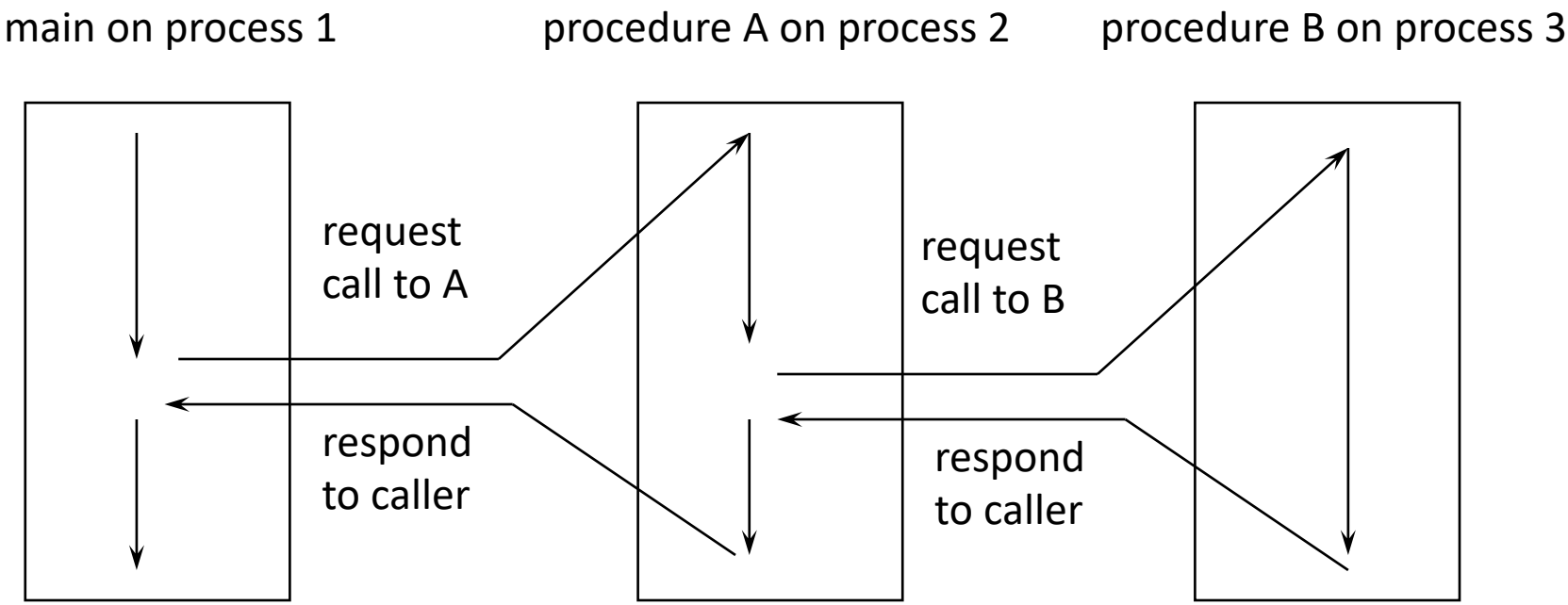
Remote Procedure Call (RPC)

Reference for study: Van Steen, Tanenbaum, "Distributed Systems", chapter 4.2

What is Remote Procedure Call?

- RPC is the transfer of the (local) procedure call mechanism to a **distributed environment:**

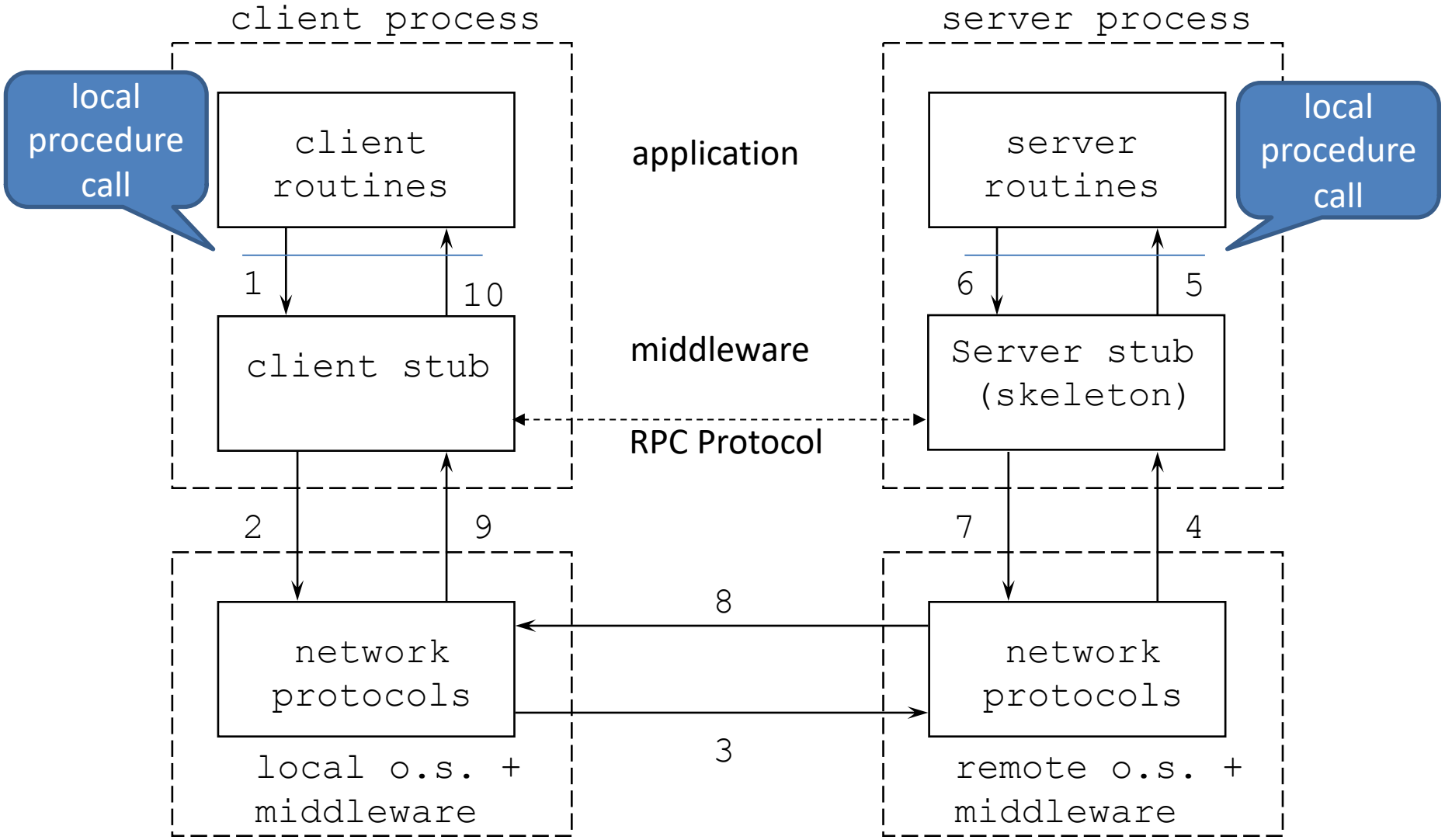


Examples of RPC Implementations

- SUN RPC (ONC) — C Language
- JAVA RMI — Java Language
- CORBA — Distributed Objects
- Web services — Distributed Services
- The Web (REST) — Distributed Resources
- gRPC

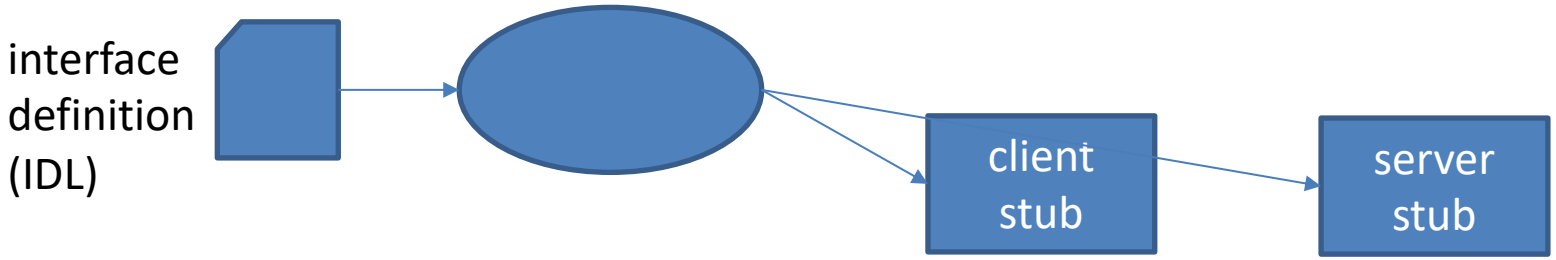
Language independent

The RPC model



The RPC Middleware

- Language-based support
  - RPC is offered as a feature of a programming language (e.g. Java RMI), exploiting language-specific serialization
  - high transparency, but limited to single-language apps
- Stub Generation



  - stubs can be generated for multiple languages

Issues to be handled by the RPC Middleware

- Caller and callee are **heterogeneous** (different languages, hw/sw platforms, etc.)
  - procedure arguments and return values must be converted (*marshaling/unmarshalling*)
- Caller and callee run on **different** processes and hosts
  - the server must be *located* before issuing the call (*dynamic linking*)
  - multiple incoming calls must be *synchronized* on the server
  - if and how arguments can be passed by reference must be defined (*address spaces are disjoint*)
  - partial failure is possible and must be managed

Managing Partial Failure (Remote Call Semantics)

- Remote call semantics differ from local call semantics:
  - A remote call may **fail** (server crashes, network outages)
  - A remote call may be executed **more than once** (message duplications)
- RPC middleware protocols can handle such problems only **partially**

  ⇒ programmers must be aware of them

  ⇒ Normally, two kinds of return are possible:
    - successful call
    - unsuccessful call

Remote call semantics

- According to what RPC protocol is used, different call semantics are obtained.
- Call Semantics Definitions:
  - **at least once**
  - **at most once**
  - **exactly once**

Typical RPC Implementation with At Least Once Semantics Calls

- Transport: UDP
- RPC protocol includes a way to associate response to request (and manages retransmissions)

- Upon successful return

  => at least once semantics

- Upon unsuccessful return

  => no guarantee given (the procedure may have been executed 0 or more times)

- Procedures must be **idempotent** (i.e. not sensitive to number of calls).
  - Often procedures can be made idempotent. Example:

    append X to file Y => write X at offset k in file Y

Typical RPC Implementations with At Most Once Semantics Calls

- The client is guaranteed that, in any case, the remote procedure will be executed at most once:
  - Upon successful return
    - exactly once semantics
  - Upon unsuccessful return
    - at most once semantics

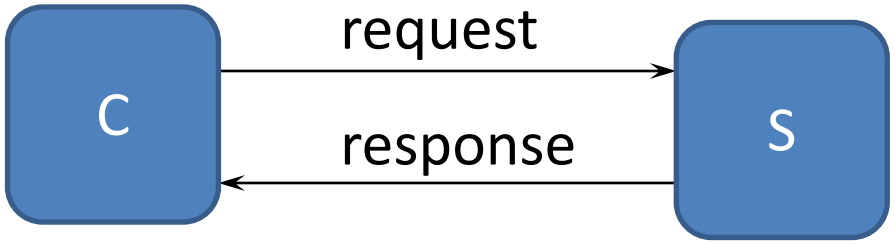
- Transport: TCP
- RPC protocol associates response to request via the connection

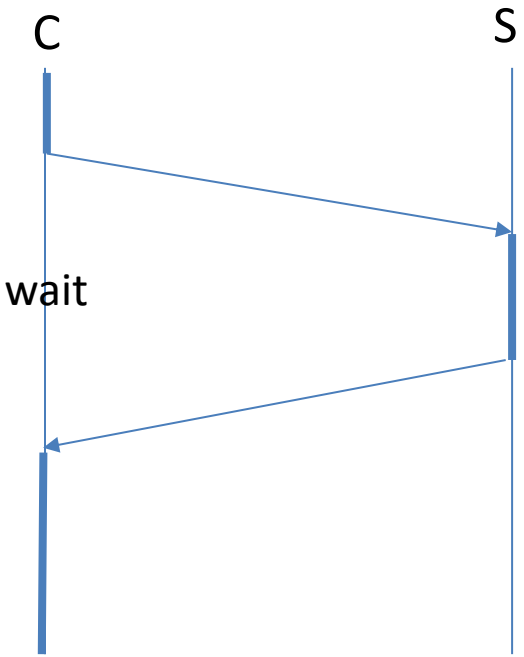
- Transport: UDP
- RPC protocol includes
  - response-request association
  - retransmission management
  - duplicate checking at server

Other RPC issues: Security and Performance

- Main *security requirements:*
  - access control
  - authentication
  - confidentiality
  - integrity
- Main *performance issues*:
  - a remote call is **orders of magnitude** slower than a local call

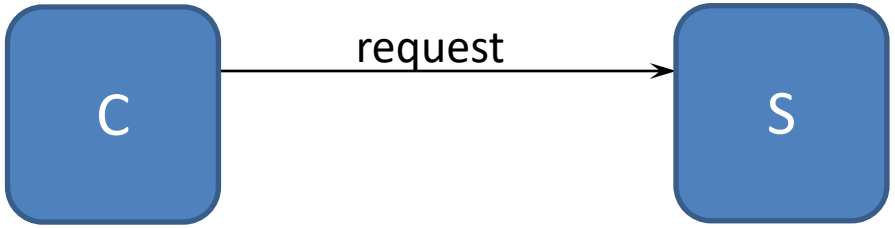
Classical (Synchronous) RPC

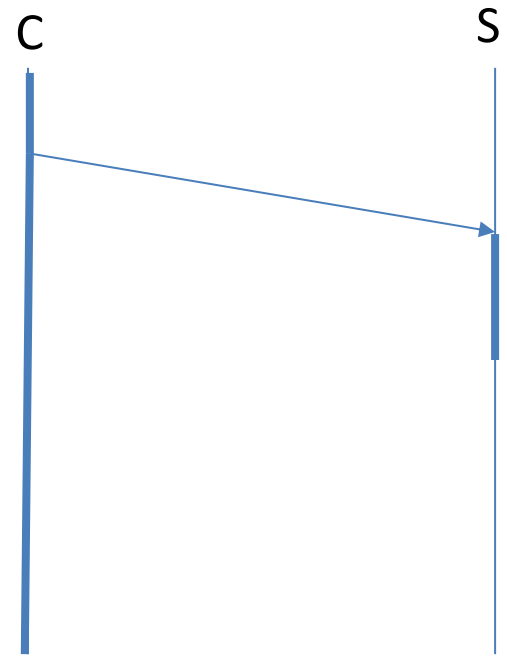




- Request:
  - Procedure identification
  - Input parameters
- Response:
  - Success/Failure indication
  - Return value and output parameters (or exceptions)

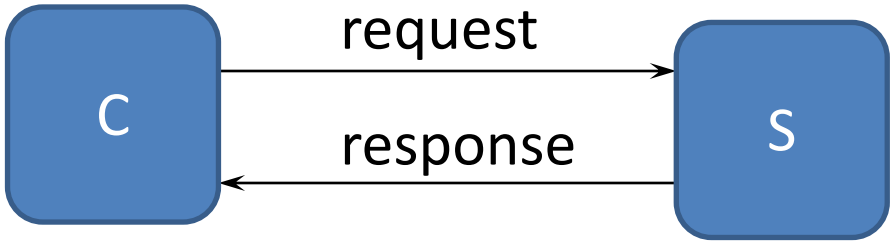
One-Way (Asynchronous) RPC

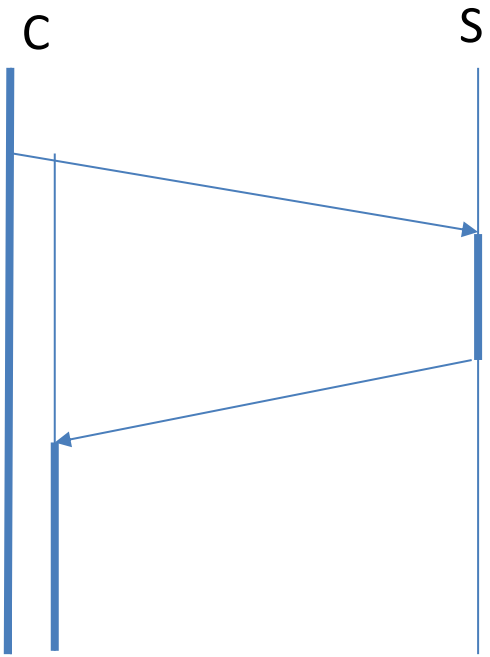




- No output arguments/return value
- No exceptions

Two-way Asynchronous RPC

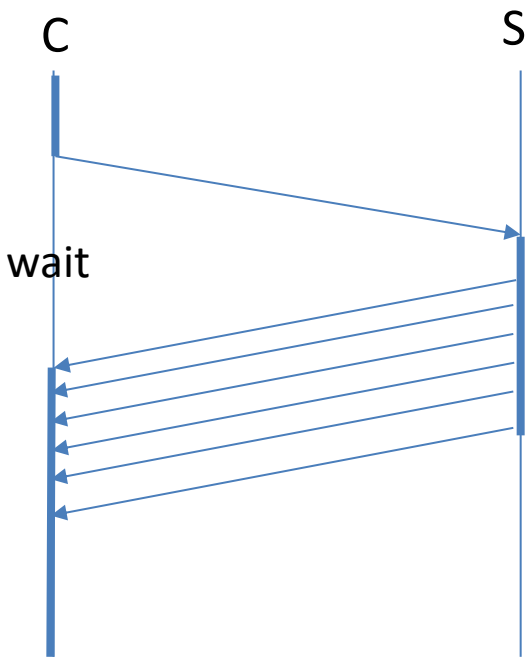


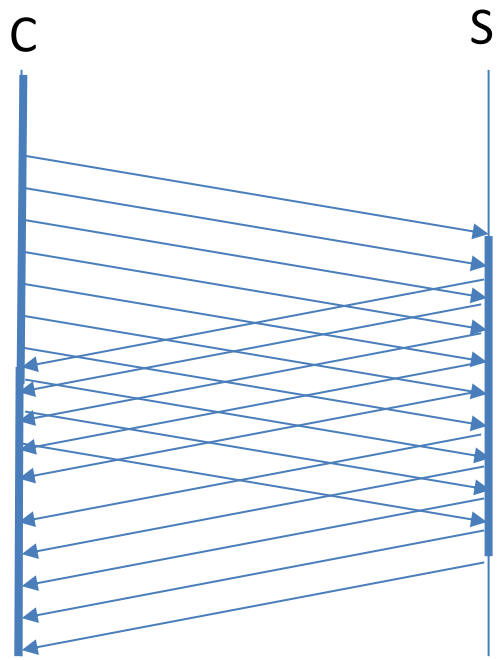


- Like Synchronous RPC
- Same protocol, but:
  - Client does not wait
  - Callback manages the response

RPC with Streaming

- All flavors of RPC can exploit **streaming** for request and/or response:
  - a request or response can be a *stream of* messages rather than *a single* message





Distributed Applications Developed on top of Socket APIs

- Sockets only provide communication services
- The implementation of the interaction protocol is up to the programmer:
  - data encoding for exchanged data
  - interaction procedures, management of partial failures, etc.
- Emphasis is more on communication than on application logic
- Normally used to implement
  - middleware
  - custom application-level protocols

Developing Applications on top of RPC

- The application can be developed as normal centralized code (but having in mind that interaction will be distributed!)
- Application modules are distributed on multiple hosts, and the application is tested in the distributed environment.
- The programmer can focus more on application logic than on communication

Developing Services on top of RPC

- Applications are conceived as combinations of services
- Services are designed and developed as reusable components that interact via RPC
- Advantages:
  - services are simple reusable components
  - scaling out can be achieved by creating multiple service instances (based on request volume)