
Distributed Systems

Winter Term 2025/26

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 30, 2025



Distributed Systems

Winter Term 2025/26

2 Middleware

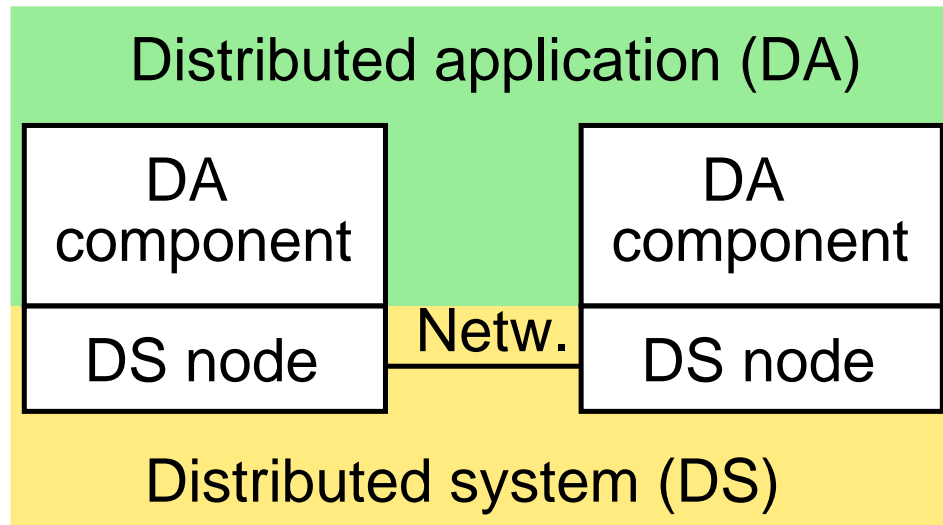


Content

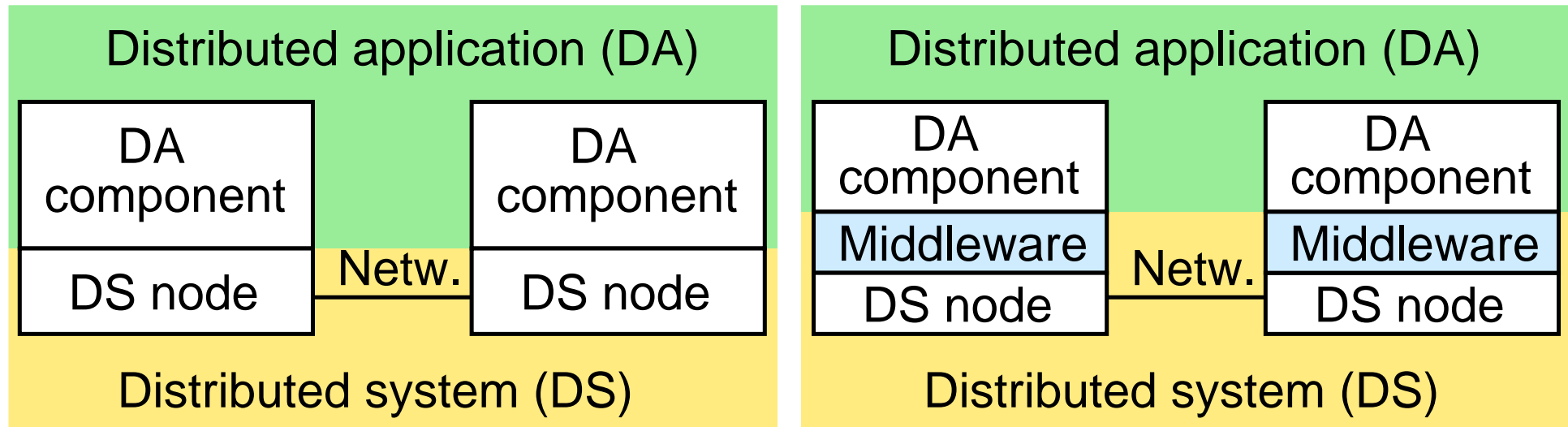
- ➔ Communication in distributed systems
- ➔ Communication-oriented middleware
- ➔ Application-oriented middleware

Literature

- ➔ Hammerschall: Ch. 2, 6
- ➔ Tanenbaum, van Steen: Ch. 2
- ➔ Colouris, Dollimore, Kindberg: Ch. 4.4



- ➔ DA uses DS for communication between its components
- ➔ DSs generally only offer simple communication services
 - ➔ direct use: **network programming**
- ➔ **Middleware** offers more intelligent interfaces
 - ➔ hides details of network programming



- ➔ DA uses DS for communication between its components
- ➔ DSs generally only offer simple communication services
 - ➔ direct use: **network programming**
- ➔ **Middleware** offers more intelligent interfaces
 - ➔ hides details of network programming



- ➔ Middleware is the interface between distributed application and distributed system
- ➔ Goal: hide distribution aspects from application
 - ➔ transparency (👉 1.3)
- ➔ Middleware can also provide additional services for applications
 - ➔ huge differences in existing middleware
- ➔ Distinction:
 - ➔ **communication-oriented middleware** (👉 2.2)
 - ➔ (only) abstraction from network programming
 - ➔ **application-oriented middleware** (👉 2.3)
 - ➔ besides communication, the focus is on support of distributed applications

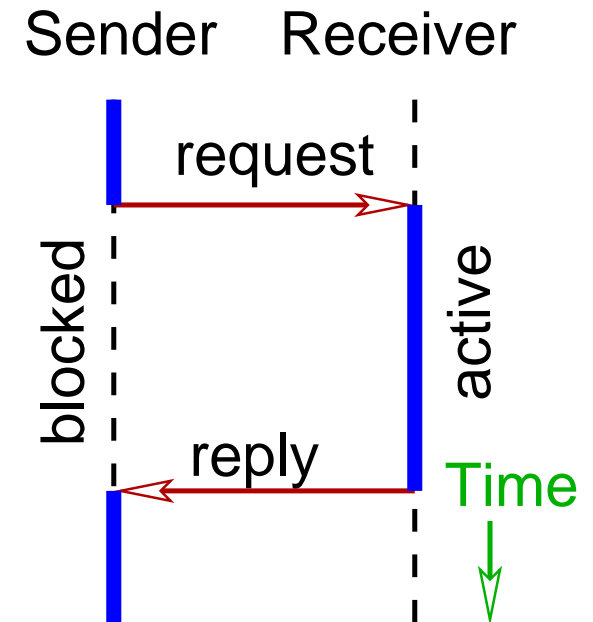
2.1 Communication in Distributed Systems

- ➔ Basis: **interprocess communication (IPC)**
 - ➔ exchange of messages between processes (👉 **BS_I: 3.2**)
 - ➔ on the same or on different nodes
 - ➔ e.g. via ports, mailboxes, streams, ...
- ➔ For distribution: network protocols (👉 **RN_I**)
 - ➔ relevant topics etc: addressing, reliability, guaranteed ordering, timeouts, acknowledgements, marshalling
- ➔ Interface for network programming: sockets (👉 **RN_II**)
 - ➔ datagrams (UDP) and streams (TCP)



Synchronous Communication

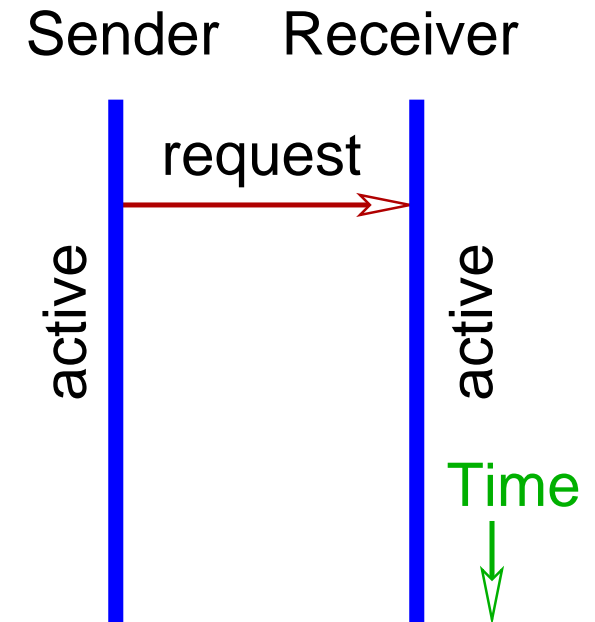
- ➔ Sender and receiver block when calling a send or receive operation
 - ➔ receiver is waiting for a request
 - ➔ sender is waiting for the reply
- ➔ Tight coupling between sender and receivers
 - ➔ advantage: easy to understand model
 - ➔ disadvantage: strong dependency, especially in case of error
- ➔ Prerequisites:
 - ➔ reliable and fast network connection
 - ➔ receiver process is available



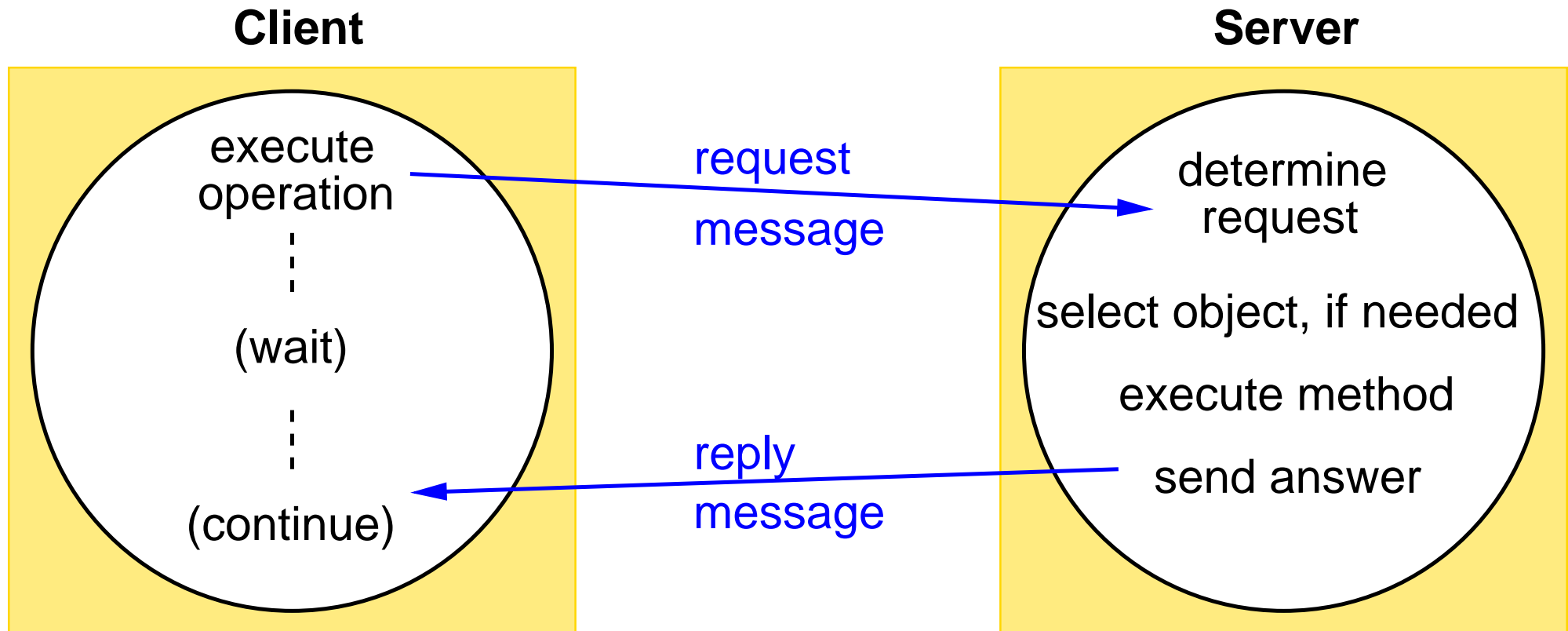


Asynchronous Communication

- ➔ Sender is not blocked, can continue immediately after sending the message
- ➔ Incoming messages are buffered at the receiver
- ➔ Answers are optional
 - ➔ receiver can reply asynchronously to the sender
- ➔ More complex implementation and use as with synchronous communication, but usually more efficient
- ➔ Only loose coupling between the processes
 - ➔ receiver does not have to be ready for reception
 - ➔ less dependent in case of errors

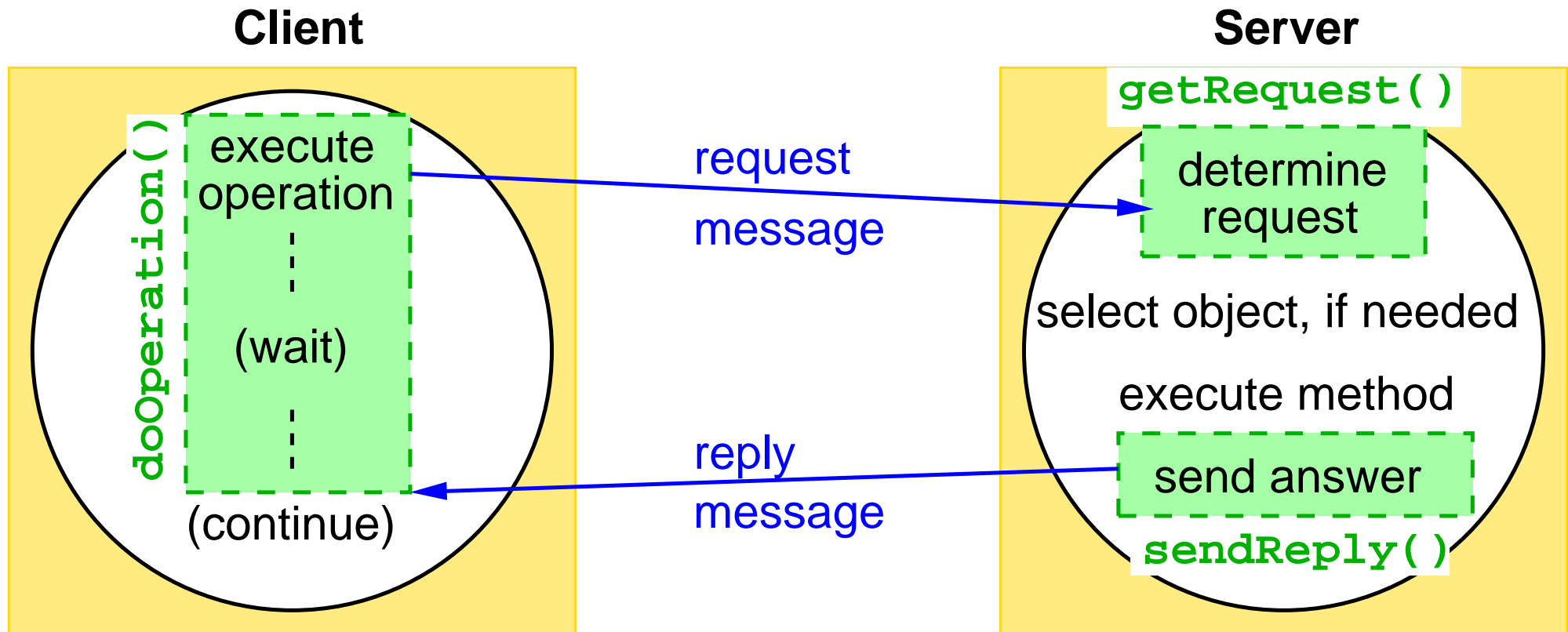


Client/Server Communication



- ➔ Mostly synchronous: client blocked until response arrives
- ➔ Variants: asynchronous (non blocking), one way (without answer)

Client/Server Communication



- ➔ Mostly synchronous: client blocked until response arrives
- ➔ Variants: asynchronous (non blocking), one way (without answer)



Client/Server Communication: Request/Response Protocol

➔ Typical operations:

- ➔ `doOperation()` – send request and wait for result
- ➔ `getRequest()` – wait for request
- ➔ `sendReply()` – send result

➔ Typical message structure:

messageType	request / reply ?
requestID	unique ID of request (usually int)
objectReference	reference to remote object (if needed)
methodID	method to be called (int / String)
arguments	arguments (usually as Byte array)

- ➔ request ID + sender ID result in unique message ID
 - ➔ e.g. to map an answer to its query



Client/Server Communication: Error Handling

- ➔ Both request and/or response messages may be lost
- ➔ Client sets a timeout when sending a request
 - ➔ after expiration, request is usually sent again
 - ➔ after a few repetitions: termination with exception
- ➔ Server discards duplicate requests if request has already been / is still being processed
- ➔ For lost response messages:
 - ➔ idempotent operations can be executed again
 - ➔ otherwise: save results of operations in a history
 - ➔ for repeated request: only resend the result
 - ➔ delete history entries when next request arrives; if necessary confirmations for results can also be used



Client/Server Communication: Semantics

➔ At most once

- ➔ request is executed **at most once** under all circumstances
- ➔ lost request (or answer) is not handled

➔ At least once

- ➔ request is executed **at least once** under all circumstances
- ➔ lost request or answer leads to a repetition of the request
- ➔ useful e.g. for idempotent requests

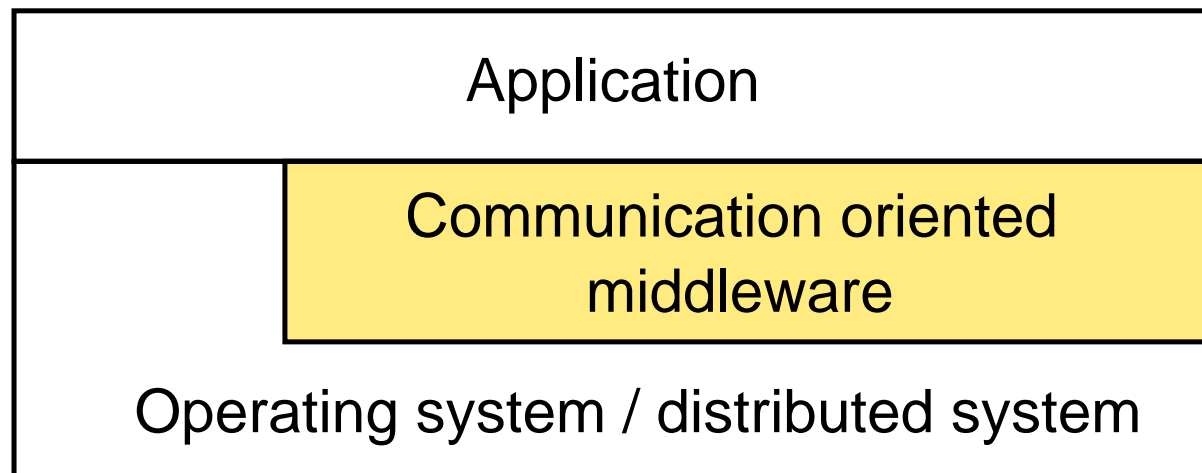
➔ Exactly once

- ➔ request is executed **exactly once** under all circumstances
- ➔ lost request or answer leads to a repetition of the request
- ➔ server must be able to handle duplicates

2.2 Communication-oriented Middleware



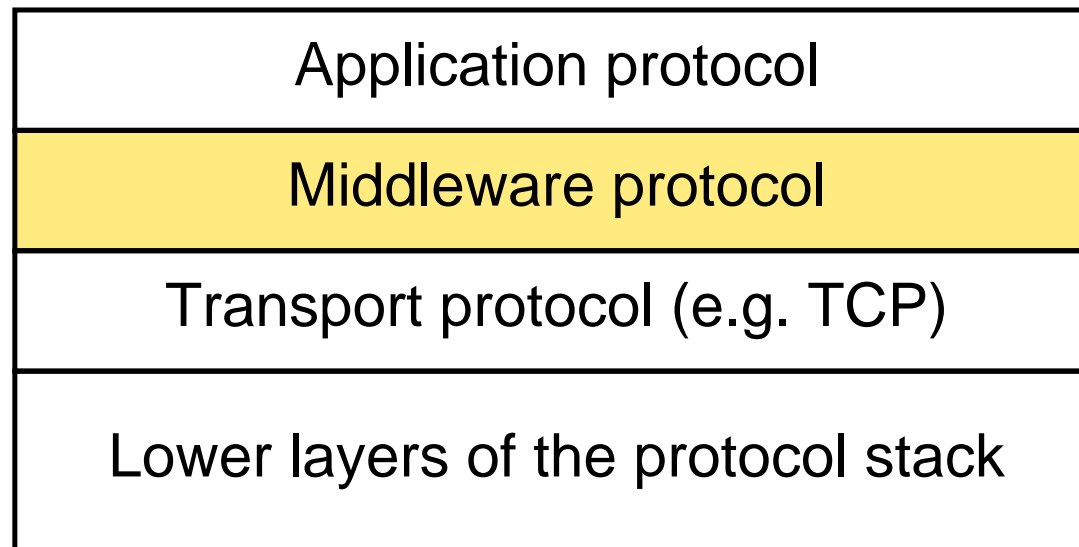
- ➔ Focus: provision of a communication infrastructure for distributed applications
- ➔ Tasks:
 - ➔ communication
 - ➔ dealing with heterogeneity
 - ➔ error handling





Communication

- ➔ Provision of a middleware protocol
- ➔ Localization and identification of communication partners
- ➔ Integration with process and thread management





Heterogeneity

- ➔ Problem with data transmission:
 - ➔ heterogeneity in distributed systems
- ➔ Heterogeneous hardware and operating systems
 - ➔ different byte order
 - ➔ little endian vs. big endian
 - ➔ different character encoding
 - ➔ e.g.. ASCII / Unicode / UTF-8 / UTF-16
- ➔ Heterogeneous programming languages
 - ➔ different representation of simple and complex data types in the main memory



Heterogeneity: Solutions (👉 RN_I)

- ➔ Use of generic, standardized data formats
 - ➔ known to all communication partners and middleware
 - ➔ platform-specific formats for middleware (e.g. CDR for CORBA) or external formats, e.g. XML
- ➔ Heterogeneity of hardware and operating system
 - ➔ is handled transparently for the applications by the middleware
- ➔ Heterogeneity of programming languages
 - ➔ applications need to convert data to higher-level format and back (**marshaling** / **unmarshaling**)
 - ➔ necessary code is usually generated automatically
 - ➔ client stub / server skeleton



Error Handling

- ➔ Possible errors due to distribution
 - ➔ incorrect transmission (incl. loss of messages)
 - ➔ handled by the protocols of the distributed system:
 - ➔ checksums, CRC
 - ➔ retransmission of packets (e.g. TCP)
 - ➔ failure of components (network, hardware, software)
 - ➔ handled by middleware or application:
 - ➔ acceptance of the error
 - ➔ retransmission of messages
 - ➔ replication of components (error avoidance)
 - ➔ controlled termination of the application



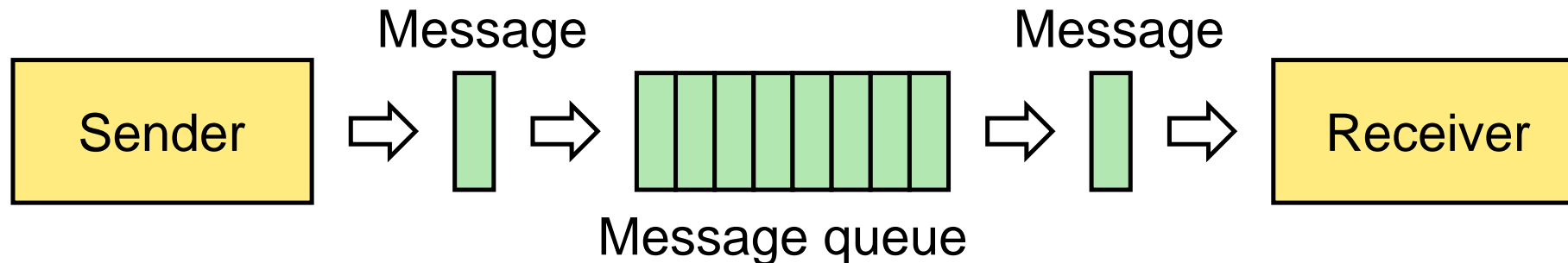
2.2.2 Programming Models

- ➔ Programming model defines two concepts:
 - ➔ communication model
 - ➔ synchronous vs. asynchronous
 - ➔ programming paradigm
 - ➔ object-oriented vs. procedural

- ➔ Three common programming models for middleware:
 - ➔ message-oriented model (asynchronous / arbitrary)
 - ➔ remote procedure call (synchronous / procedural)
 - ➔ remote method invocation (synchronous / object-oriented)

Message-Oriented Model

- ➔ Sender puts message in receiver's queue



- ➔ Receiver accepts message as soon as he is ready
- ➔ Extensive decoupling of transmitter and receiver
- ➔ No method or procedure calls
 - ➔ data is packed and sent by the application
 - ➔ no automatic reply message

Distributed Systems

Winter Term 2025/26

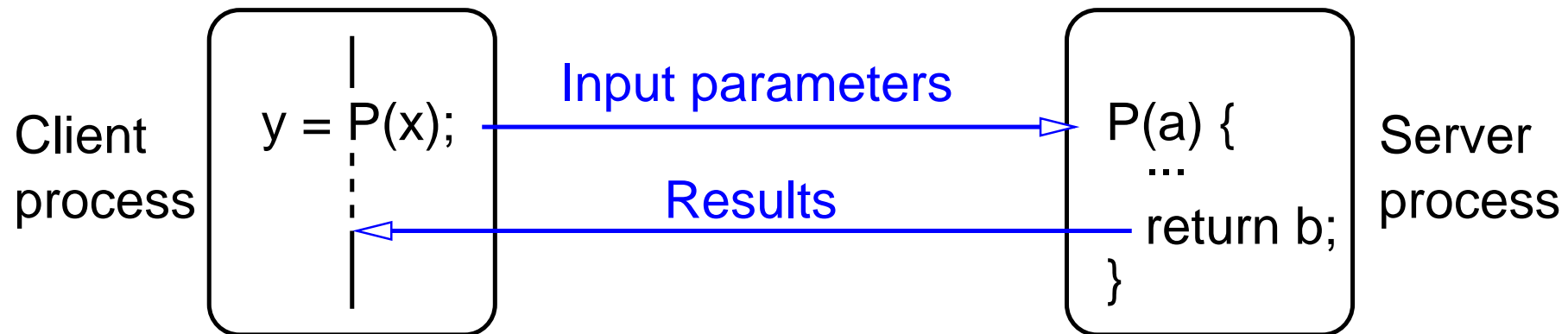
30.10.2025

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: October 30, 2025

Remote Procedure Call (RPC)

- ➔ Allows a client to call a procedure in a remote server process



- ➔ Communication according to request / response principle

Remote Method Invocation (RMI)

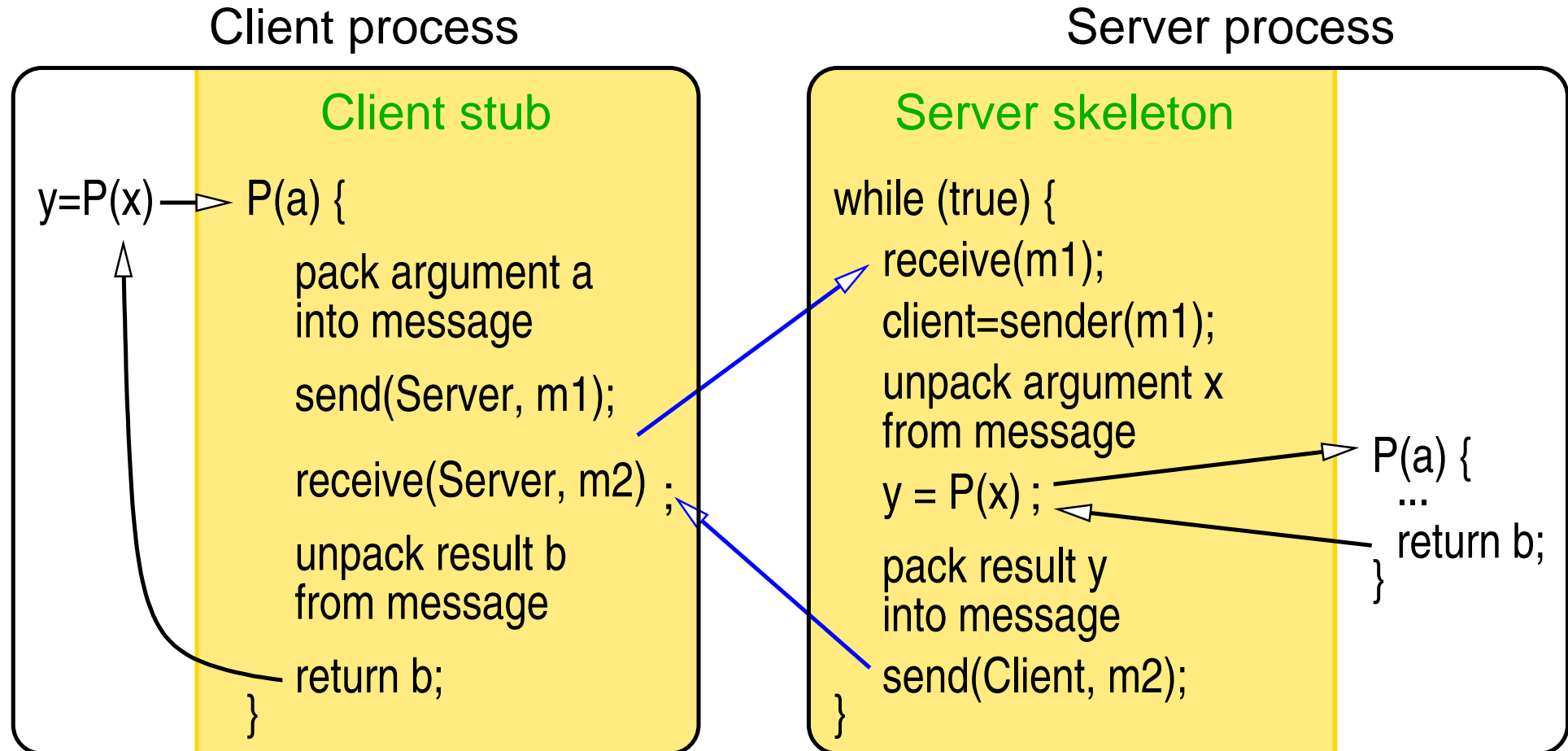
- ➔ Allows an object to call methods of a remote object
- ➔ In principle very similar to RPC



Common Basic Concepts of Remote Calls

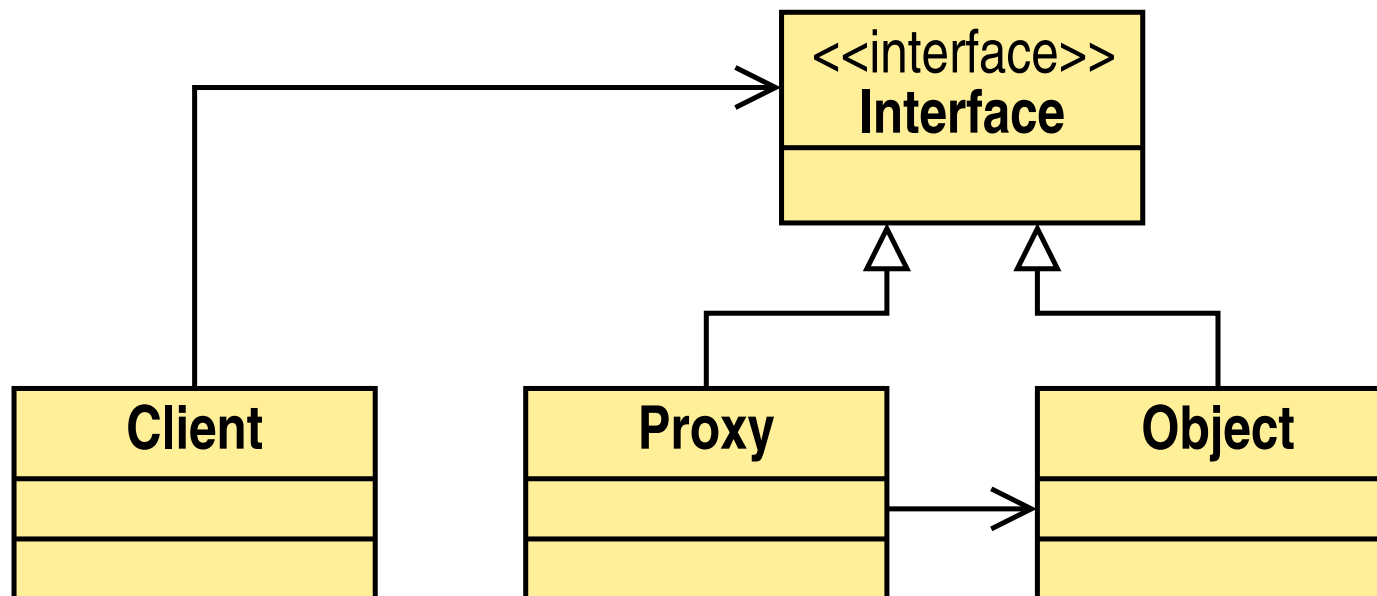
- ➔ Client and server are decoupled by interface definition
 - ➔ defines names of calls, parameters and return values
- ➔ Introduction of **client stubs** and **server skeletons** as an access interface
 - ➔ are automatically generated from interface definition
 - ➔ IDL compiler (IDL = interface definition language)
 - ➔ are responsible for marshaling / unmarshaling as well as for the actual communication
 - ➔ realize access and location transparency

How Client Stub and Server Skeleton Work (RPC)

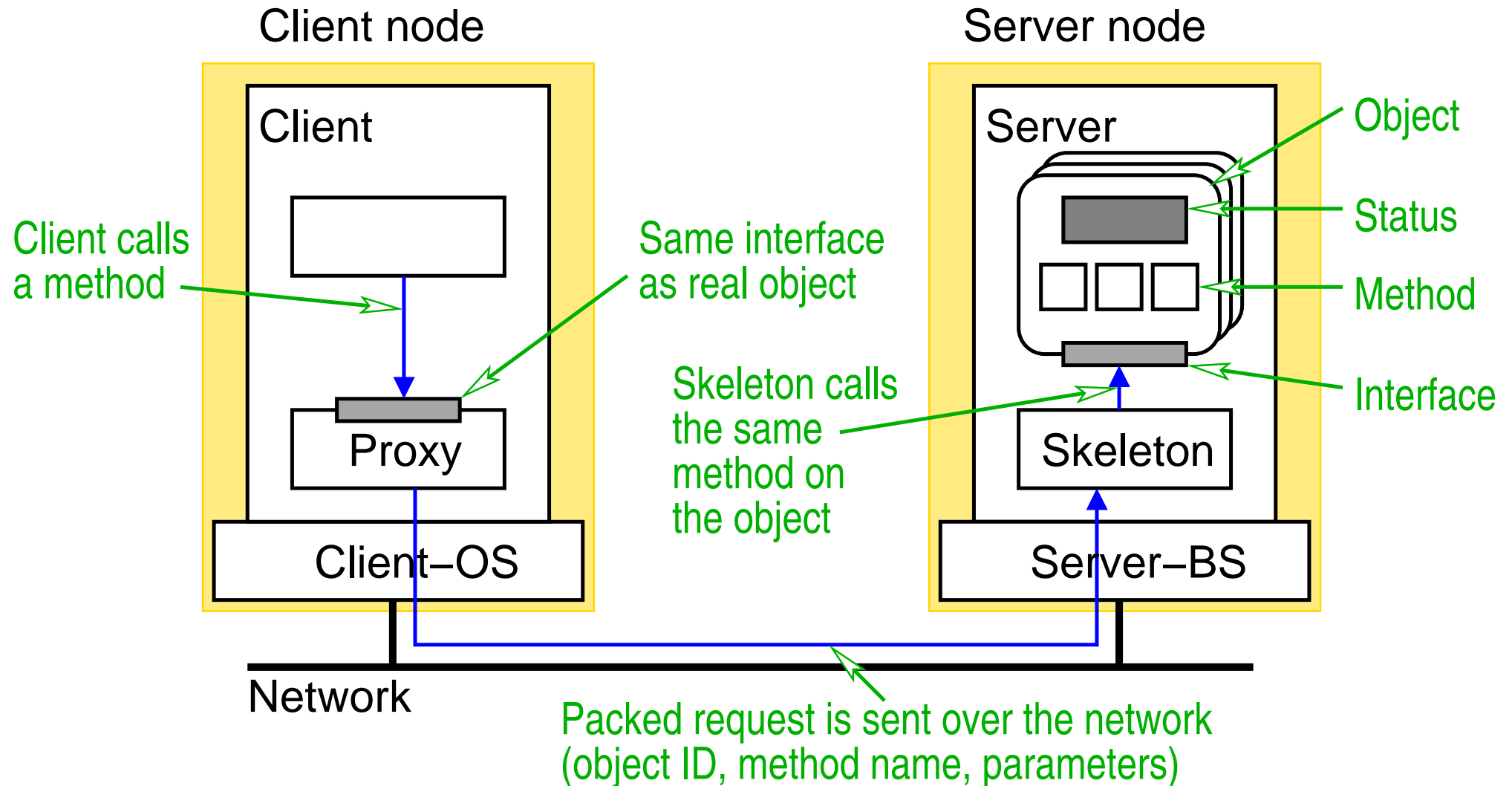


Basis of RMI: The Proxy Pattern

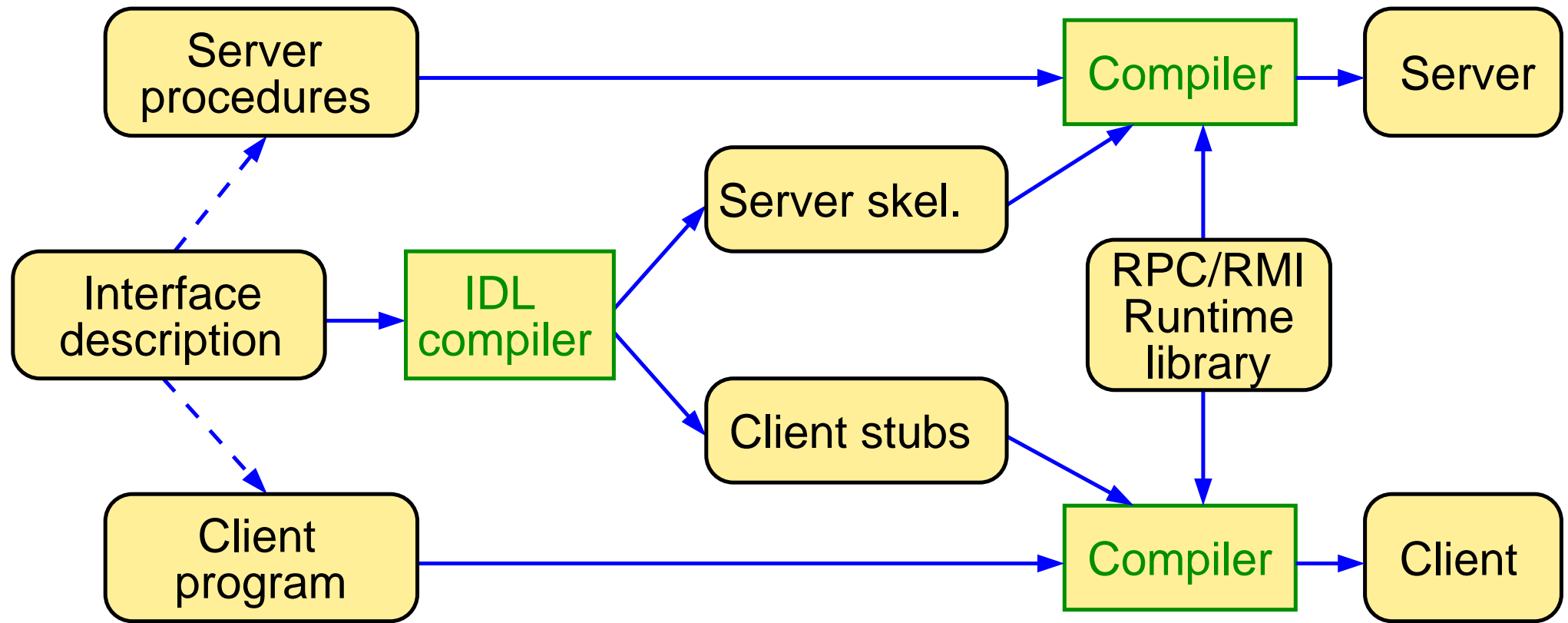
- ➔ Client works with a deputy object (**proxy**) of the actual server object
- ➔ proxy and server object implement the same interface
- ➔ client only knows / uses this interface



Flow of a Remote Method Call



Creation of a Client/Server Program



➔ Applies in principle to all realizations of remote calls



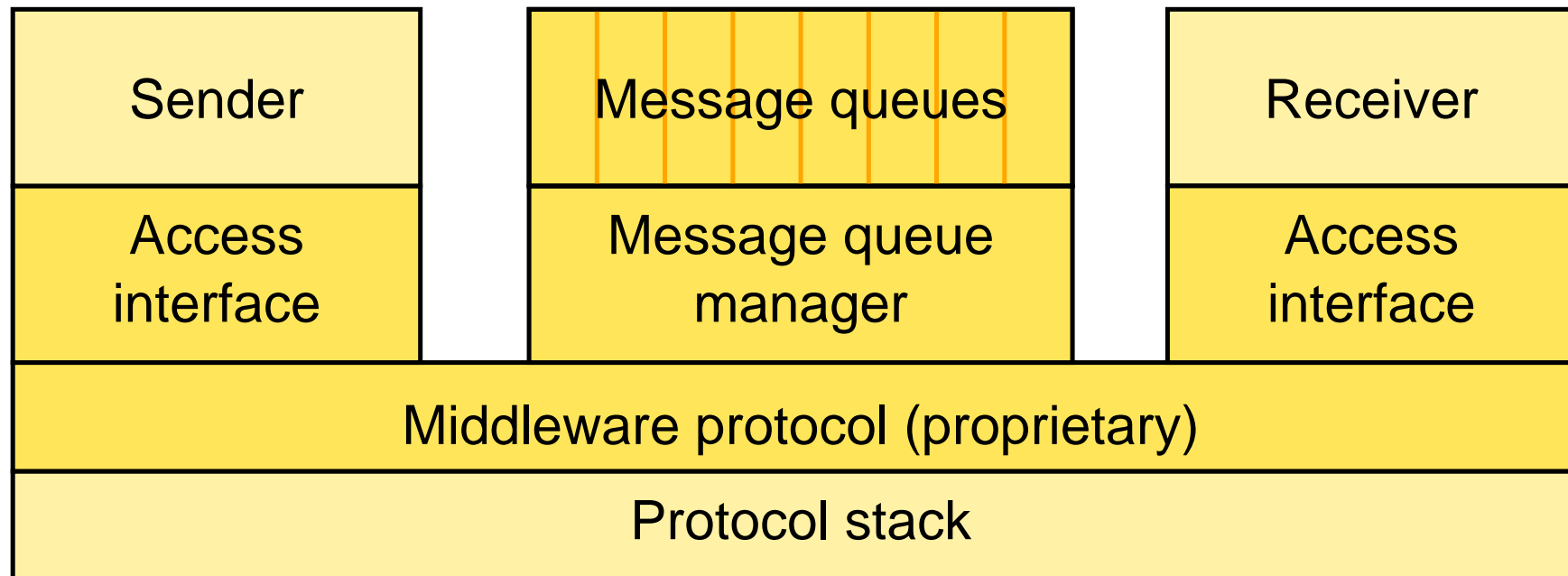
2.2.3 Middleware Technologies

- ➔ Realize (at least) one of the programming models
 - ➔ rely on open standards / standardized interfaces
- ➔ Remote procedure call
 - ➔ SUN RPC, DCE RPC, Web Services, gRPC (👉 **3.2**), ...
- ➔ Remote method invocation
 - ➔ Java RMI (👉 **3.1**), CORBA, ...
- ➔ Message-oriented middleware technologies
 - ➔ MOM: message oriented middleware, messaging systems
 - ➔ mainly for EAI
 - ➔ Java Message Service, WebSphereMQ (MQSeries), ...



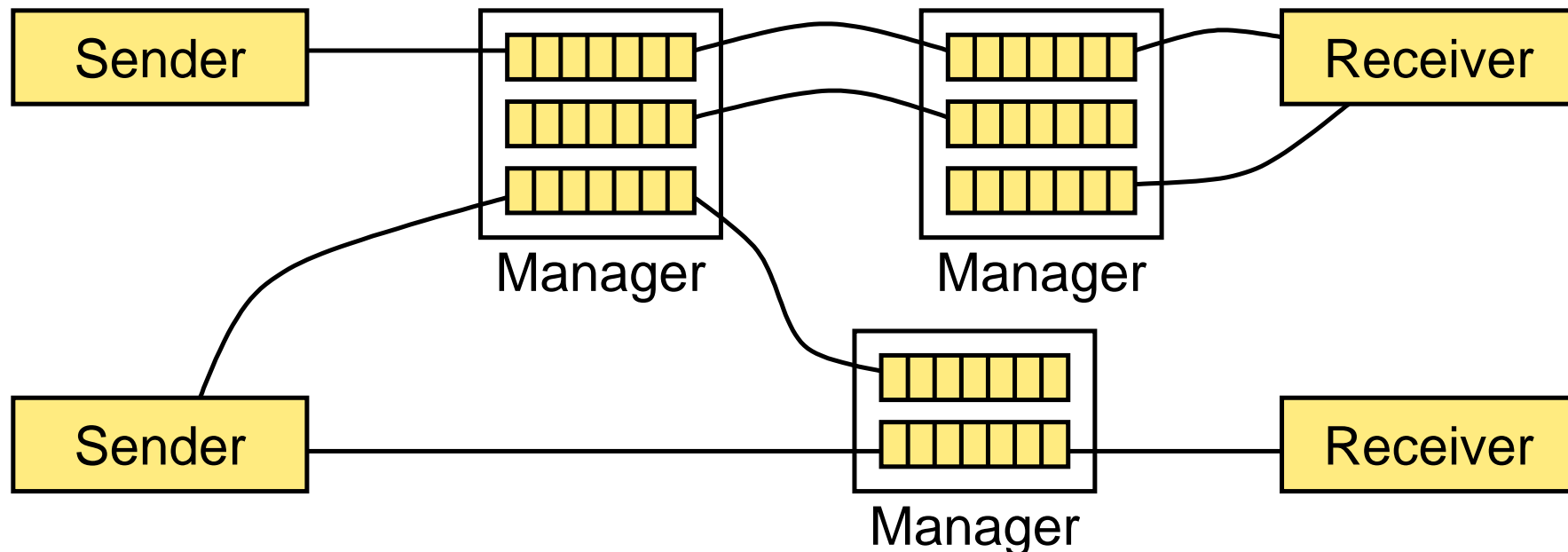
2.2.4 Message Oriented Middleware (MOM)

- ➔ Middleware technology for the message-oriented model
- ➔ In addition to message exchange also other services, especially queue management



Message Queue Infrastructure

- ➔ Access to queues is only possible locally
 - ➔ local: same computer or same subnet
- ➔ Transport of messages across subnet boundaries by queue administrators (routers)





Variants of message exchange

- ➔ Point-to-point communication
 - ➔ communication between two defined processes
 - ➔ simplest model: asynchronous communication
 - ➔ enhancement: request/reply model
 - ➔ enables synchronous communication via asynchronous middleware
- ➔ Broadcast/multicast communication
 - ➔ message is sent to all reachable receivers
 - ➔ one implementation: publish/subscribe model
 - ➔ publishers publish messages/news on a topic
 - ➔ subscribers subscriber to certain topics
 - ➔ mediation via a broker



Example: Java Message Service

- ➔ Part of the Java Enterprise Edition (Java EE)
- ➔ Unified Java interface for MOM services
- ➔ Distinguishes two roles:
 - ➔ JMS provider: the respective MOM server
 - ➔ JMS client: sender or receiver of messages
- ➔ JMS supports:
 - ➔ asynchronous point-to-point communication
 - ➔ request/reply model
 - ➔ publish/subscribe model
- ➔ JMS defines corresponding access objects and methods



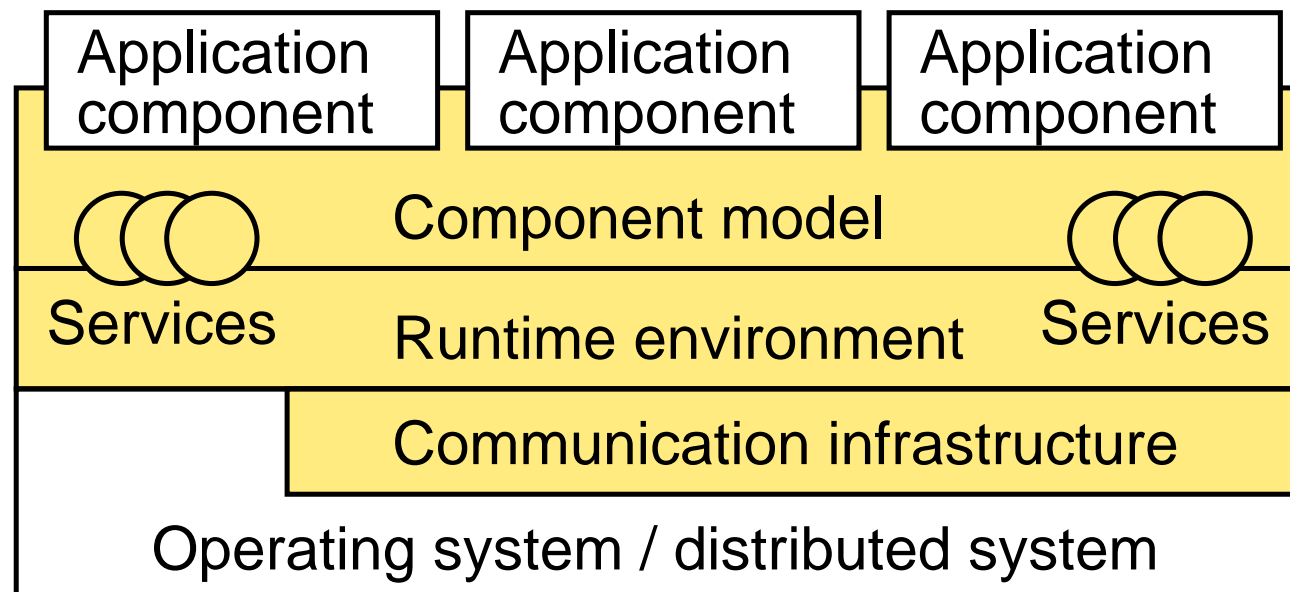
2.2.5 Summary

- ➔ Tasks: Communication, dealing with heterogeneity, error handling
- ➔ Programming models:
 - ➔ message-oriented model (asynchronous)
 - ➔ basis: message queues
 - ➔ refinements:
 - ➔ request/reply model (synchronous)
 - ➔ publish/subscribe model (broadcast)
 - ➔ remote procedure or method calls
 - ➔ synchronous: request and response
 - ➔ generated stubs for (un-)marshaling

2.3 Application-oriented Middleware



- ➔ Based on communication-oriented middleware
- ➔ Extends it by:
 - ➔ runtime environment
 - ➔ services
 - ➔ component model





- ➔ Based on node operating systems of the distributed system
 - ➔ Operating system (OS) manages processes, memory, I/O, ...
 - ➔ provides basic functionality
 - ➔ starting / stopping processes, scheduling, ...
 - ➔ interprocess communication, synchronization, ...
- ➔ Runtime environment extends functionality of the OS:
 - ➔ improved resource management
 - ➔ e.g. concurrency, connection management
 - ➔ improved availability
 - ➔ improved security mechanisms



Resource management

- ➔ Middleware goes beyond simple OS functionality
 - ➔ e.g. independently managed main memory areas with individual security criteria
 - ➔ pooling of processes, threads, connections
 - ➔ are created for stock and made available as required
 - ➔ possible, since middleware is specific to certain classes of applications
- ➔ Goal: improved performance, scalability and availability



Concurrency

- ➔ Concurrency in this context:
 - ➔ isolated parallel processing of requests
- ➔ Concurrency can be implemented via processes or threads
 - ➔ threads (lightweight processes): concurrent activities within processes
 - ➔ threads in the same process share all resources
 - ➔ advantages and disadvantages:
 - ➔ processes: high resource requirements, not well scalable, good protection, with low concurrency
 - ➔ threads: well scalable, no mutual protection, with high concurrency



Concurrency ...

- ➔ Middleware takes over automatic generation / administration of threads in the case of concurrent orders, e.g.
 - ➔ *single-threaded*
 - ➔ only one thread, sequential processing
 - ➔ *thread-per-request*
 - ➔ a new thread is created for each request
 - ➔ *thread-per-session*
 - ➔ a new thread is created for each session (client)
 - ➔ *thread pool*
 - ➔ fixed number of threads, incoming requests are distributed automatically
 - ➔ saves thread generation costs
 - ➔ limits resource consumption



Connection management

- ➔ Connection here means: endpoints of communication channels
 - ➔ occur at tier boundaries (between process spaces)
 - ➔ e.g. client/server interface, database access
 - ➔ are assigned to a process/thread, if in the active state
 - ➔ require resources (memory, processor time)
 - ➔ opening and closing connections is costly
- ➔ To save resources: pooling of connections
 - ➔ connections are initialized to stock and placed in pool
 - ➔ each thread/process receives a connection on demand
 - ➔ after use: return connection to pool



Availability

- ➔ Requirement to the application,
but mainly implemented by the runtime environment
- ➔ Downtimes are caused by
 - ➔ failure of a hardware or software component
 - ➔ overload of a hardware or software component
 - ➔ maintenance of a hardware or software component
- ➔ Frequent technology for ensuring availability: cluster
 - ➔ replication of hardware and software
 - ➔ cluster appears externally as one unit
 - ➔ two types: fail-over cluster / load-balancing cluster



Security

- ➔ Distributed applications are vulnerable due to their distribution
- ➔ Middleware supports different security models
- ➔ Security requirements:
 - ➔ **authentication:**
 - ➔ proves the identity of the user / a component
 - ➔ e.g. by password query (for users) or cryptographic techniques and certificates (for components)
 - ➔ **authorization:**
 - ➔ definition of access rights for users to specific services
 - ➔ or more fine grained: methods and attributes
 - ➔ requires secure authentication



Security ...

➔ Security requirements ...:

➔ **confidentiality**

- ➔ information cannot be intercepted during transmission in the network
- ➔ technique: encryption

➔ **integrity**

- ➔ transmitted data cannot be changed without being noticed
- ➔ techniques: cryptographic checksum (message digest, fingerprint), digital signature
 - ➔ digital signature also ensures authenticity of the sender



Security ...

- ➔ Security mechanisms:
 - ➔ encryption
 - ➔ symmetric (e.g. AES, IDEA)
 - ➔ same key for encryption and decryption
 - ➔ asymmetric (public key algorithms, e.g. RSA)
 - ➔ public key for encryption
 - ➔ private key for decrypting
 - ➔ digital signature
 - ➔ ensures integrity of a message and authenticity of the sender as well as nonrepudiation
 - ➔ certificate
 - ➔ certifies that public key and person (or component) belong together

Name service (directory service) (👉 4)

- ➔ Publication of available services
 - ➔ in the intranet or Internet
- ➔ Assignment of names to references (addresses)
 - ➔ name serves as a unique / unchangeable identifier
 - ➔ the client can request the address of a service via its name
 - ➔ address can change e.g. at restart
 - ➔ goal: decoupling of client and server
- ➔ Examples: JNDI, RMI registry, CORBA interoperable naming service, UDDI registry, LDAP server, Active Directory, ...



Session management

- ➔ In interactive systems: each instance of a client is assigned its own **session**
 - ➔ deleted when logging out or closing the client
- ➔ Session stores all relevant data (in main memory)
 - ➔ e.g. identification of the user, browser type, "shopping cart", ...
 - ➔ data stored in the server or in the client
 - ➔ transient data: deleted at the end of the session
 - ➔ persistent data: is written to a data carrier (database) at the end of the session.
- ➔ Middleware implements/supports the assignment of requests to sessions (often transparent)
 - ➔ e.g. cookies, HTTP-sessions, session beans, ...



Transaction management (👉 8.4)

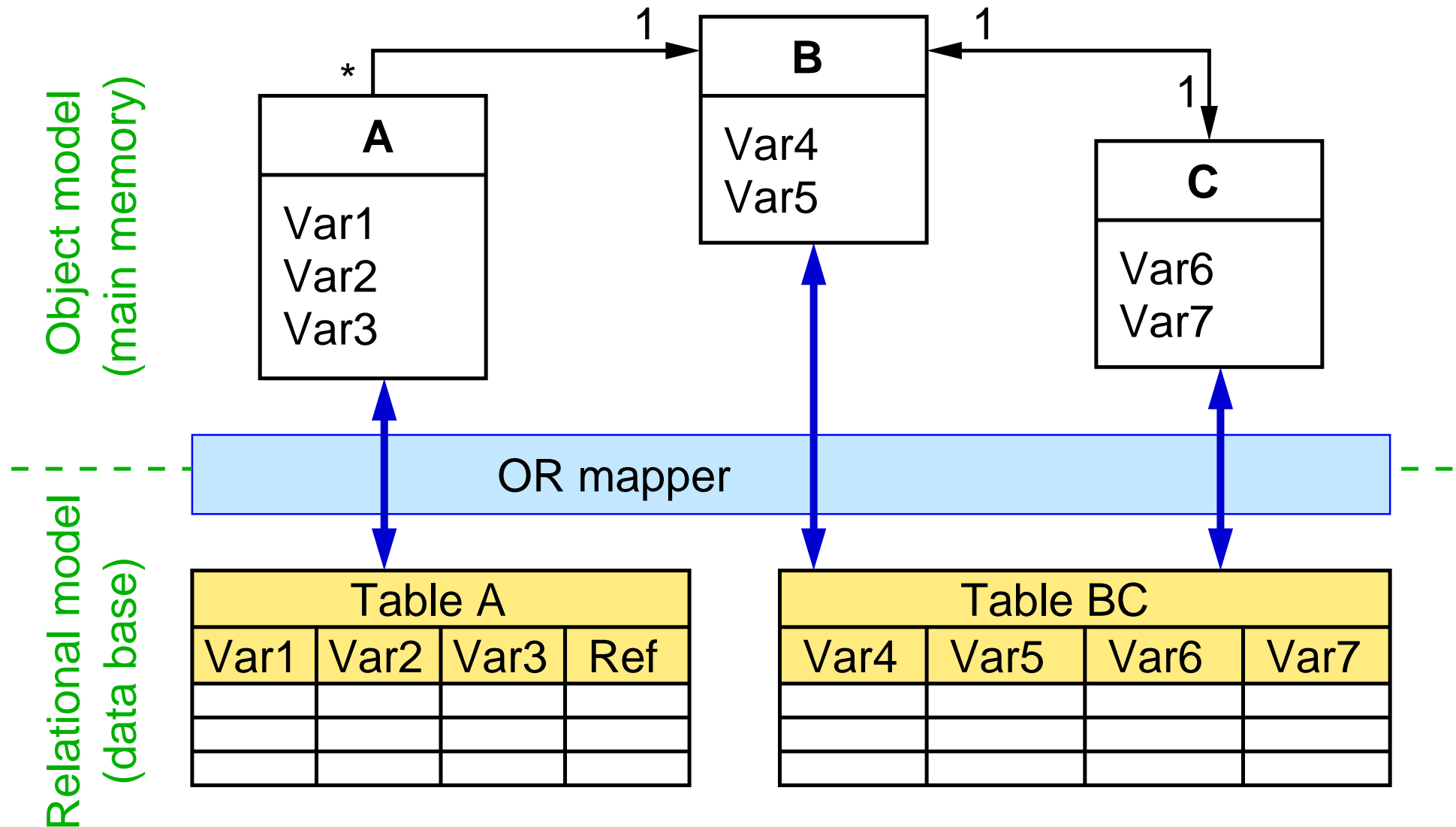
- ➔ Service for interactive, data-centric applications
 - ➔ consistency / integrity of data is important
 - ➔ this means that the entire (maybe distributed) dataset must represent a valid state in itself
- ➔ Typical sequence in applications:
 1. client requests data
 2. client changes the data
 3. client requests that the data be rewritten
 - ➔ problem: steps 1-3 could be performed by two clients at the same time
- ➔ Transaction management allows execution of a sequence of actions as an atomic unit



Persistence service

- ➔ Persistence: all measures for the permanent storage of main memory data
- ➔ Persistence service: intelligent interface to the database
 - ➔ integrated in middleware or as an independent component
 - ➔ most important service for data-centered applications besides transaction management
- ➔ Most common type: object-relational mapper (OR-Mapper)
 - ➔ maps objects in memory to tables in a relational database
 - ➔ class → table
 - ➔ attribute → column
 - ➔ object → row
 - ➔ mapping rules are controlled by application developer

Persistence service ...



2.3.3 Component model



- ➔ Components: “large” objects for structuring applications
- ➔ A component model defines:
 - ➔ the term “component”
 - ➔ structure and properties of the components
 - ➔ mandatory and optional interfaces
 - ➔ interface contracts
 - ➔ how do components interact with each other and with the runtime environment?
 - ➔ component runtime environment
 - ➔ management of the life cycle of components
 - ➔ implicit provision of services: component only specifies its requirements (e.g. persistence)



- ➔ Object request broker (ORB)
 - ➔ distributed objects, remote method calls
 - ➔ variety of services, only basic runtime environment
 - ➔ example: CORBA

- ➔ Application server
 - ➔ focus: support of application logic (middle tier)
 - ➔ services, runtime environment, and component model
 - ➔ today only as part of a middleware platform

- ➔ Middleware platforms
 - ➔ extension of application servers: support of all tiers
 - ➔ distributed applications as well as EAI
 - ➔ examples: Java EE/EJB, .NET/COM, CORBA 3.0/CCM



Application-oriented middleware

- ➔ Runtime environment
 - ➔ resource management, availability, security
- ➔ Services
 - ➔ name service, session management, transaction management, persistence service
- ➔ Component model
 - ➔ definition of components, interface contracts, runtime environment