Introduction to Communication Networks and Distributed Systems

Unit 3 - Introduction to Distributed Systems -

Fachgebiet Telekommunikationsnetze

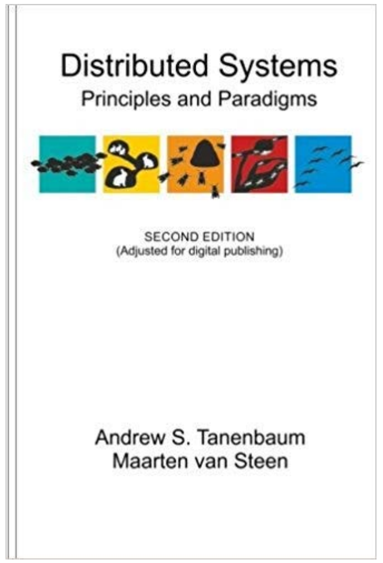
Prof. Dr. A. Zubow

zubow@tkn.tu-berlin.de

Technische Universität Berlin

We can now transport meaningful messages …

… but how to use them?



Distributed Systems
Principles and Paradigms

SECOND EDITION
(Adjusted for digital publishing)

Andrew S. Tanenbaum
Maarten van Steen

Tanenbaum/Steen: „**Distributed Systems: Principles and Paradigms**“, chapter 1.

Distributed System

- A distributed computing system consists of multiple **autonomous** processors that do not share primary memory but **cooperate** by sending messages over a communication network.

  Henri Bal / Colouris

- A distributed system is one in which the **failure** of a computer which you didn‘t even know existed can render your own computer unusable.

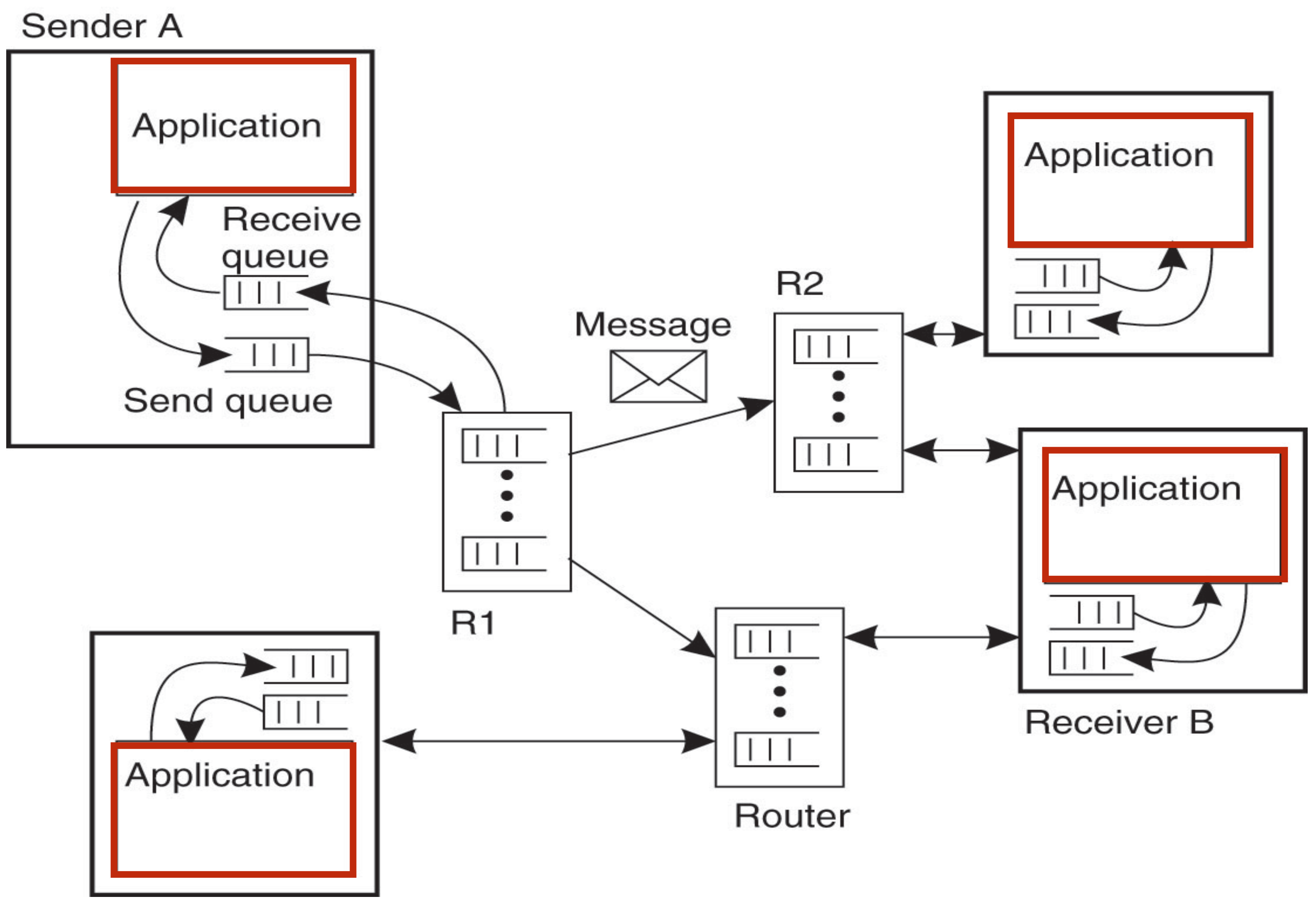
  Leslie Lamport

- A distributed system is a collection of **independent** computers that **appears** to its users as a **single coherent** system.

  A. S. Tanenbaum

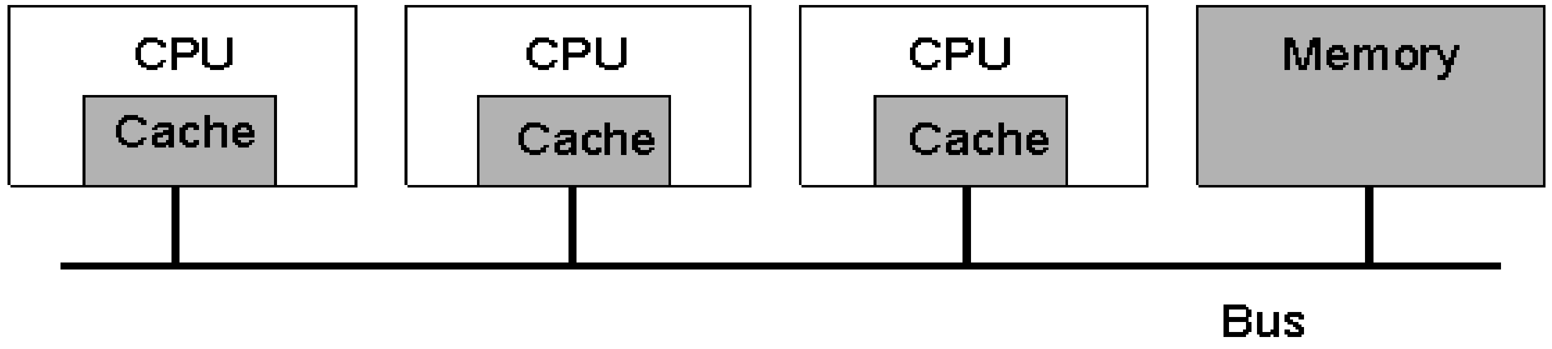
A Distributed System

[Tannenbaum, op. cit.]



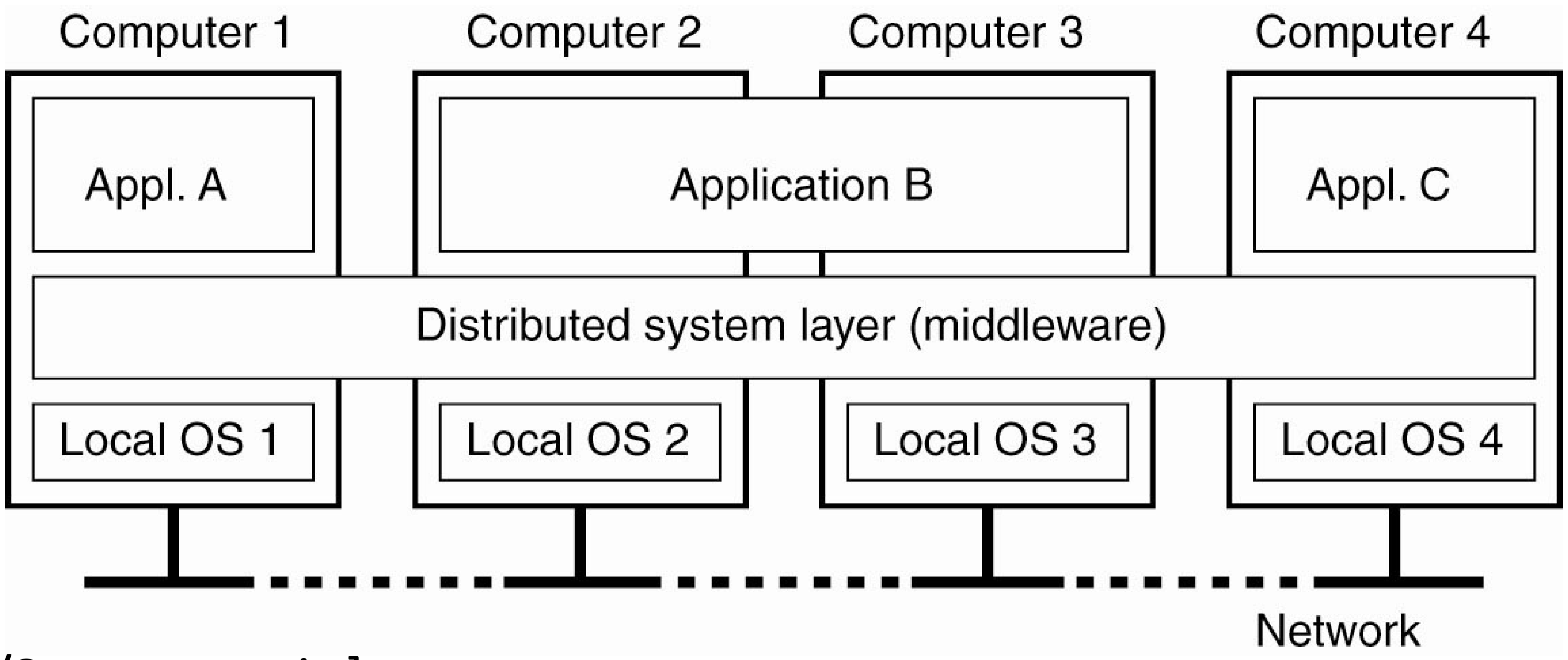
Multiprocessors vs. Distributed Systems

- A bus-based multiprocessor with typical features:
  - Physical access to a common memory
  - Resources under same management
  - Very fast (bus, switching matrix) based local communication



Distributed System organized as Middleware

- **Middleware layer** = an additional layer of software to support heterogeneous computers and networks while offering a single system view



[Tannenbaum/Steen, op. cit.]

Two important Features of Distributed Systems

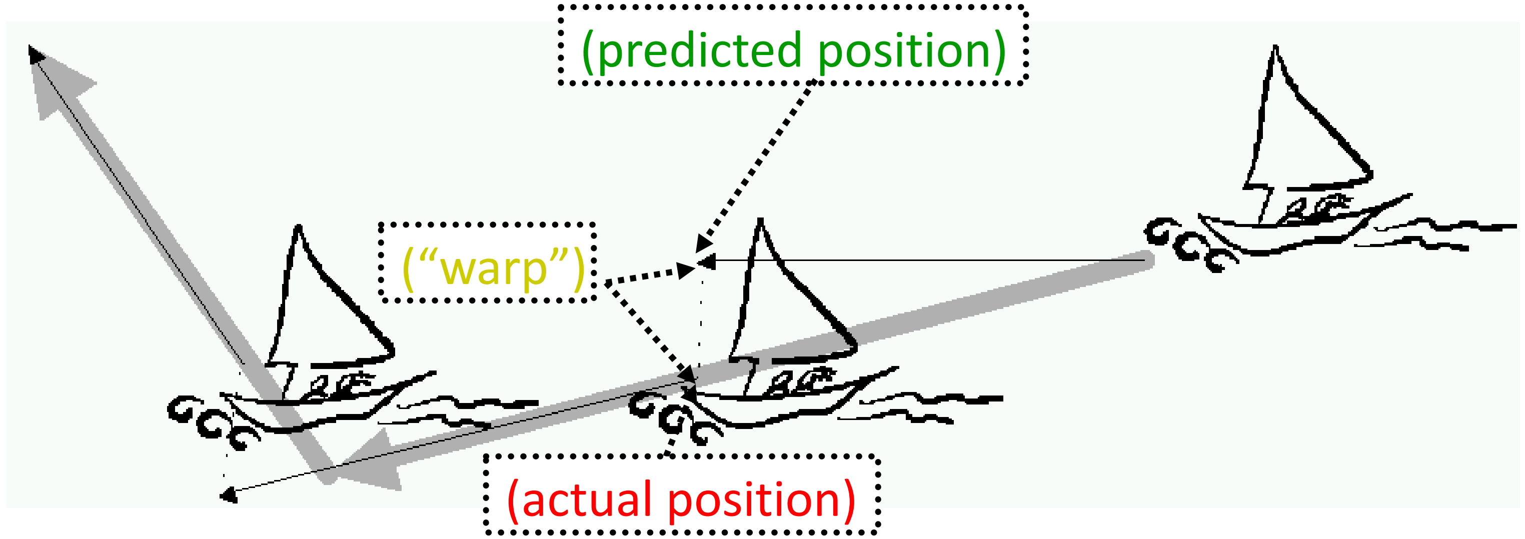
- **Autonomy**
  - A distributed system consists of autonomous, independent entities (usually cooperative ones, sometimes antagonistic ones)
  - Each individual entity is typically a full-fledged, operational system of its own
  - Individual entities might follow local policies, be subject to local constraints…
- **Transparency**
  - The fact that the distributed systems is indeed a conglomerate of different (simpler) systems is of no interest and should be of no concern to the user

A Dead Man that Shoots [Mouve, op. cit.]

- Consider an action game played over the network: player **A** shoots player **B**:
  - Application **A** creates a bullet entity with a certain heading and velocity. It will then transmit over network the state of that bullet entity.
  - Upon receiving the state of the bullet application **B** will start to check whether any entity under its control is hit by the bullet.
- There is, however, a transmission delay
  - This **network delay** may be large (~>100ms) although the players in “virtual space” are close
  - During this time player **B** might take actions,
    - shoot at another player **C**, even though “he’s already dead”.
    - move so that the bullet would not hit him - from his point of view!
- Q.: Which hits should be scored? Players A, B, and C may therefore disagree about whether a hit has been scored on B or not!

Dead Reckoning

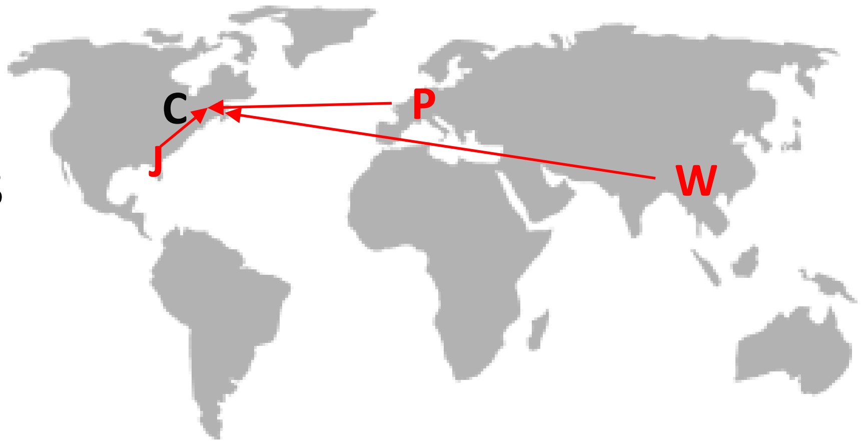
- Based on ocean navigation techniques
- Predict position based on last known position plus direction
  - Can also only send updates when deviates past a threshold



- When prediction differs, get "warping" or "rubber-banding" effect

Distributed Stock Exchange

- Traders distributed all over the (connected) world:
  - Each of the traders sets his computer to:
    - Buy stocks of company *HOPE* if they fall under X $
- The central unit (located at Wall Street, NY) defines the value of the *HOPE* stocks and announces them
- Due to different transmission delays trader John at Wall Street will always **react quicker** than Paul in Frankfurt or Wendong in Peking!
  - Will he really?
- Will traders located close to each other receive the information simultaneously?
  - The notion of fairness



Pitfalls when Developing Distributed Systems

- Wrong assumptions are made:
  - The network is reliable
  - The network is secure
  - The network is homogeneous
  - The topology does not change
  - **Latency is zero**
  - **Bandwidth (= bit/data rate) is infinite**
  - Transport cost is zero
  - There is one administrator

Characteristics of Decentralized Algorithms

- No machine has complete information about the system state.
- Machines make decisions based only on local information.
- There is no implicit assumption that a global clock (i.e. precise common understanding of time) exists.
- **Note**: Failure of one or more machines **should not** ruin the algorithm

The larger a system the larger the uncertainty

Design Challenges: Openness [Hou, op. cit.]

- Degree to which new **resource-sharing** services can be added & used.
- Requires **publication** of interfaces for access to shared resources.
- Requires **uniform communication** mechanism.
- Conformance of each component to the published standard must be tested and verified.

Design Challenges: Scalability

[Hou, op. cit.]

- A system is said to be scalable if it will remain effective when there is a significant increase in the number of **resources** and **users**:
  - Controlling the cost of resources
  - Controlling the performance loss
  - Preventing software resources running out (e.g., IP addresses)

Transparency in a Distributed System [Tanenbaum, op. cit.]

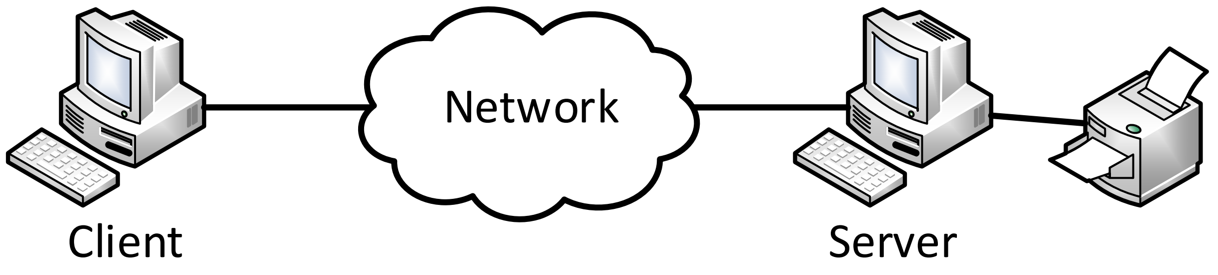
Transparency	**Description**
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) is in memory or on disk

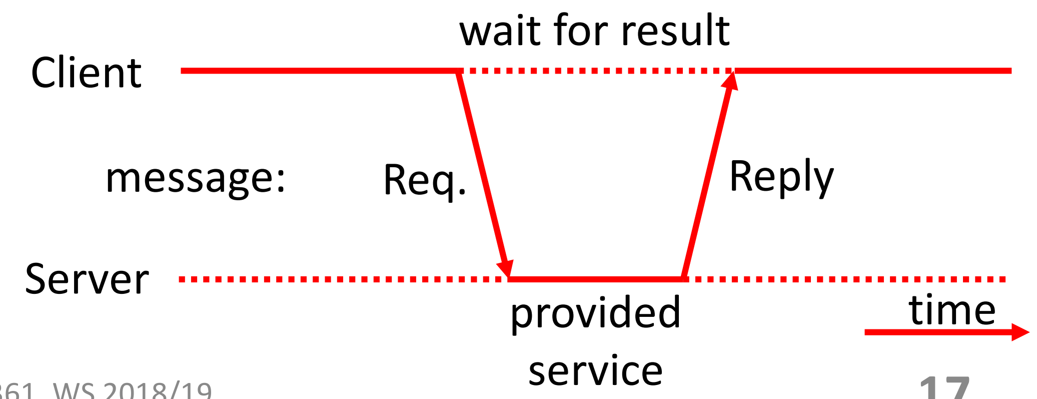
- **Note**: There is a difference between the feature, mechanism and policies
- Take openness and scaling into account … different forms of transparency in a distributed system (ISO, 1995)

The Client-Server Approach

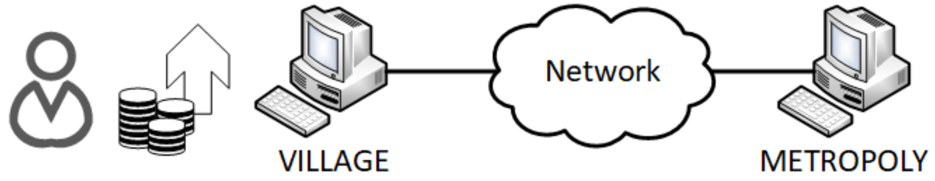
Client-Server Approach

- Is the message passing/stream interface enough? What about more complex cooperation patterns….
- **Example** of client/server approach:
  - A process wants to perform a specific action, e.g. to print a file. The local computer can not do it (there is no printer), but there is another computer able to do it (the printer is connected to that other computer). The other one is the server.
  - Client transmits a request message to the server (including the file to be printed), asking the server to perform the service
  - Server receives this messages and performs (probably) the appropriate action (prints file).
  - Confirmation about printout is send back to the client via a reply message.





Another Example - A Bank Transfer



- Consider a bank with headquarters in METROPOLY which has a branch in VILLAGE
- Client: Has 5000 € on his account. He enters the branch in VILLAGE to cash 1000€.
- **Send-and-Wait** will be used to transmit this request:
  - Clerk 1 (branch): Posts and repeats upon timer expiration!

    If Balance > 1000 €

    then (subtract 1000 € and acknowledge),
    otherwise call the police in VILLAGE.
  - Clerk 2 (headquarters): Executes the request.
- Imagine the acknowledgement is lost, several times in sequence! duplicates

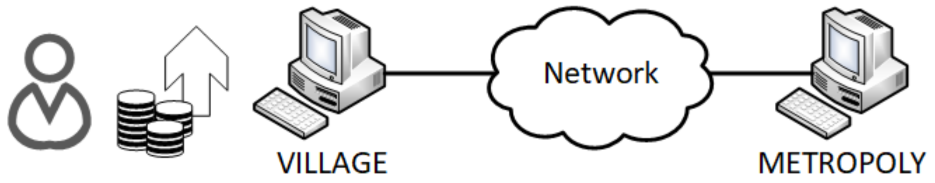
Client – Server Model (distributed computing)

- The clerk 2 is „only“ executing the requests posted by the clerk 1,
- Conceptually, clerk 1 might have done it himself,
  - But did not have the data „handy“,
  - And, possibly, could serve next customer while clerk 2 has been processing his request - **pipelining**
- Such schema of operation is called „client-server“ model
- We will discuss this in detail later …

But how to solve our problem?

- Remedy?
  - One way to go: Change the way of interaction between the clerks!
  - Another way to go: Replace the Send-and-wait by some better solution (without duplications!)

1st Option - Restructuring the Interaction



- Check the balance
  - Clerk 1: Posts a following message and sets a timer!

    Post me the balance , account no. X // comment: can be repeated several times
  - Clerk 2: Executes the request
- Set new balance
  - Clerk 1: Computes the new balance;
  - Clerk 1: Posts and repeats upon timer expiration!

    New balance for account no. X is 4000 $, confirm
  - Clerk 2 : Executes the request
- **Idempotent actions**: activities which result does not change in case of repetitive execution.

2nd Option - Restructuring the Interaction

- Add to each request of clerk 1 a **transaction identifier** (TxID)
  - Use the same TxID for an original request and for any possible repetition of this request
  - Do not use the same TxID for different transactions!
- Good news: It seems to work.
- What happens if messages might be **heavily delayed**
  - Clerk 2 should memorize the list of all the “already used” TxIDs for each account number
  - He has something to memorize …
- Construction of such protocol is not trivial. We will discuss this later in this class!

The Concept of **State**: intuitively

- Information relevant to the progress of some activity. Access to this information is frequently necessary to assure proper continuation of this activity.
- Send-and-wait
  - Sender has to be aware of the timer!
  - Receiver is **stateless**
- In the first solution – being stateless is a desired feature of a server!
  - Consider changing the server? Server collapse?
  - Consider **scalability**: if there are many clients and many accounts - state has been memorized for each of them?

Hard State vs. Soft State

- Introducing **state** might be necessary. Typically if some resources have to be kept available …
    - Consider the restructured banking example. If several clients would use the same account a „**lock**“ on access would be needed between *Checking Balance* and setting *New Balance*.
- **Hard state** approach:
    - State information has to be deleted as result of a proper action!
    - What happens with the “lock” if the computer of clerk 1 brakes down after “checking the balance”?
- **Soft state** approach:
    - The state information removed if not reactivated since X → timeout
    - Analogy: like putting shoes in the store on hold for 2 days …

Service

- Service: Any act or performance that one party can offer to another that is essentially intangible and does not result in the ownership of anything. Its production may or may not be tied to a physical product.

D. Jobber, Principles and Practice of Marketing

- Focus is on the ***output***, i.e. the *result* of the service.
- Not the means to achieve it.

Service-Oriented Architecture (SOA)

- SOA establishes an **architectural model** that considers services as the primary means through which (enterprise) solution logic is represented
- It follows a software **design pattern** of *"separation of concerns"* and *"interoperability",* i.e. partitioning solution logic into capabilities, each designed to solve an individual concern, while following a set of core principles
- It tries to address the deficiencies of the traditional application-specific solution logic design:
  - High levels of functional redundancy
  - Loss of efficiency through building and rebuilding existing logic
  - Increasing hosting, maintenance, and administration demands
  - Diversity of technologies hampers evolution and scalability
  - High efforts needed for post-factum integration and data sharing

High-level SOA Goals

- Increased consistency in how functionality and data is represented
- Reduced dependencies between units of solution logic
- Reduced external awareness of underlying solution logic design and implementation details
- Increased opportunities to use a piece of solution logic for multiple purposes
- Increased opportunities to combine units of solution logic into different configurations
- Increased behavioral predictability
- Increased availability and scalability
- Increased awareness of available solution logic

SOA Principles

- Service Abstraction
  - This principle emphasizes the need to hide as much of the underlying details of a service as possible. This enables the **loosely coupled dependencies** and promotes design based on **service compositions**
- Standardized Service Contracts
  - Services express their purpose and capabilities via a ***service contract*** defined through a ***public interface*** that expresses the functionality, as well as the associated *data types, data models* and *use policies*
- *Service Loose Coupling*
  - The goal is **reducing** (“loosening”) **dependencies** between the *service contract*, its *implementation*, and its *service consumers,* promoting independent design and evolution of a service’s logic and implementation while still guaranteeing interoperability with existing service consumers

SOA Principles (II)

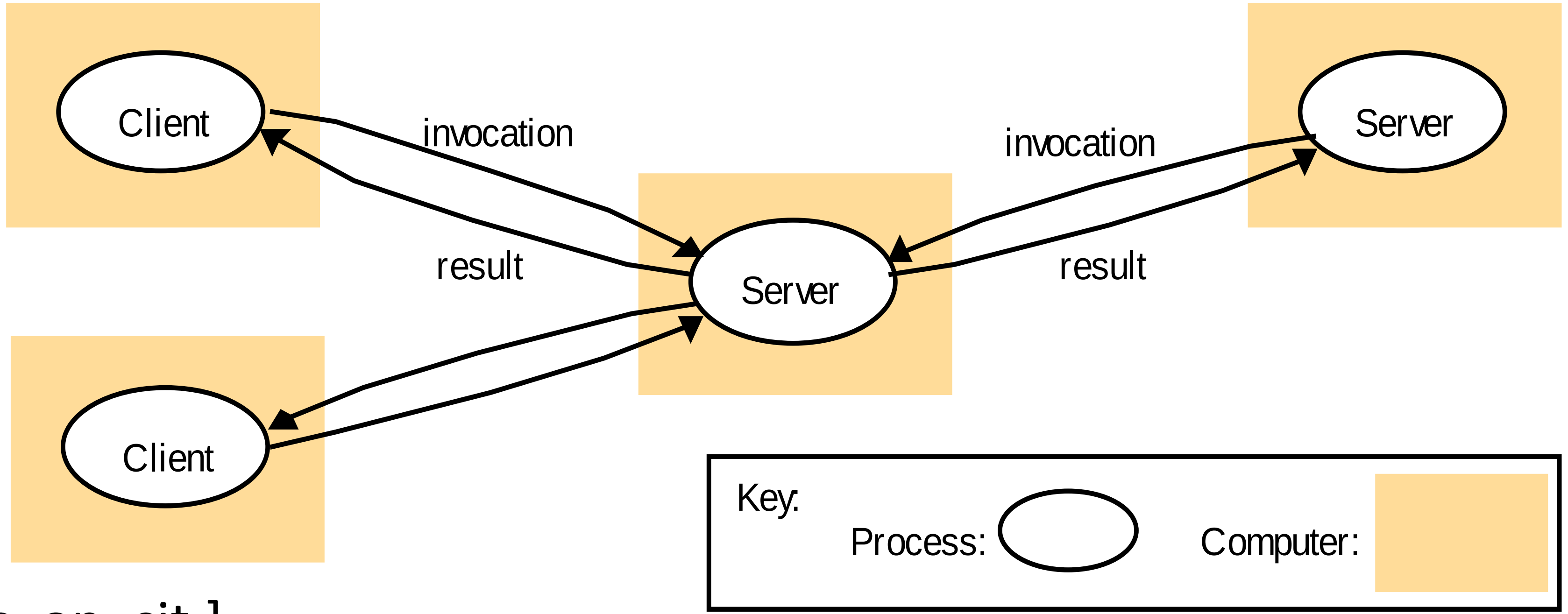
- Service Reusability
  - The aim is to design services as resources with agnostic functional contexts which maximizes the reuse potential of multi-purpose logic
- Service Autonomy
  - For services to carry out their capabilities consistently and reliably, their underlying solution logic needs to have a significant degree of control over its environment and resources
- Service Statelessness
  - The management of large state information can compromise the availability of a service and undermine its scalability potential. Services are therefore ideally designed to remain stateful only when required

SOA Principles (III)

- Service Discoverability
  - Services need to be easily identified and understood by consumers or by service designers (for reuse). This can be supported by introducing a dedicated discovery mechanism such as a service registry
- Service Composability
  - Services are expected to be capable of participating as effective composition members, regardless of whether they need to be immediately enlisted in a composition

System Architecture: Client-Server Model

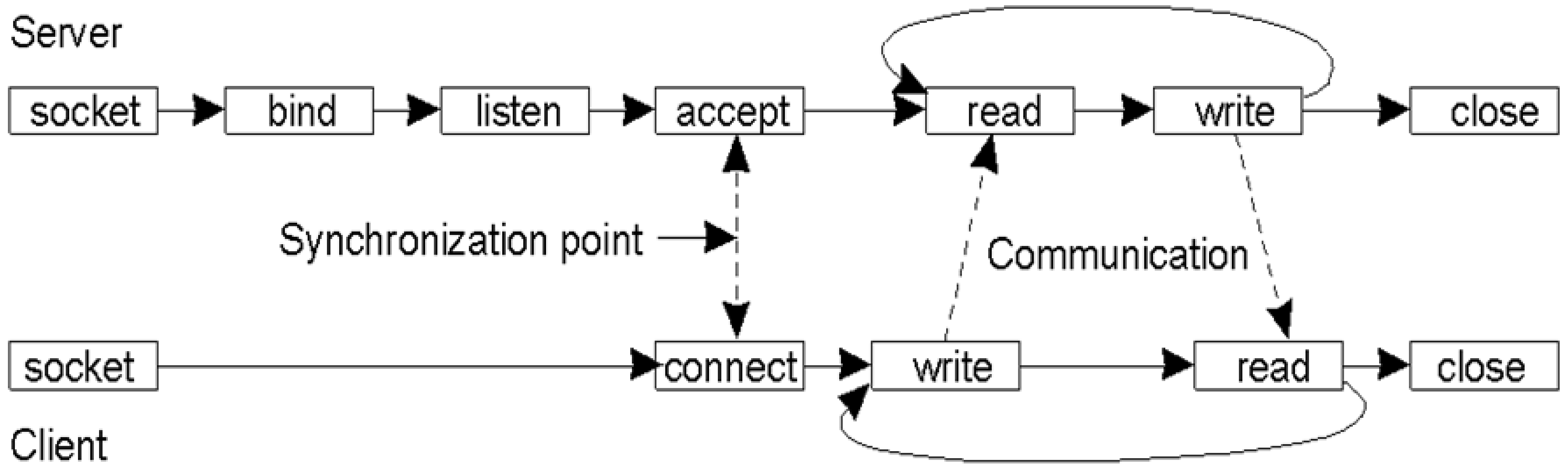
- A server process can act as client for an other server
- A service is not bound to a specific computer
- A single computer can host some clients and some servers



[Colouris, op. cit.]

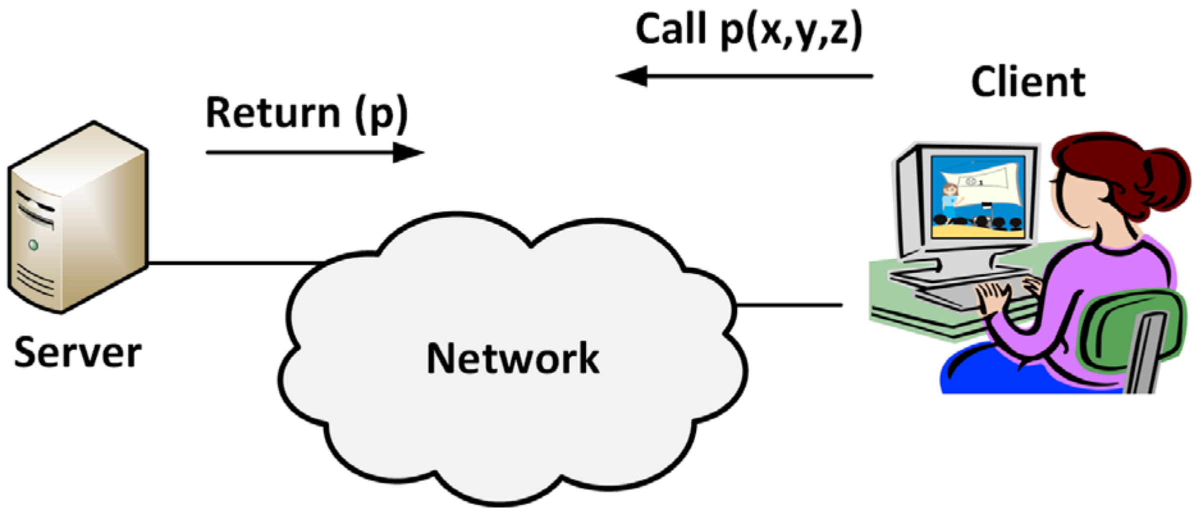
Berkeley Sockets for Client-Server

- Berkeley sockets is a Unix API for **Internet sockets** and Unix domain sockets, used for inter-process communication (IPC).



Remote Procedure Calls (RPC)

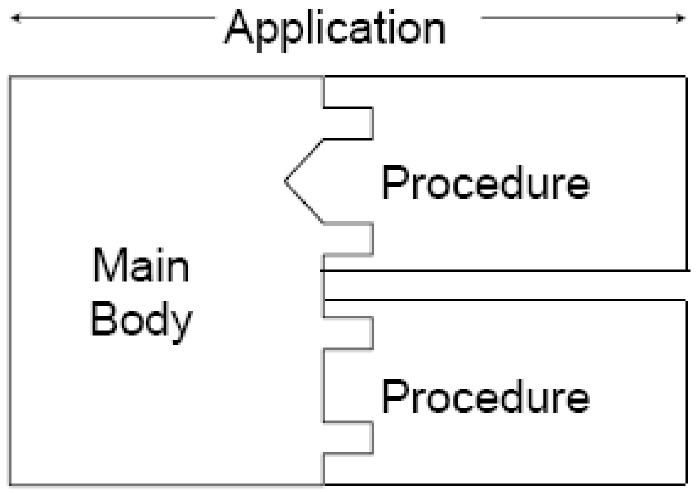
- Remote Procedural Calls are the preferred way to implement the client-server model.
- In classical procedure calls the code of the procedure is located on the same computer (in the same address space) as the calling program, in an RPC the code is located on another (**remote**) **computer**.



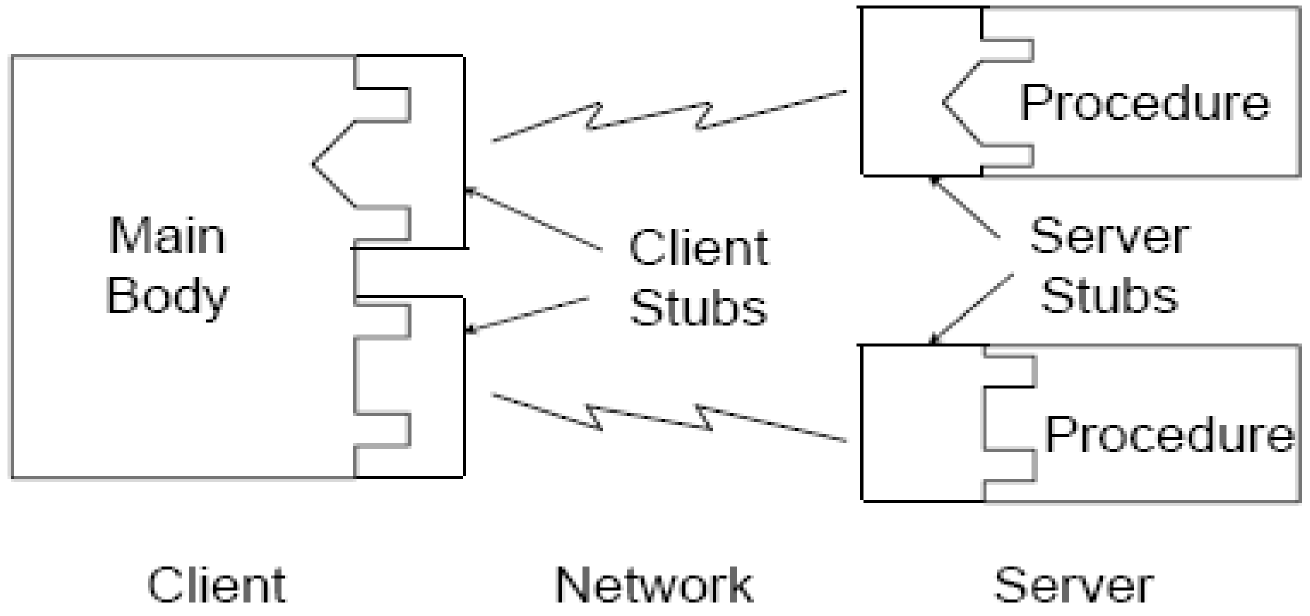
Remote Procedure Calls (II)

- One major design goal of an RPC system is **transparency**: ideally the caller should not know if the callee is located locally or remotely.
- So in RPC we have to consider the following topics:
  - Parameter handling and marshalling
  - Semantics
  - Addressing
- An RPC system is attractive for the users because automatic support for the conversion from local to remote procedural call can be supported by some framework (see below).

Local vs. Remote Procedural Call



vs.

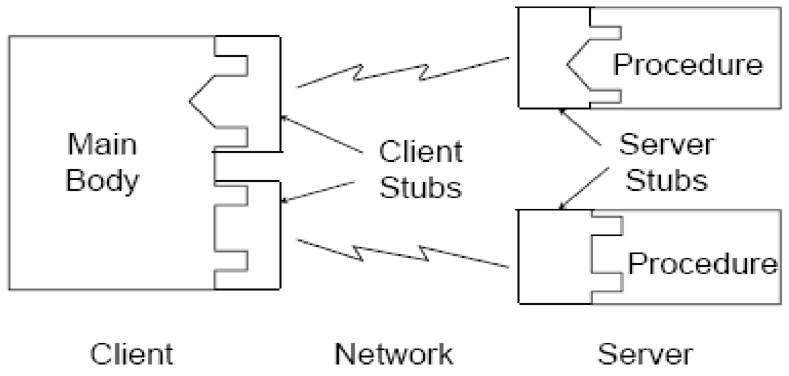


Basic RPC Operation

- The **caller uses** a specified **procedure interface** (e.g. ANSI-C function prototype), where all necessary procedure parameters are declared with their datatypes.
- In case of a local procedure, the callee performs the requested operation, in the **RPC** case the **callee is a stub** procedure that takes the given parameters, forms a message out of them (***marshalling***) and sends it to the appropriate server, i.e. procedure call is transformed into message sent over the network.
- In the **server** another **stub** procedure (also often called a skeleton procedure) receives the message, extracts the parameter values (***unmarshalling***) and calls the (local) procedure.

Basic RPC Operation (II)

- The **server stub** takes the results of this call and sends them back to the client (again as a message).
- The **client stub** returns the result extracted from the received message to its caller.

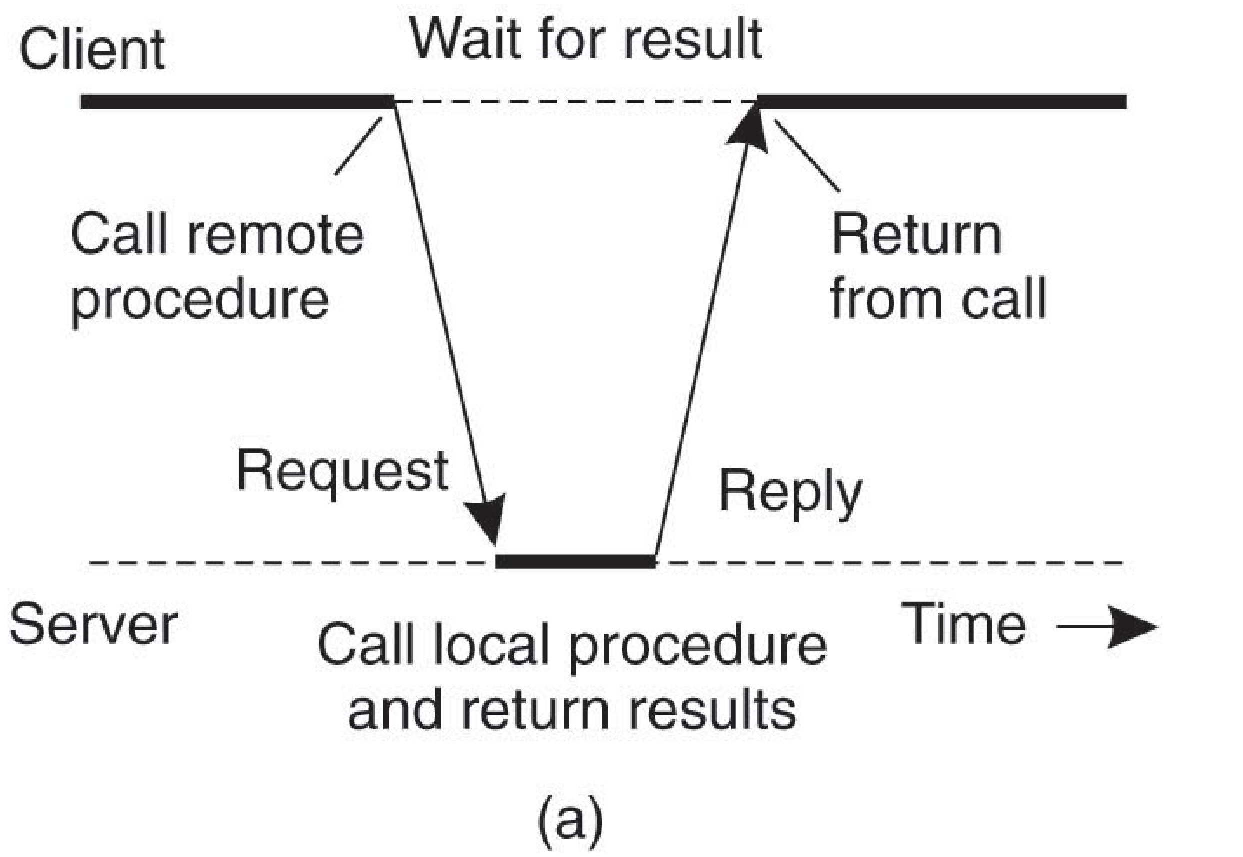


- Note: client/server stubs are automatically created by tools

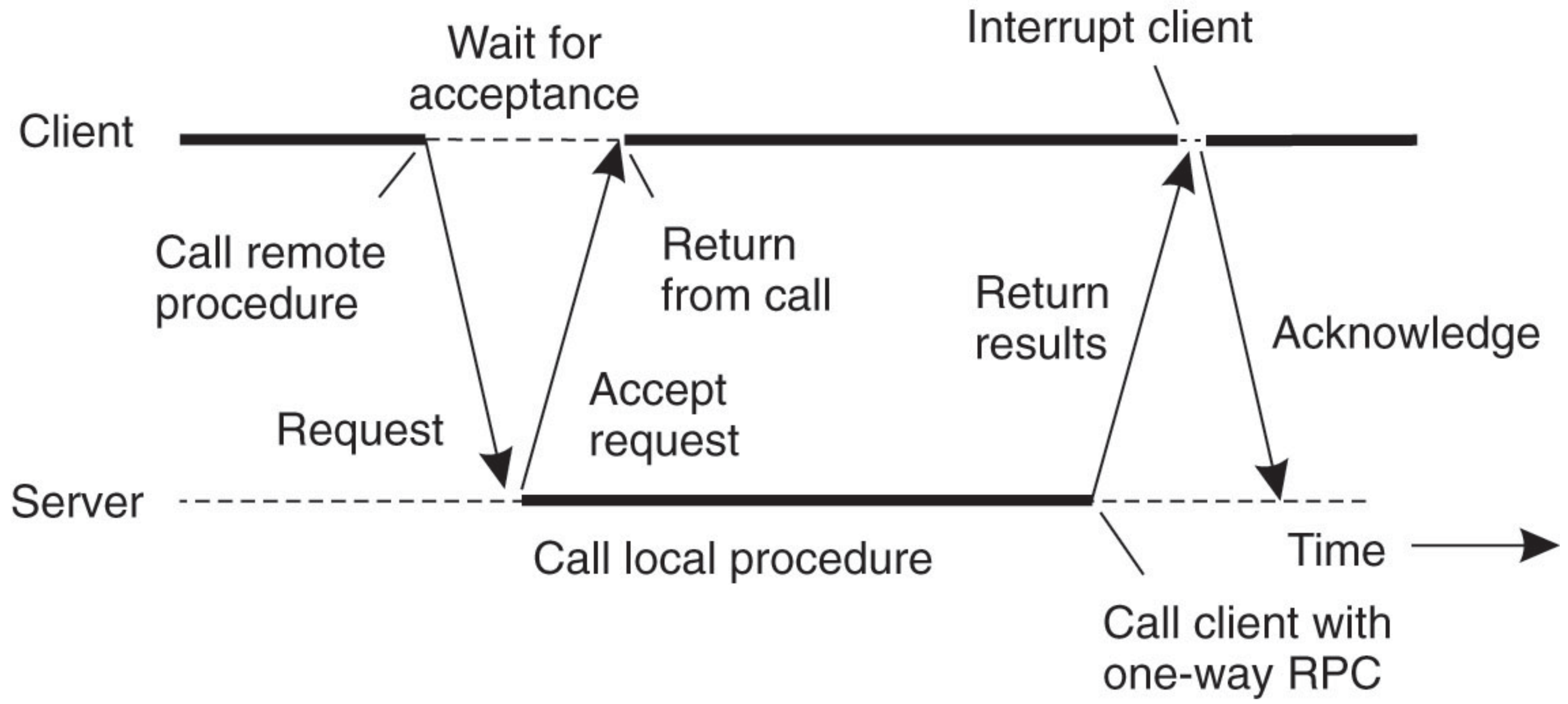
RPC Parameter Passing

- Procedures in common programming languages have different types of parameters and **calling conventions**, which have to be treated in a RPC:
  - Simple **call-by-value** parameters are passed “as is“ (e.g. simple integer values)
  - **Call-by-reference** parameters are pointers ➔ IN RPC pointers are allowed only to objects accessible globally – by both requesting party and the serving party!
    - If different address spaces are used by sender and receiver, the denoted value (e.g. a buffer) has to be completely transmitted (so its length and its type must be known in advance).
- The stub procedures must use a common encoding convention for different parameter types.

“Traditional” synchronous RPC



Client & Server interacting through asynchronous RPCs



Local Call vs. RPC - Differences

- In the local case it is assumed that a procedure call returns correctly (unless the system fails).
- This assumption is not valid for RPC systems. Several problems can arise:
  - **Addressing** (the client could not find an appropriate service)
  - The client or the server can fail
  - **Message loss**
- If the client could not find an appropriate server, a kind of exception handling is needed, thus **violating the transparency requirement**.

RPC Semantics

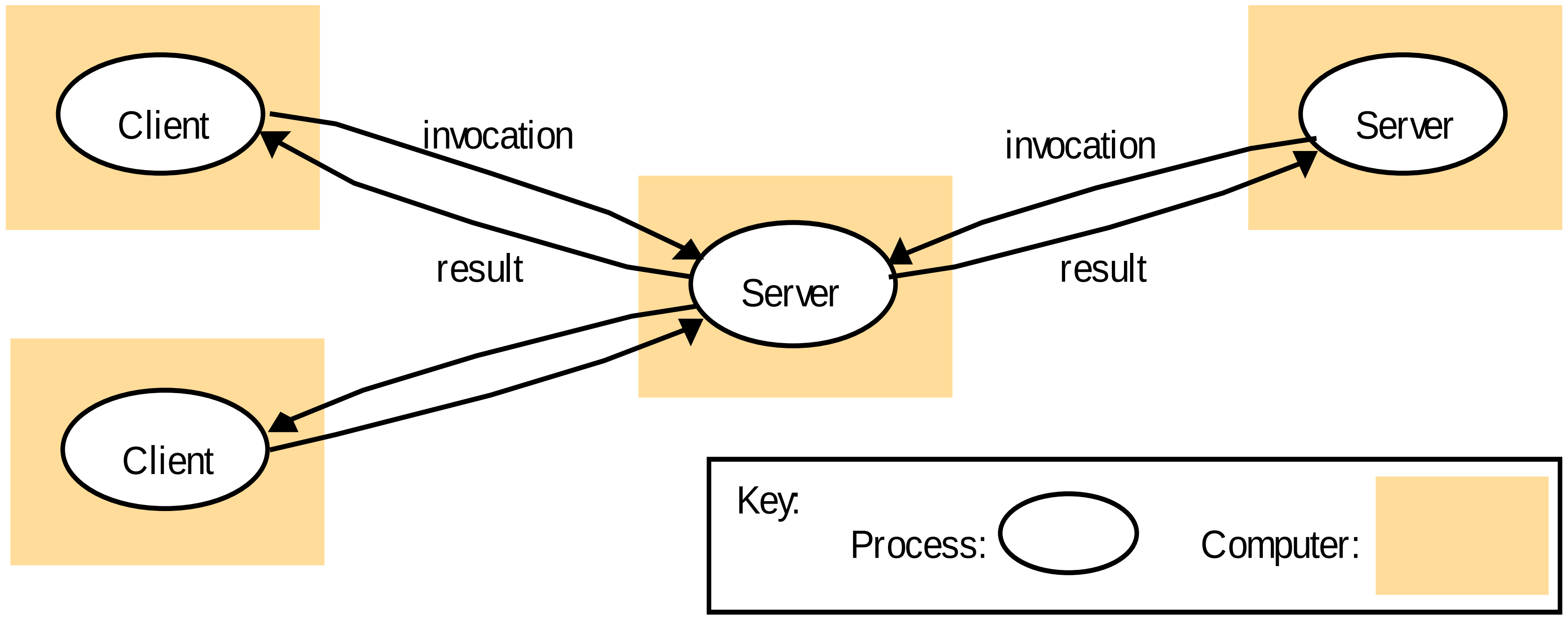
- The client stub has three possibilities for further behavior if the result (reply message) is missing:
    - He can retransmit his request until he receives a correct reply message (the server can be restarted or another server was found). In case that the server crashed after execution of the command, it could be executed twice. This is called „**at least once semantics**“.
    - He can stop after transmitting one message and report an error to the client. This is called „**at most once semantics**“
    - Achieving the “**exactly once semantics**” is not possible in general …

Reminder - Idempotent Operations [Mullender, op. cit.]

- An operation is **idempotent** if
  - Doing it twice has the same effect as doing it once
  - Doing it partially (several times, possibly) and then doing it whole has the same effect as doing it once
- Example: writing a block to disk
  - Doing it partially, results in a bad checksum for the block (so the block becomes unreadable)
  - Doing it whole makes the block readable
  - Doing it again doesn’t matter (it’s the same block again)
- If all RPC actions are idempotent, the RPC semantics **“At least once”** can be used, since every request can be repeated without harm.

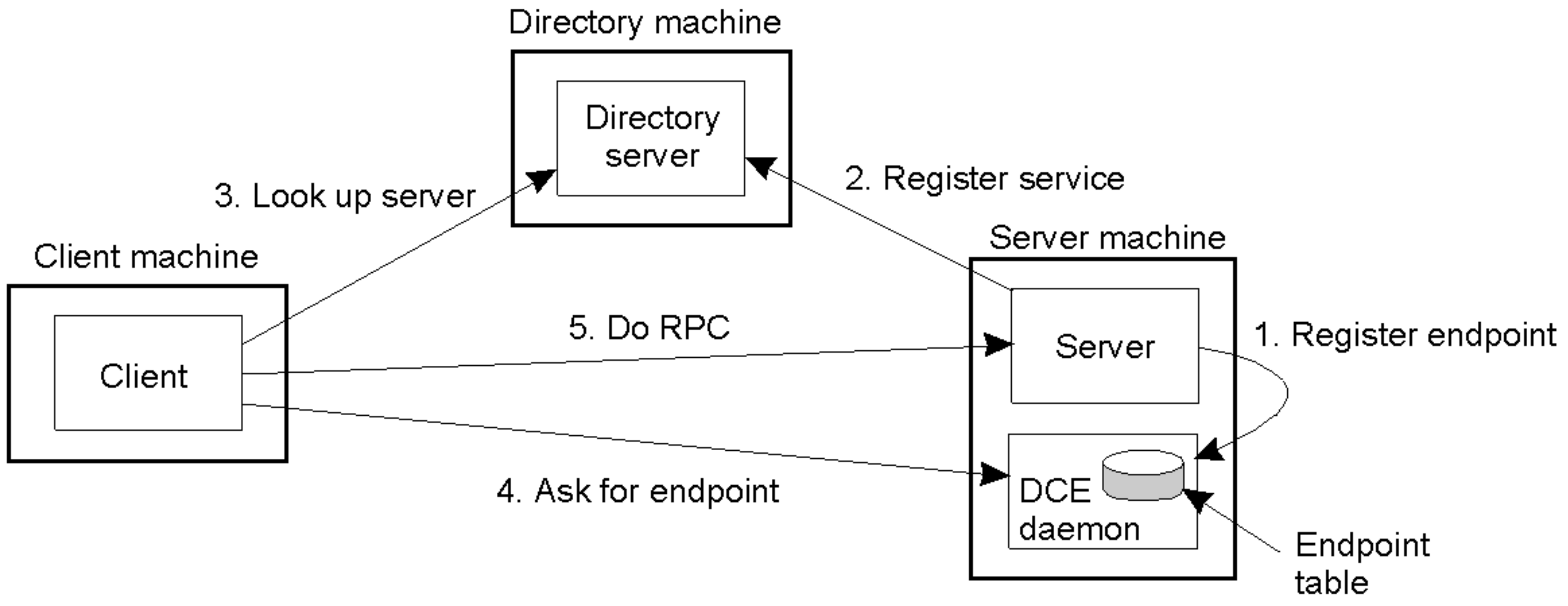
System Architecture: Client-Server Model

- A server process can act as client for an other server
- A service is not bound to a specific computer. How to get to the right one?



Distributed Computing Environment

- Client-to-server binding in Distributed Computing Environment (DCE)
  - Usage of a directory server (like a phone book)



Features of Client-Server Systems

- **Advantages**
    - The work can be *distributed* among different machines
    - The clients can access the server's functionality from a *distance*
    - The client and server can be *designed separately*
    - Simple, well known programming paradigm: call a function!
    - The server can be accessed *simultaneously* by many clients
    - Modification of the server easy …

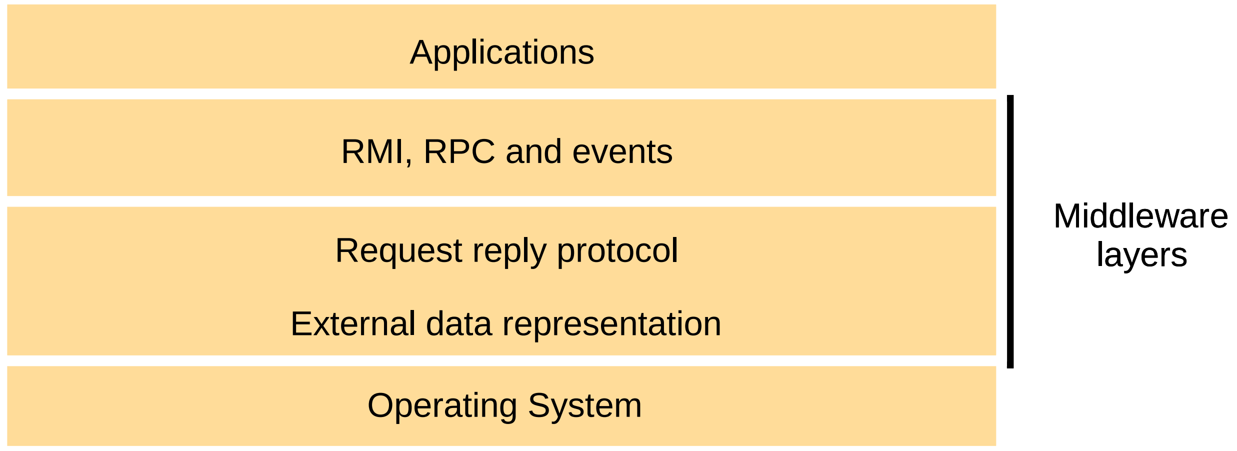
Features of Client-Server Systems (II)

- **Disadvantages**
  - Server can become a bottleneck
  - Latency in communication might be significant
  - Resources (e.g. disc space) of clients might be wasted
  - **How to find a server for a given service?**
  - The service might be disabled if a server fails.

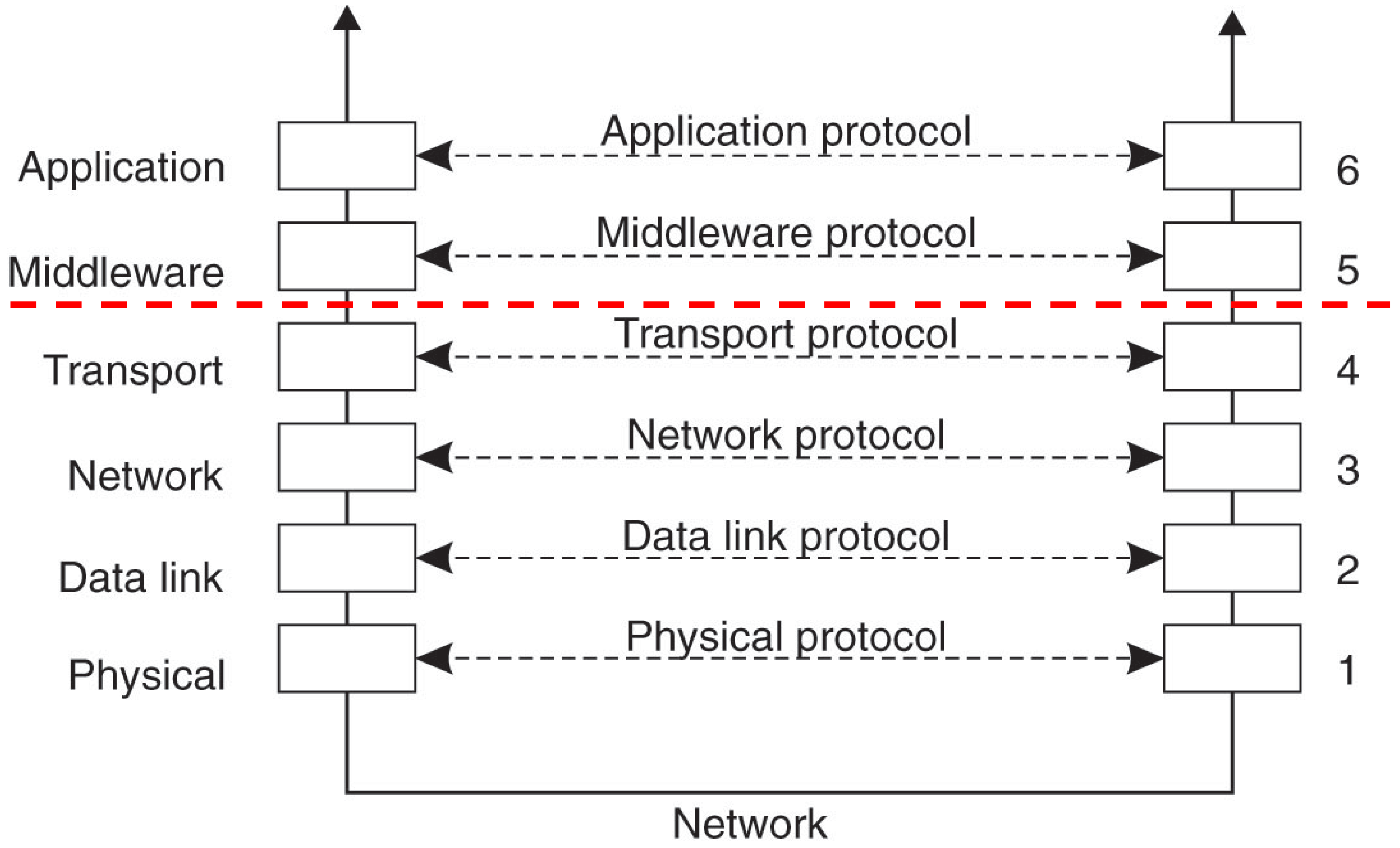
RPC-based Middleware

[Karl, op. cit.]

- **RPCs** present an **abstracter view** of a distributed system than a request/reply protocol directly realized with sockets
  - New programming model!
- Collection of software realizing such a new programming model is a ***middleware***
  - Can achieve, e.g., transparency towards location, communication protocols, hardware, operating system, different programming languages, …



An adapted Communication Model [Tanenbaum, op. cit.]


Application
Middleware
Transport
Network
Data link
Physical
Application protocol
Middleware protocol
Transport protocol
Network protocol
Data link protocol
Physical protocol
6
5
4
3
2
1
Network