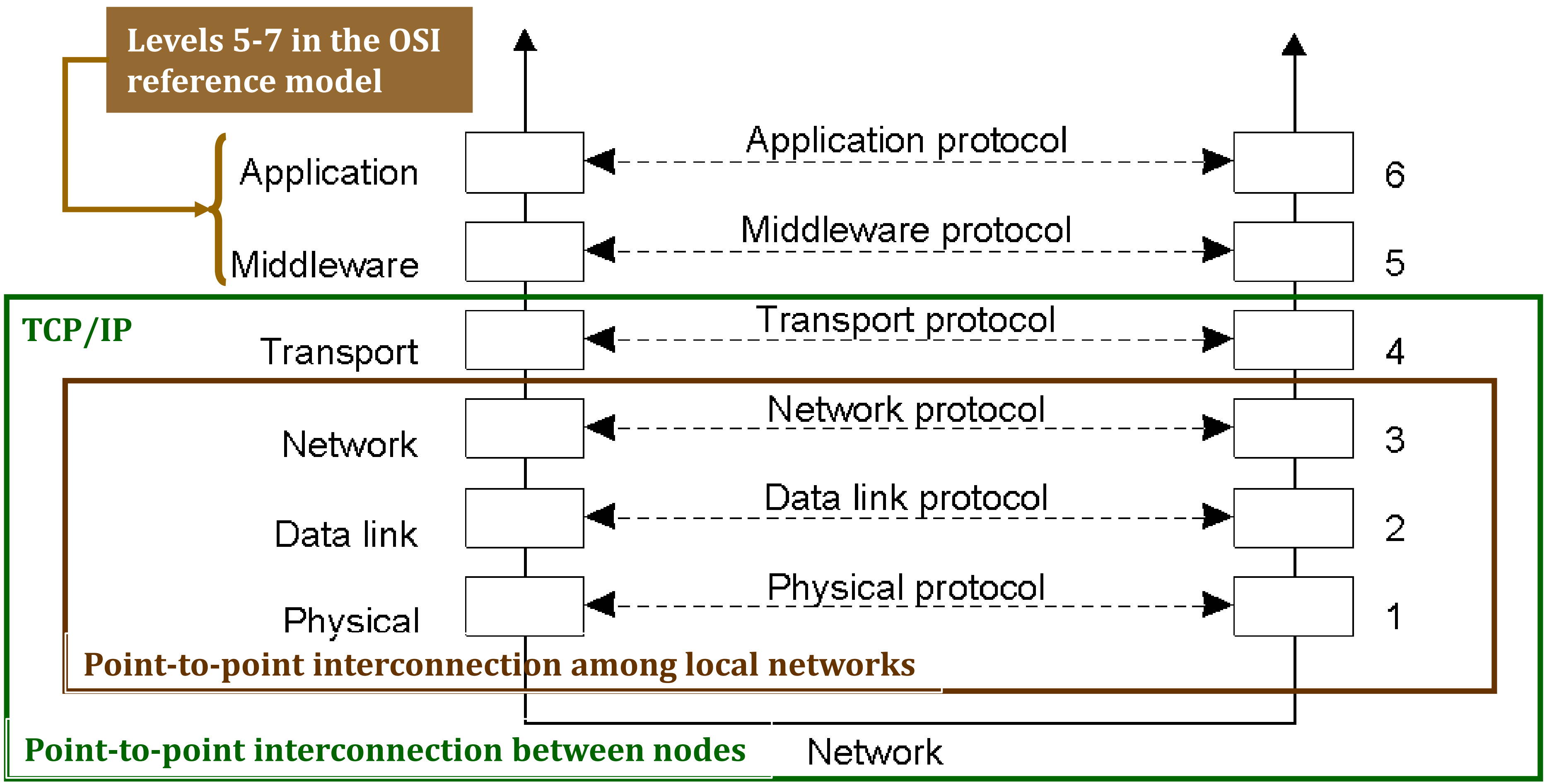
Distributed communications

Runtimes for concurrency and distribution

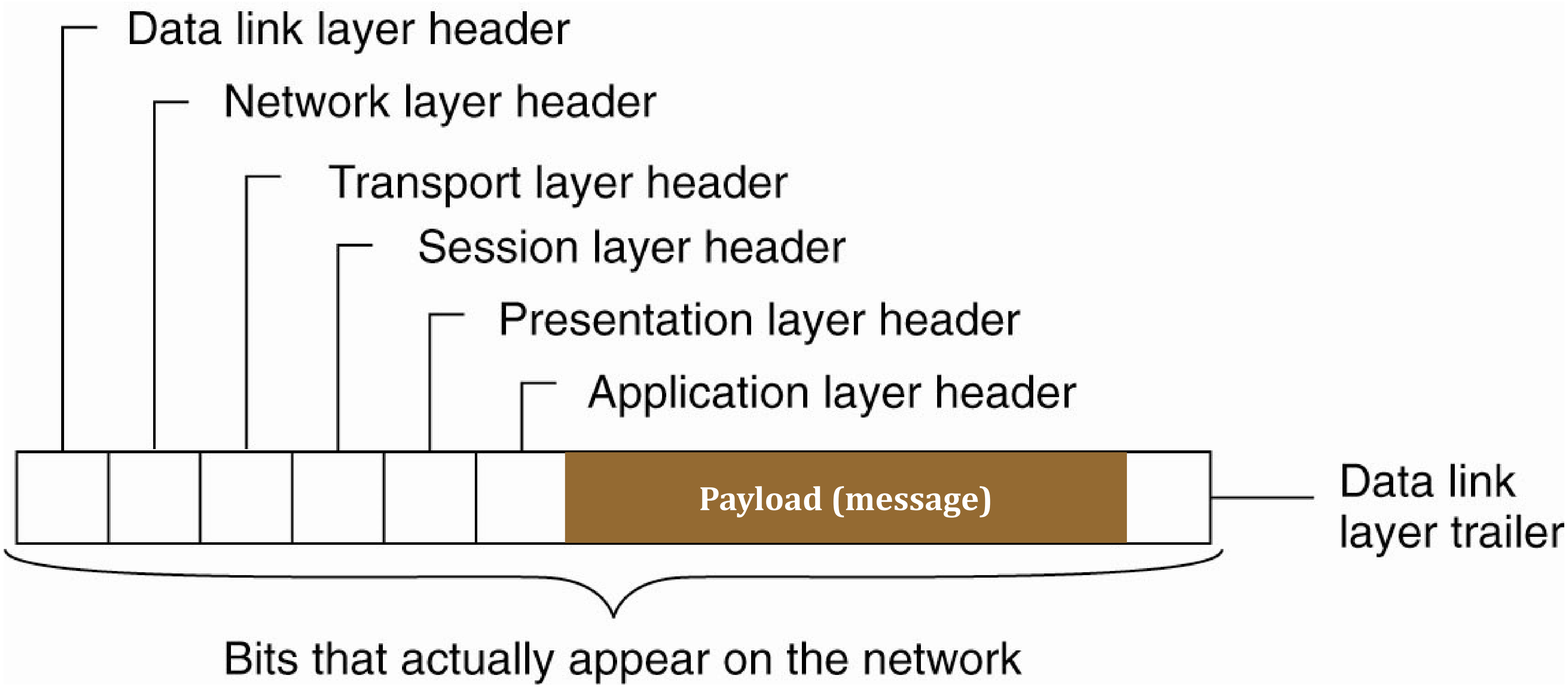
Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2020/2021

A layered view of networked communication – 1



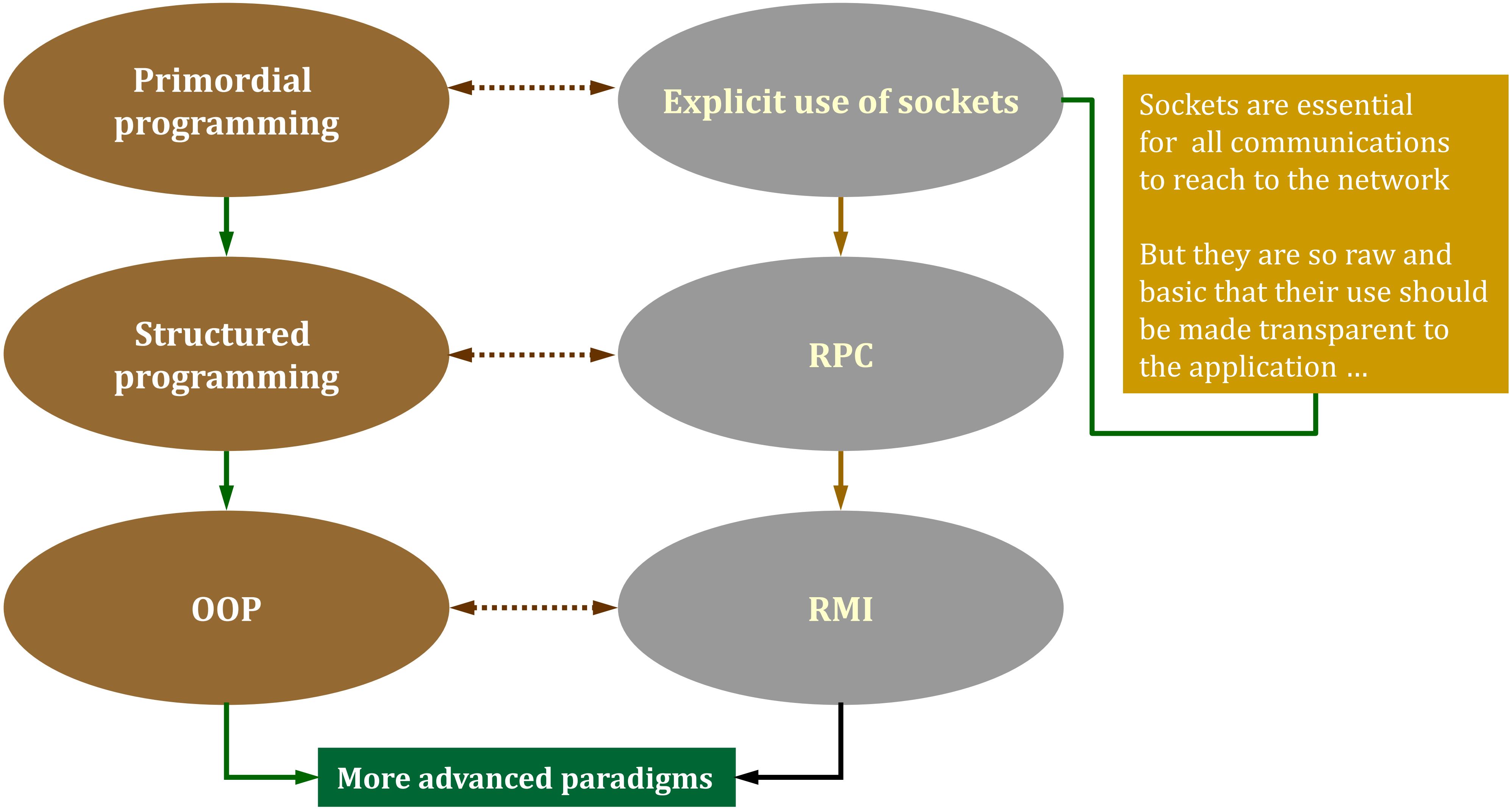
A layered view of networked communication – 2



Models of distributed communication

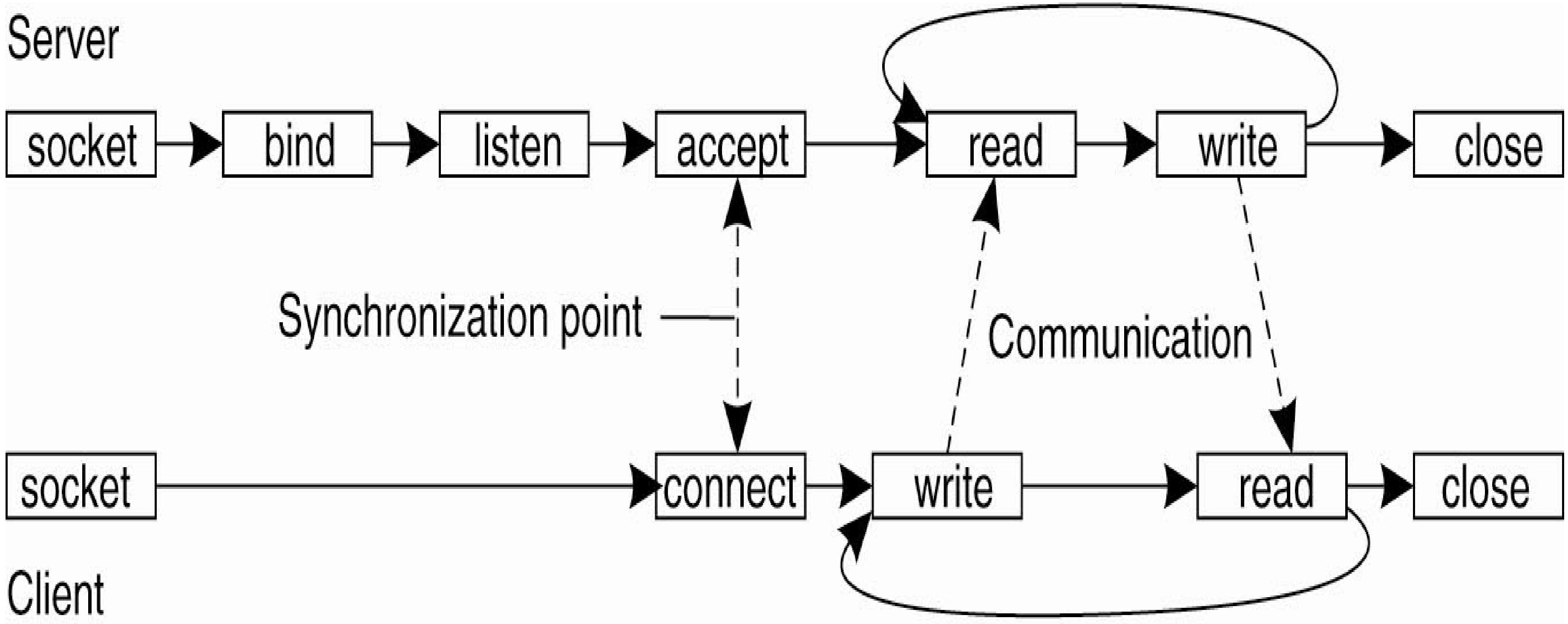
- **Remote procedure call** (RPC)
  - Transparency of all the message passing that realizes the caller-callee interaction at the application level
- **Remote (object) method invocation** (RMI)
  - As above, except leveraging interfaces
- **Middleware-mediated message passing**
  - Language-specific (e.g., event-based, reactive)
  - Internet-based (over HTTP, pull or push)

Analogies …


Primordial programming
Explicit use of sockets
Sockets are essential for all communications to reach to the network
But they are so raw and basic that their use should be made transparent to the application …
Structured programming
RPC
OOP
RMI
More advanced paradigms

The negation of abstraction

Socket-based communication has nearly no prescribed syntax or semantics, which are left to sender and receiver at the application level



Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

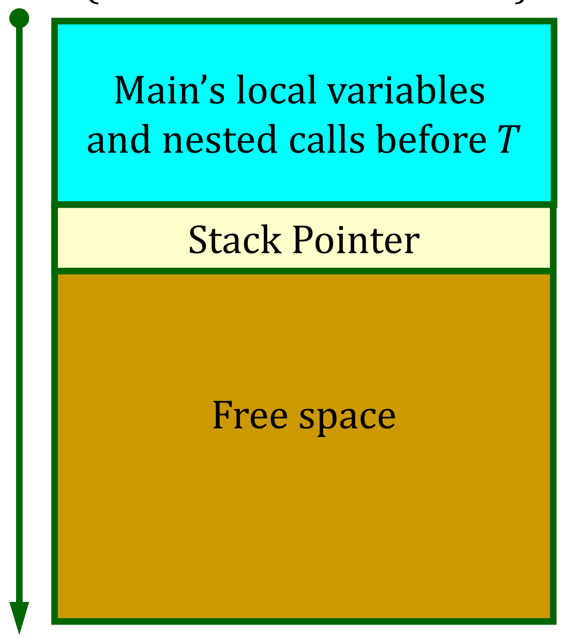
Anatomy of RPC – 1

- RPC allows a caller (a process) on one node to invoke **locally** a procedure in an address space owned by a **remote** process
  - Transparent networking kicks in necessarily
  - Caller and callee should not know of what happens under the hood of the call
- As in normal procedure calls, the caller "*stays on the call*" until the called procedure returns
  - The caller is suspended throughout
  - The `in` parameters travel from caller to callee
  - The call executes at the callee side, and returns
  - The `out` parameters travel back to the caller

Γνωθι σεαυτον (Know thyself)

- That's how a local procedure call works …

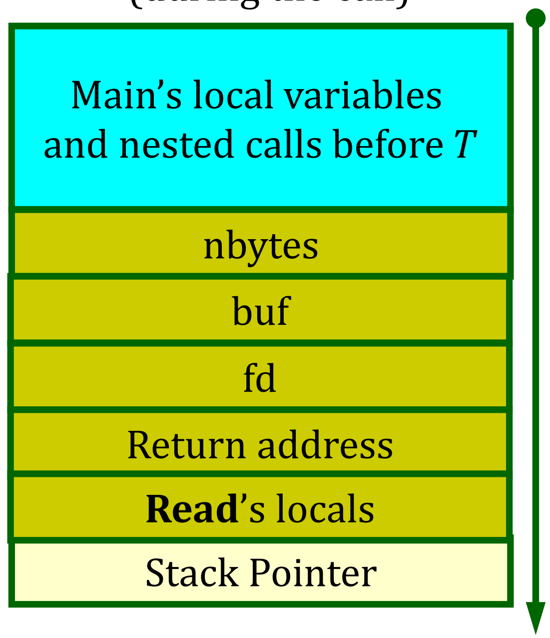
Stack of the caller
(at time *T* before the call)



Read(fd,buf,nbytes)

The C language places params on the stack in reverse order …
Every language has its own call conventions
(e.g., **cdecl**)

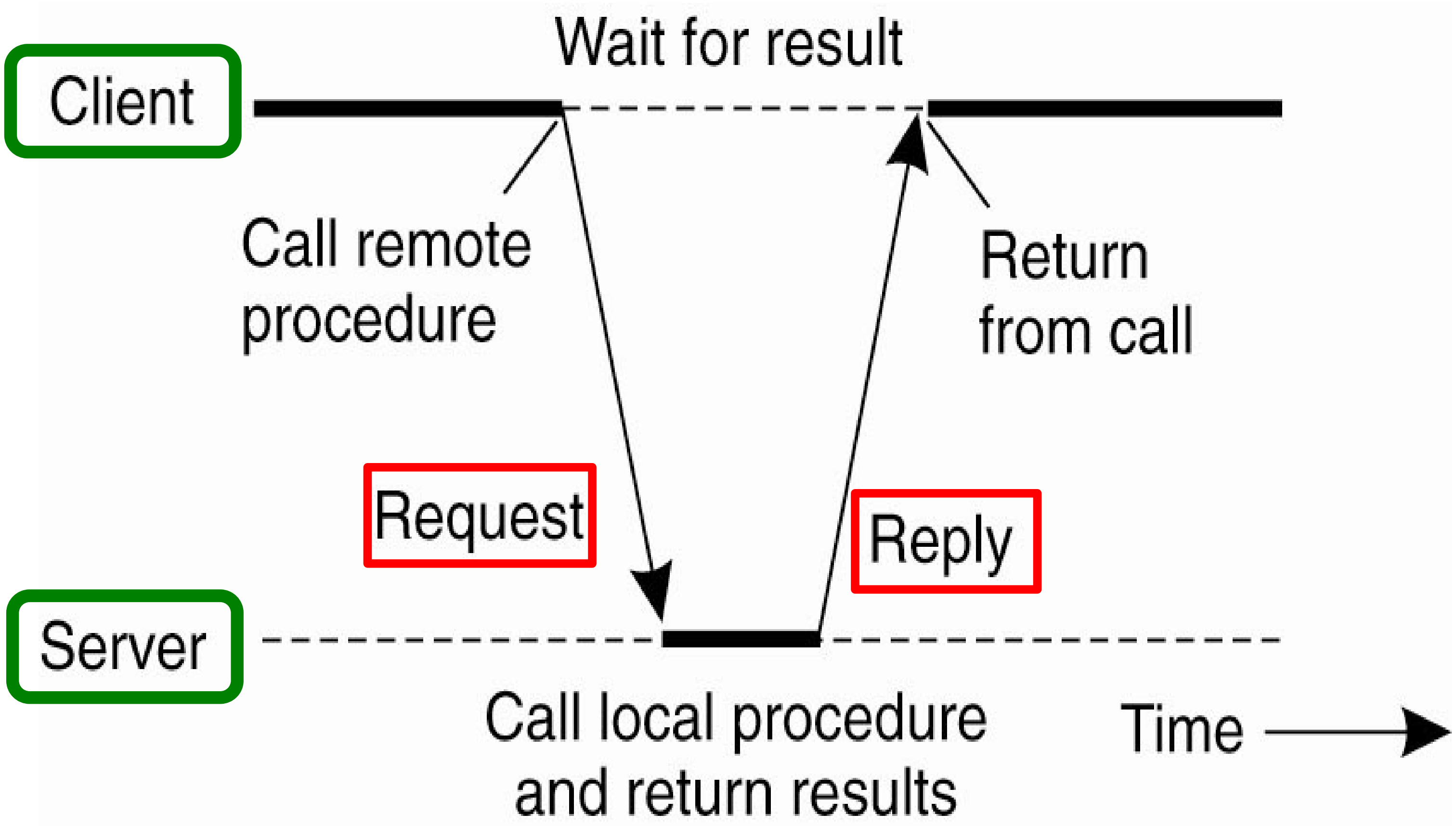
Stack of the caller
(during the call)



Anatomy of RPC – 2

- The call parameters may be either ***by-value***
  - They are copied on the stack of the callee
- Or ***by-reference***
  - They are addresses that point back to the caller's address space
  - Every update to them should be reflected back immediately at the caller's end
- Or ***by-value-result***
  - Only the latest updates propagate back at the call return

Anatomy of RPC – 3

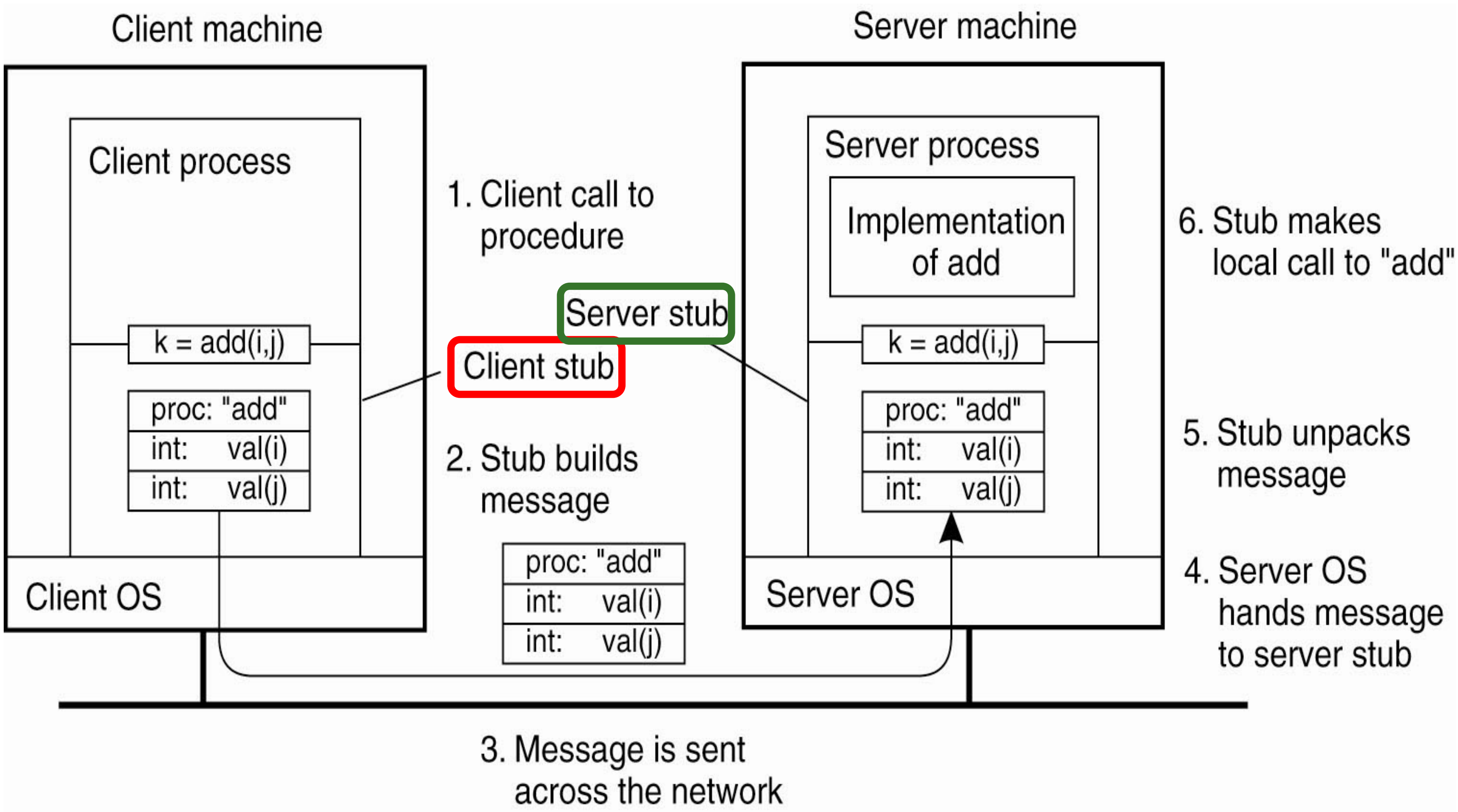


Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Anatomy of RPC – 4

- At the caller's side, remote calls appear local
  - The call is "posted" on the caller's stack according to local conventions
  - The ***client stub*** creates the corresponding call descriptor and forwards it across the network, using a mechanism called *parameter marshalling*
- At the callee's side, the arrival of the remote call activates a local "caller"
  - The ***server stub*** transforms the call descriptor into a call on the local stack, awaits the return and sends it back across the network
  - On call arrival, this uses the reverse mechanism, called *parameter unmarshalling*

Anatomy of RPC – 5



Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Anatomy of RPC – 6

- The RPC mechanics involves several important decisions
  - On the format of messages between stubs
  - On the encoding of the data exchanged by caller and callee
  - On the network protocol to use for such messages (TCP, UPD)
  - On how the client stub can locate the server stub
- The latter is difficult to address transparently
  - The server side **must register itself** (`IP address : port`) at a given registry as a "provider" of the target procedure
    - Registering what? The "procedure" is strictly a server-side concept ….
  - The client side must retrieve "that" information and establish a (TCP) connection to the corresponding network location
    - But then the server side should listen at all times for incoming calls and also permanently seize the target port: not very nice …

Anatomy of RPC – 7

- The RPC is intrinsically **synchronous**
  - It can be asynchronous *only* for calls *without* return parameters
    - The caller might proceed as soon as the call has been issued
    - Without knowing whether the call actually succeeded …
- The eventuality of network errors requires adding optional mechanisms to either stubs
  1. The client side may retry requests that did not return
  2. If it did so, the server side would have to recognize and filter out call duplicates
  3. The server side should also retransmit results in case the client did not ack them

Anatomy of RPC – 8

- Such provisions yield diverse **request-reply protocol semantics**
  - Best effort, with no safeguard mechanism
    - No guarantee on call execution and effects
  - At least once, with just request-retry at client side
    - Retry until success, without knowing how many executions at server side
  - At most once, with all mechanisms in use
    - Failure only if server is unreachable
  - Exactly once, when all guarantees are in place
    - Including hot-redundant server

Language-neutral RPC

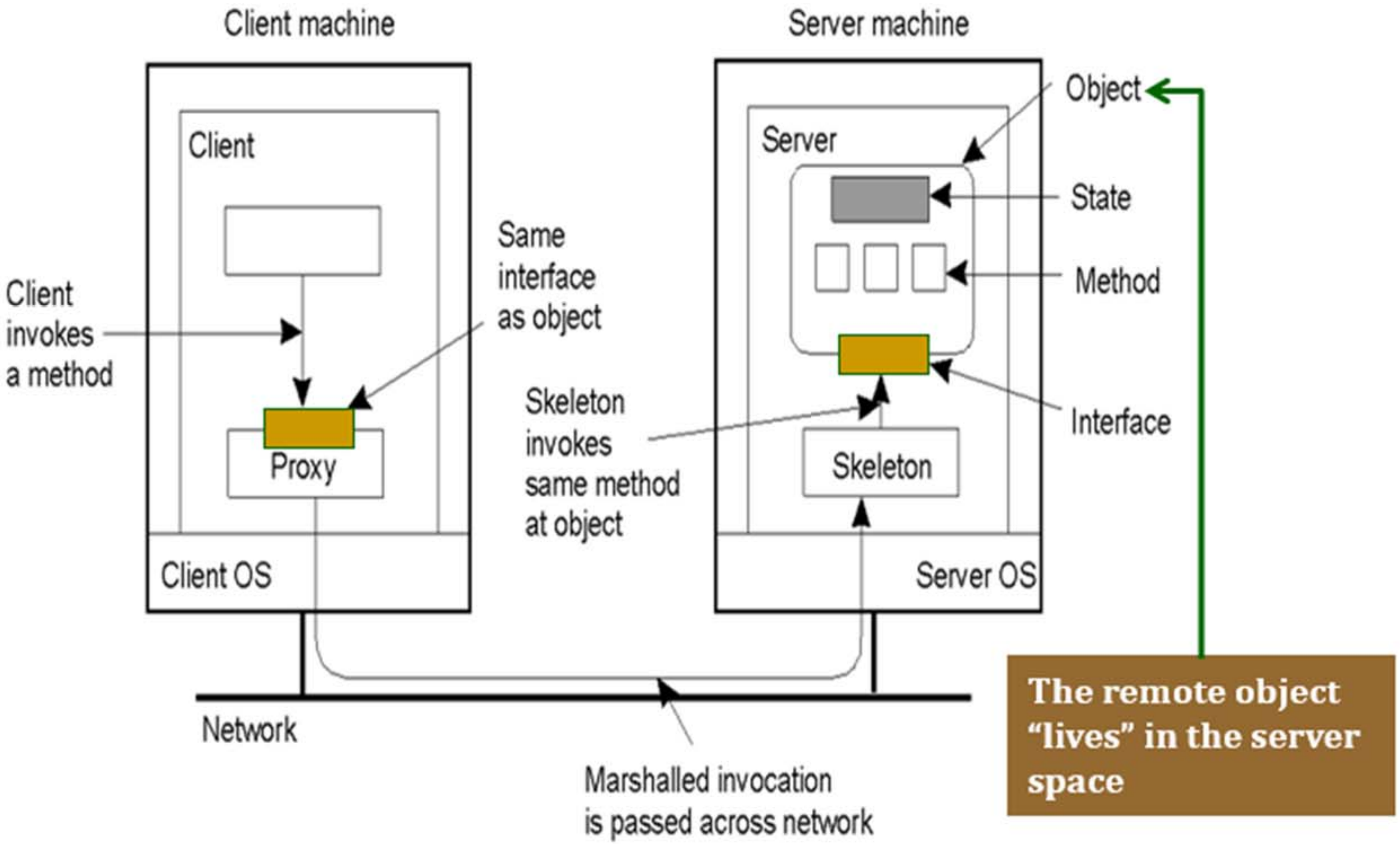
- All “historic” RPC support based on TCP
  - Which was rather limiting: HTTP not understood as a programming interface back then
- And was language-specific
  - Short-sighted: the immediate need was for individual languages to support distributed programming
- Then came **interoperability**
  - **CORBA**: Common Object Request Broker Architecture, better in concept than in practice …
    - https://corba.org/faq.htm
- Finally, RPC was lifted to **HTTP/2.0**
  - **gRPC**: check it out at https://grpc.io/

Differential anatomy of RMI – 1

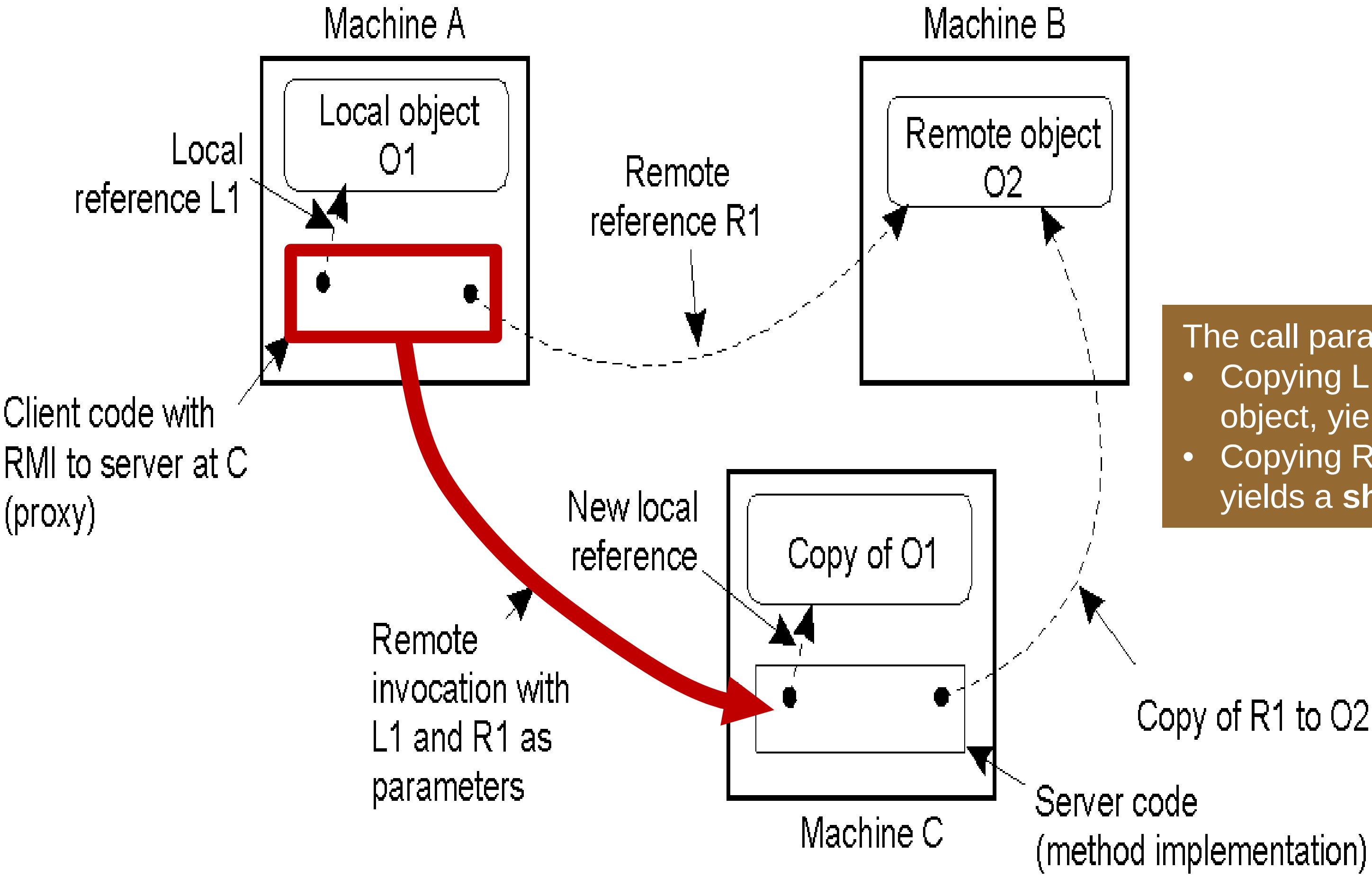
- The LSP* separation between (service) interface and object (implementation) is naturally conducive to distribution
  - The interface is a lightweight entity that can be exposed remotely easily and naturally
  - Objects live (long) in the heap: their scope is global
  - These traits earn RMI more transparency than RPC
    - So much so that RMI interaction can be enabled *at run time* by wrapping "object-lookalike" over non-object resources (CORBA)
- The client stub becomes the ***proxy***
  - Which can be loaded in *dynamically* when the client **binds** with the target implementation
    - Binding is generally **explicit**, hence not transparent
- The server side becomes the ***skeleton***
  - Compile-time provision, derived from the remote interface

*: **L**iskov **S**ubstitution **P**rinciple

Differential anatomy of RMI – 2



Differential anatomy of RMI – 3

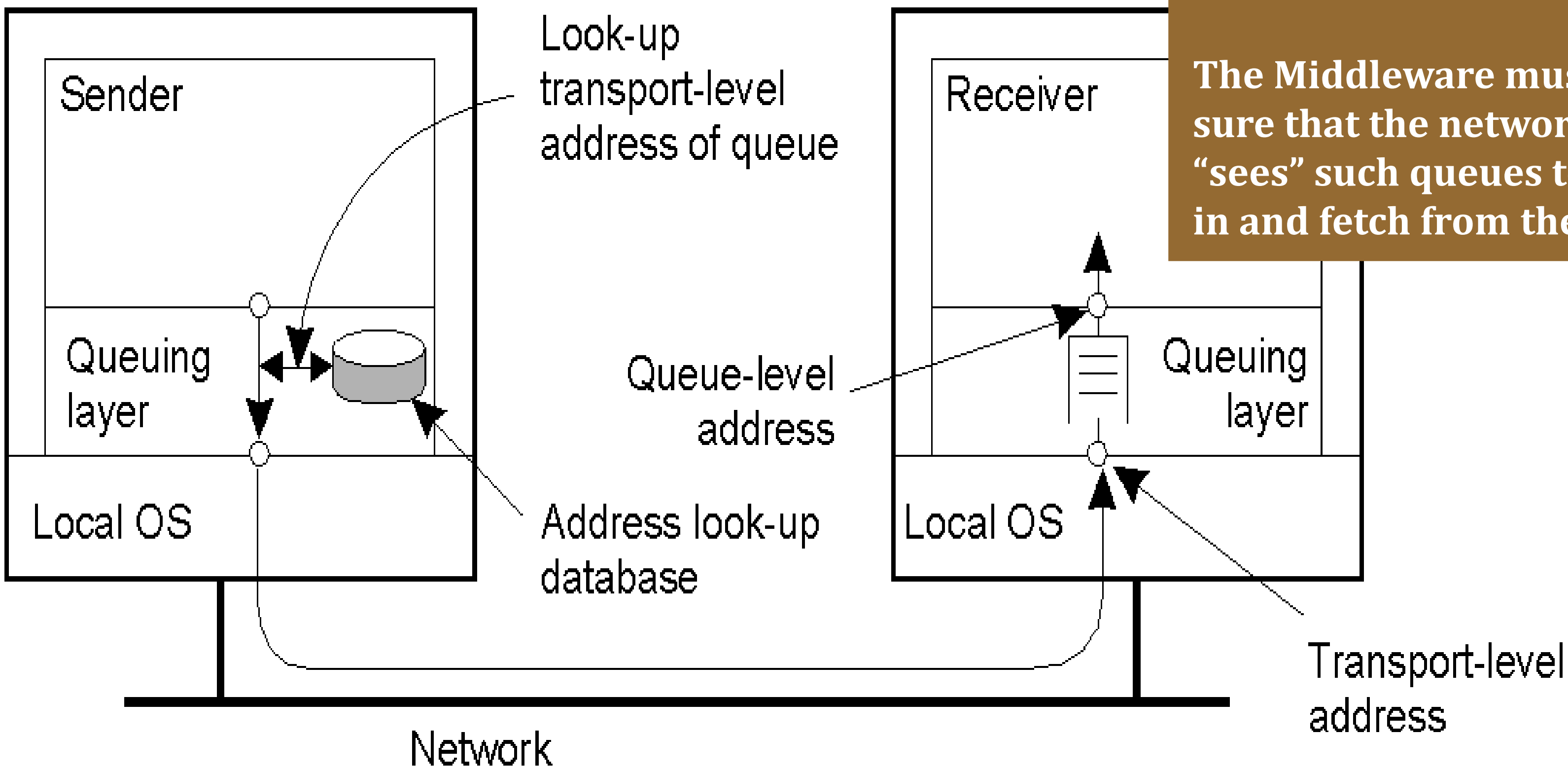


The call parameters (A to C) are **by-value**:
- Copying L1, which refers to a local object, yields a **deep copy** of O1
- Copying R1, which is a remote reference, yields a **shallow copy** of O2

Middleware-based message-passing – 1

- Applications can communicate by placing messages in Middleware-supported queues
- Very easy to realize
  - Distinct queues at either side (or along the way), depending on the desired support for **persistency**
  - With blocking events contingent on synchronization behaviour
- **Send** maps to a non-blocking `Put`
  - Becomes blocking if MW wants to prevent overwrites on full queue
  - The send queue handler acts as a proxy
- **Receive** maps to a blocking (guarded) `Get`
  - A **callback** mechanism should be provided to decouple the receiver from the queue
  - The receive queue handler acts as a skeleton

Middleware-based message-passing – 2

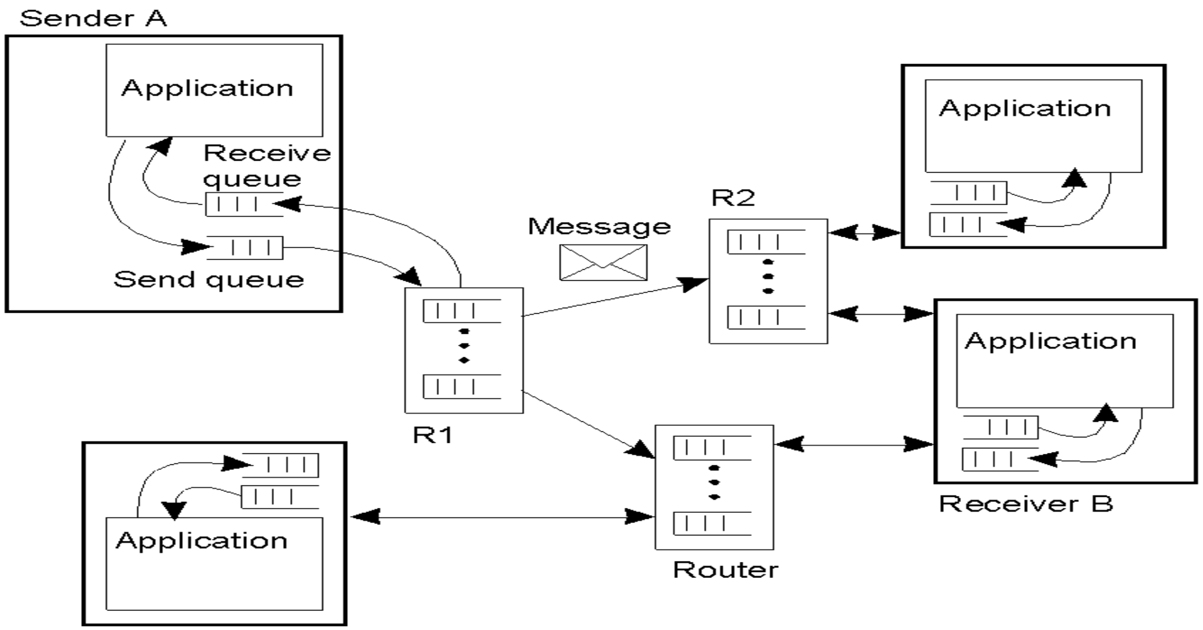


Each queue appears as local to sender and receiver

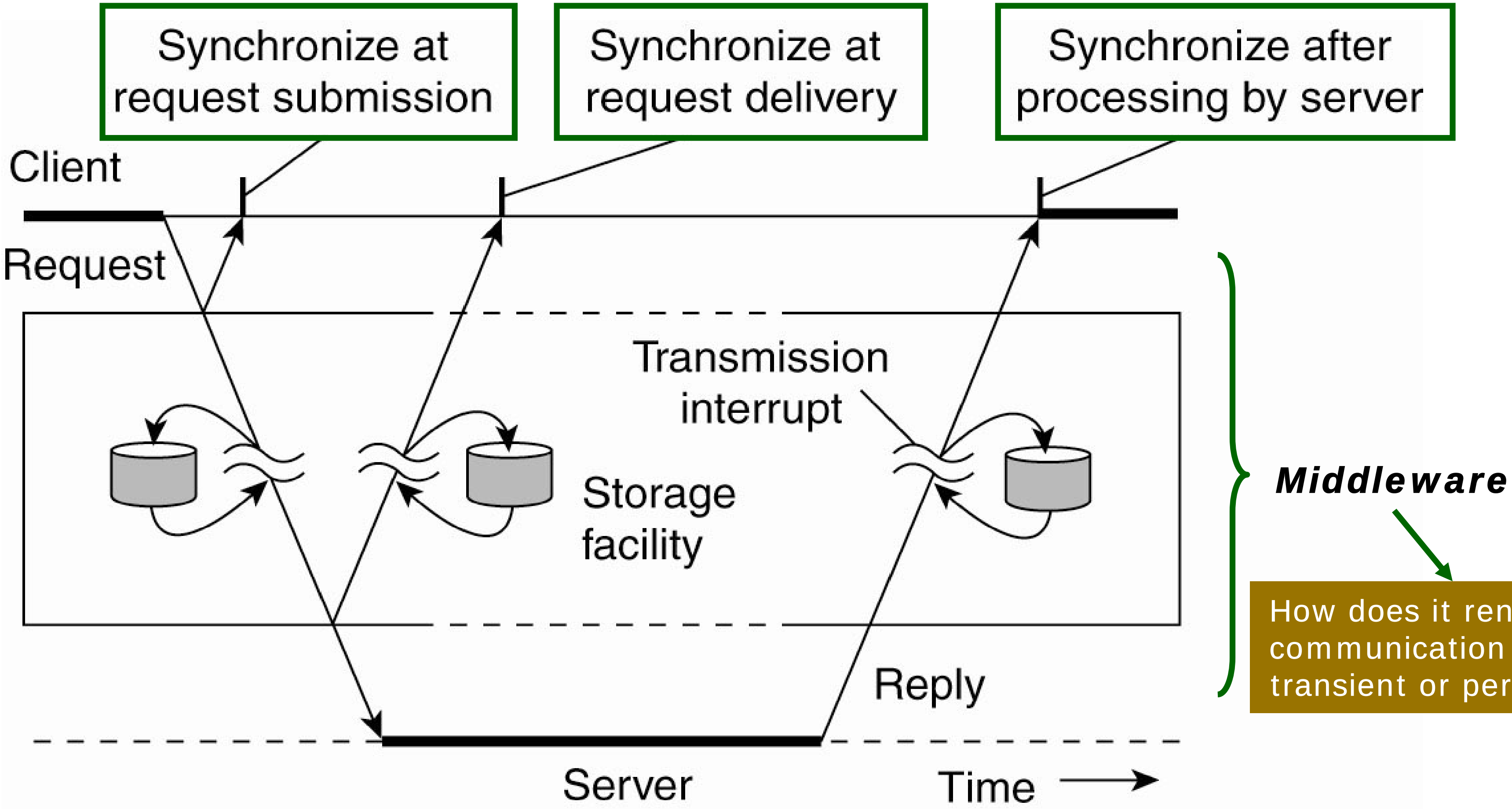
The Middleware must make sure that the network layer "sees" such queues to deposit in and fetch from them

Middleware-based message-passing – 3

- The Middleware overlays its own network over the underlying internet (lowercase ‘I’)
  - With its own static or dynamic topology and routing
- A ***broker*** acts at all points in which the overlay network traffic needs to become internet traffic
  - Similar in nature to the **gateway** nodes of the classic Internet



Middleware-based message-passing – 4

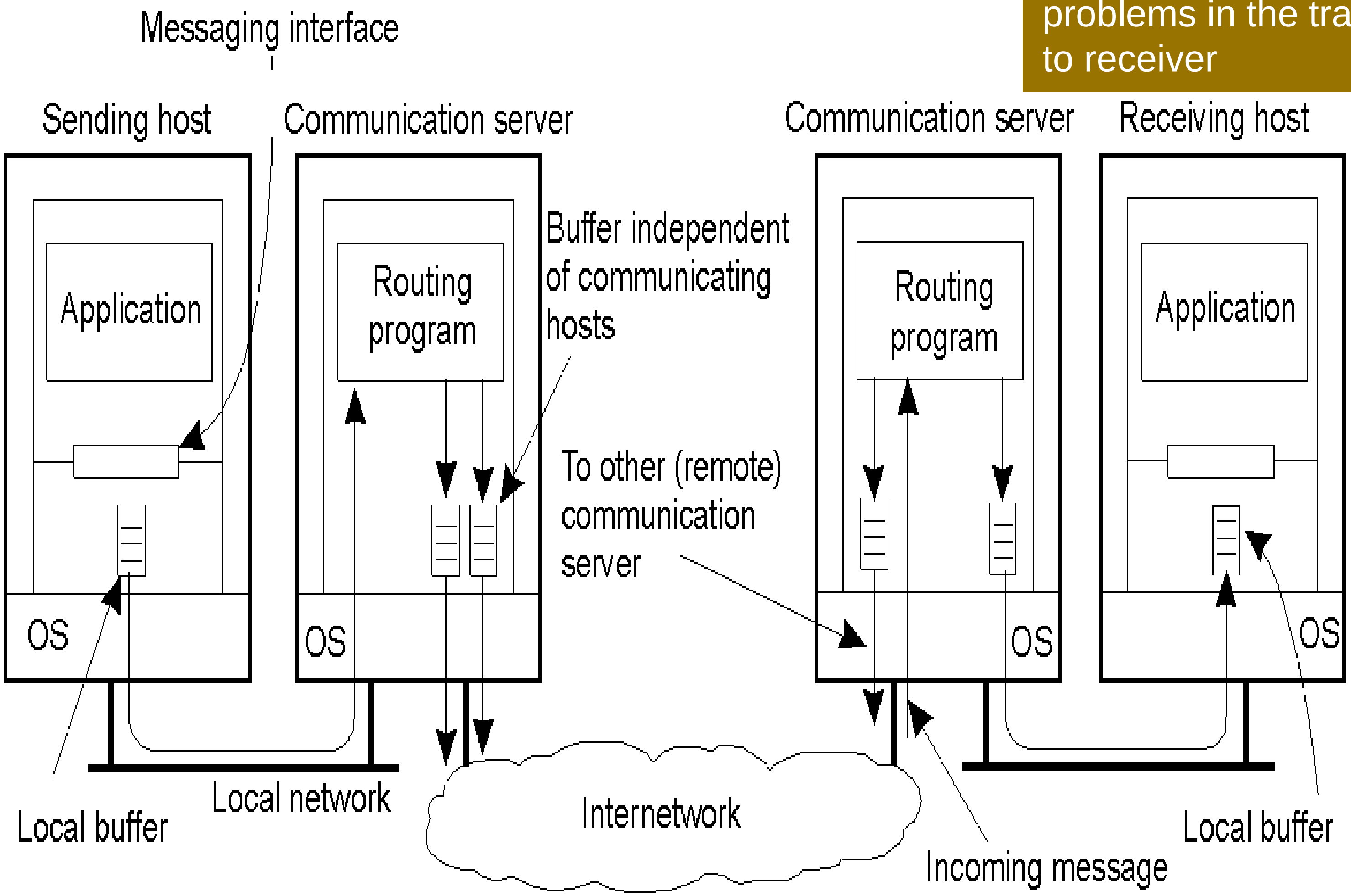


How does it render communication transient or persistent?

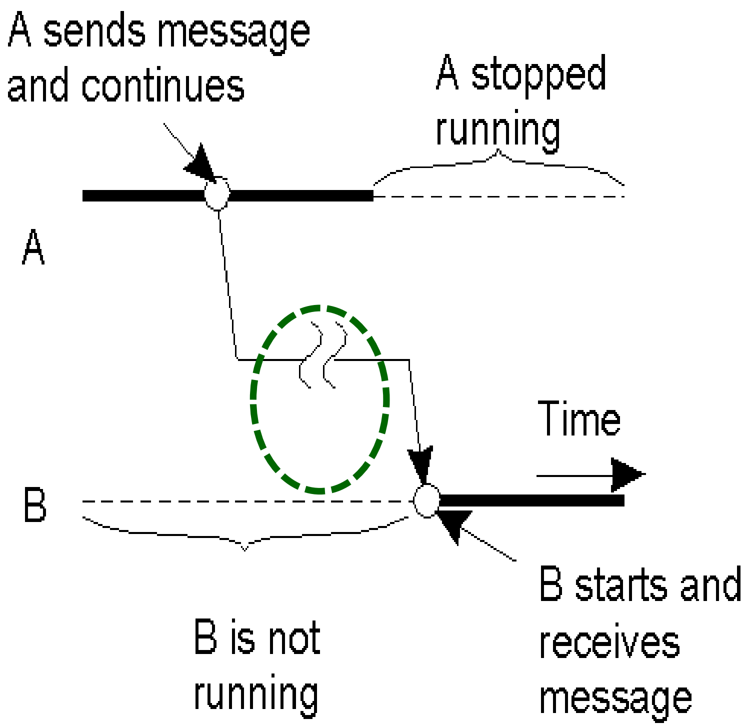
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Middleware-based message-passing – 5

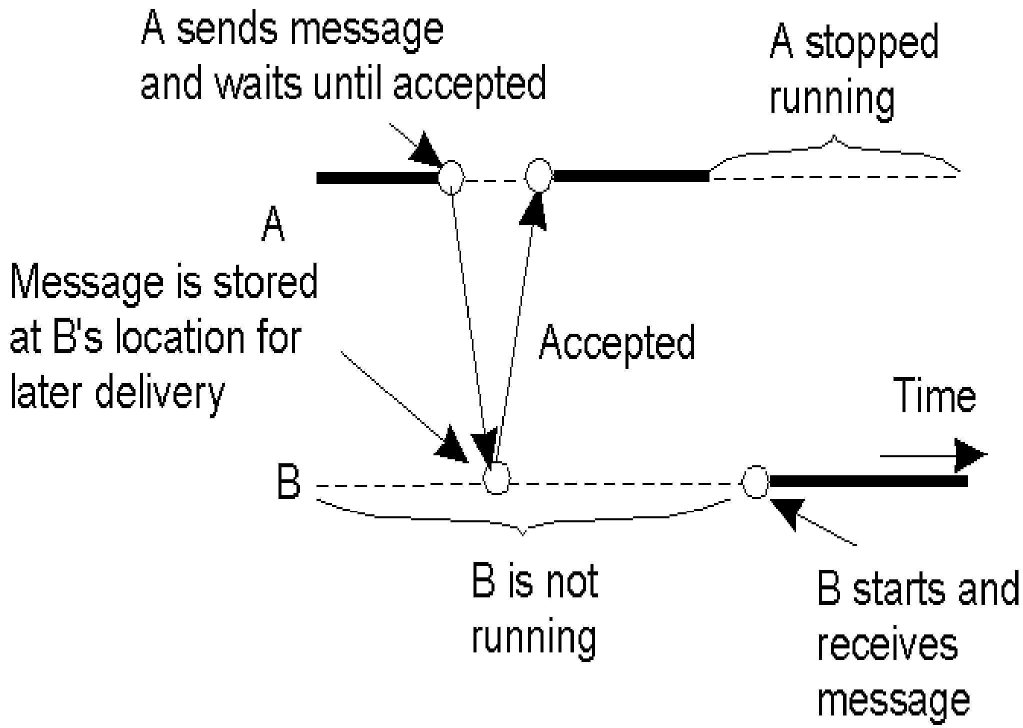
Distributed message passing incurs persistency and synchronization problems in the transit from sender to receiver



Middleware-based message-passing – 6



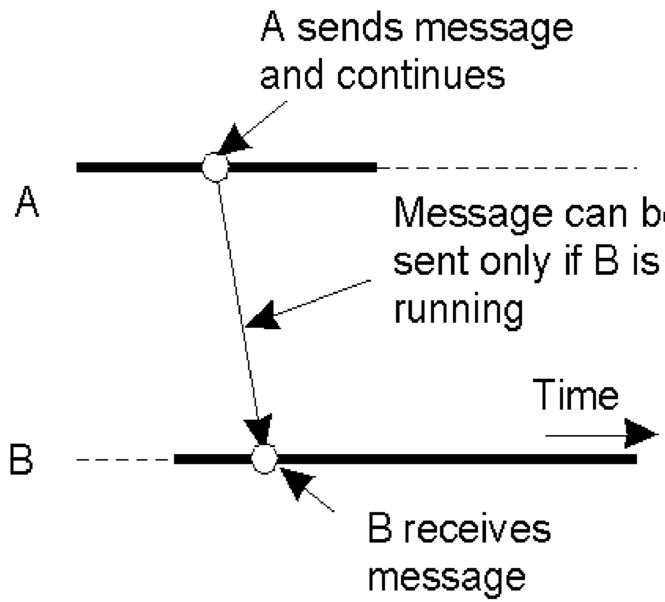
Asynchronous, persistent (?)



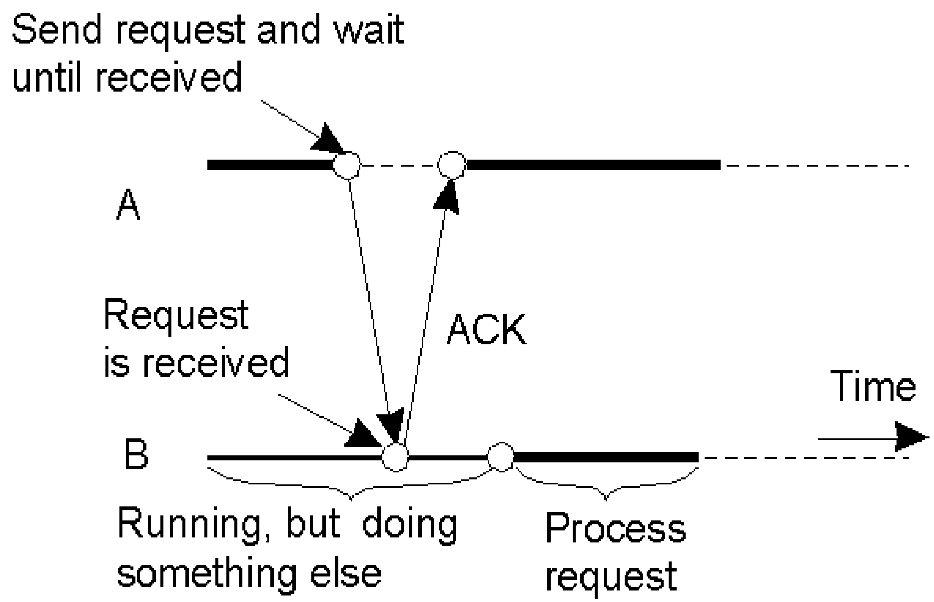
Persistent, synchronous (?)

Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Middleware-based message-passing – 7



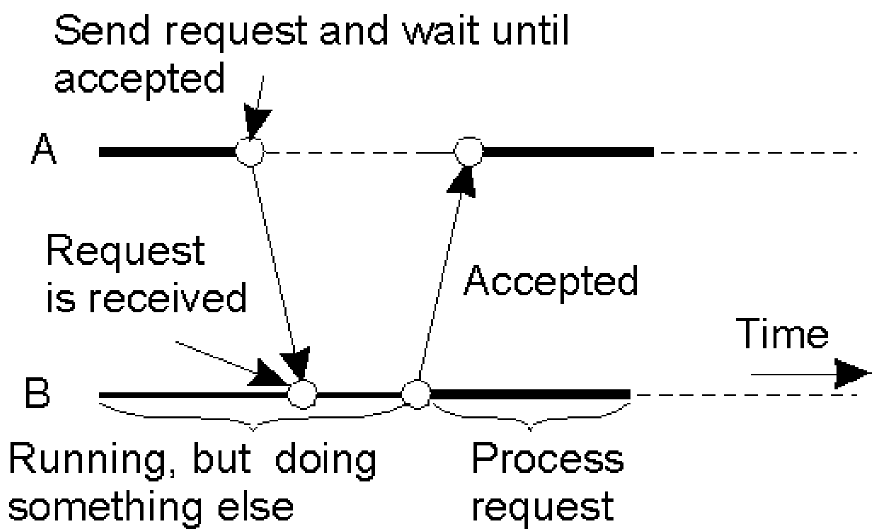
Asynchronous, persistent (?)

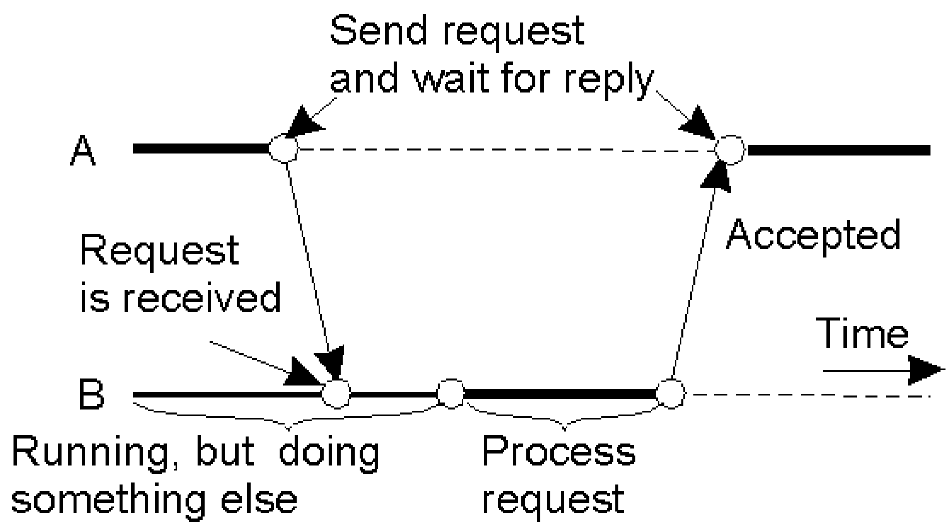


Persistent, synchronous (?)

Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Middleware-based message-passing – 8





Synchronous, persistent (?)

Synchronous, persistent

Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

What is happening to the Internet?

- With **HTTP/1.1** (textual) when a web browser loads a web page, it can requests one resource at a time per TCP connection to the server
  - The original Web assumed few heavy-weight connections, all pull based
  - The Web of today features a zillion of light-weight connections, also in push mode
- **WebSocket** allows full-duplex communication, making “the HTTP/1.1 layer” a two-way road
- **HTTP/2** (binary) **multiplexes** multiple requests over a single connection to the same server, to allow receiving multiple responses at once
  - TCP does not know about it, which causes needless retransmissions …
- HTTP/2 also allows the server to **push** contents into the client proactively, without it requesting so (aka Server-Sent Events)
- **QUIC** replaces TCP with
  - Default authentication and encryption, plus faster handshake
  - Direct support for multiplexed transport streams delivered independently (resend on packet loss becomes specific)
  - Use of UDP, *in user space*, with far less execution overhead
- **HTTP/3** is HTTP/2 adapted to QUIC

Variants of middleware (repeat)

- Distributed file system
  - UNIX-like NFS
- Remote procedure call (RPC)
- Distributed objects (RMI)
- Distributed documents: Web 1.0
  - All TCP based
- Distributed everything: Web 2.0 (all over HTTP)
  - Resource-centric: REST
  - Data-centric: GraphQL
  - Collaboration-centric: gRPC
  - Stream-oriented: WebRTC

Past

Present & Future