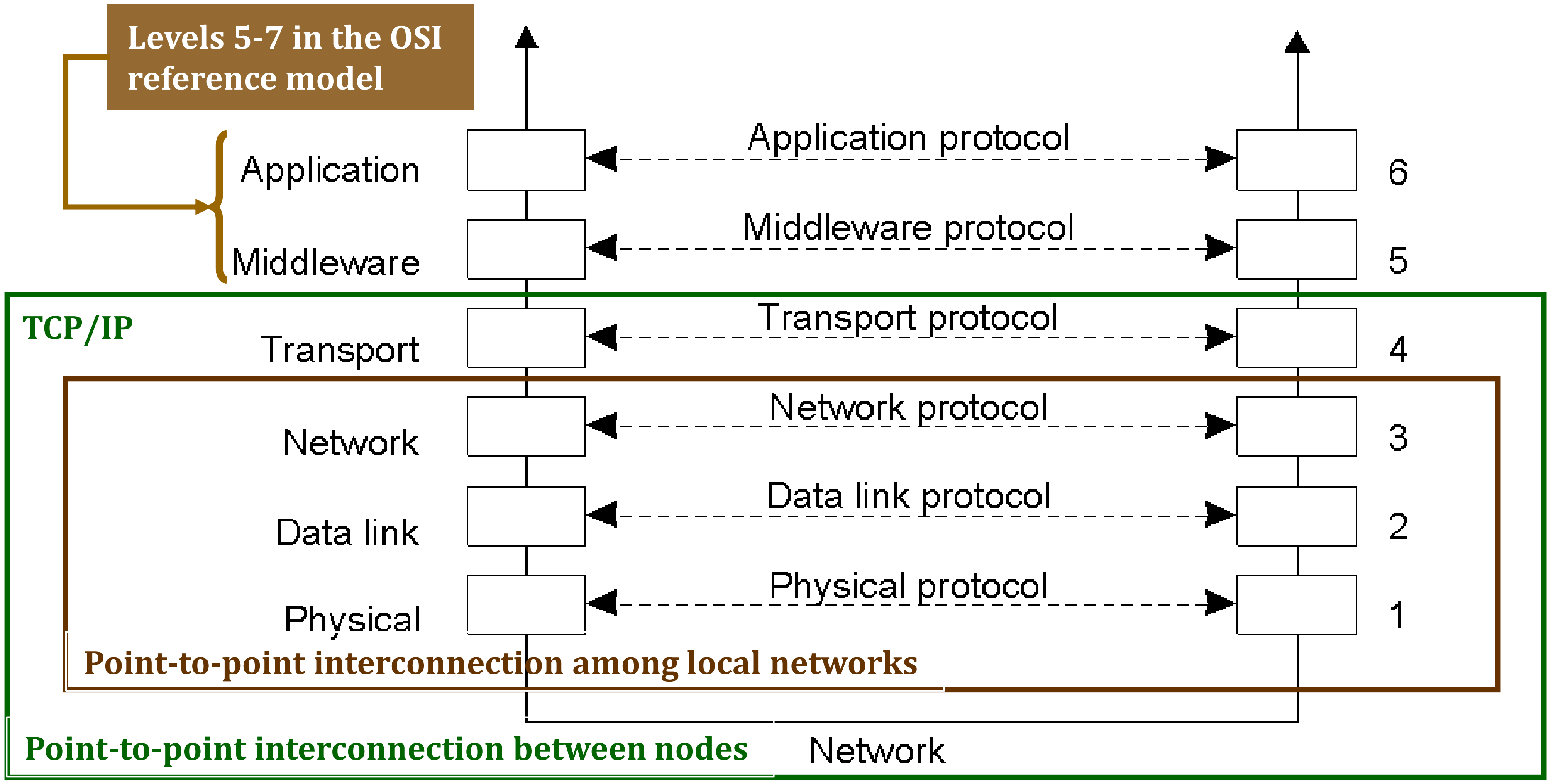
Distributed communications

Runtimes for concurrency and distribution

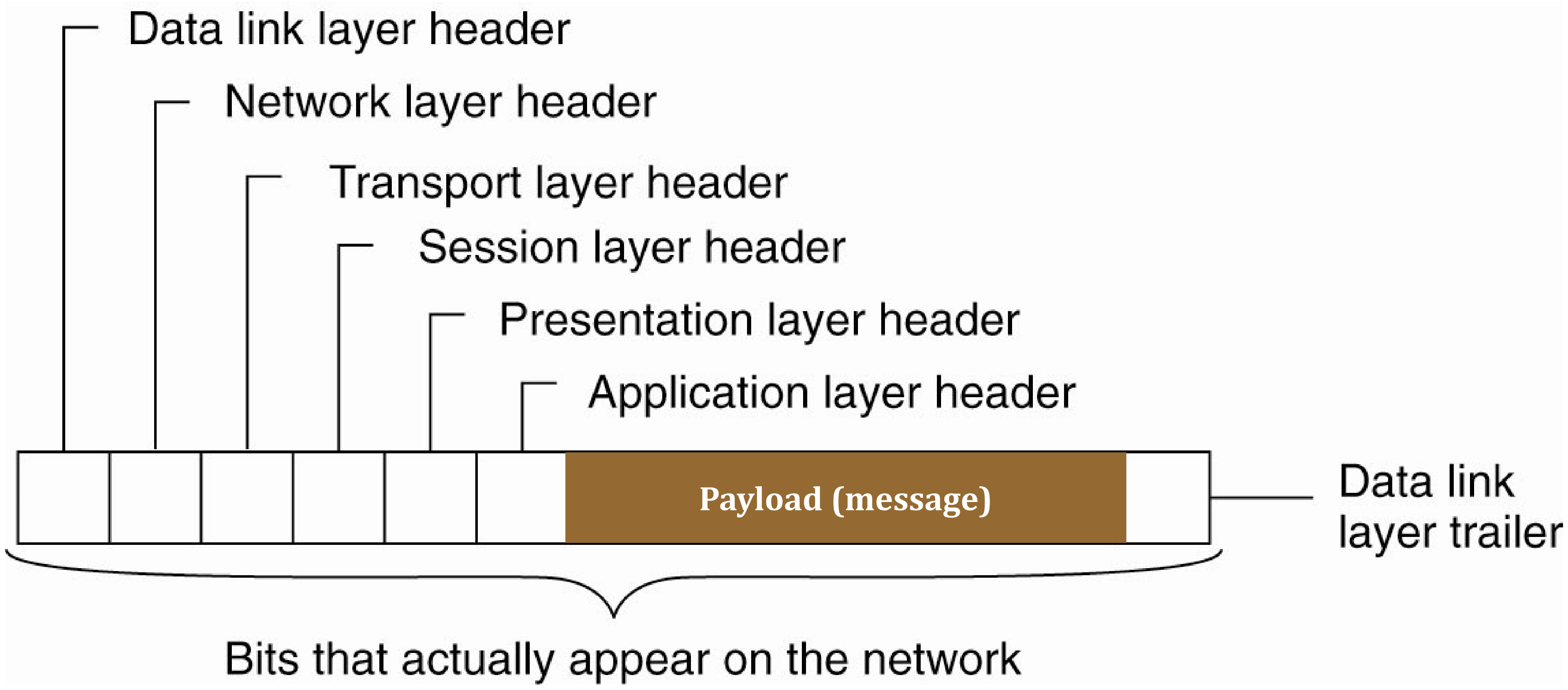
Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

A layered view of networked communication – 1



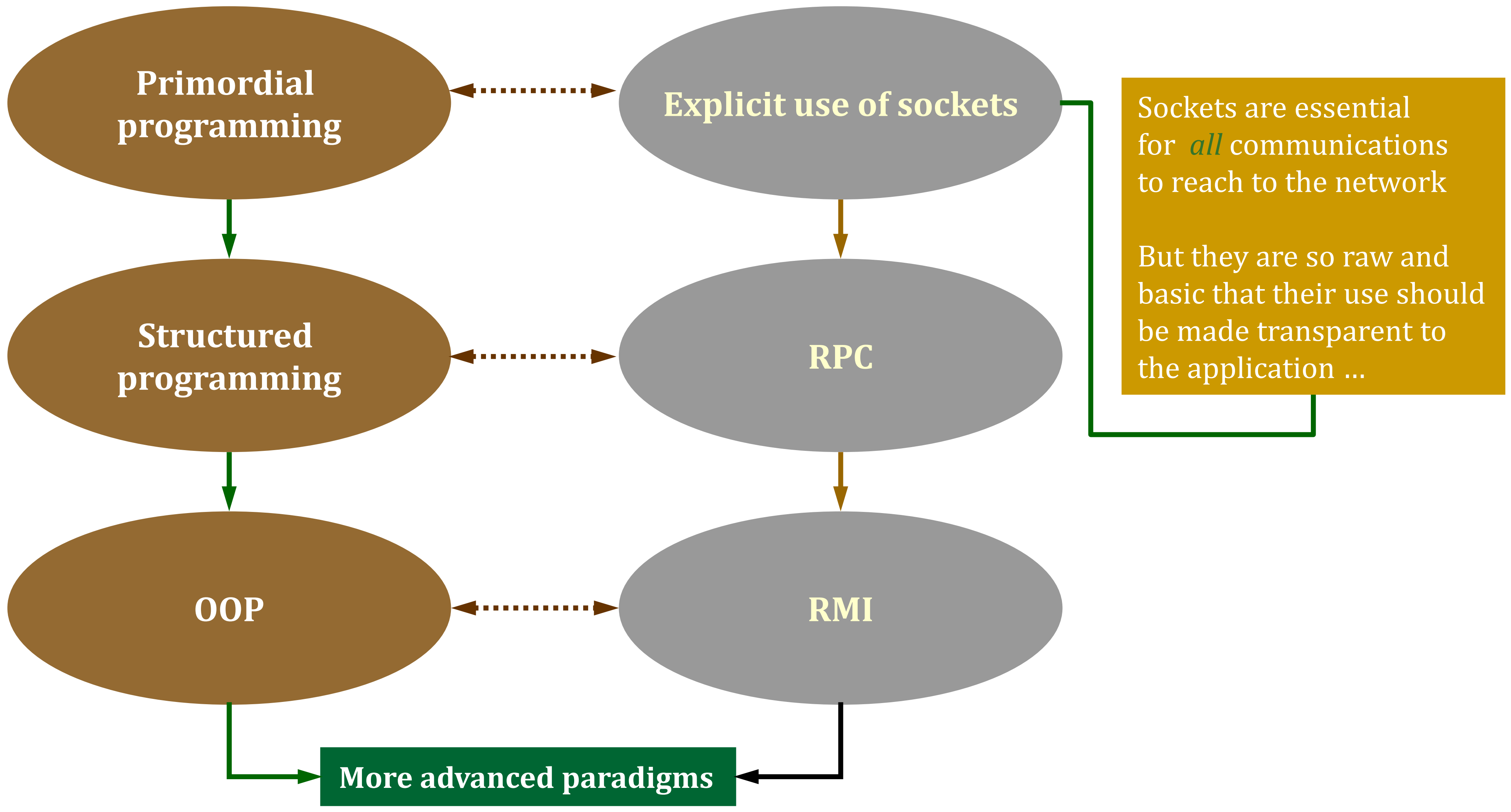
A layered view of networked communication – 2



Models of distributed communication

- **Remote procedure call** (RPC)
  - Transparency of all coordination-related 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 …

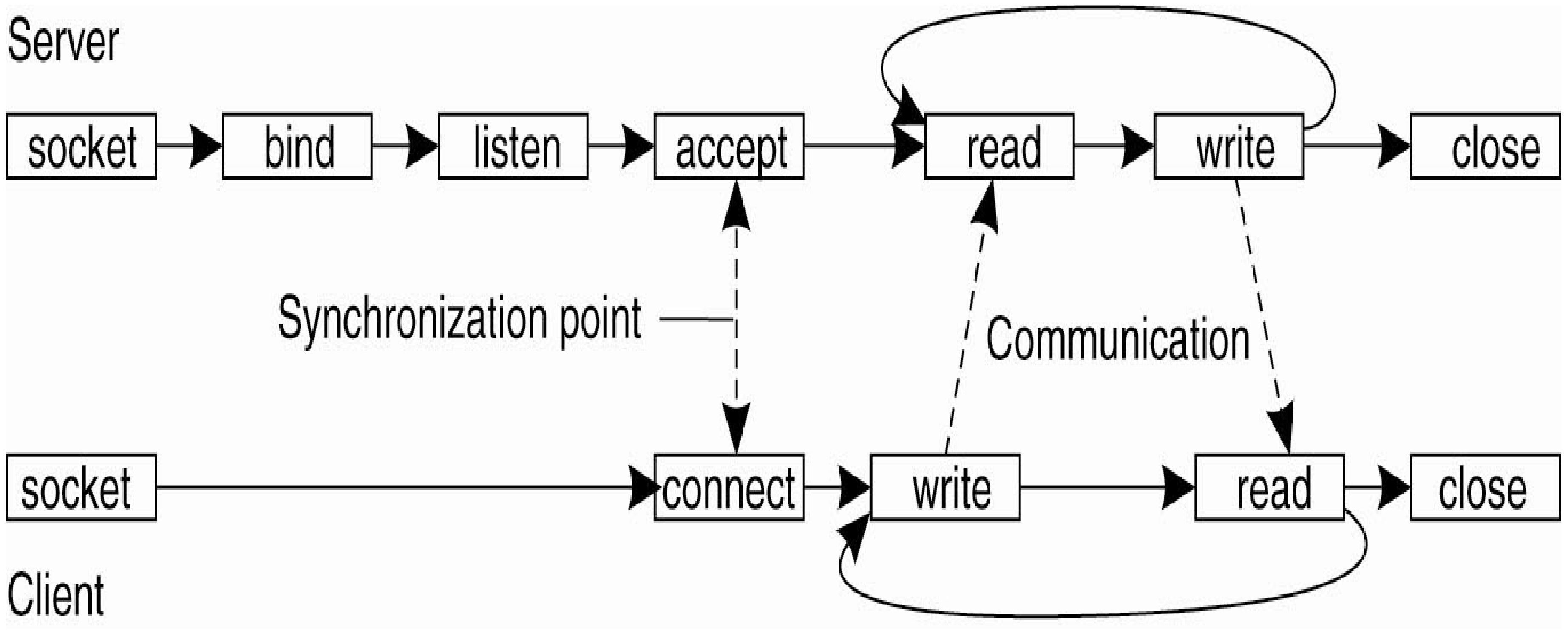


Sockets are essential for *all* communications to reach to the network

But they are so raw and basic that their use should be made transparent to the application …

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



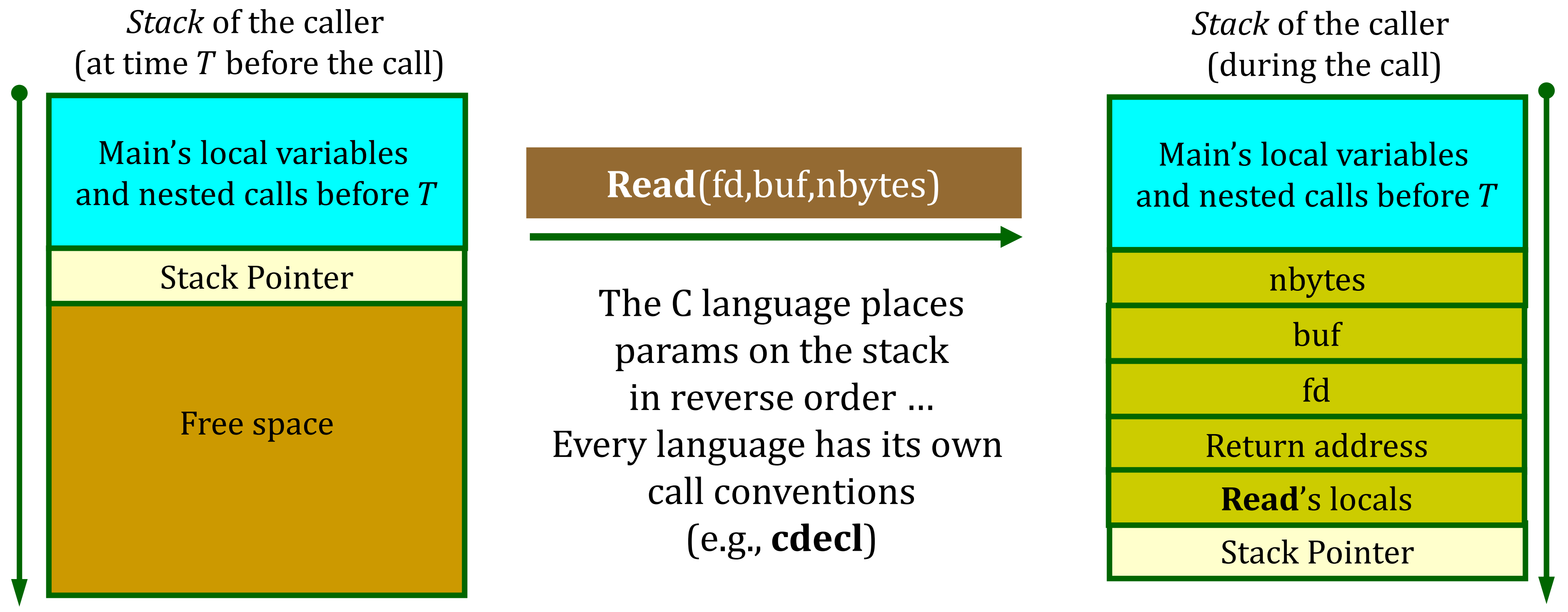
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Anatomy of RPC – 1

- RPC allows a caller (process) on one node to invoke **locally** a procedure in an address space owned by a **remote** callee (process)
  - Transparent networking kicks in necessarily
  - Caller and callee should *not* know what happens under the hood of the call
- As in normal procedure calls, the caller "*stays on the call*" until the callee 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 …

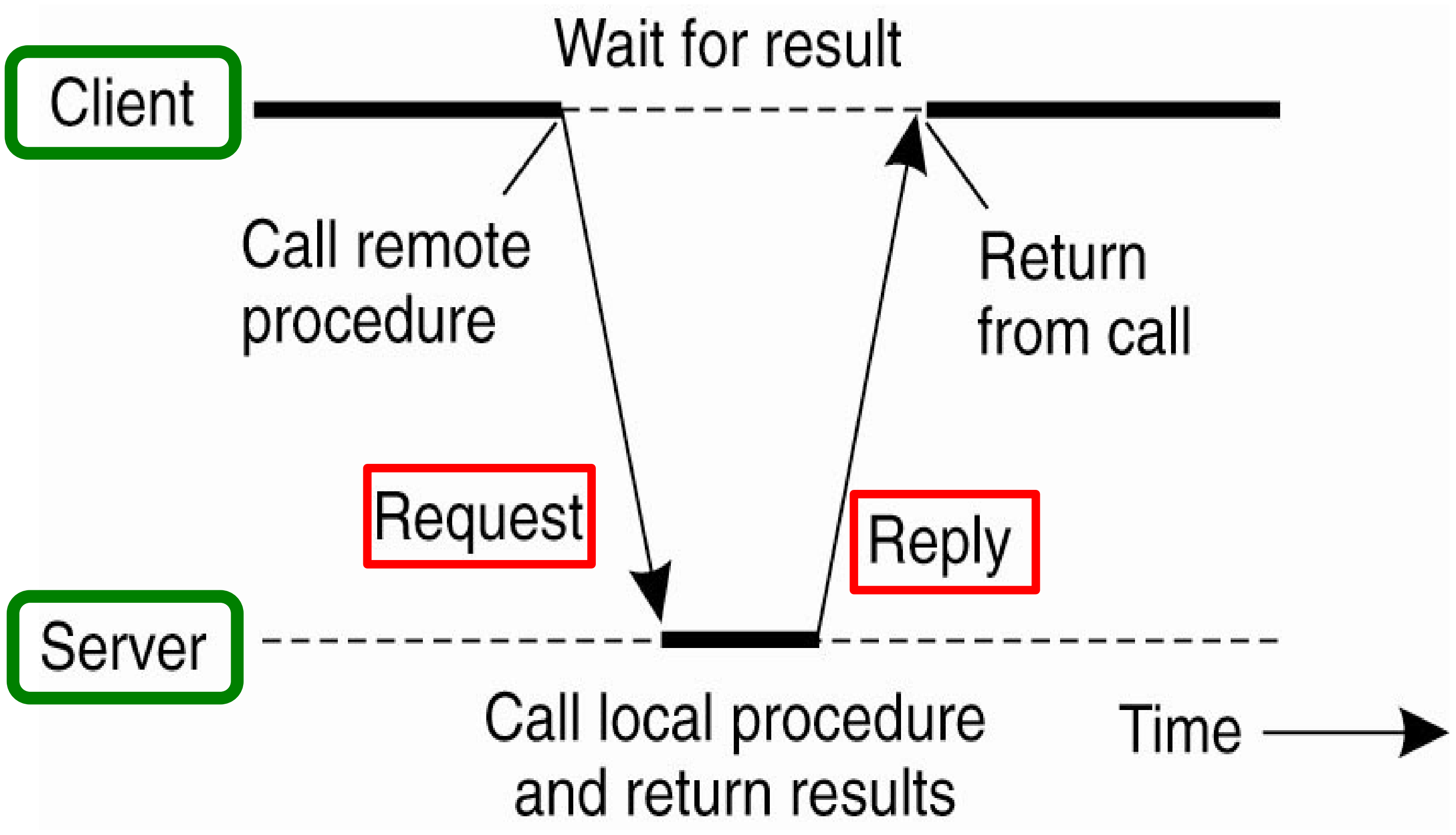


The C language places params on the stack in reverse order …
Every language has its own call conventions
(e.g., **cdecl**)

Modes of call parameters

- ***By-value***
  - Copied on the stack of the callee
- ***By-reference***
  - Locations in the caller's address space
  - Every write to them should be reflected back immediately at the caller's end
- ***By-value-result***
  - Only the latest updates propagate back to the caller, at the return of the call

Anatomy of RPC – 2

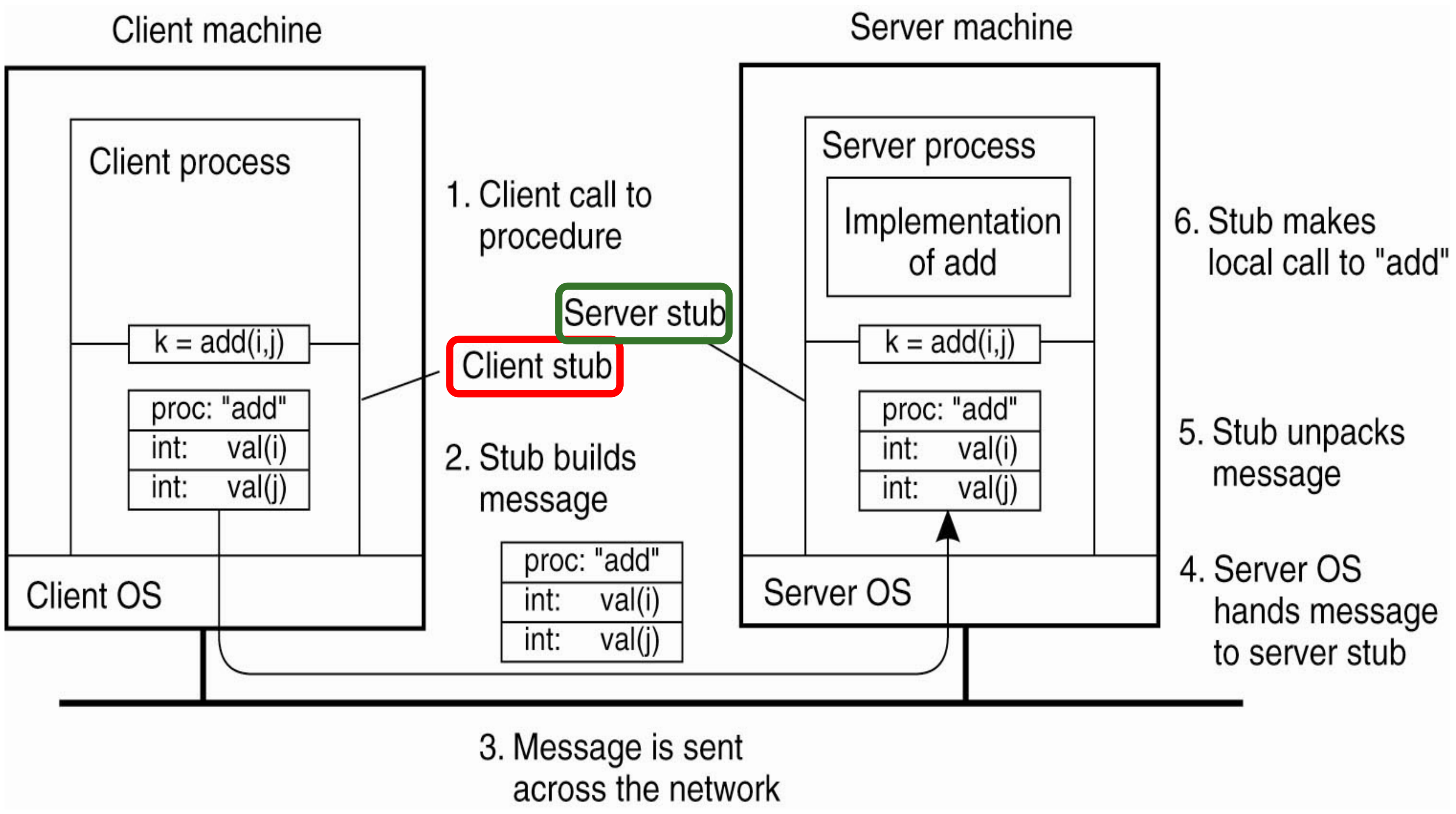


Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Anatomy of RPC – 3

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

Anatomy of RPC – 4



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

Anatomy of RPC – 5

- 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 problem is difficult to address transparently
  - Server side **must register itself** (`IP address : port`) as a "provider" of target procedure
    - Registering what? The "procedure" is strictly a server-side concept ….
  - Client side must retrieve that registry entry and establish a (TCP) connection to it
  - Server side should listen at all times for incoming calls and permanently seize the target port
    - Not very nice …

Anatomy of RPC – 6

- The RPC is intrinsically **synchronous**
  - Asynchronous *only* for calls *without* return parameters
    - Caller might proceed as soon as call has been issued
    - Without knowing whether the call actually succeeded …
- The eventuality of network errors requires adding optional capabilities to either stubs
  1. Client side may retry requests on missing returns
  2. In that case, server side should be able to detect and filter out call duplicates (*sliding window protocol*?)
  3. Server side should also retransmit results (without recomputing!) if client side did not ack them

Anatomy of RPC – 7

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

Cloud-fit retry management strategies

- Exponential back-off
  - https://dzone.com/articles/understanding-retry-pattern-with-exponential-back
  - https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- Circuit breaker
  - https://martinfowler.com/bliki/CircuitBreaker.html
- Backpressure
  - https://www.tedinski.com/2019/03/05/backpressure.html
- Throttling
  - https://docs.microsoft.com/en-us/azure/architecture/patterns/throttling
  - https://aws.amazon.com/premiumsupport/knowledge-center/dynamodb-table-throttled/

Language-neutral RPC

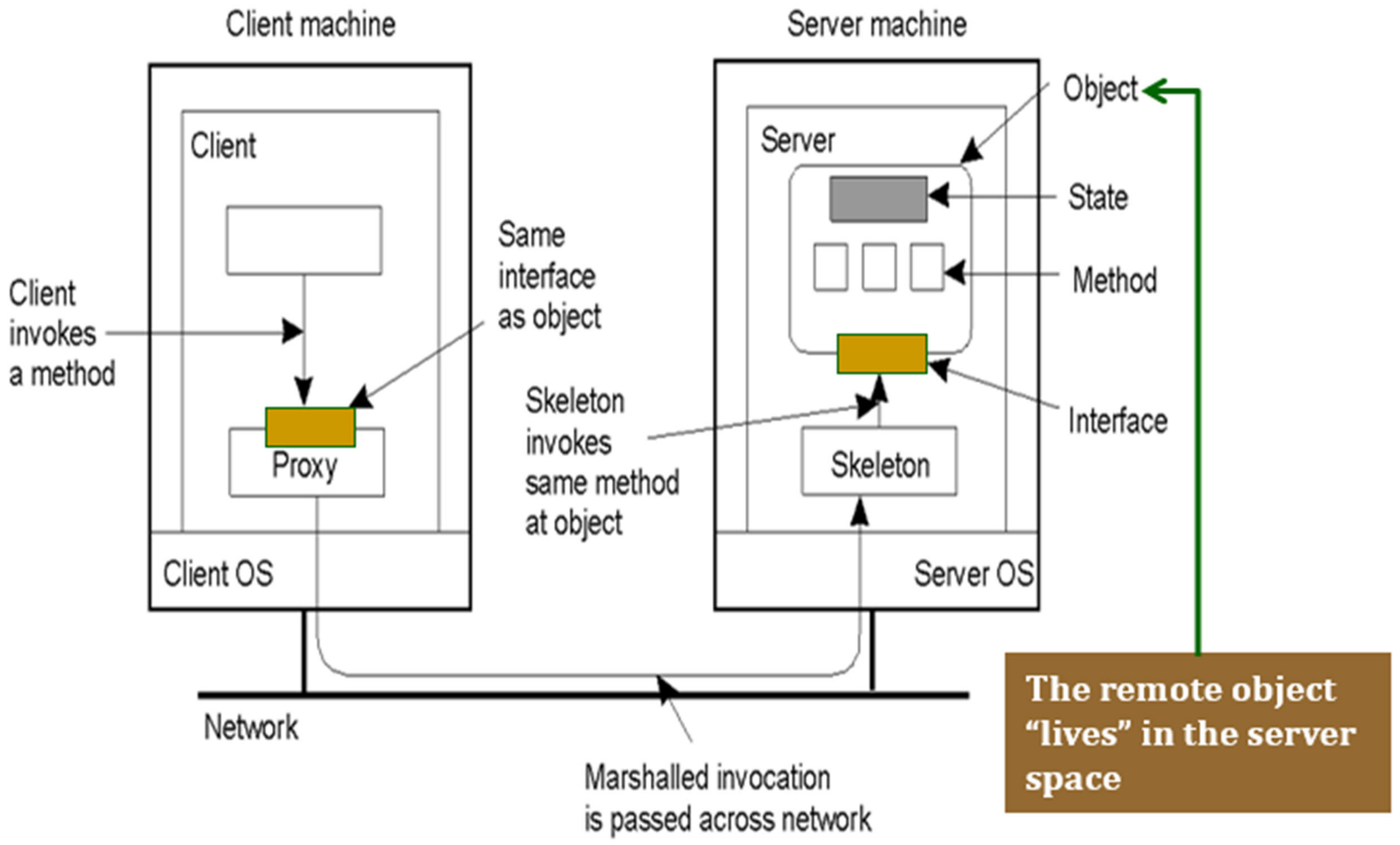
- All "historic" RPC support based on TCP
  - Seriously limiting: HTTP was not understood as a *programming interface*
- And was language-specific
  - Short-sighted: the immediate need was for individual languages to support distributed programming
- Then came **interoperability**
  - **CORBA**: Common 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) fits distribution very well
  - The interface is a lightweight entity that can be exposed remotely in a most natural way
  - 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)
- Server side becomes the ***skeleton***
  - Compile-time provision, derived from remote interface
- Client stub becomes the ***proxy***
  - Loaded *dynamically* (as an implementation artefact) when client **binds** with target server side explicitly
    - No transparency in that act

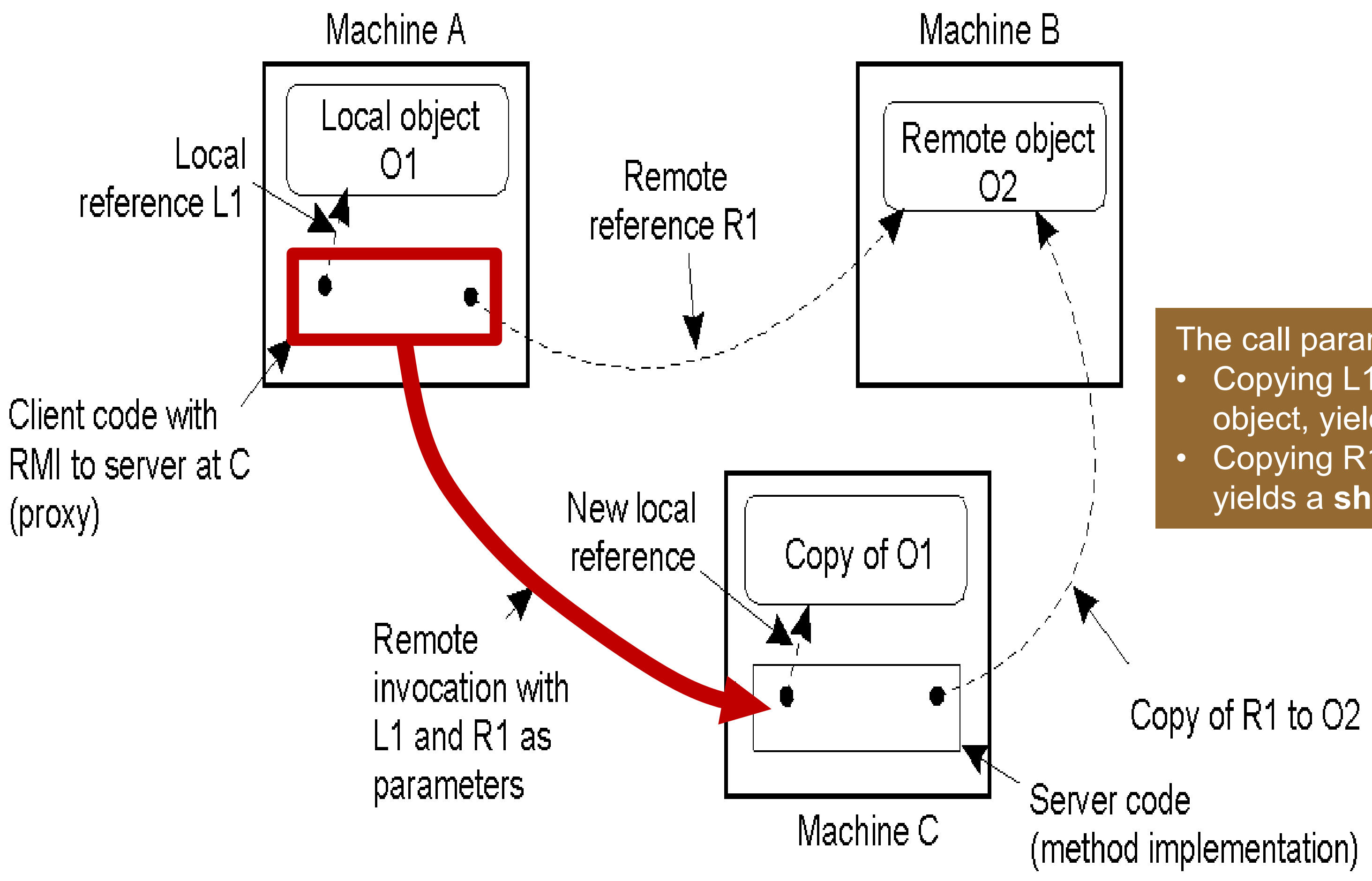
-#Olvnry Vxevwlwxwlrq#Sulqflsoh#

Differential anatomy of RMI – 2



The remote object "lives" in the server space

Differential anatomy of RMI – 3

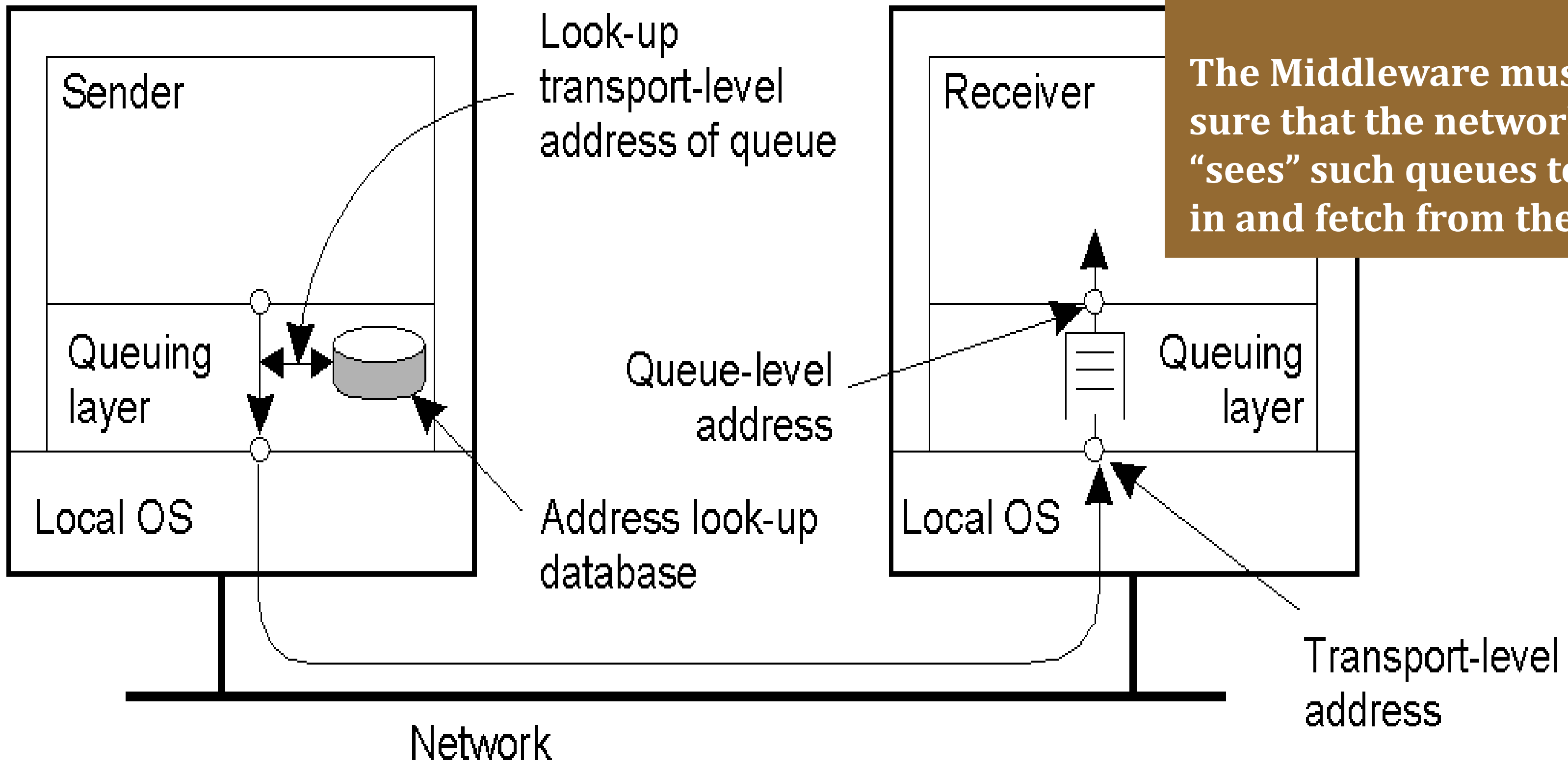


The call parameters (A to C) are **by-value**:
- Copying L1, which refers to a local object, yields a **deep copy** of O1
- Copying R1, which is a remote reference, yields a **shallow copy** of O2

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 non-blocking `Put`
  - Blocking if MW wants to prevent overwrites on full queue
  - Handler of send queue acts as proxy
- **Receive** maps to blocking (guarded) `Get`
  - A **callback** mechanism should be provided to decouple receiver from receive queue
  - Handler of receive queue acts as skeleton

Middleware-based message-passing – 2

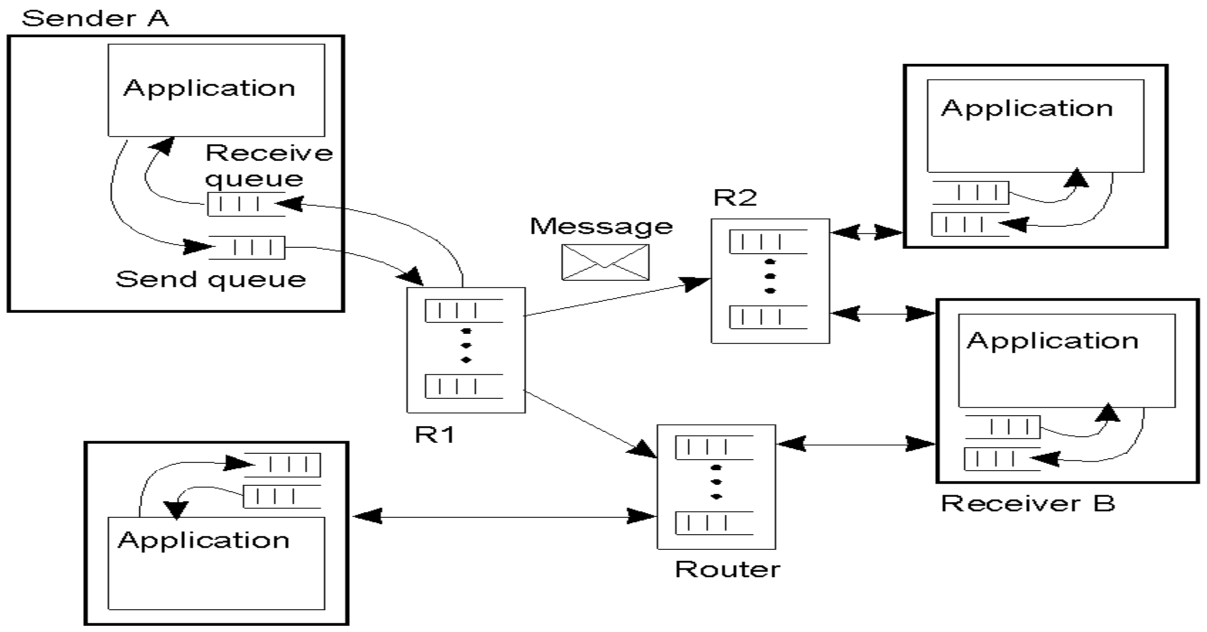


Each queue appears as local to sender and receiver

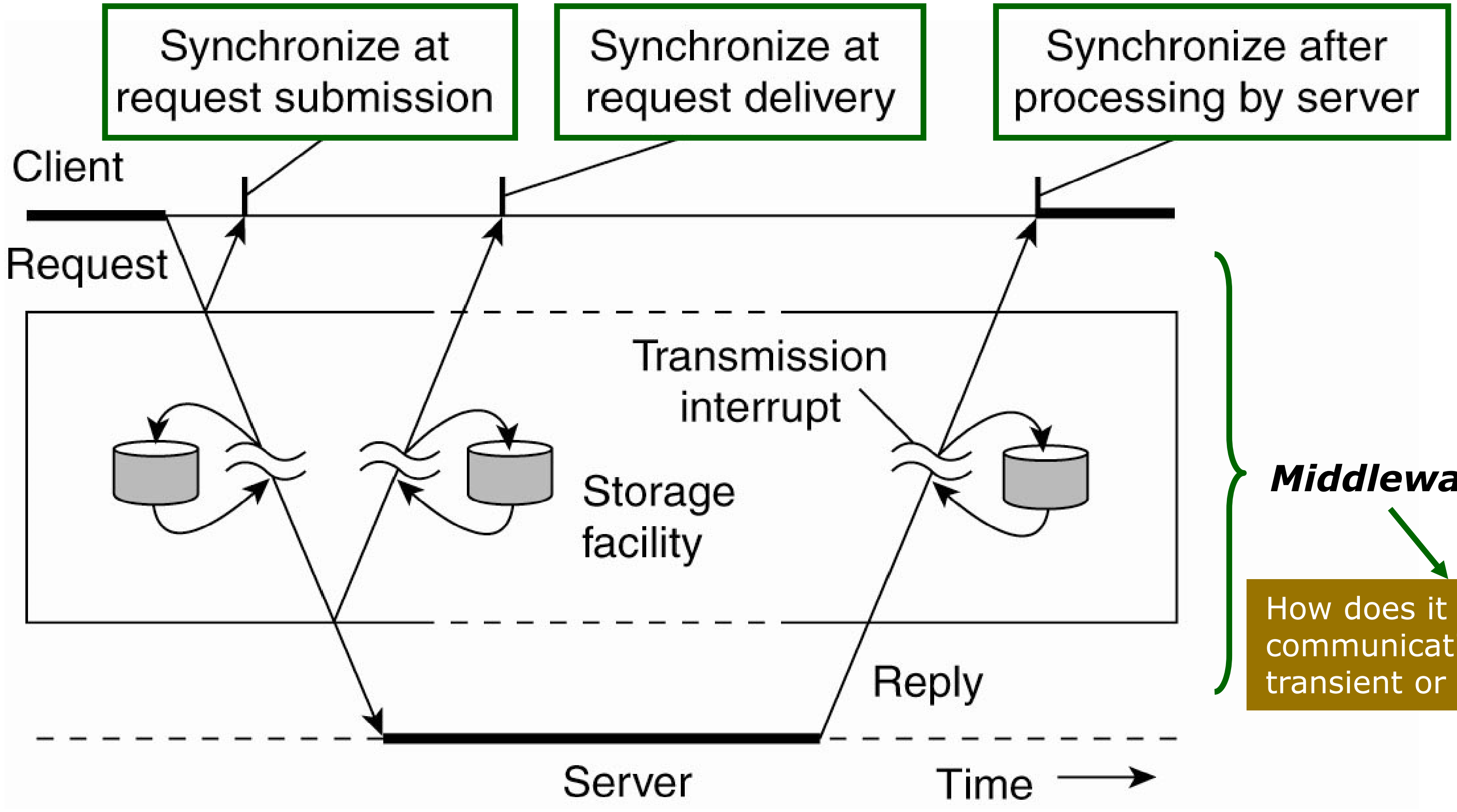
The Middleware must make sure that the network layer “sees” such queues to deposit in and fetch from them

Middleware-based message-passing – 3

- When Middleware overlays its own network over 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 classic Internet



Middleware-based message-passing – 4



Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

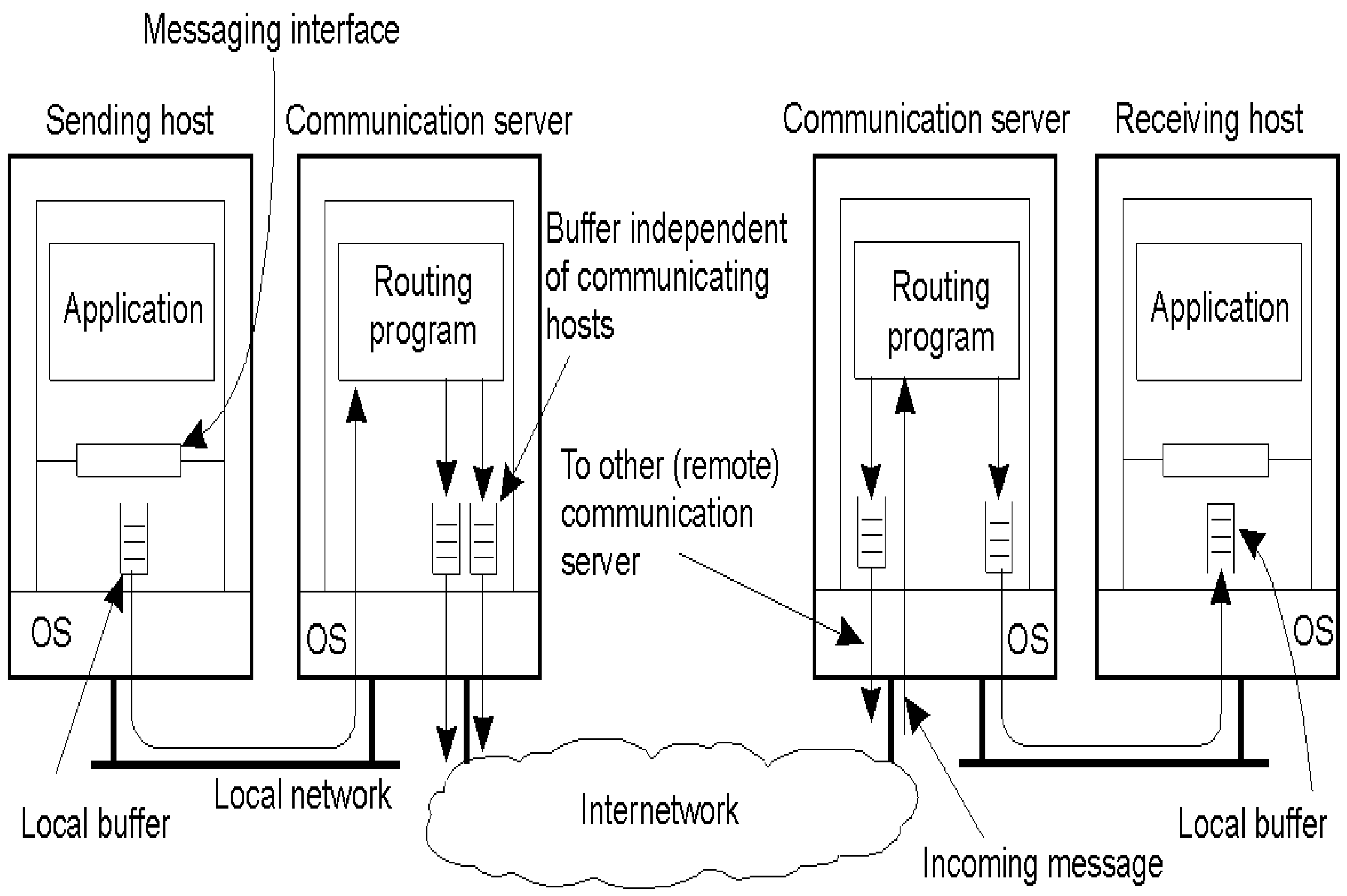
Middleware

How does it render communication transient or persistent?

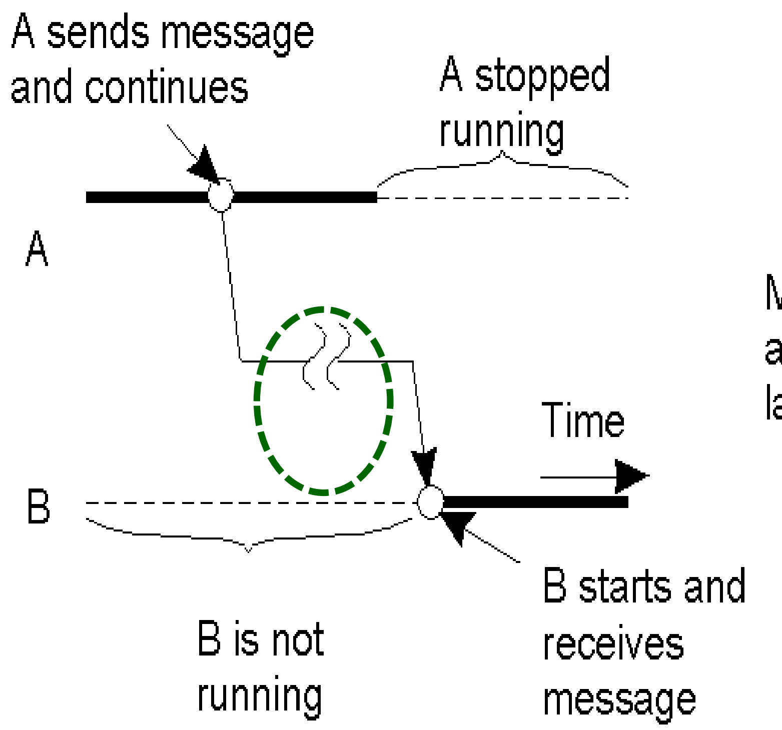
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Middleware-based message-passing – 5

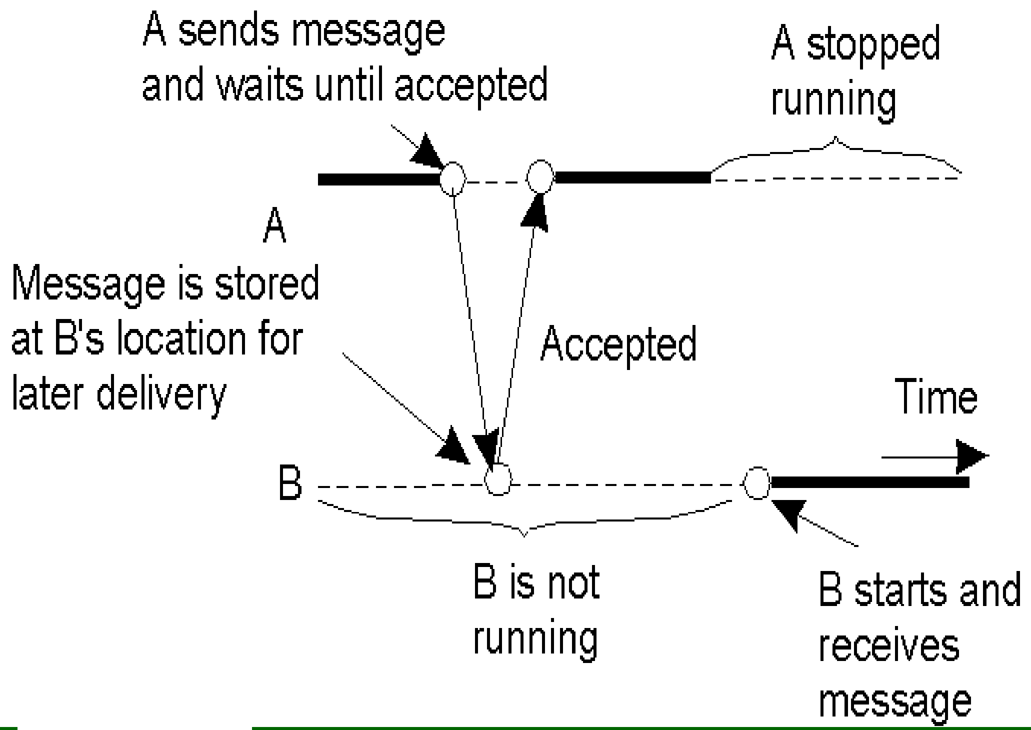
Distributed message passing incurs persistency and synchronization problems in the transit from sender to receiver



Middleware-based message-passing – 6



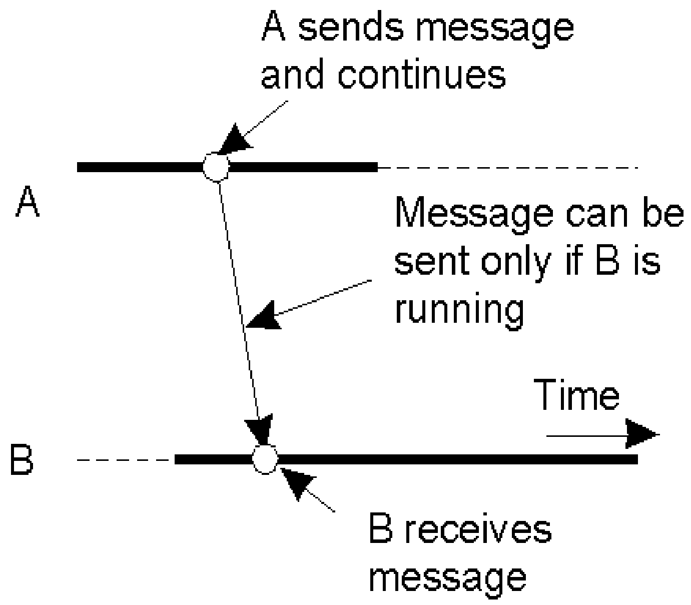
Async send (@MW), sync receive


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

Sync send (@MW), async receive

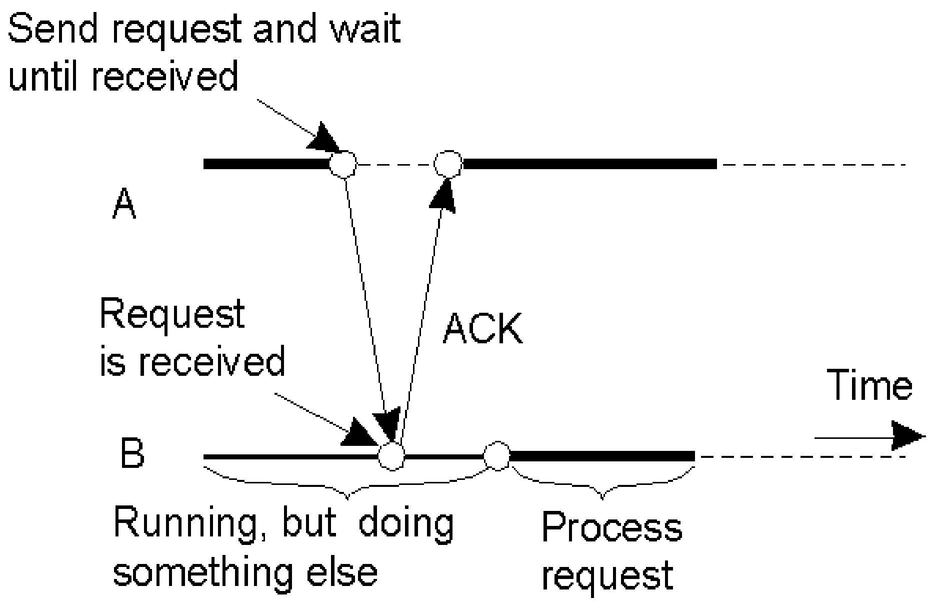
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Middleware-based message-passing – 7



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

Async send, sync receive

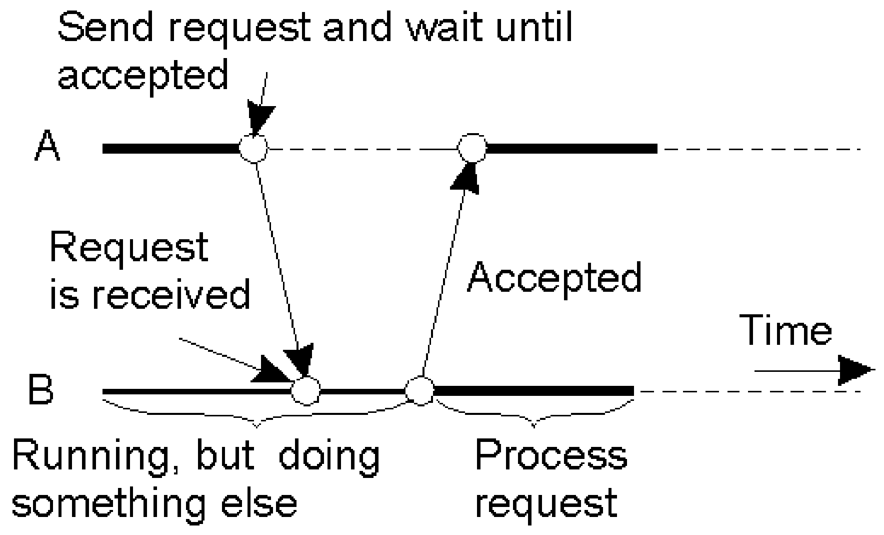


Send request and wait until received
A
Request is received
ACK
Time
B
Running, but doing something else
Process request

Partially sync send, async receive

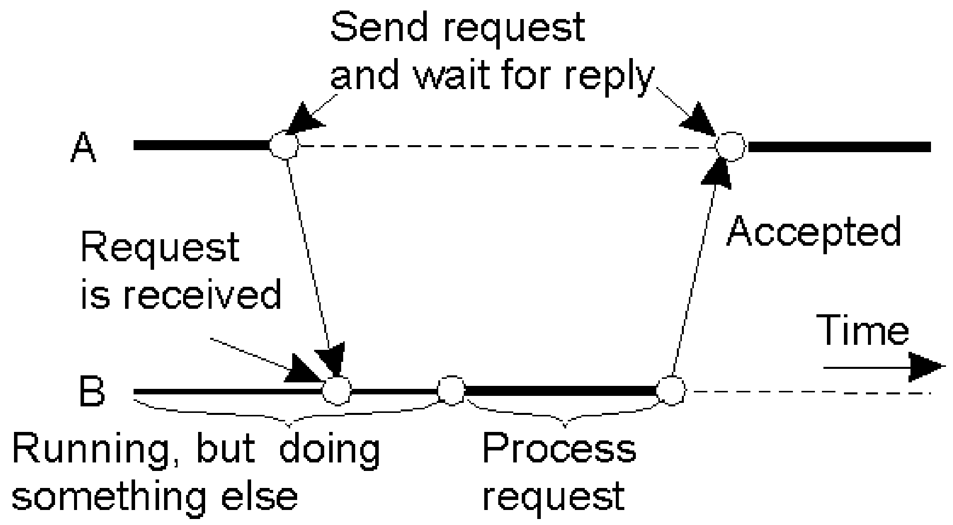
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Middleware-based message-passing – 8



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

Partially sync send, deferred receive



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

Sync send, deferred receive

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 client browser loads a page, it can request one *resource* at a time per TCP connection to server
  - Original Web assumed few heavy-weight connections, all pull based
  - Today’s Web features zillions of light-weight connections, also in push mode
- **WebSocket** allows full-duplex communication, making “HTTP/1.1 layer” a two-way road
- **HTTP/2** (binary) **multiplexes** multiple requests over a single connection to same server, to allow receiving *multiple* responses at once
  - But TCP does not know about it, which causes needless retransmissions …
- HTTP/2 also allows server to **push** contents into client without it requesting so (aka Server-Sent Events)
- **QUIC** (https://www.chromium.org/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* for 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
    - Check out WSO2 Rest API Design Guidelines
  - Data-centric: GraphQL
  - Collaboration-centric: gRPC
  - Stream-oriented: WebRTC

Past
Present & Future