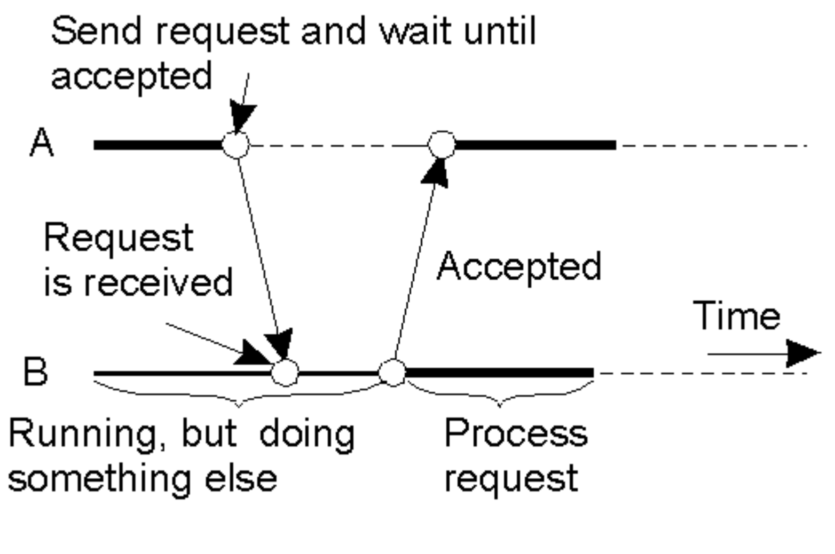
Persistence and Synchronicity in Communication (4)


A sends message and continues
A
Message can be sent only if B is running
Time
B
B receives message
(c)

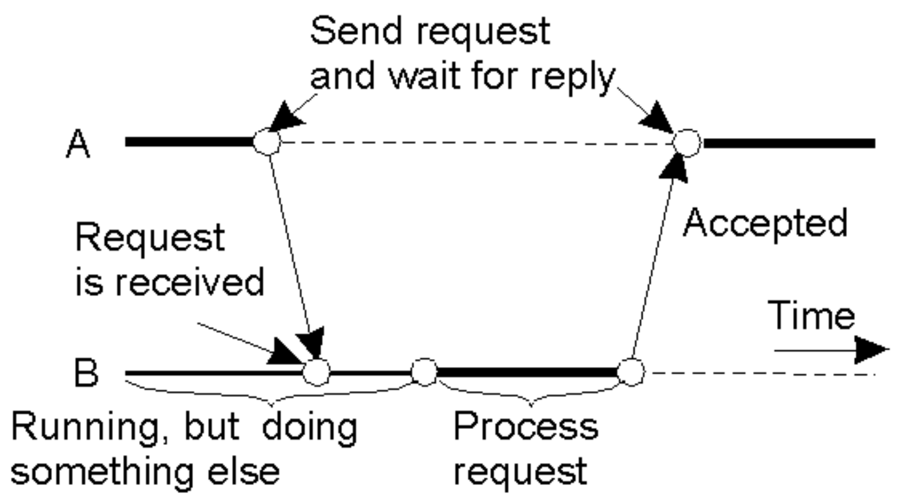

Send request and wait until received
A
Request is received
ACK
Time
B
Running, but doing something else
Process request
(d)

c) Transient asynchronous communication
d) Receipt-based transient synchronous communication

Persistence and Synchronicity in Communication (5)


Send request and wait until accepted
A
Request is received
Accepted
Time
B
Running, but doing something else
Process request

(e)


Send request and wait for reply
A
Request is received
Accepted
Time
B
Running, but doing something else
Process request

(f)

e) Delivery-based transient synchronous communication at message delivery

f) Response-based transient synchronous communication