



Distributed Systems

Winter Term 2025/26

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

Stand: January 8, 2026



Distributed Systems

Winter Term 2025/26

0 Organisation

- ➔ Studies in Computer Science, Techn. Univ. Munich
 - ➔ Ph.D. in 1994, state doctorate in 2001
- ➔ Since 2004 Prof. for Operating Systems and Distributed Systems
- ➔ **Research:** Secure component based systems; Pattern recognition in network data; Parallel and distributed systems
- ➔ Head of Examination Board
- ➔ **E-mail:** roland.wismueller@uni-siegen.de
- ➔ **Tel.:** 0271/740-4050
- ➔ **Room:** H-B 8404
- ➔ **Office Hour:** Mon., 14:15-15:15



Andreas Hoffmann

andreas.hoffmann@uni-...

0271/740-4047

H-B 8405

- ➔ E-assessment and e-labs
- ➔ IT security
- ➔ Web technologies
- ➔ Mobile applications



Felix Breitweiser

felix.breitweiser@uni-...

0271/740-4719

H-B 8406

- ➔ Operating systems
- ➔ Programming languages
- ➔ Virtual machines



Sven Jacobs

sven.jacobs@uni-...

0271/740-2533

H-B 8407

- ➔ E-assessment and e-labs
- ➔ Generative artificial intelligence
- ➔ Web technologies

Lectures/Labs

- ➔ Rechnernetze I, 6 CP (Bachelor, summer term)
- ➔ Rechnernetze Praktikum, 6 CP (Bachelor, winter term)
- ➔ Rechnernetze II, 6 CP (Master, summer term)

- ➔ Betriebssysteme und nebenläufige Programmierung, 6 CP (Bachelor, summer term)
- ➔ Parallel processing, 6 CP (Master, winter term)
- ➔ Distributed systems, 6 CP (Bachelor, winter term)



Project Groups

- ➔ e.g., secure cooperation of software components
- ➔ e.g., concepts for secure management of Linux-based thin clients

Theses (Bachelor, Master)

- ➔ Topic areas: secure virtual machine, parallel computing, pattern recognition in sensor data, e-assessment, ...

Seminars

- ➔ Topic areas: IT security, programming languages, pattern recognition in sensor data, ...
- ➔ Procedure: block seminar (30 min. talk, 5000 word paper)
- ➔ Master: attend the lecture “Scientific Working” beforehand!
 - ➔ block course end of Feb. / beginning of March



➔ Lecture:

➔ Thursday, **08:30 - 10:00**, room H-C 6321

➔ Exercises:

➔ Thursday, 10:15-11:45, room H-C 6321

➔ start: 30.10.2025

➔ includes programming exercises (Java, python)



Information, Slides and Announcements

➔ On the course's webpage:

<http://www.bs.informatik.uni-siegen.de/lehre/vs>

➔ If necessary, updates/supplements shortly before the lecture

➔ look at the date!

➔ Exercise sheets will be put online as PDF

➔ please print and process them yourself!

➔ There is also a moodle course

➔ submission of mandatory exercise solutions

➔ self tests

Registration for “Course Achievement” (Studienleistung)

- ➔ Passing the course requires successful completion of homework:
 - ➔ 10 exercise sheets with two mandatory exercises
 - ➔ 6 exercise sheets must be successfully processed
- ➔ You must register for “4INFBA303-S - Coursework Distributed Systems” **before you can submit a solution!** (do it right now!)
 - ➔ independent of the registration to the course and the lab!
 - ➔ if you cannot complete the course work: **deregister** again!



The screenshot shows a list of courses in a registration system. The courses are:

- 4INFBA303 - Verteilte Systeme - empf. FS 4 - Wahlpflicht - 6,0 Credits
- 4INFBA303-VG1 - Verteilte Systeme (Vorlesung) - empf. FS 4 - Pflicht
- 4INFBA303-VG2 - Verteilte Systeme (Übung) - empf. FS 4 - Pflicht
- 4INFBA303-S - Studienleistung Verteilte Systeme - empf. FS 4 - Pflicht - 2,0 Credits
- 4INFBA303-P - Prüfungsleistung Verteilte Systeme - empf. FS 4 - Pflicht - 4,0 Credits

Registration buttons labeled "Anmelden" are visible next to the last two courses.

- ➔ Oral examination
 - ➔ duration about 30-40 minutes

- ➔ Registration:
 - ➔ first register at the campus management system (unisono)
 - ➔ at least 1 week before the exam date (better 3-4 weeks)
 - ➔ then fix a date with my secretary (Ms. Zetzsche, H-B 8403)
 - ➔ at least 1 week before the exam date (better 3-4 weeks)
 - ➔ phone: -4048
 - ➔ email: bsvs.zetzsche@eti.uni-siegen.de
 - ➔ cancellation is possible up to 7 days before the exam
 - ➔ via unisono
 - ➔ please inform me, too!



- ➔ Introduction
- ➔ Middleware
- ➔ Distributed programming (Java RMI, gRPC)
- ➔ Name services
- ➔ Process management
- ➔ Time and global state
- ➔ Fault tolerance
- ➔ Coordination
- ➔ Replication and consistency
- ➔ Distributed file systems

(Some changes compared to last year's lecture!)



- ➔ Understand the properties of distributed systems
 - ➔ absence of a global state
 - ➔ problems with synchronization and with consistency of replicated data
- ➔ Understand the approaches to solve the problems and be able to apply them to given challenges
- ➔ Be able to develop simple distributed programs
- ➔ Distinguish architecture models for distributed systems as well as different types and tasks of middleware and be able to assess their usability for given problems

- ➔ Andrew S. Tanenbaum, Marten van Steen. *Verteilte Systeme, Grundlagen und Paradigmen*. Pearson Studium, 2003.
(English: *Distributed Systems: Principles and Paradigms, 2nd Edition*. Pearson Education, 2016. Available [online](#).)
- ➔ Ulrike Hammerschall. *Verteilte Systeme und Anwendungen*. Pearson Studium, 2005.
- ➔ George Coulouris, Jean Dollimore, Tim Kindberg. *Verteilte Systeme, Konzepte und Design, 3. Auflage*. Pearson Studium, 2002.
(English: *Distributed Systems: Concepts and Design, 5th Edition*. Pearson Education, 2012.)
- ➔ Andrew S. Tanenbaum., Herbert Bos *Moderne Betriebssysteme, 5. Auflage*. Pearson Studium, 2025.
- ➔ William Stallings. *Betriebssysteme – Prinzipien und Umsetzung, 4. Auflage*. Pearson Studium, 2003.



- ➔ Jim Farley, William Crawford, David Flanagan. *Java Enterprise in a Nutshell*. O'Reilly 2002.
- ➔ Cay S. Horstmann, Gary Cornell. *Core Java 2, Band 2 – Expertenwissen*. Sun Microsystems Press / Addison Wesley, 2008.
- ➔ Robert Orfali, Dan Harkey. *Client/Server-Programming with Java and Corba*. John Wiley & Sons, 1998.
- ➔ Torsten Langner. *Verteilte Anwendungen mit Java*. Markt + Technik, 2002.

Zentrum für akademisches Schreiben (ZefaS)

Deine Anlaufstelle für alle Fragen zum Schreiben im Studium

Angebote für Bachelor- und Masterstudierende aller Fächer:

- Individuelle Peer-to-Peer-Beratungen
- Workshops
- Schreib-Peer-Tutor*innen-Ausbildung
- ZefaS-Schreib- und Lesezeit

Mehr Informationen:



UB Universitätsbibliothek
Siegen

U Universität
Siegen



Distributed Systems

Winter Term 2025/26

1 Introduction



Contents

- ➔ What is a distributed system?
- ➔ Software architecture
- ➔ Architecture models
- ➔ Cluster

Literature

- ➔ Hammerschall: 1
- ➔ Tanenbaum, van Steen: 1
- ➔ Colouris, Dollimore, Kindberg: 1, 2
- ➔ Stallings: 13.4



1.1 What is a distributed system?

In a distributed system, components located on different computers work together to coordinate their actions by exchanging messages.

G. Coulouris

A distributed system is a set of independent computers that appear to the user as a single, coherent system.

A. Tanenbaum

A distributed system is a collection of processors that neither share main memory nor a clock.

A. Silberschatz

A distributed system is one on which I can't do any work because some machine I've never heard of has crashed.

L. Lamport

1.1 What is a distributed system? ...



- ➔ A distributed system is a **system**
 - ➔ in which **hardware and software components** are based on **networked computers**, and
 - ➔ communicate and coordinate their actions only via the **exchange of messages**.
- ➔ The boundaries of the distributed system are defined by a common application
- ➔ Best known example: Internet
 - ➔ communication via the standardized Internet protocols
 - ➔ IP and TCP / UDP (👉 lecture Computer Networks)
 - ➔ users can use services / applications, regardless of the present location



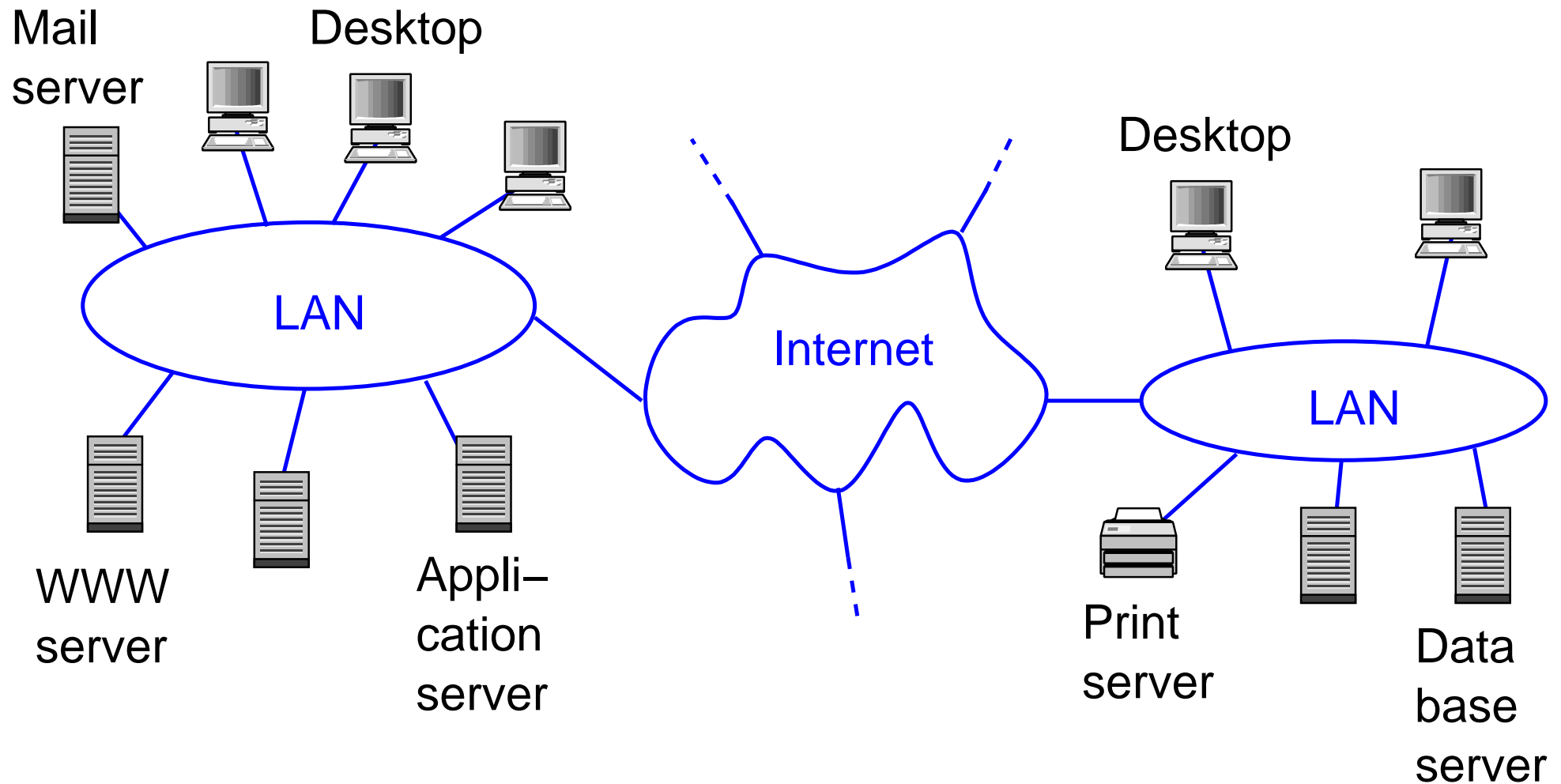
What is a distributed application?

- ➔ Application that uses a distributed system to create a self-contained functionality
- ➔ Application logic distributed among several, largely independent components
- ➔ Components often executed on different machines
- ➔ Examples:
 - ➔ simple internet applications (e.g. WWW, FTP, email)
 - ➔ distributed information systems (e.g. flight booking)
 - ➔ SW intensive, data centered, interactive, highly concurrent
 - ➔ distributed embedded systems (e.g. in the car)
 - ➔ distributed mobile applications (e.g. for handhelds)

1.1 What is a distributed system? ...



A typical distributed system





Why distribution?

- ➔ Central, non-distributed applications are
 - ➔ generally safer and more reliable
 - ➔ generally more performant
- ➔ Main reason for distribution: sharing of resources
 - ➔ hardware resources (printer, scanner, ...)
 - ➔ cost saving
 - ➔ data and information (file server, database, ...)
 - ➔ information exchange, data consistency
 - ➔ functionality (centralization)
 - ➔ error avoidance, reuse

1.2 Characteristics of distributed systems



- ➔ Resources (e.g. computers, data, users, ...) are distributed
 - ➔ sometimes worldwide
- ➔ Cooperation via message exchange
- ➔ Concurrency
 - ➔ but: parallel processing of **a single** request is not the primary goal
- ➔ No global clock (more precisely: no global time)
- ➔ Distributed status information
 - ➔ no uniquely determined global state
- ➔ Partial errors are possible (independent failures)



Parallel vs. distributed systems

➔ Parallel system:

- ➔ motivation: higher performance through parallel execution
- ➔ multiple tasks (processes/threads) working on one job
- ➔ tasks are fine-grained: frequent communication
- ➔ tasks work simultaneously (parallel)
- ➔ homogeneous hardware / OSs, regular network structure

➔ Distributed system:

- ➔ motivation: distributed resources (computers, data, users)
- ➔ multiple tasks (processes/threads) working on one or many jobs
- ➔ tasks are coarse grained: communication less frequent
- ➔ tasks work synchronized (usually one after the other)
- ➔ inhomogeneous (processors, networks, OSs, ...)



- ➔ **Heterogeneity:** computer hardware, networks, OSs, programming languages, implementations by different developers, ...
 - ➔ solution: **middleware**
 - ➔ software layer that hides heterogeneity by providing a unified programming model
 - ➔ e.g. CORBA: distributed objects, remote method invocation
 - ➔ e.g. web services: remote procedure calls (services)
- ➔ **Openness:** easy extensibility (with new services)
 - ➔ requirements:
 - ➔ key interfaces are published / standardized
 - ➔ uniform communication mechanisms / protocols
 - ➔ components must conform to standards

[Coulouris, 1.4]



➔ Security

- ➔ information: confidentiality, integrity, availability
 - ➔ esp. with mobile code
- ➔ users: authentication, authorization

➔ Scalability: number of resources or users can grow without negative impact on performance and cost

➔ Error handling (partial errors)

- ➔ error detection (e.g. checksums)
- ➔ error masking (e.g. retransmission of a message)
- ➔ error tolerance (e.g. browser: “server not available”)
- ➔ recovery (of data) after errors
- ➔ redundancy (of hardware and software)



➔ Concurrency

- ➔ synchronization, consistency of replicated data
- ➔ lack of global time / global state

➔ Transparency

- ➔ access~: local and remote accesses identical
 - ➔ location~: no need to know the location
 - ➔ mobility~: transparent relocation of resources
 - ➔ replication~: transparent replication of resources
 - ➔ concurrency~: shared use of resources without disruptions
 - ➔ error~: hiding errors due to component failure
 - ➔ performance~: performance is largely independent of the load
 - ➔ scaling~: system scales without negative impact on users
- } network~

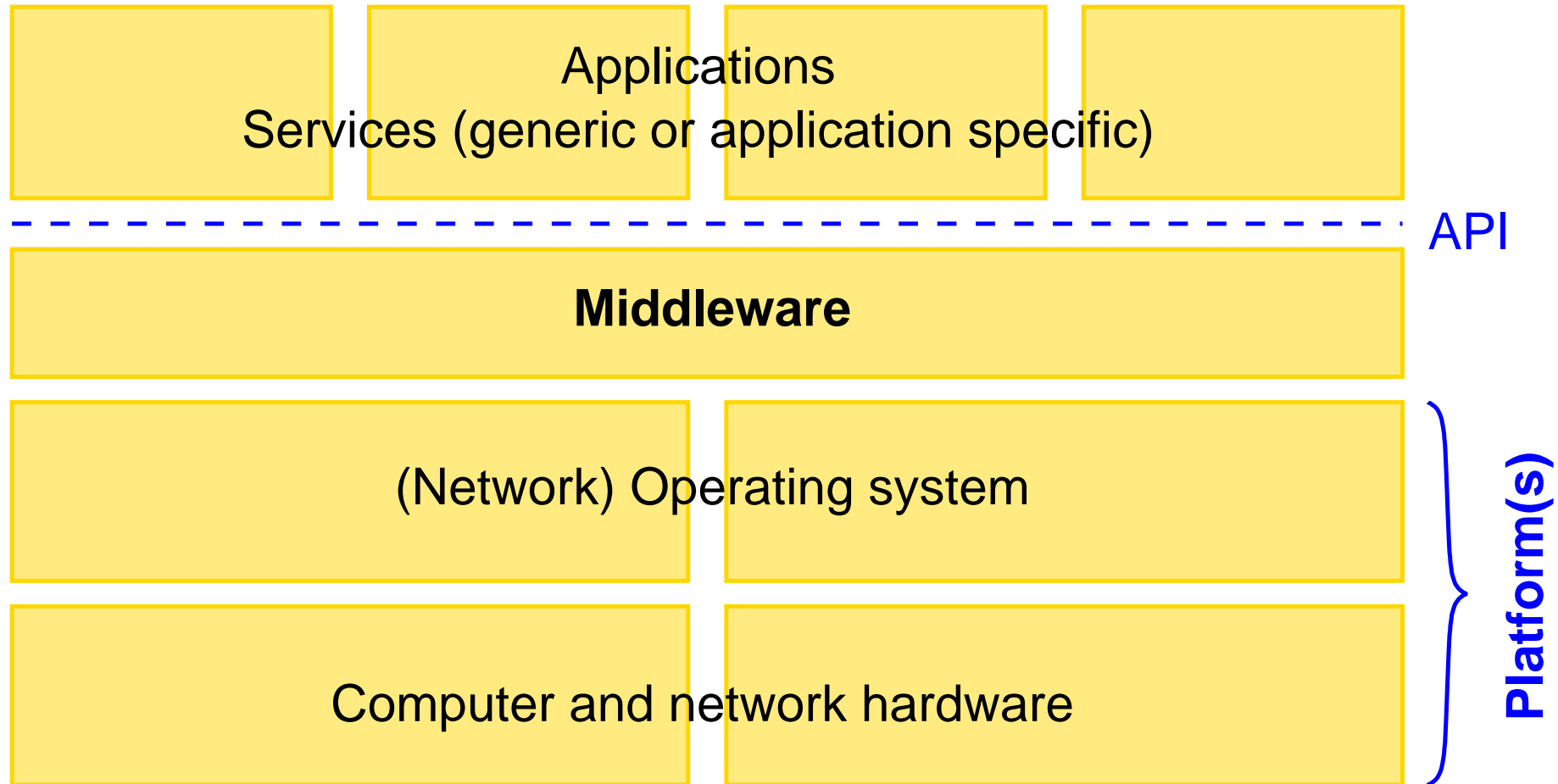


Types of Operating Systems for Distributed Systems

- ➔ Network operating system:
 - ➔ traditional OS, extended by support for network applications (API for sockets, RPC, ...)
 - ➔ each computer has its own OS, but can use services of other computers (file system, email, ssh, ...)
 - ➔ the existence of the other computers is visible
- ➔ Distributed operating system:
 - ➔ uniform OS for a network of computers
 - ➔ transparent for the user
 - ➔ requires cooperation of the OS kernels
 - ➔ so far mainly research projects



Typical layers in a distributed system



[Coulouris, 2.2.1]



Middleware

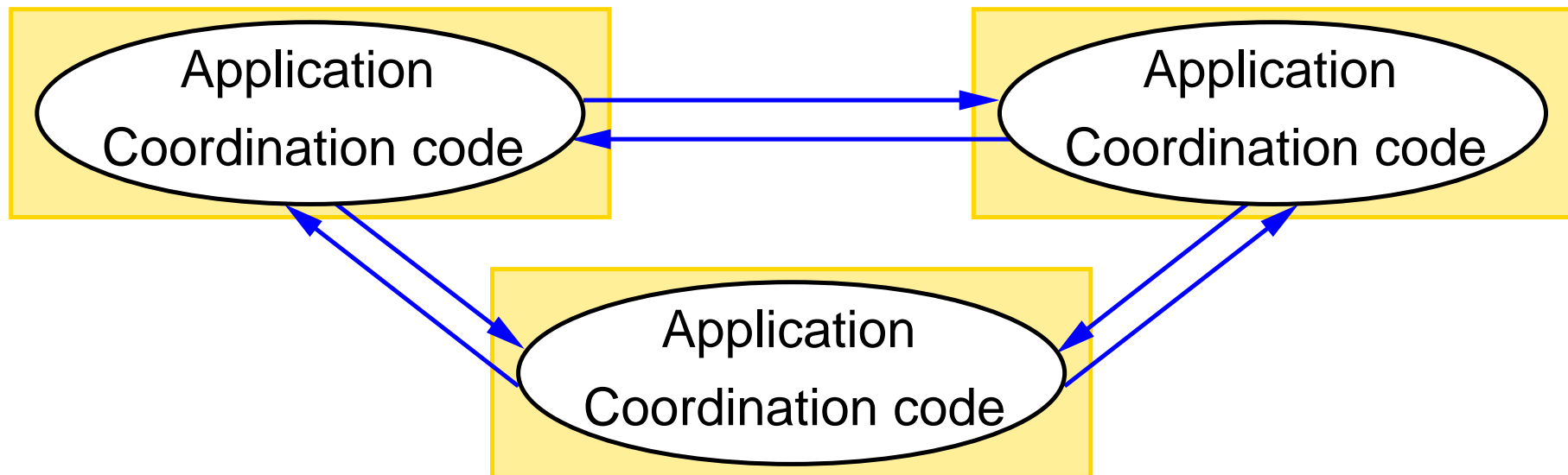
- ➔ Tasks:
 - ➔ hiding of distribution and heterogeneity
 - ➔ providing a common programming model / API
 - ➔ provision of general services
- ➔ Functions e.g:
 - ➔ communication services: remote method calls, group communication, event notifications
 - ➔ replication of shared data
 - ➔ security services
- ➔ Examples: CORBA, EJB, .NET, Web Services, gRPC, ...



- ➔ An architecture model characterizes:
 - ➔ roles of an application component within the distributed application
 - ➔ relationships between application components
- ➔ Role defined by the type of process the component is running in:
 - ➔ client process
 - ➔ short-lived (for the duration of use by the user)
 - ➔ acts as initiator of interprocess communication (IPC)
 - ➔ server process
 - ➔ lives 'unlimited'
 - ➔ acts as a service provider for an IPC
 - ➔ peer process
 - ➔ short-lived (for the duration of use by the user)
 - ➔ acts as initiator and service provider

Peer-to-Peer Model

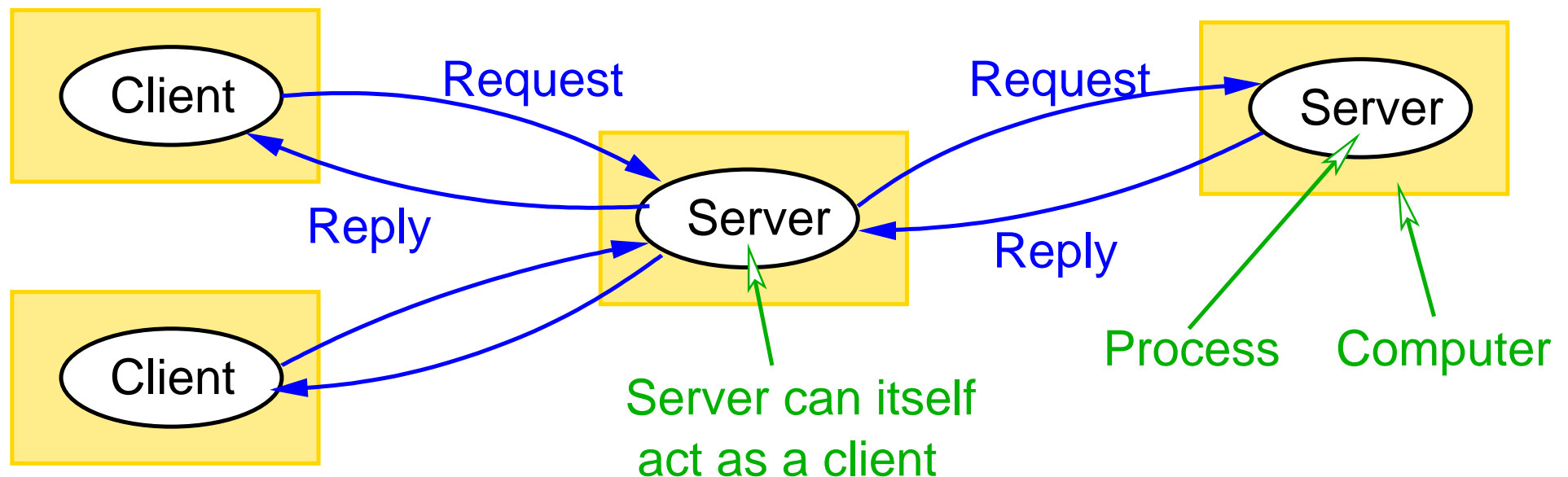
- ➔ Collaboration of peer processes for a distributed activity
 - ➔ each process manages a local part of the resources
 - ➔ distributed coordination and synchronization of actions at application level



- ➔ E.g.: file sharing applications, routers, ...

Client/Server Model

- ➔ Asymmetric model: Servers provide services that can be used by (multiple) clients.
- ➔ servers usually manage resources (centralized)

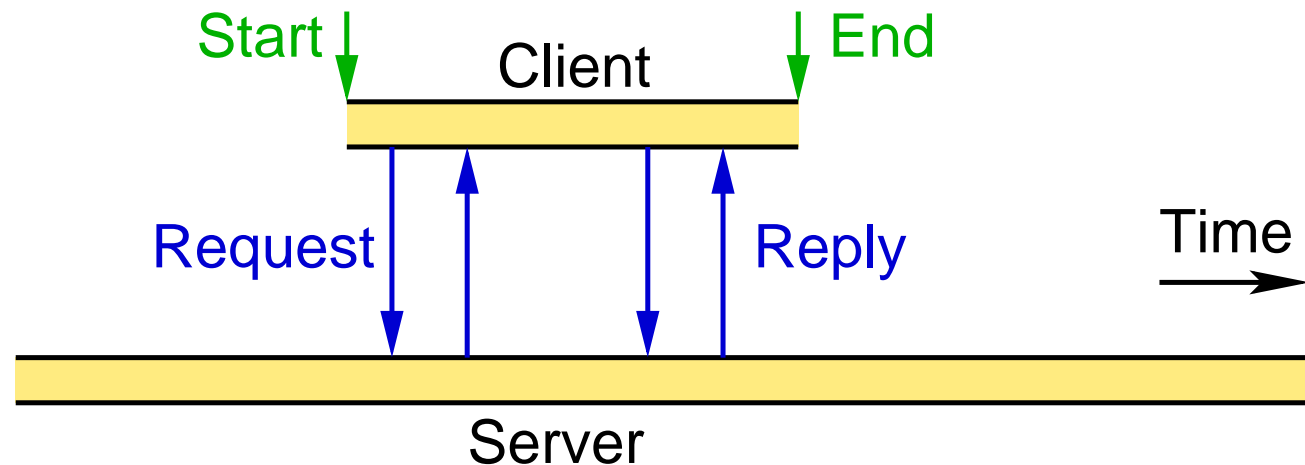


- ➔ Most common model for distributed applications (ca. 80 %)



Client/Server Model ...

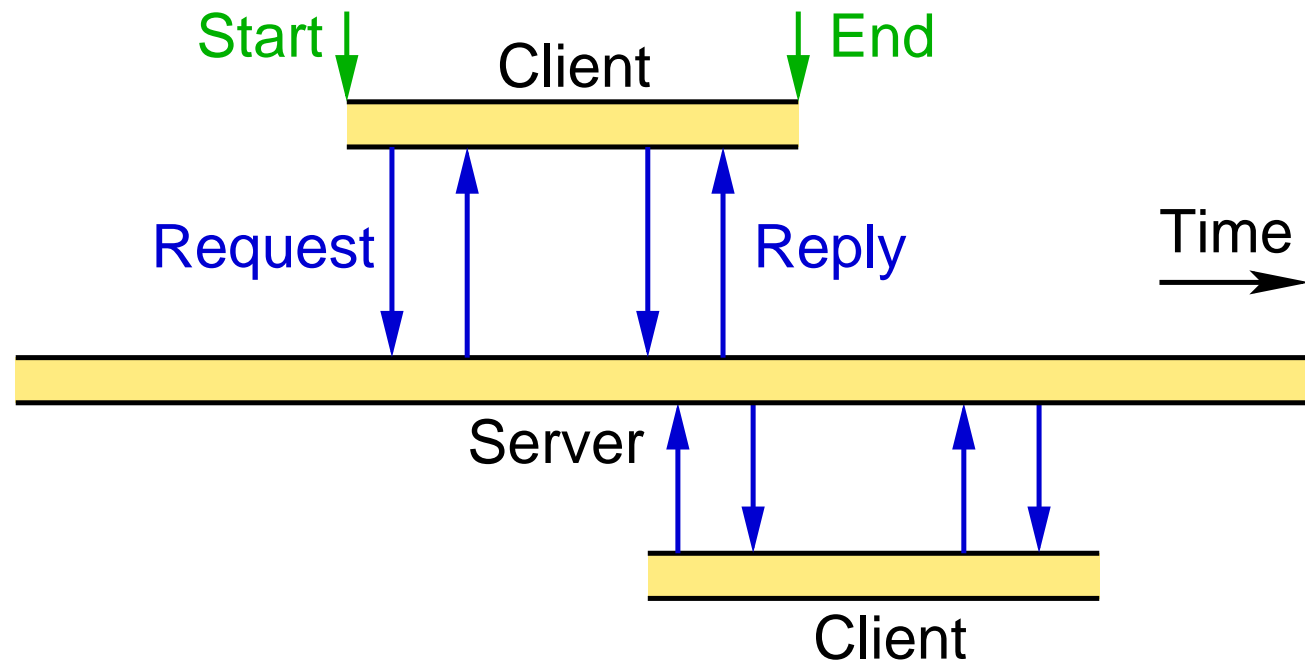
- ➔ Usually concurrent requests from several client processes to the server process



- ➔ Examples: file server, web server, database server, DNS server,
...

Client/Server Model ...

- ➔ Usually concurrent requests from several client processes to the server process

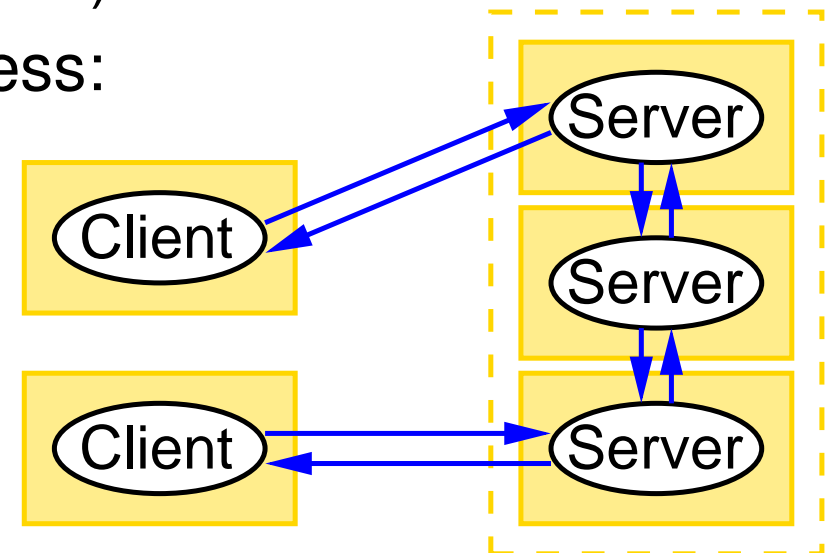


- ➔ Examples: file server, web server, database server, DNS server,
...

Variants of the client/server model

➔ Cooperating servers

- ➔ Network of servers transparently processes a request
- ➔ Example: Domain Name Server (DNS)
 - ➔ if server cannot determine address:
request is transparently
forwarded to another server



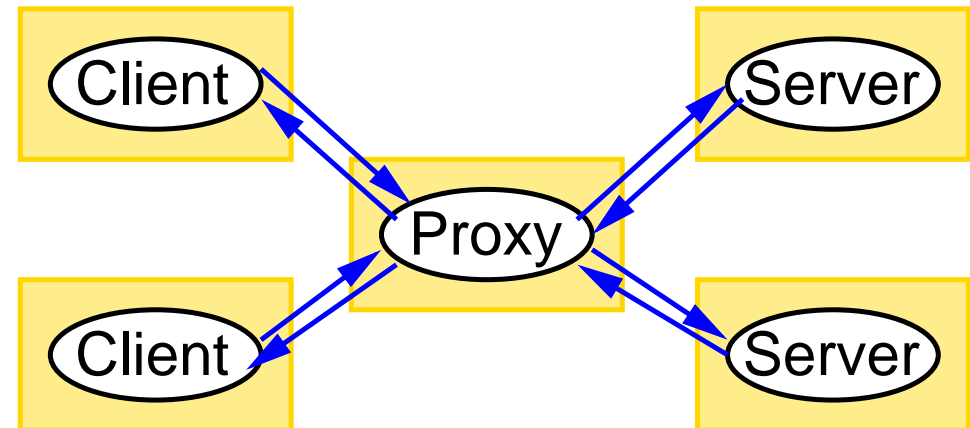
➔ Replicated servers

- ➔ replicas of server processes are provided
 - ➔ transparent replicas (often in clusters)
 - ➔ requests are automatically distributed to the servers
 - ➔ public replicas (e.g. mirror servers)
- ➔ goals: better performance, reliability

Variants of the client/server model ...

➔ Proxy-Server / Caches

- ➔ proxy is a delegate for the server
- ➔ task often is caching of data / results
- ➔ e.g. web proxy



➔ Mobile code

- ➔ executable server code migrates to client on request
- ➔ code is executed by the client
- ➔ best-known example: JavaScript / WebAssembly in the WWW

➔ Mobile agents

- ➔ agent contains code and data, moves through the network and performs actions on local resources

Distributed Systems

Winter Term 2025/26

23.10.2025

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

Stand: January 8, 2026



n-Tier Architectures

- ➔ Refinements of Client/Server Architecture
- ➔ Models for distributing an application to the nodes of a distributed system
- ➔ Mainly used in information systems
- ➔ **Tier** (german: Schicht / Stufe) denotes an independent process space within a distributed application
 - ➔ process space can, but does not have to, correspond to a physical host
 - ➔ several process spaces on one computer are possible

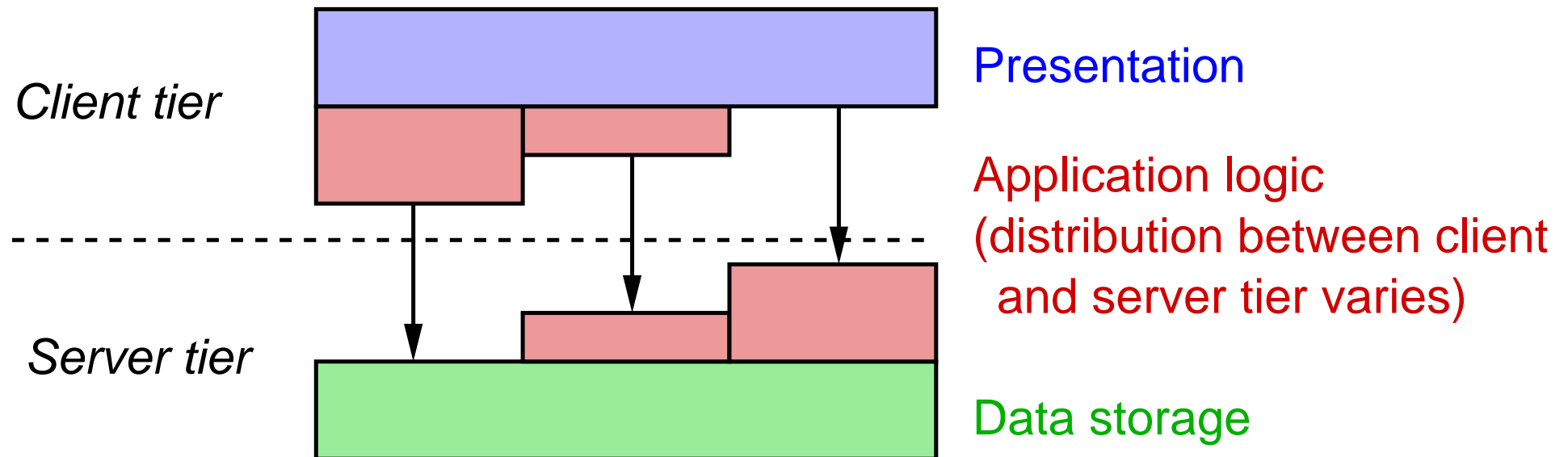


The Tier Model

- ➔ Typical tasks in an information system:
 - ➔ presentation – interface to the user
 - ➔ application logic – actual functionality
 - ➔ data storage – storage of data in a database
- ➔ The tier model determines:
 - ➔ assignment of tasks to application components
 - ➔ distribution of application components on tiers
- ➔ Architectures:
 - ➔ 2-tier architectures
 - ➔ 3-tier architectures
 - ➔ 4-or-more-tier architectures

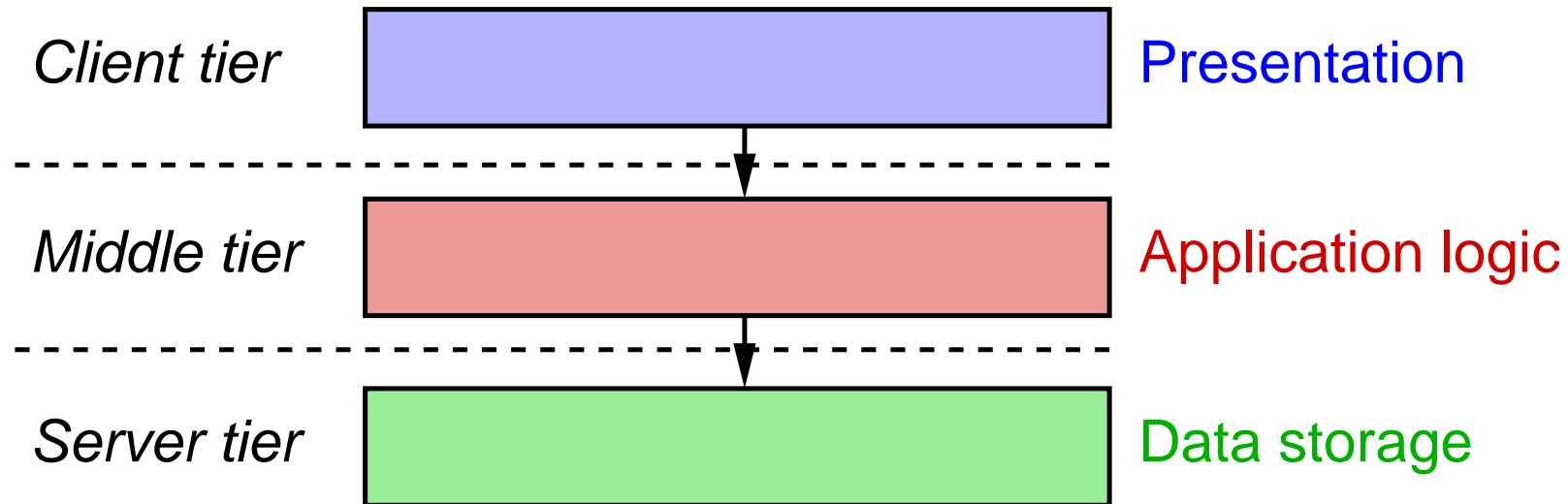
2-Tier Architecture

- ➔ Client and server tier
- ➔ No own tier for the application logic



- ➔ Advantage: simple, high performance
- ➔ Disadvantage: difficult to maintain, poorly scalable

3-Tier Architecture



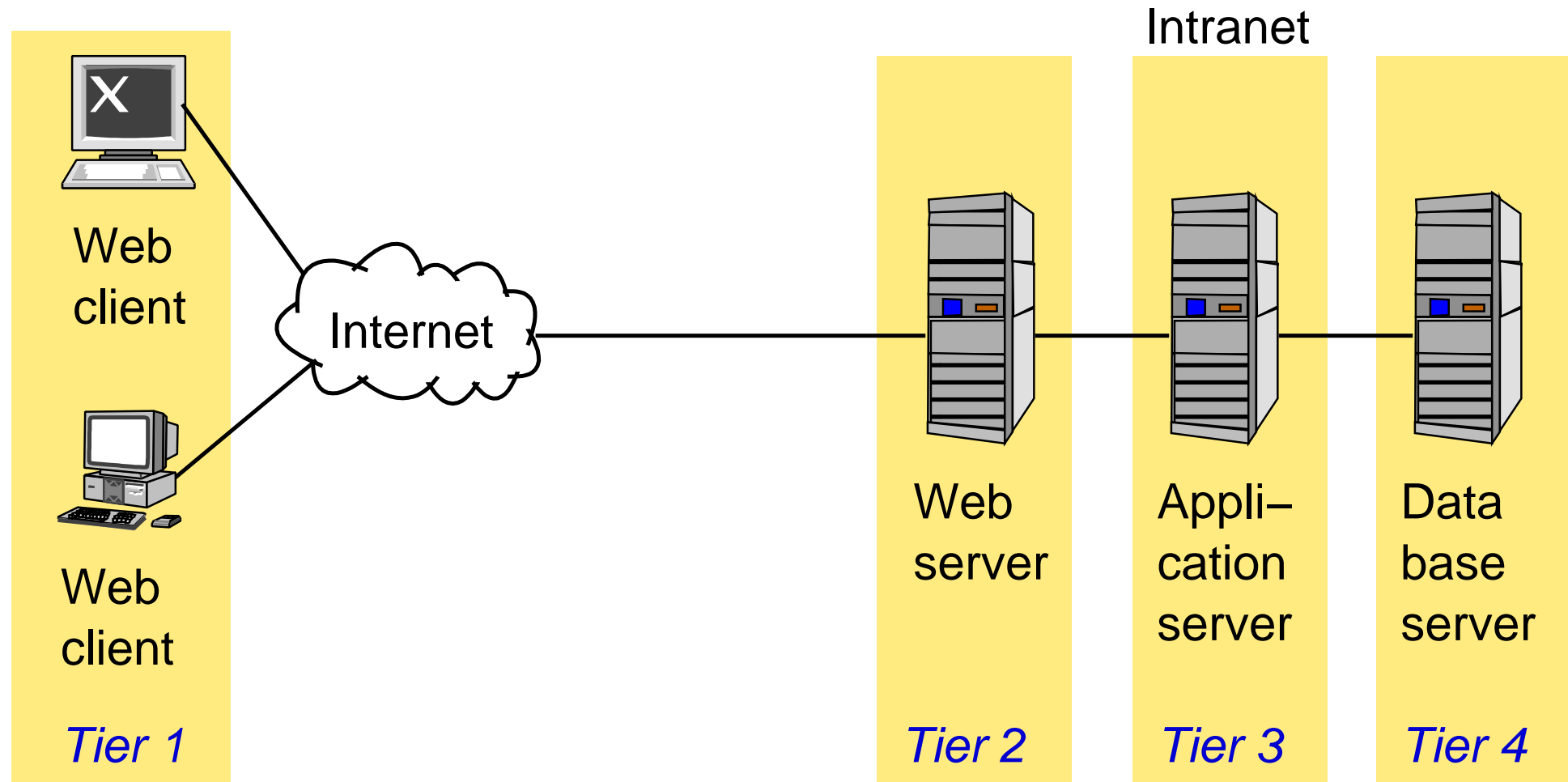
- ➔ Standard distribution model for simple web applications:
 - ➔ client tier: web browser for display
 - ➔ middle tier: web server with JSP / ASP / PHP ...
 - ➔ server tier: database server
- ➔ Advantages: central administration of application logic, scalable



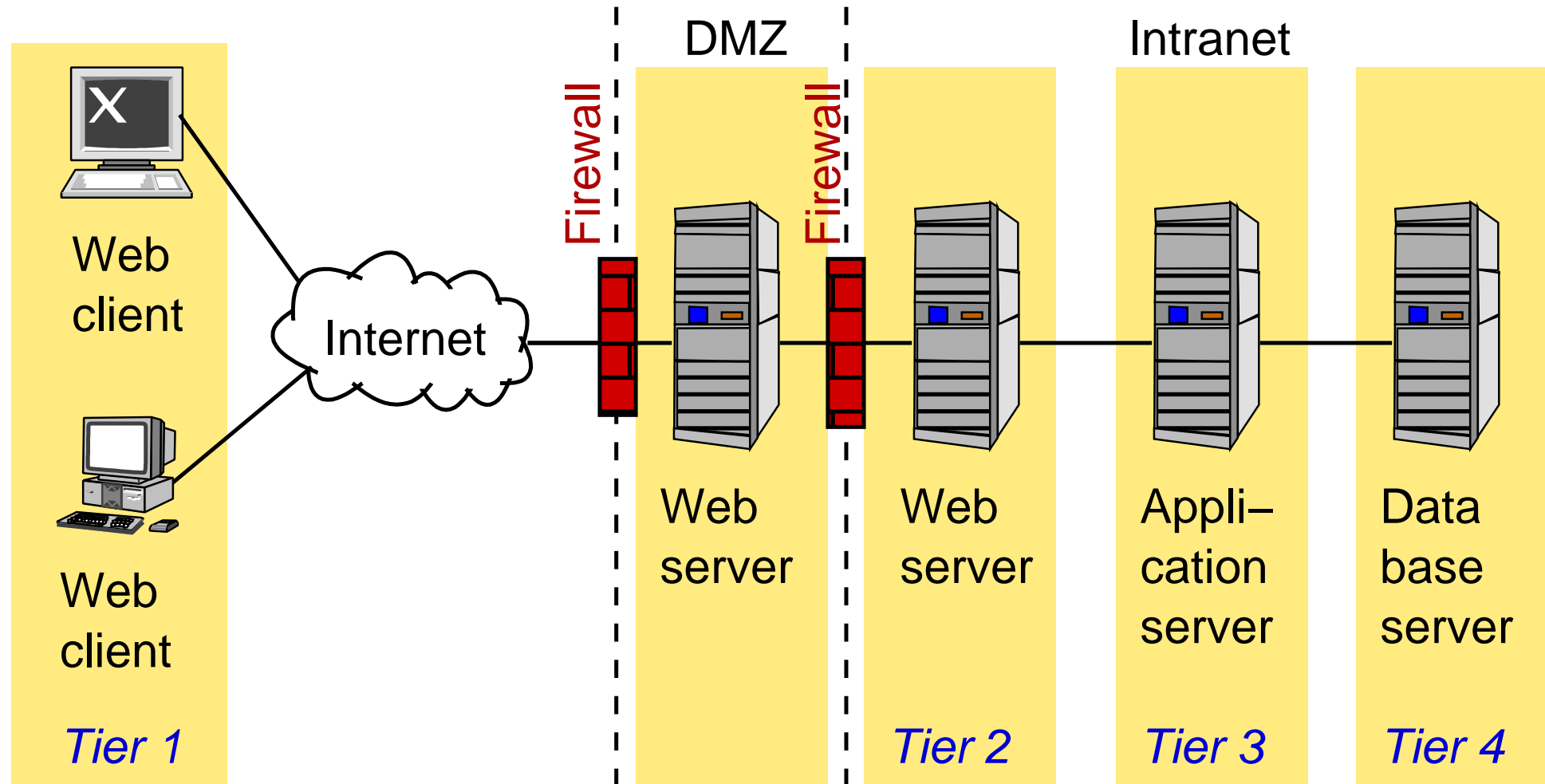
4-or-more-Tier Architectures

- ➔ Difference to 3-tier architecture:
 - ➔ application logic distributed across multiple tiers
- ➔ Motivation:
 - ➔ minimization of complexity (divide and conquer)
 - ➔ better protection of individual application parts
 - ➔ reusability of components
- ➔ Many distributed information systems have 4-or-more-tier architectures

Example: Typical Internet Application



Example: Typical Internet Application





Thin and fat clients

- ➔ Characterizes complexity of the application component on the client tier
- ➔ Ultra-thin client
 - ➔ client tier only for presentation: pure display of dialogs
 - ➔ presentation component: web browser
 - ➔ only possible with 3-or-more-tier architectures
- ➔ Thin client
 - ➔ client tier for presentation only: display of dialogs, preparation of data for display
- ➔ Fat client
 - ➔ parts of the application logic on the client tier
 - ➔ usually with 2-tier architectures



Distinction from Enterprise Application Integration (EAI)

- ➔ EAI: integration of different applications
 - ➔ communication, exchange of data
- ➔ Goals similar to distributed applications / middleware
 - ➔ middleware is often used for EAI as well
- ➔ Differences:
 - ➔ distributed applications: application components, high degree of coupling, usually little heterogeneity
 - ➔ EAI: complete applications, low degree of coupling, mostly great heterogeneity (different technologies, systems, programming languages, ...)

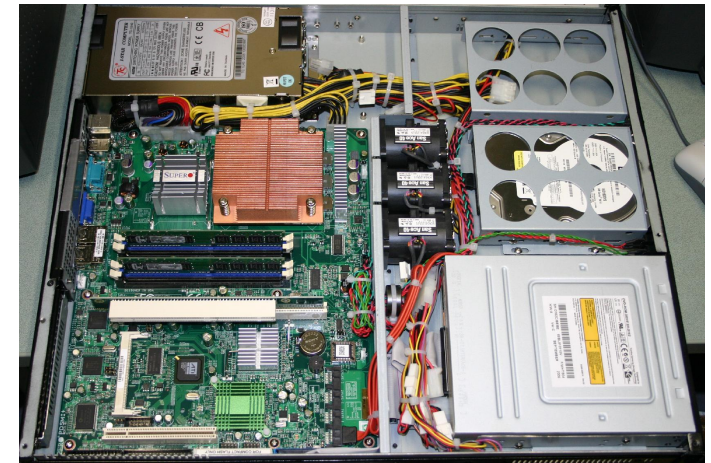
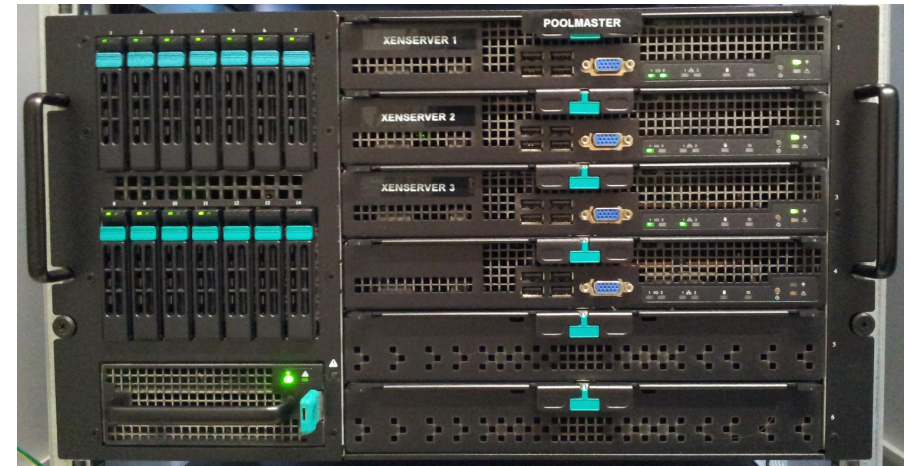
1.6 Cluster



- ➔ Cluster: group of networked computers that acts as a unified computing resource
 - ➔ i.e. multicomputer system
 - ➔ nodes usually standard PCs or blade server

- ➔ Application mainly as high performance server

- ➔ Motivation:
 - ➔ (step-by-step) scalability
 - ➔ high availability
 - ➔ good price/performance ratio



[Stallings, 13.4]



Uses for Clusters

- ➔ High availability (HA) clusters
 - ➔ improved reliability
 - ➔ when a node is faulty: services are migrated to other nodes (failover)
- ➔ Load balancing cluster
 - ➔ incoming requests are distributed to different nodes of the cluster
 - ➔ usually by a (redundant) central instance
 - ➔ frequently with WWW or email servers
- ➔ High performance computing cluster
 - ➔ cluster as parallel computer



Cluster configurations

- ➔ Passive standby (no actual cluster)
 - ➔ processing of all requests by primary server
 - ➔ secondary server takes over tasks (only) in case of failure
- ➔ Active standby
 - ➔ all servers process requests
 - ➔ enables load balancing and improved reliability
 - ➔ problem: access to data of other / failed server
 - ➔ alternatives:
 - ➔ replication of data (a lot of communication)
 - ➔ shared hard disk system (usually mirrored disks or RAID system for fail-safe operation)



Active Standby Configurations

- ➔ Separate servers with data replication
 - ➔ separate disks, data is continuously copied to secondary servers
- ➔ Server with shared hard disks
 - ➔ shared nothing cluster
 - ➔ separate partitions for each server
 - ➔ in case of server failure: reconfiguration of the partitions
 - ➔ shared disc cluster
 - ➔ simultaneous use by all servers
 - ➔ requires lock manager software to lock files or records



- ➔ Distributed system
 - ➔ HW and SW components on networked computers
 - ➔ no shared memory, no global time
 - ➔ motivation: use of distributed resources
- ➔ Challenges
 - ➔ heterogeneity, openness, security, scalability
 - ➔ error handling, concurrency, transparency
- ➔ Software architecture: middleware
- ➔ Architectural models:
 - ➔ peer-to-peer, client/server
 - ➔ n-tier models
- ➔ Cluster: high availability, load balancing



Distributed Systems

Winter Term 2025/26

2 Middleware

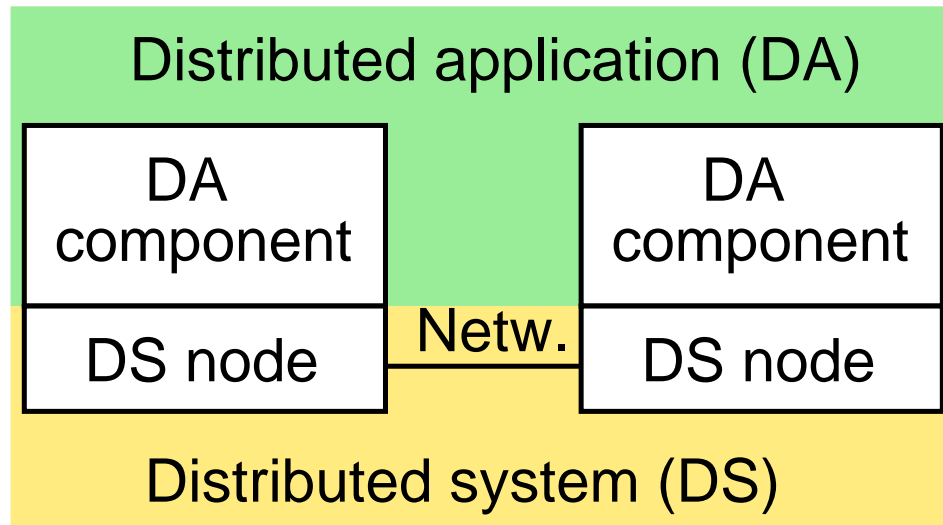


Content

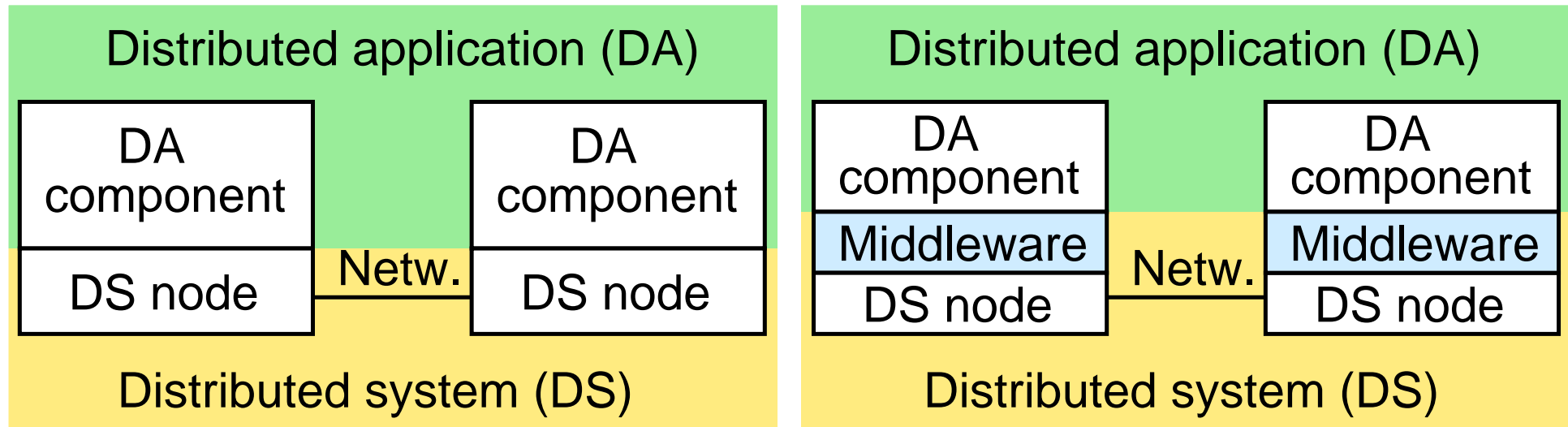
- ➔ Communication in distributed systems
- ➔ Communication-oriented middleware
- ➔ Application-oriented middleware

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



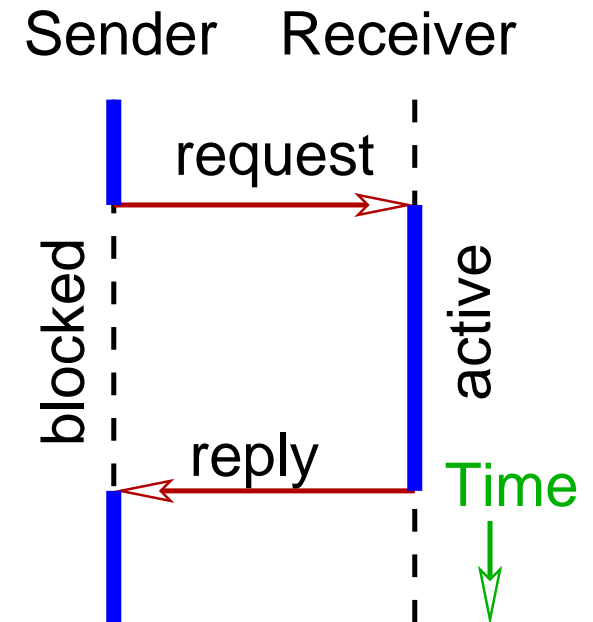
- ➔ Middleware is the interface between distributed application and distributed system
- ➔ Goal: hide distribution aspects from application
 - ➔ transparency (👉 1.3)
- ➔ Middleware can also provide additional services for applications
 - ➔ huge differences in existing middleware
- ➔ Distinction:
 - ➔ **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

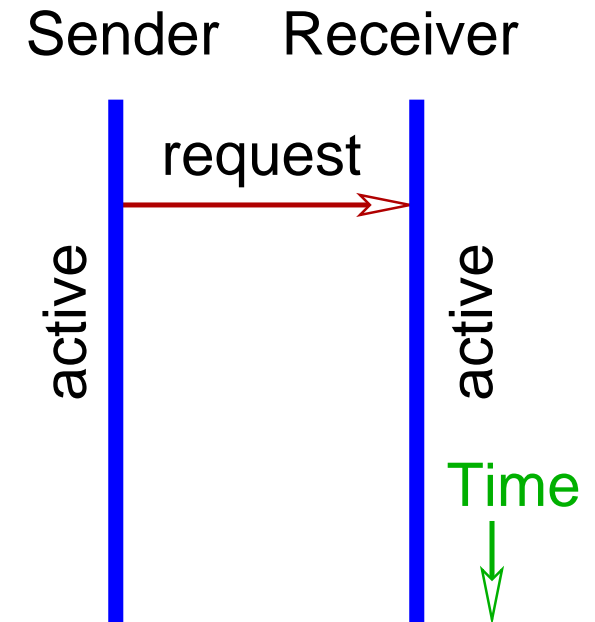
- ➔ Sender and receiver block when calling a send or receive operation
 - ➔ receiver is waiting for a request
 - ➔ sender is waiting for the reply
- ➔ Tight coupling between sender and receivers
 - ➔ advantage: easy to understand model
 - ➔ disadvantage: strong dependency, especially in case of error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



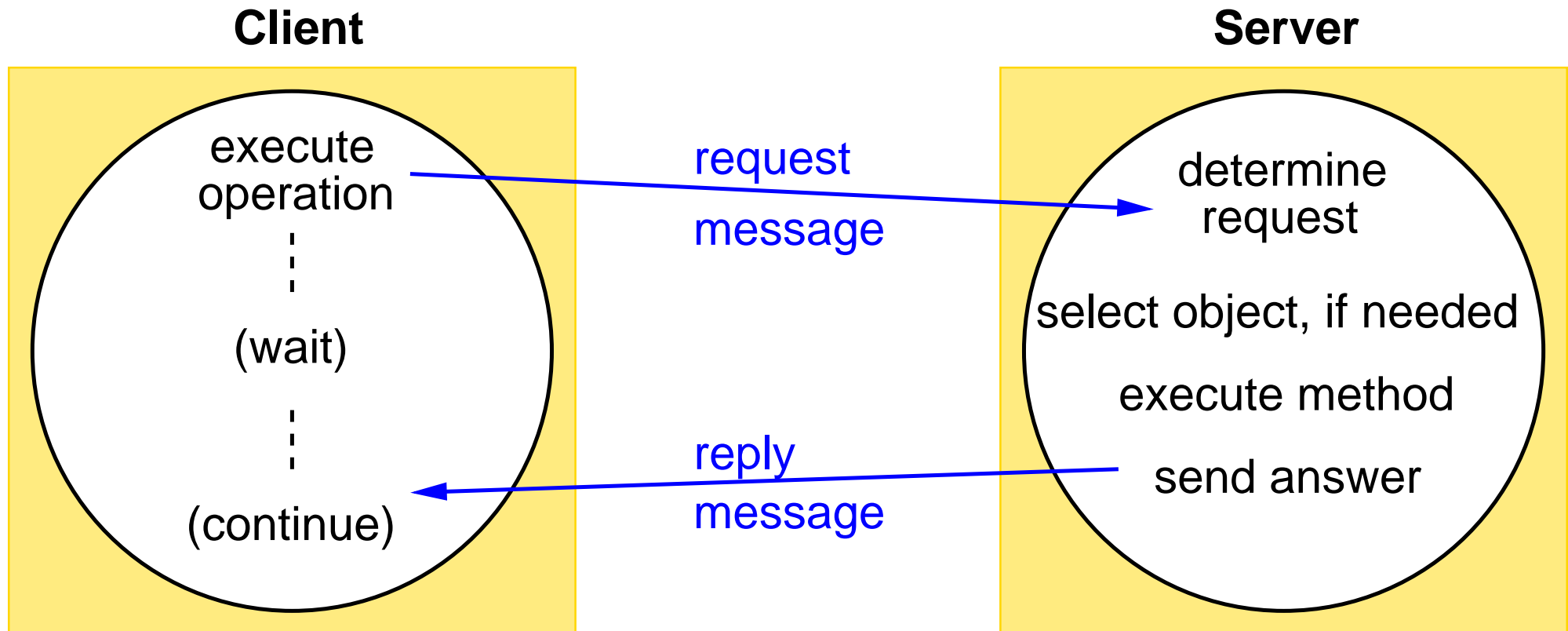


Asynchronous Communication

- ➔ 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

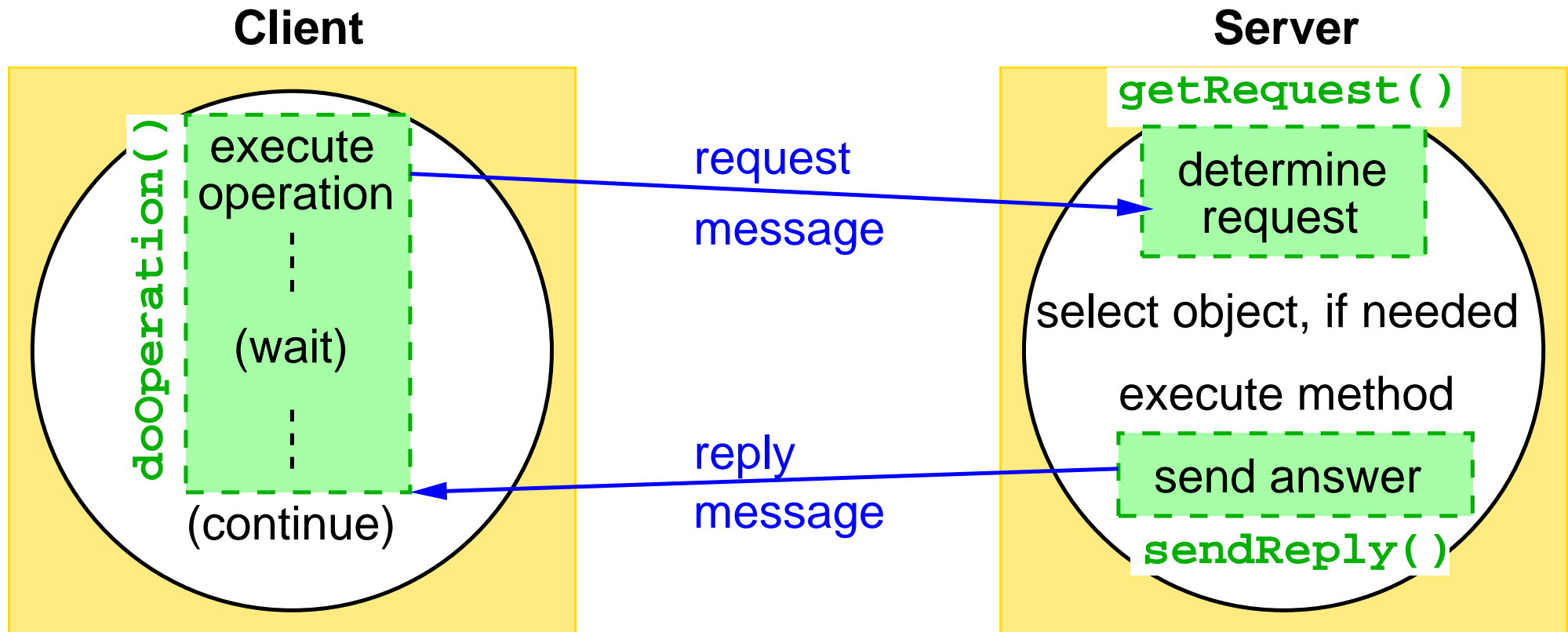


Client/Server Communication



- ➔ 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

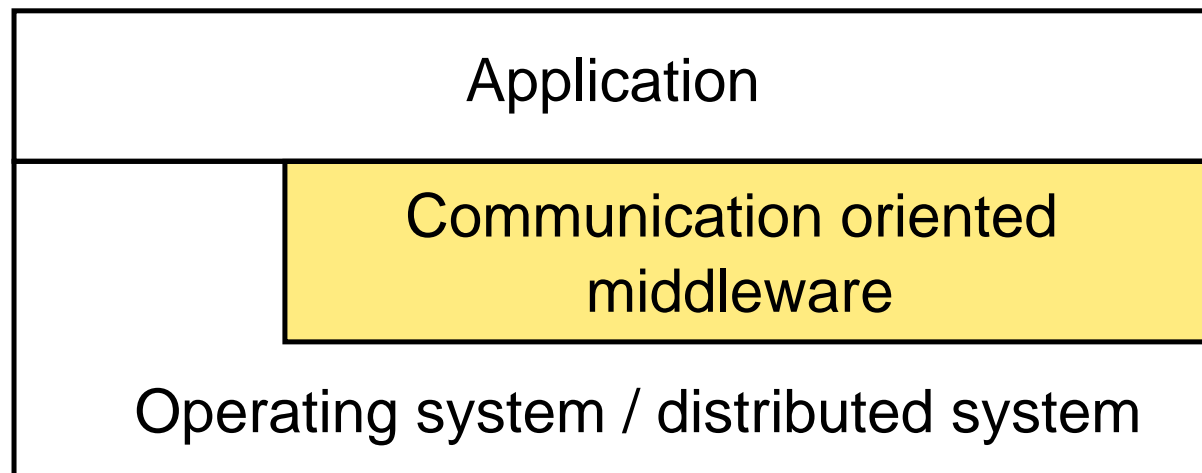
➔ Exactly once

- ➔ 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



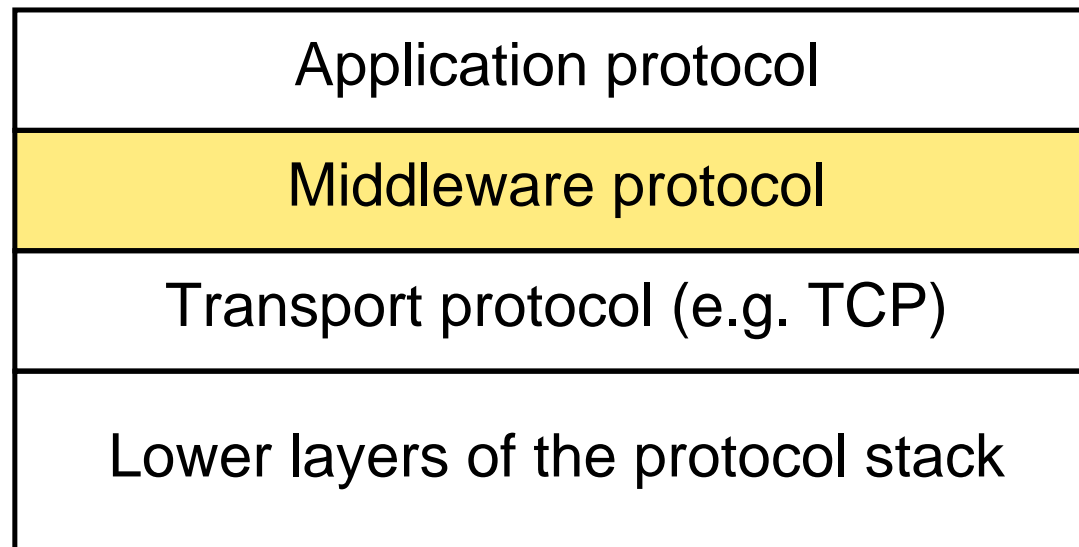
- ➔ Focus: provision of a communication infrastructure for distributed applications
- ➔ Tasks:
 - ➔ communication
 - ➔ dealing with heterogeneity
 - ➔ error handling





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



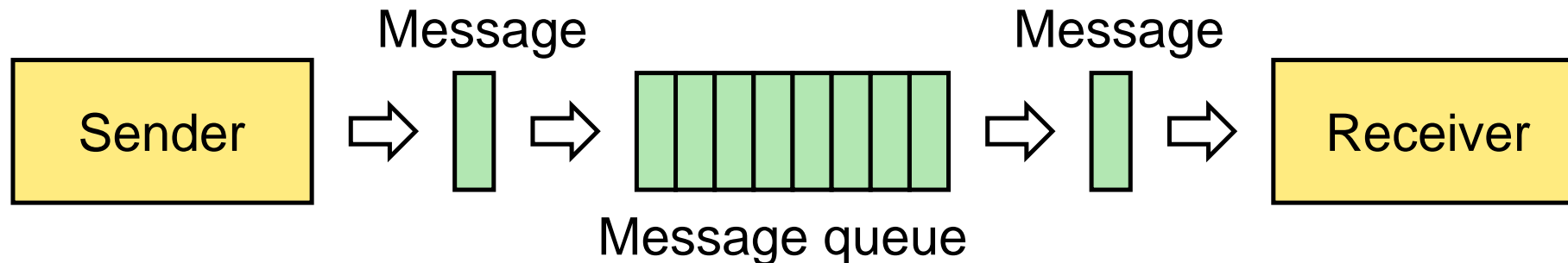
2.2.2 Programming Models

- ➔ 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

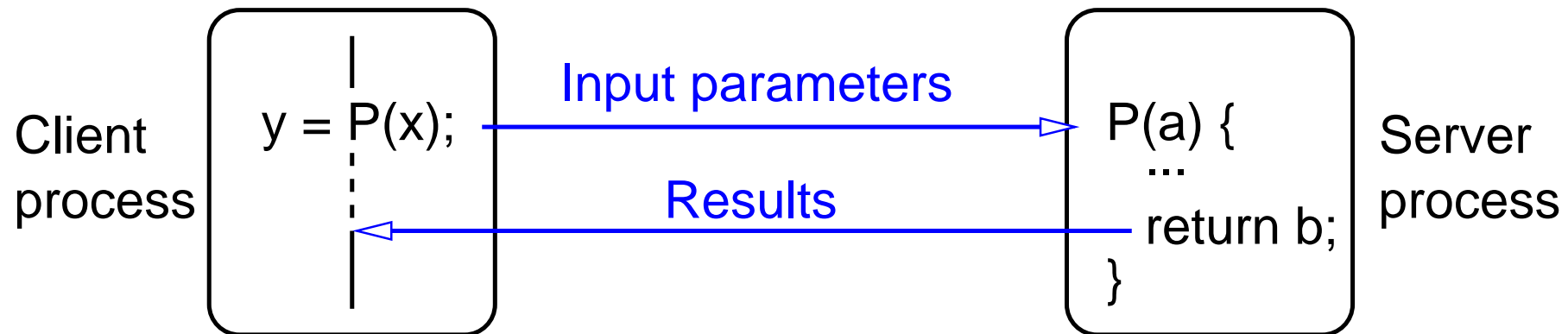
30.10.2025

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

Stand: January 8, 2026

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)

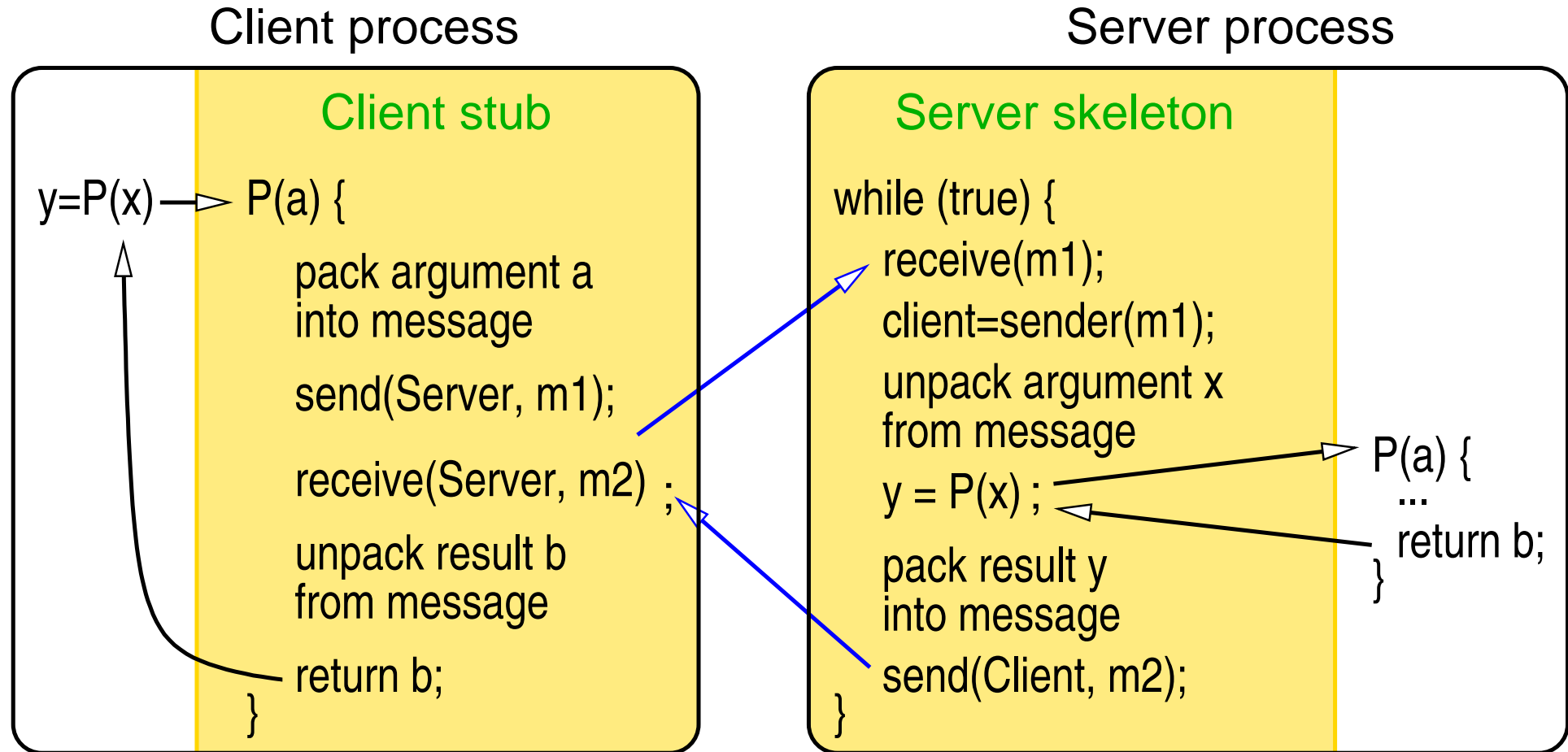
- ➔ Allows an object to call methods of a remote object
- ➔ In principle very similar to RPC



Common Basic Concepts of Remote Calls

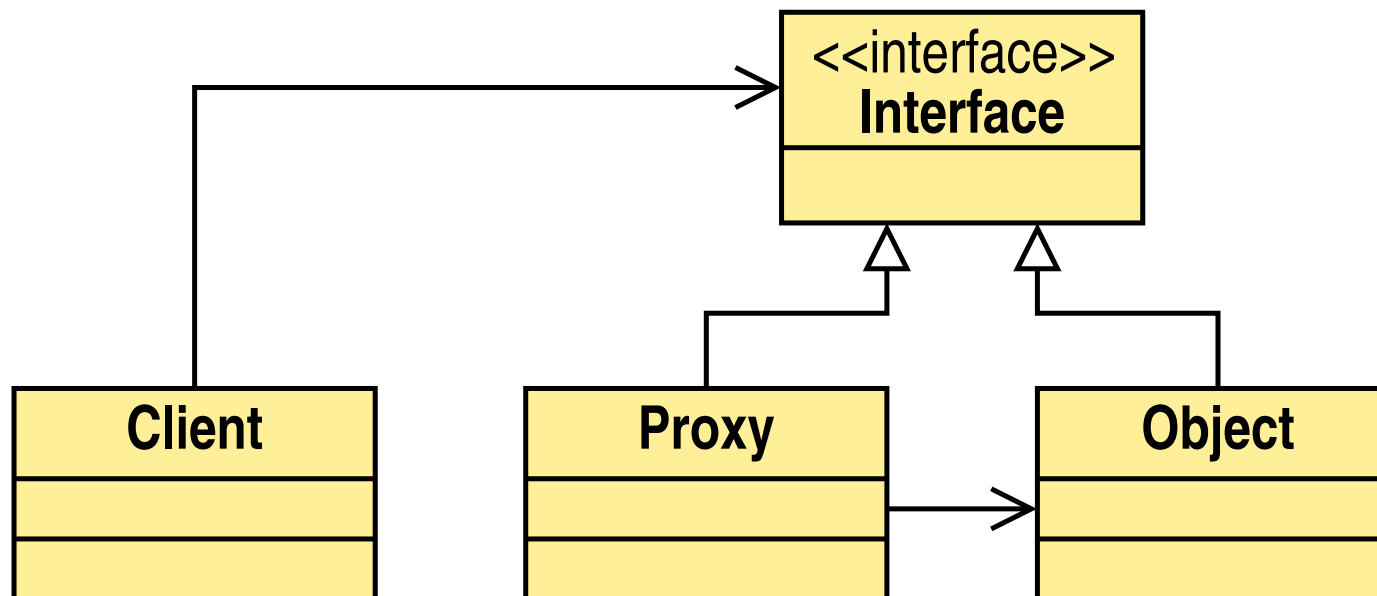
- ➔ Client and server are decoupled by interface definition
 - ➔ defines names of calls, parameters and return values
- ➔ Introduction of **client stubs** and **server skeletons** as an access interface
 - ➔ are automatically generated from interface definition
 - ➔ IDL compiler (IDL = interface definition 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)

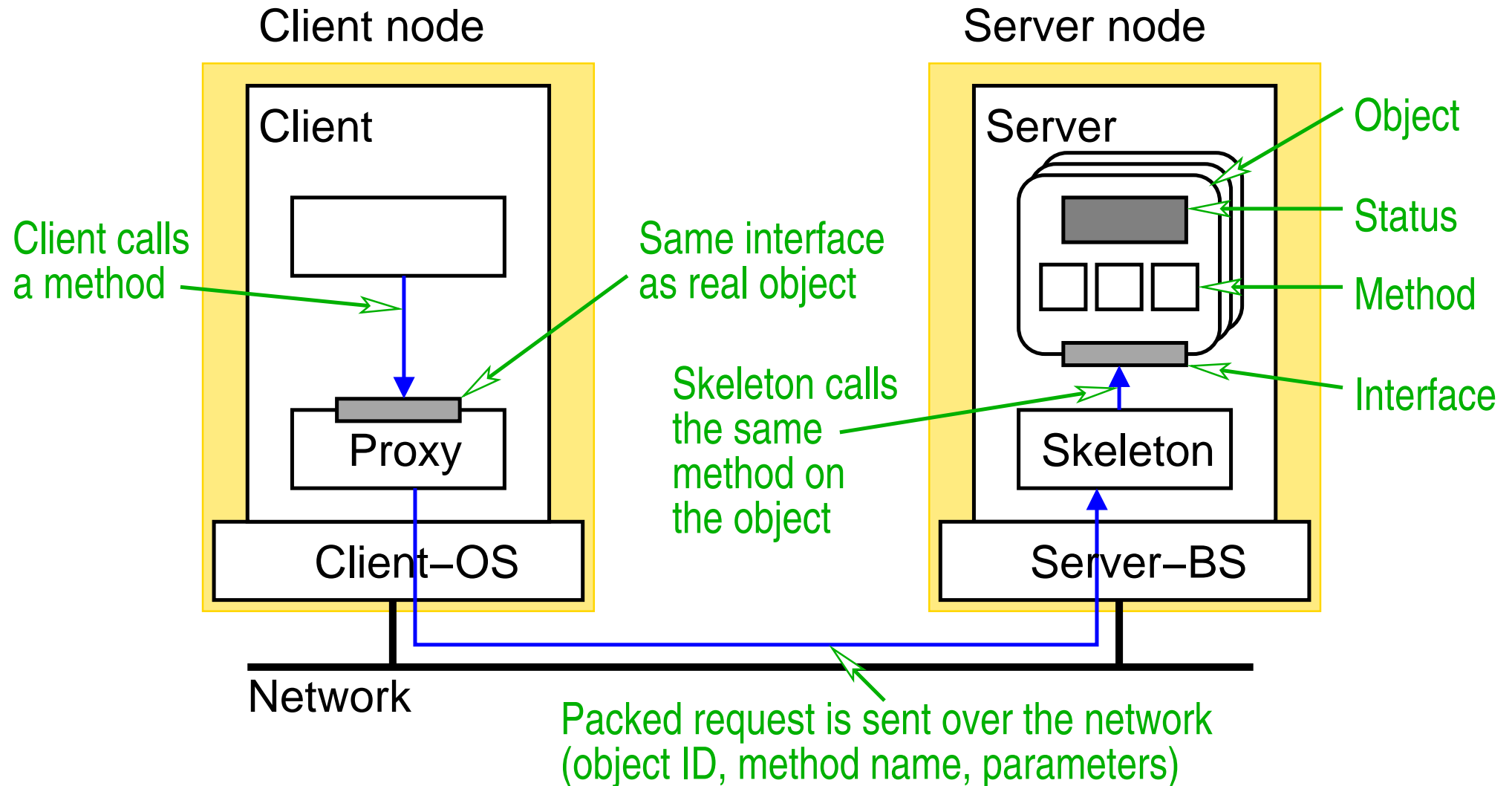


Basis of RMI: The Proxy Pattern

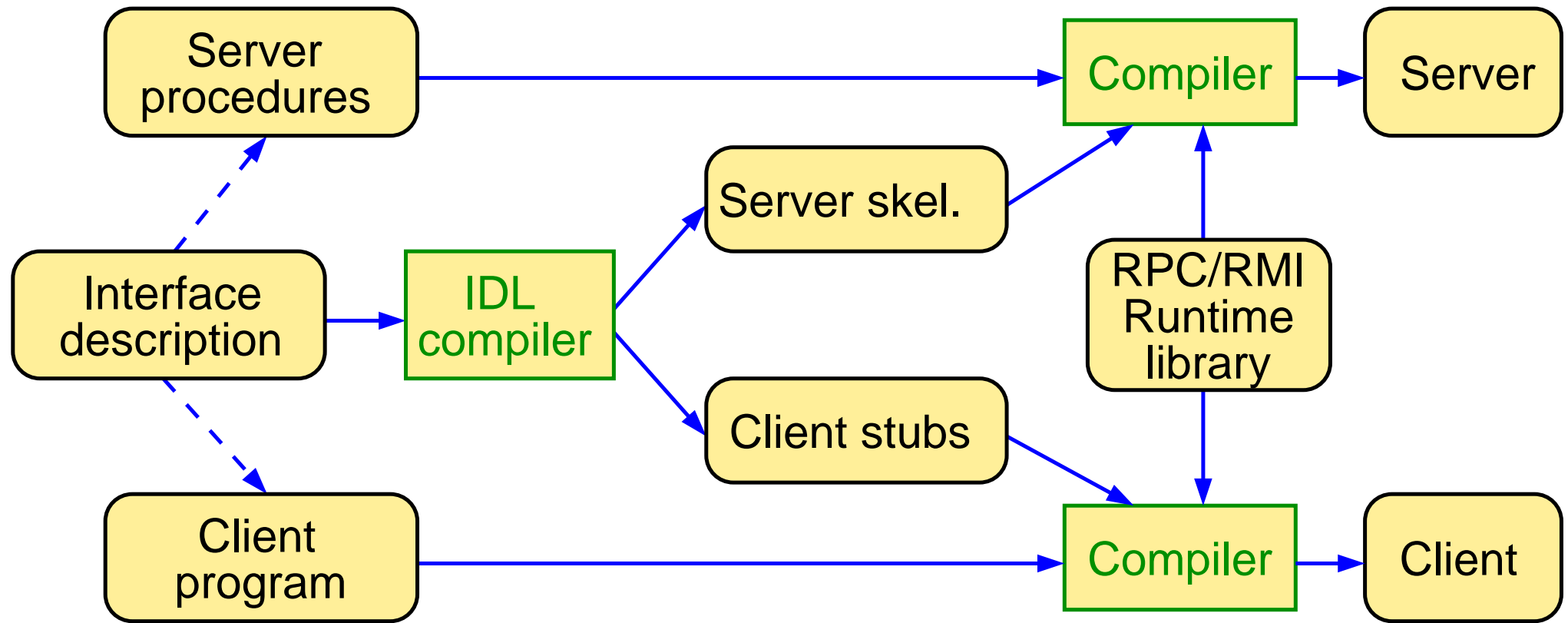
- ➔ Client works with a deputy object (**proxy**) of the actual server object
- ➔ proxy and server object implement the same interface
- ➔ client only knows / uses this interface



Flow of a Remote Method Call



Creation of a Client/Server Program



➔ Applies in principle to all realizations of remote calls



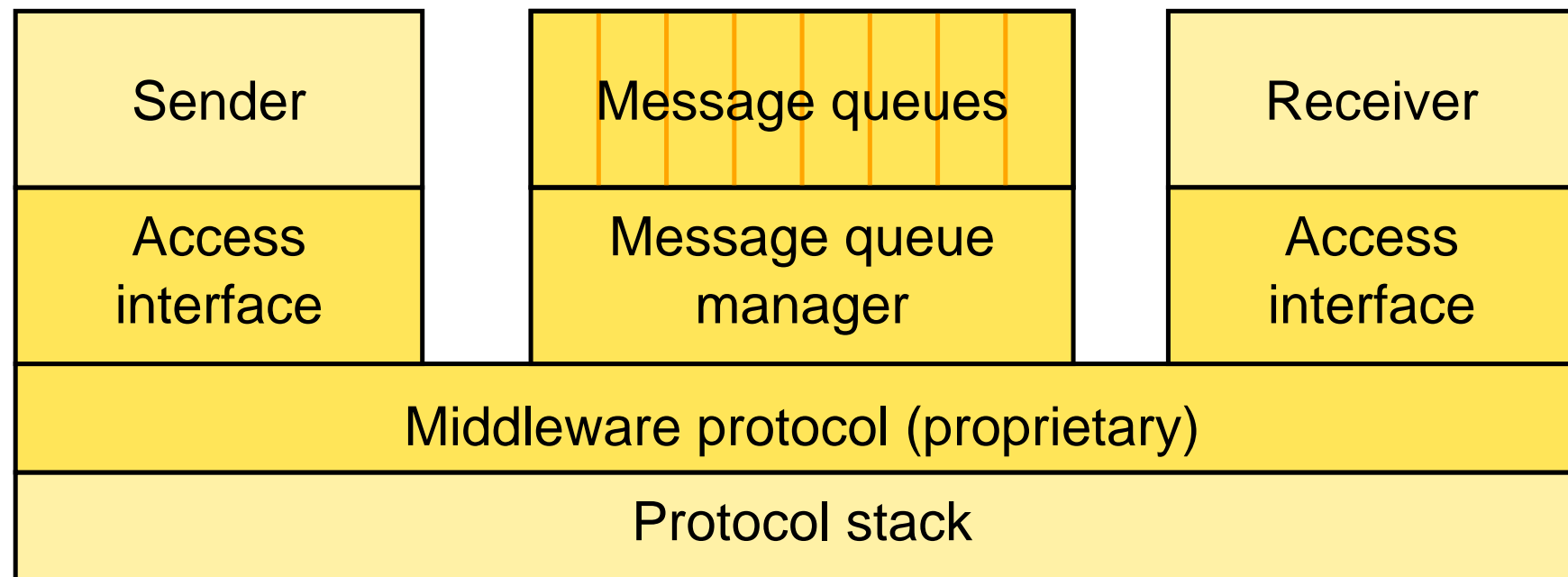
2.2.3 Middleware Technologies

- ➔ 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), ...



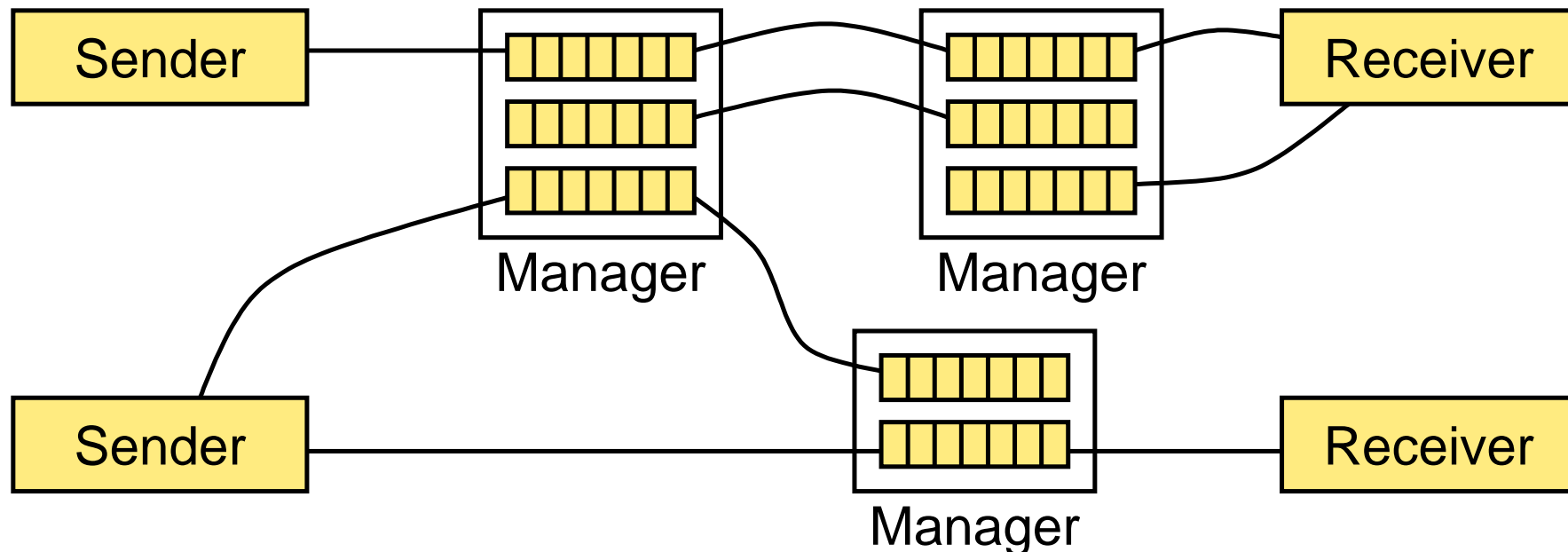
2.2.4 Message Oriented Middleware (MOM)

- ➔ 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



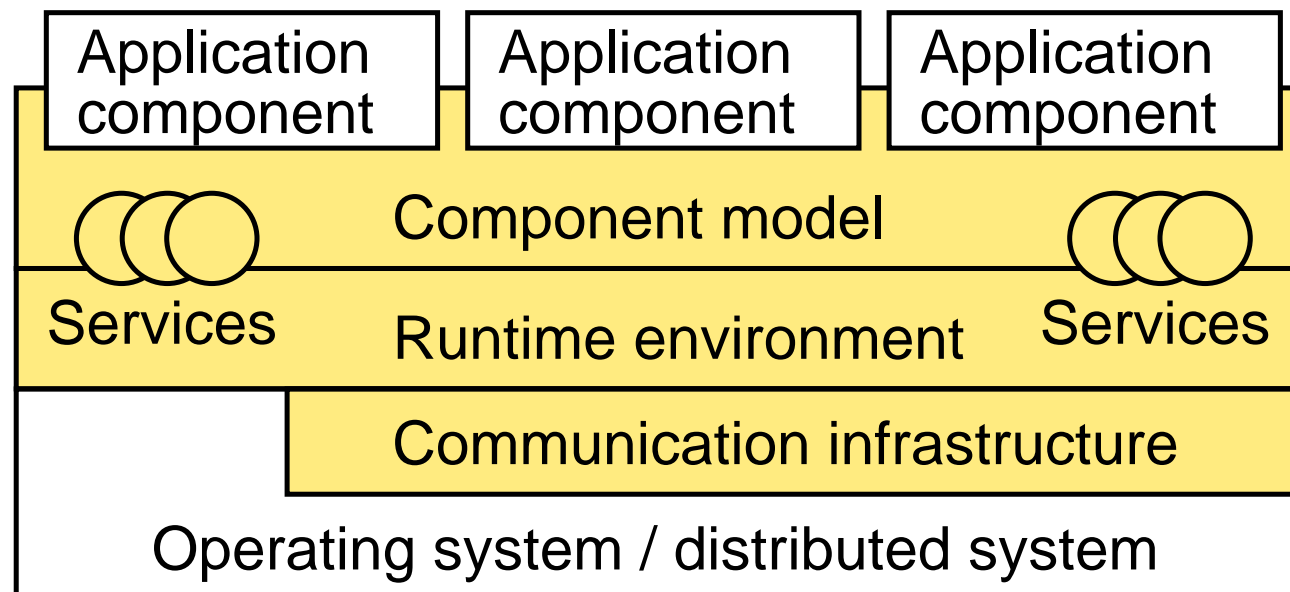
2.2.5 Summary

- ➔ 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)

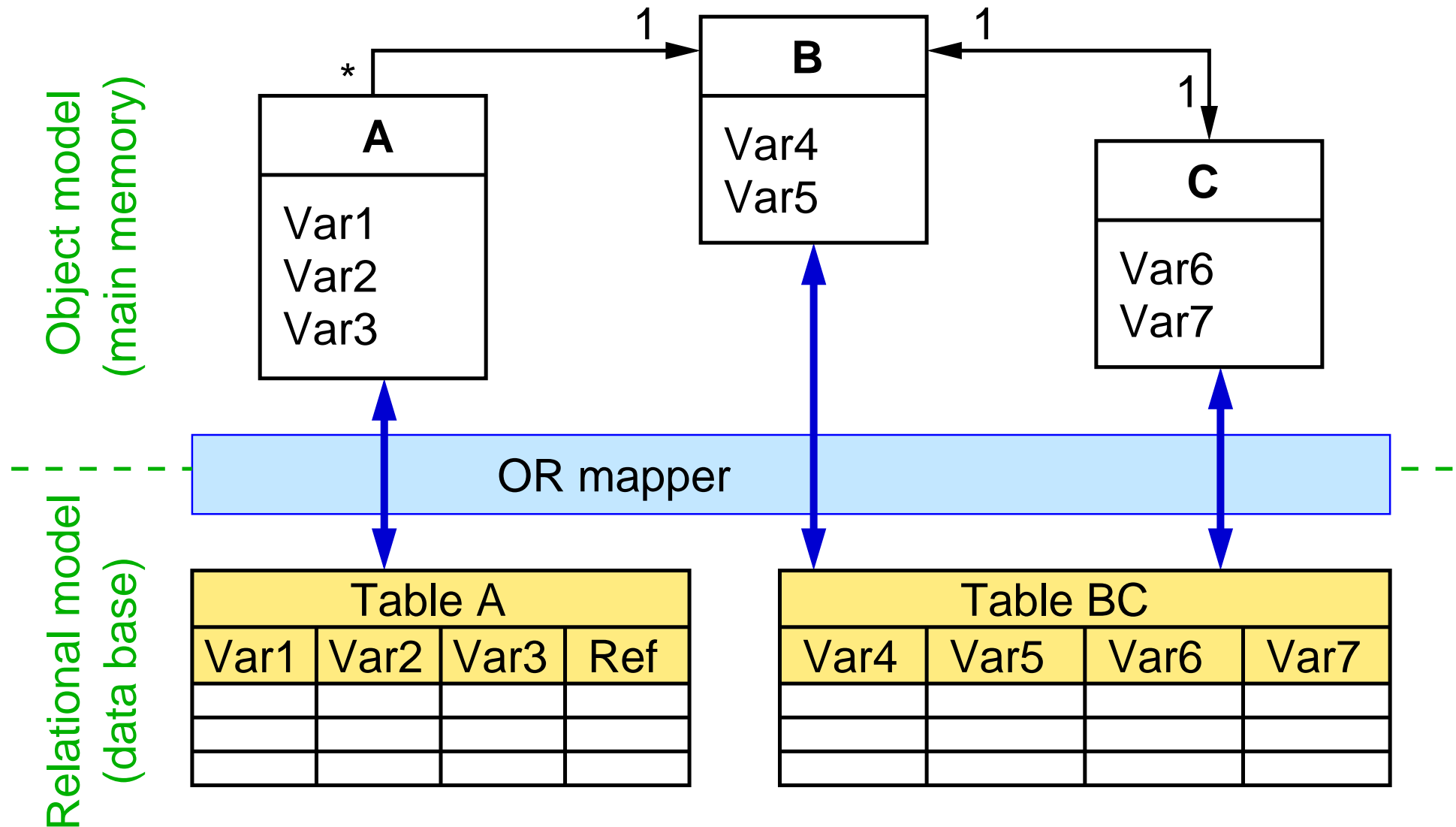
- ➔ Service for interactive, data-centric applications
 - ➔ consistency / integrity of data is important
 - ➔ this means that the entire (maybe distributed) dataset must represent a valid state in itself
- ➔ Typical sequence in applications:
 1. client requests data
 2. client changes the data
 3. client 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



Distributed Systems

Winter Term 2025/26

3 Distributed Programming



Content

➔ RMI

- ➔ Introduction

- ➔ *Hello World* with RMI

- ➔ RMI in more detail

- ➔ classes and interfaces, stubs, name service, parameter passing, factories, callbacks, ...

➔ gRPC



Literature

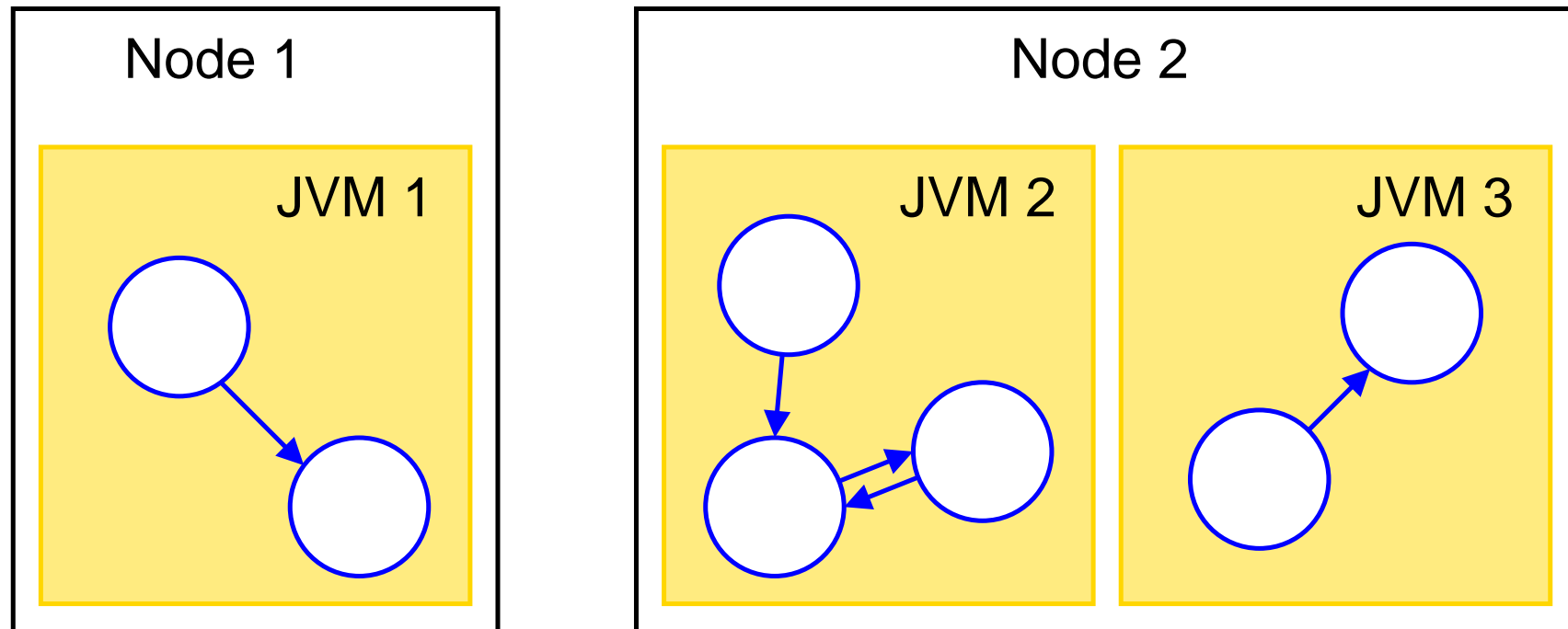
- ➔ WWW documentation and tutorials from Oracle
 - ➔ <https://docs.oracle.com/en/java/javase/17/docs/api/java.rmi/java/rmi/package-summary.html>
 - ➔ <https://docs.oracle.com/javase/8/docs/technotes/guides/rmi>
- ➔ Hammerschall: Ch.. 5.2
- ➔ Farley, Crawford, Flanagan: Ch. 3
- ➔ Horstmann, Cornell: Ch. 5
- ➔ Orfali, Harkey: Ch. 13
- ➔ gRPC home page
 - ➔ <https://grpc.io>



3.1.1 Introduction

- ➔ Java RMI supports the use of remote objects in Java
 - ➔ integral part of Java (package `java.rmi`)
- ➔ Essential elements:
 - ➔ remote object implementations
 - ➔ client interfaces (stubs) to remote objects
 - ➔ server skeletons for remote object implementations
 - ➔ name service to locate objects in the network
 - ➔ communication protocol
- ➔ All objects (i.e., client and server) must be programmed in Java
 - ➔ allows seamless integration into the language (including distributed garbage collection)
 - ➔ RMI/IIOP allows interoperability with CORBA

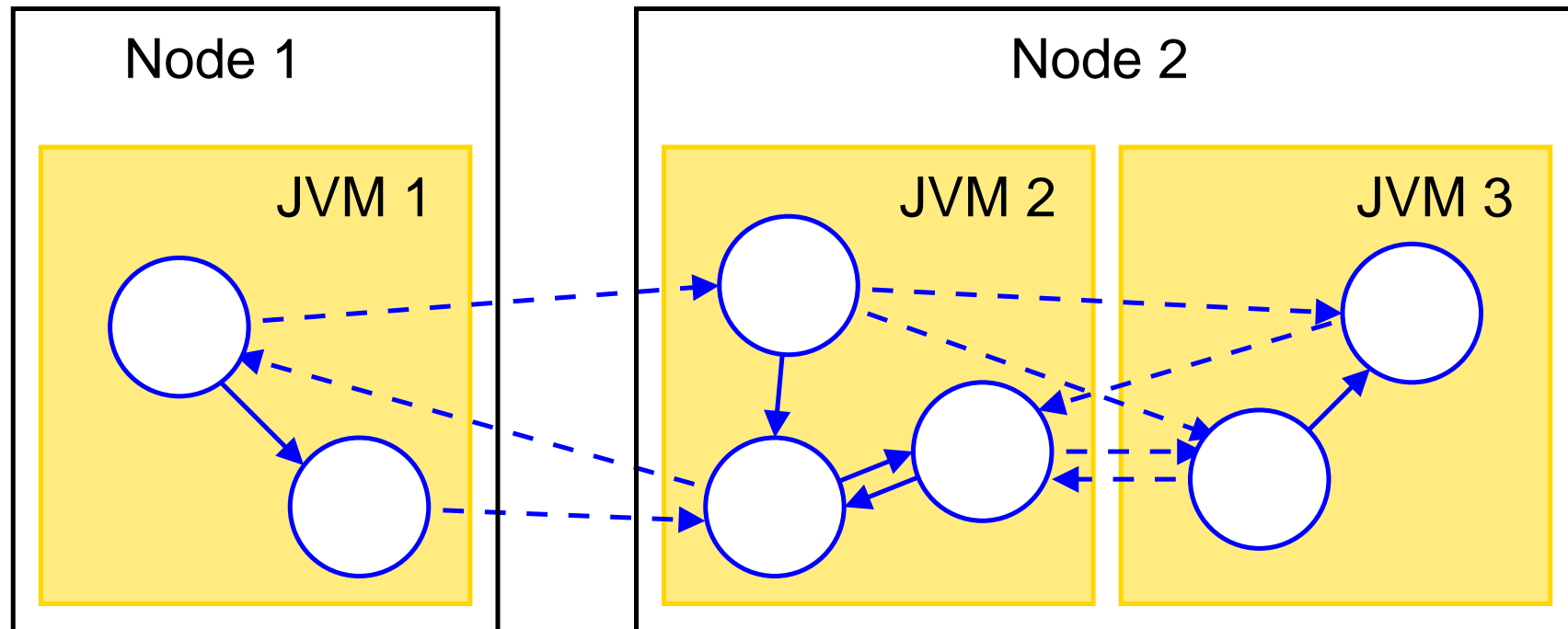
Distributed Objects



→ local reference

- ➔ Remote references can be used just like local references
- ➔ Objects can occur in client and server roles

Distributed Objects



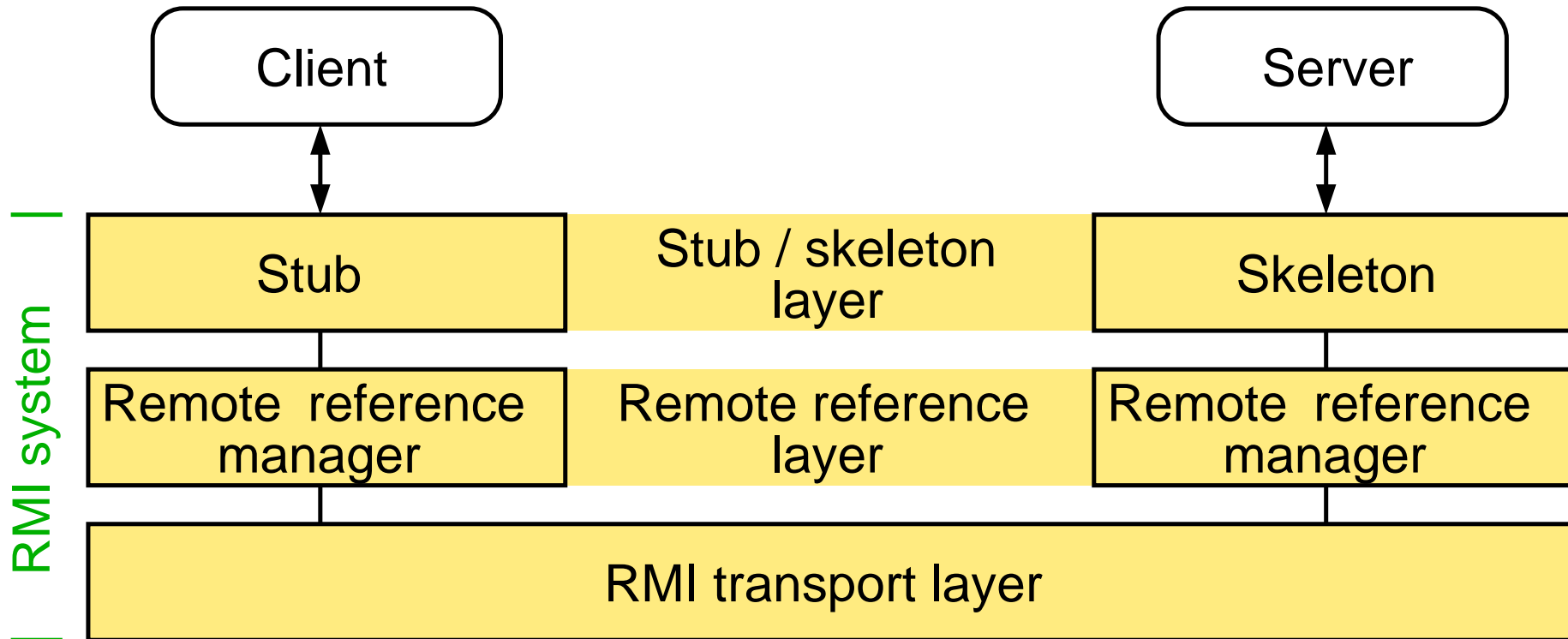
—→ local reference

- - -→ remote reference

- ➔ Remote references can be used just like local references
- ➔ Objects can occur in client and server roles



3.1.2 RMI Architecture





Stub/Skeleton Layer

- ➔ Stub and skeleton classes are automatically generated at runtime
 - ➔ from a Java interface

Remote Reference Layer

- ➔ Defines call semantics (unicast, multicast, object activation)
- ➔ Also: connection management, distributed garbage collection

Transport Layer

- ➔ Proprietary protocol: Java Remote Method Protocol (JRMP)
 - ➔ allows tunneling via HTTP (due to firewalls)
 - ➔ allows to use TLS via a socket factory
- ➔ Alternative: RMI-IIOP



3.1.3 RMI Services

➔ Name service: RMI Registry

- ➔ registers remote references to RMI objects under freely selectable unique names
- ➔ a client can then get the corresponding reference for a name
 - ➔ registry sends a serialized proxy object (client stub) to the client.
- ➔ RMI can also be used with other naming services, e.g. via JNDI (Java Naming and Directory Interface)

Distributed Systems

Winter Term 2025/26

06.11.2025

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

Stand: January 8, 2026



➔ Distributed Garbage Collection

- ➔ automatic garbage collection of Java also works for remote objects
- ➔ server-side JVM manages a list of remote references to objects
- ➔ references are “leased” for a certain time
- ➔ reference counter of the object is decremented, if
 - ➔ client deletes the reference (e.g., end of the lifetime of the reference variable), or
 - ➔ client does not renew the lease in time
 - ➔ reason: remote reference layer cannot explicitly “log off” an object, if the client crashes
 - ➔ default setting: 10 min.



3.1.4 Example: *Hello World* with Java RMI

Client JVM

Interface

```
interface Hello {  
    String sayHello();  
}
```

Server JVM

Client class

```
class HelloClient {  
    ...  
    Hello h;  
    ...  
    s = h.sayHello();  
    ...  
}
```

Server class

```
class HelloServer  
    implements Hello {  
    String sayHello() {  
        return "Hello World";  
    }  
    ...  
}
```



Development Process:

1. Design the interface for the server object
2. Implement the server class
3. Develop the server application to include the server object
4. Develop the client application with calls to the server object
5. Compile and start the system



Designing the Interface for the Server Object

- ➔ Specified as normal Java interface
- ➔ Must extend `java.rmi.Remote`
 - ➔ no inheritance of operations, only marking as remote interface
- ➔ Each method must declare to raise the exception `java.rmi.RemoteException` (or a base class of it)
 - ➔ base class for all errors that may occur
 - ➔ in the client, during transmission, in the server
- ➔ No restrictions compared to local interfaces
 - ➔ but: semantic differences (parameter passing!)



Hello-World Interface

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Hello extends Remote {  
    String sayHello() throws RemoteException;  
}
```

Marker interface,
contains no methods,
marks interface as
RMI interface

RemoteException
indicates error in the
remote object or
during communication



Implementing the Server Class

- ➔ A class that is to be usable remotely must:
 - ➔ implement a given remote interface
 - ➔ usually extend `java.rmi.server.UnicastRemoteObject`
 - ➔ defines call semantics: point-to-point
 - ➔ have a constructor that declares to throw a `RemoteException`
 - ➔ creation of object must be done in a try-catch block
- ➔ Methods usually do not need to specify `throws RemoteException`
 - ➔ because they don't throw the exception themselves



Hello-World Server (1)

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloServer extends UnicastRemoteObject
    implements Hello {
    public HelloServer() throws RemoteException {
        super();
    }
    public String sayHello() {
        return "Hello World!";
    }
}
```

Remote method



Development of the Server Application to Include the Server Object

- ➔ Tasks:
 - ➔ creating a server object
 - ➔ registering the object with the name service
 - ➔ under a specified public name
- ➔ Typically not a new class, but `main` method of the server class



Hello-World Server (2)

```
public static void main(String args[]) {
    try {
        HelloServer obj = new HelloServer();
        Naming.rebind("rmi://localhost/Hello-Server", obj);
    }
    catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Create the server object

Register the server object under the name "Hello-Server" with the name server (RMI registry, local host, port 1099)



Development of the Client Application with Calls to the Server Object

- ➔ Client must first use the name service to get a reference to the server object from the name service
 - ➔ type cast to the correct type required
- ➔ Then: any method can be called
 - ➔ no syntactical differences to local calls
- ➔ Note: client can get remote references in other ways as well
 - ➔ e.g. as return value of a remote method



Hello-World Client

```
import java.rmi.*;

public class HelloClient {
    public static void main(String args[]) {
        try {
            Hello obj =
                (Hello)Naming.lookup("rmi://bspc02/Hello-Server");
            String message = obj.sayHello();
            System.out.println(message);
        }
        catch (Exception e) {
            ...
        }
    }
}
```

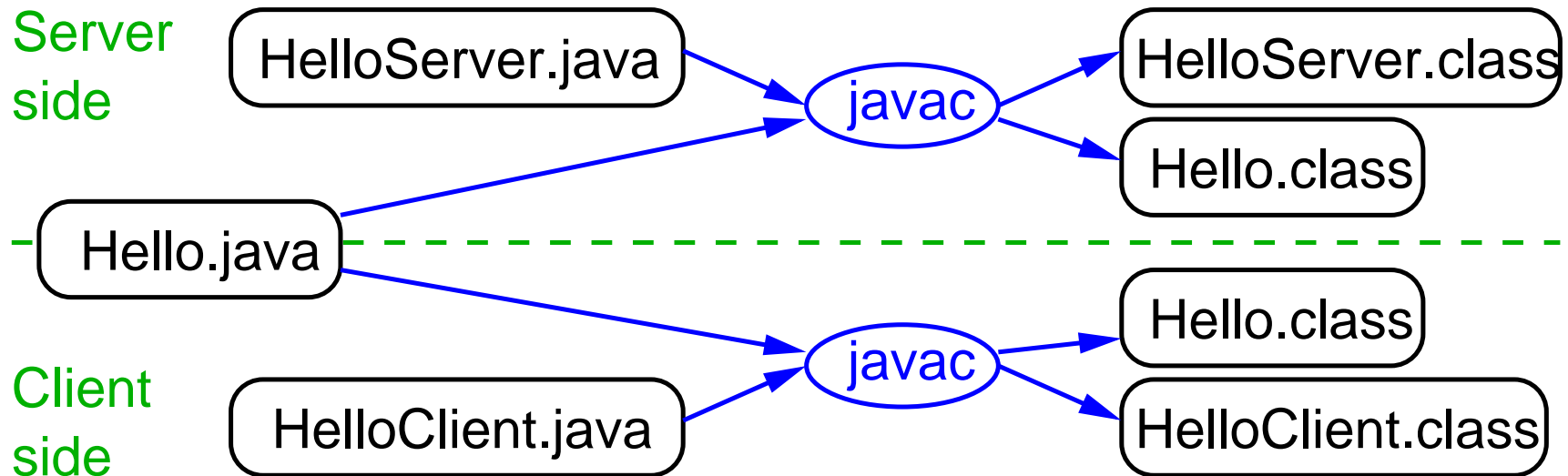
Get object reference from name server

Call the method on the remote object



Compiling and Starting the System

➔ Compiling:



➔ Starting the naming service: `rmiregistry [port]`

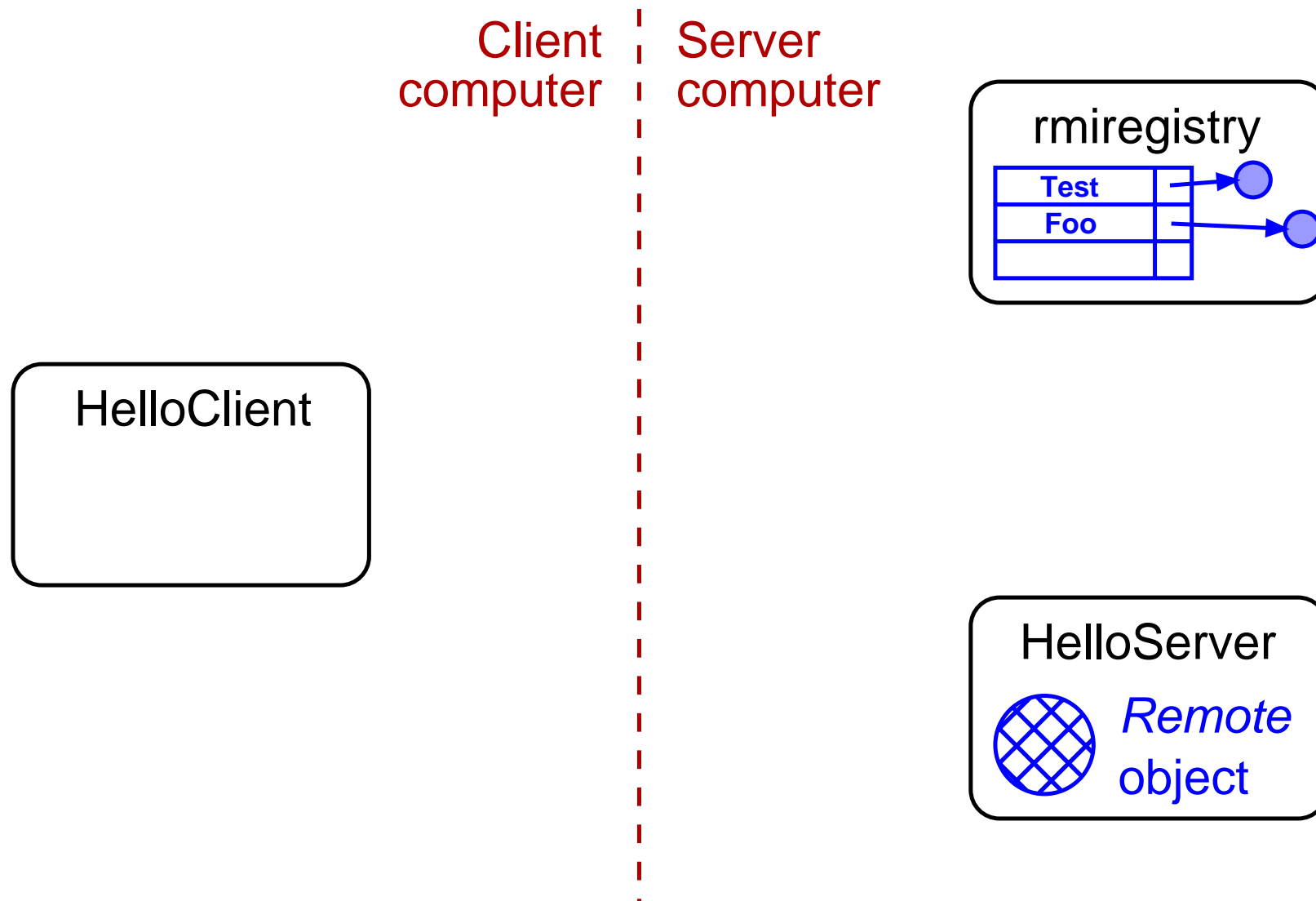
➔ for security reasons, objects can only be registered on the local host, i.e. RMI registry must run on server computer

➔ Starting the server: `java HelloServer`

➔ Starting the client: `java HelloClient`

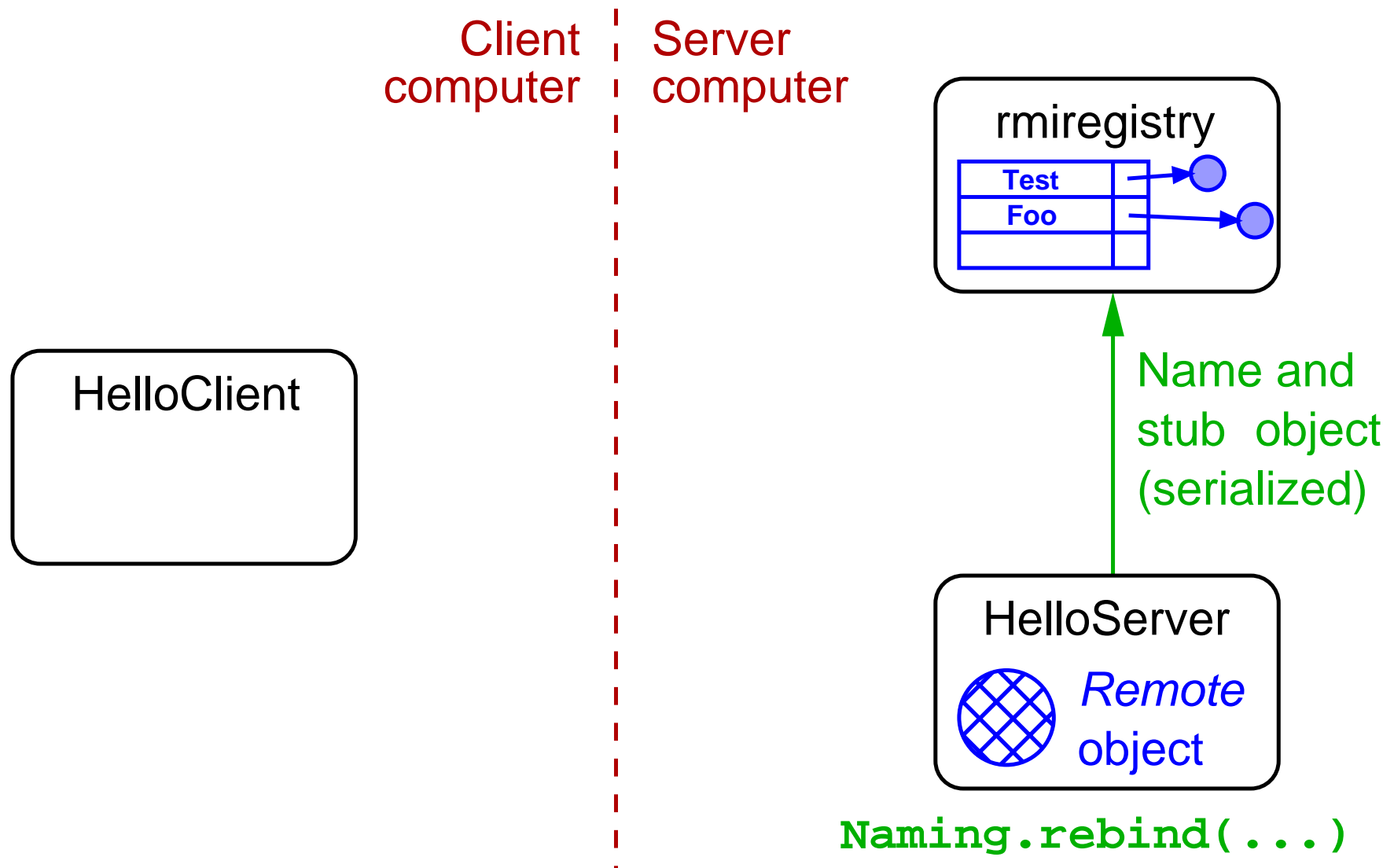


Execution of the Example



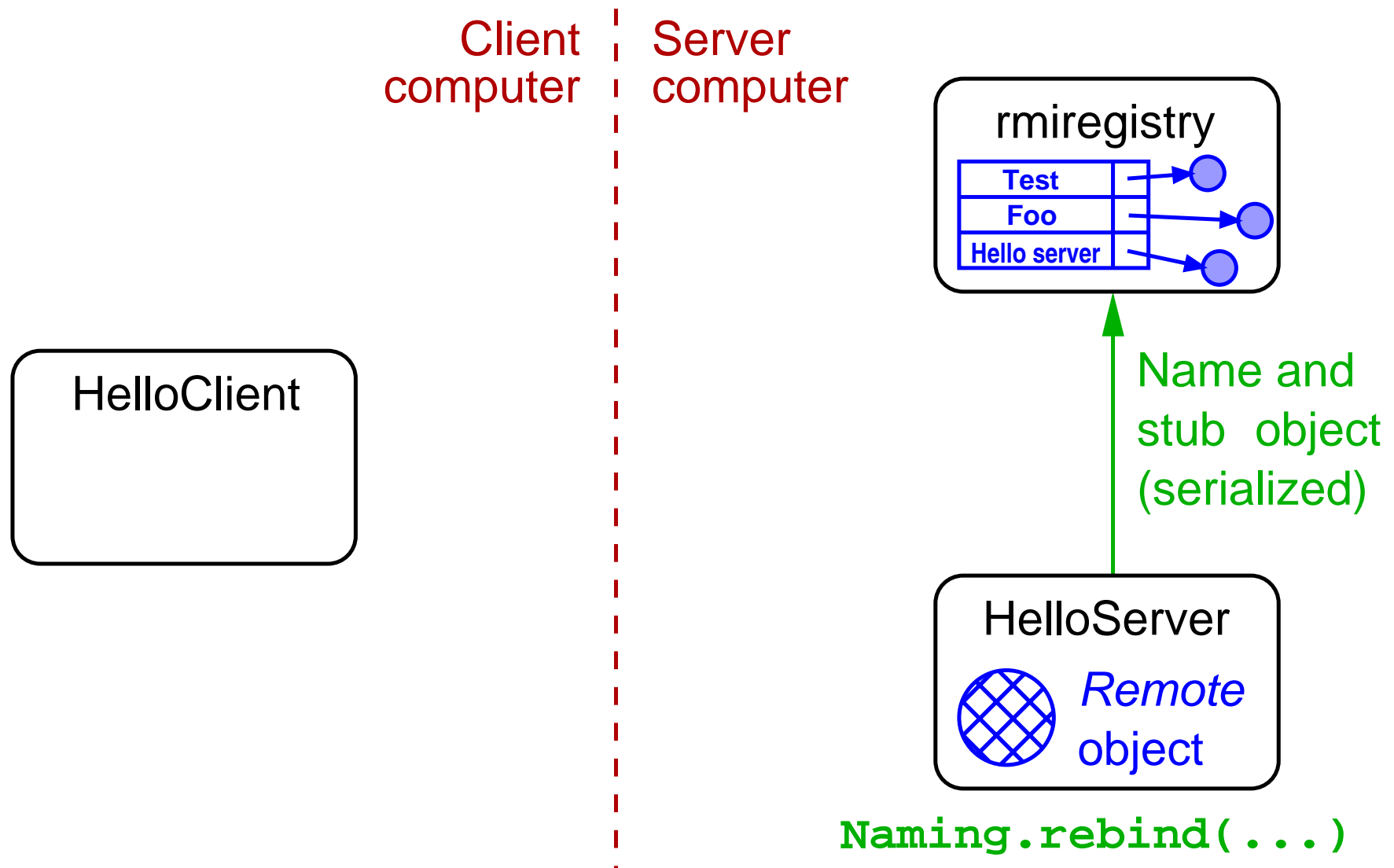


Execution of the Example



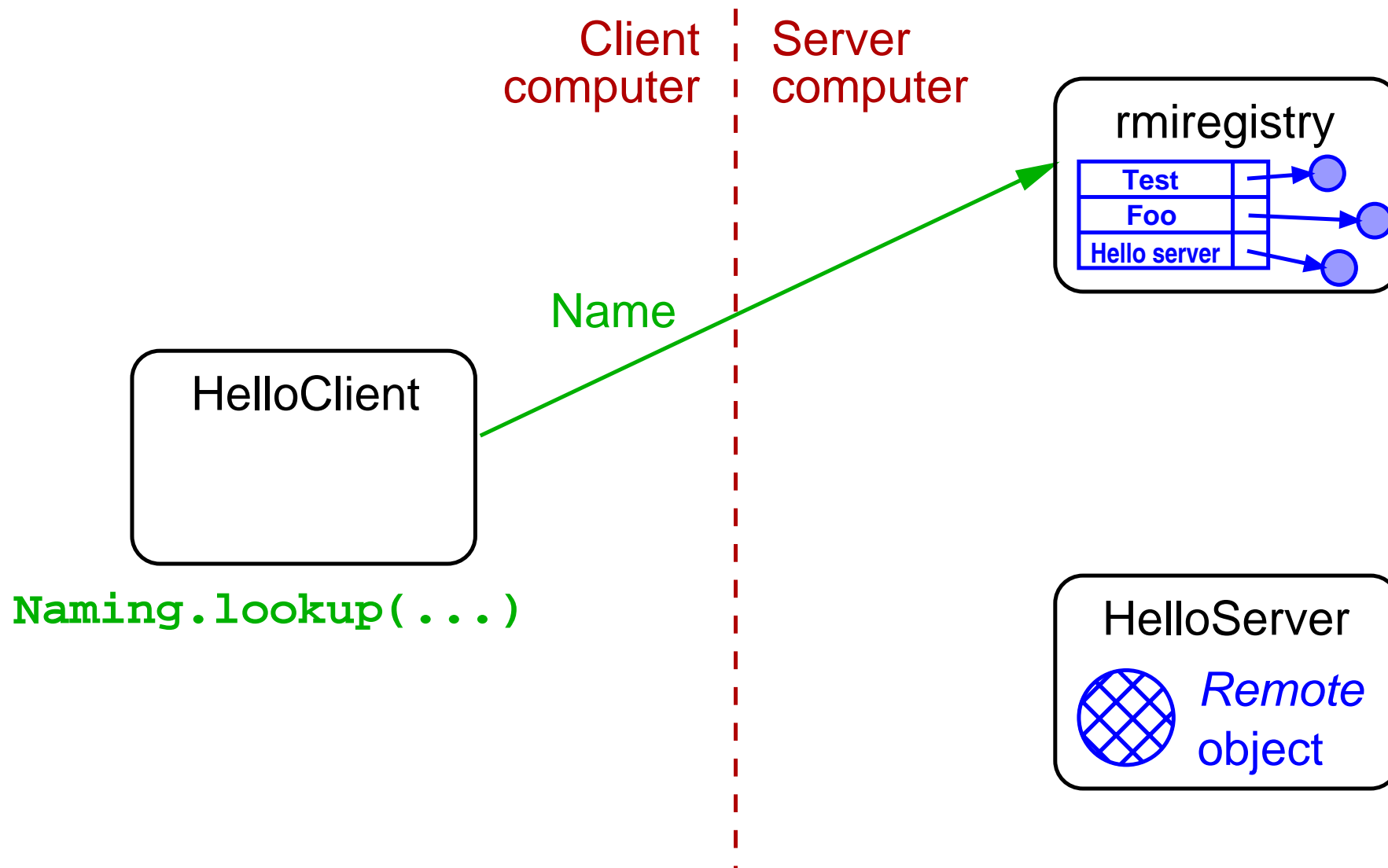


Execution of the Example



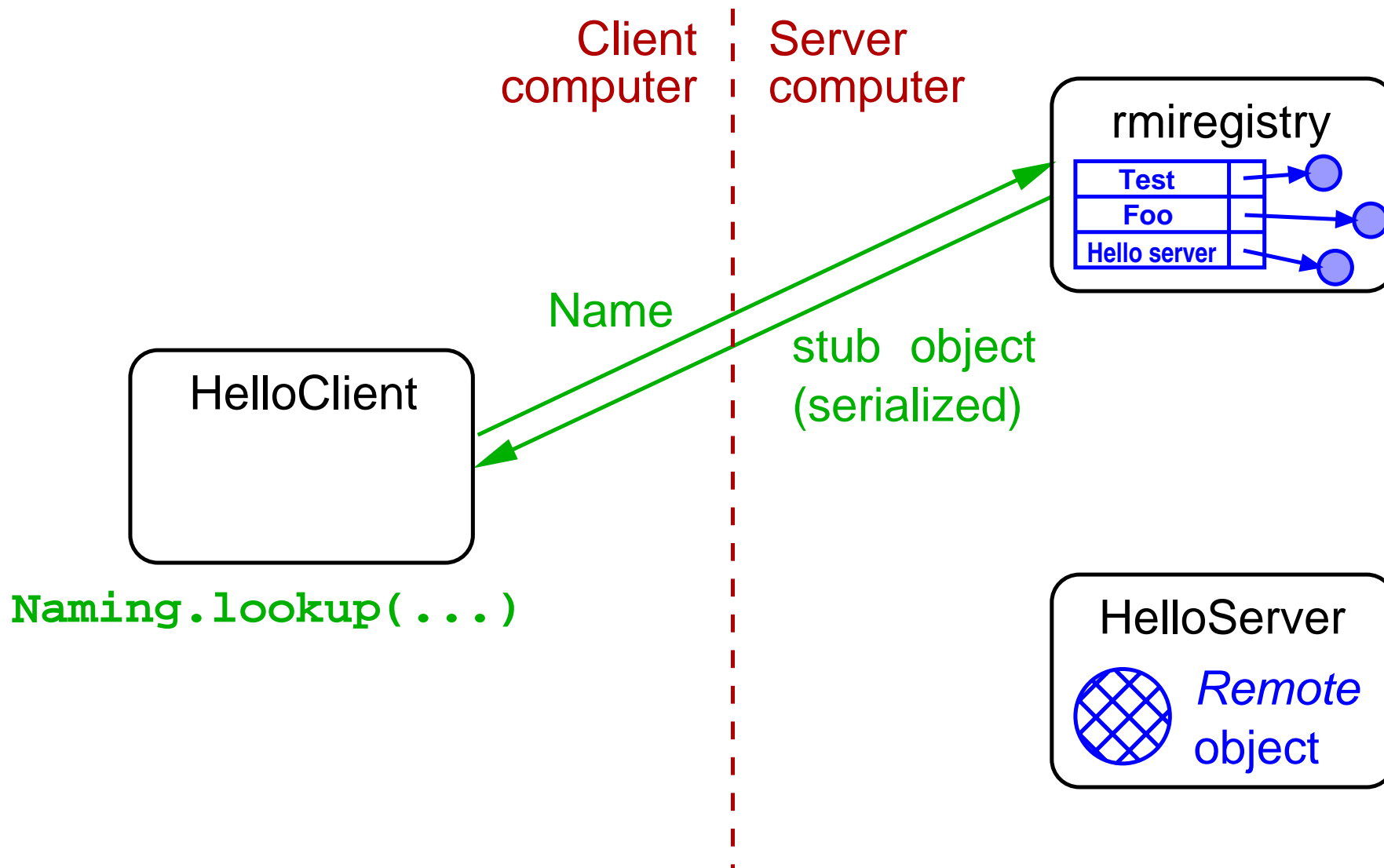


Execution of the Example



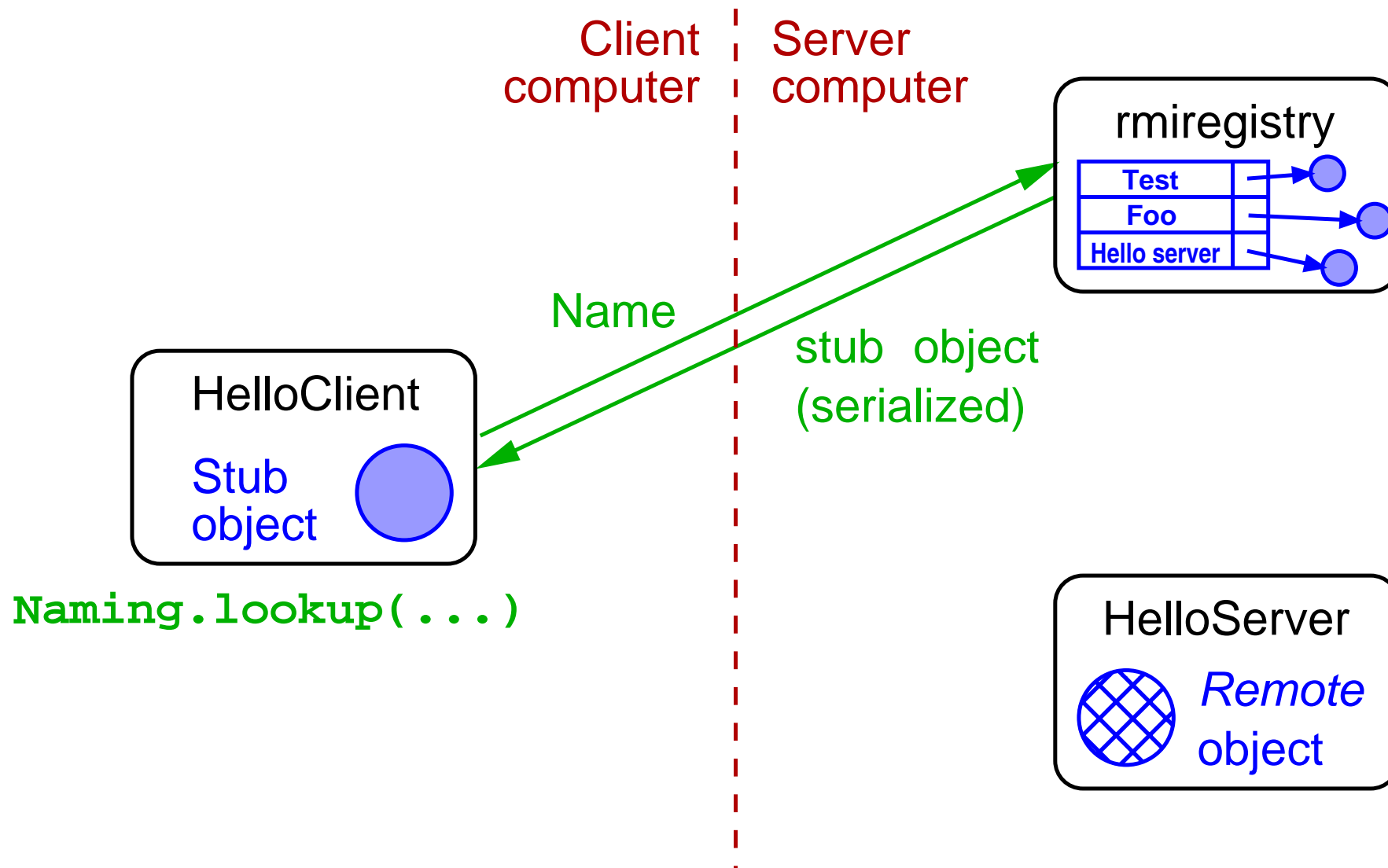


Execution of the Example





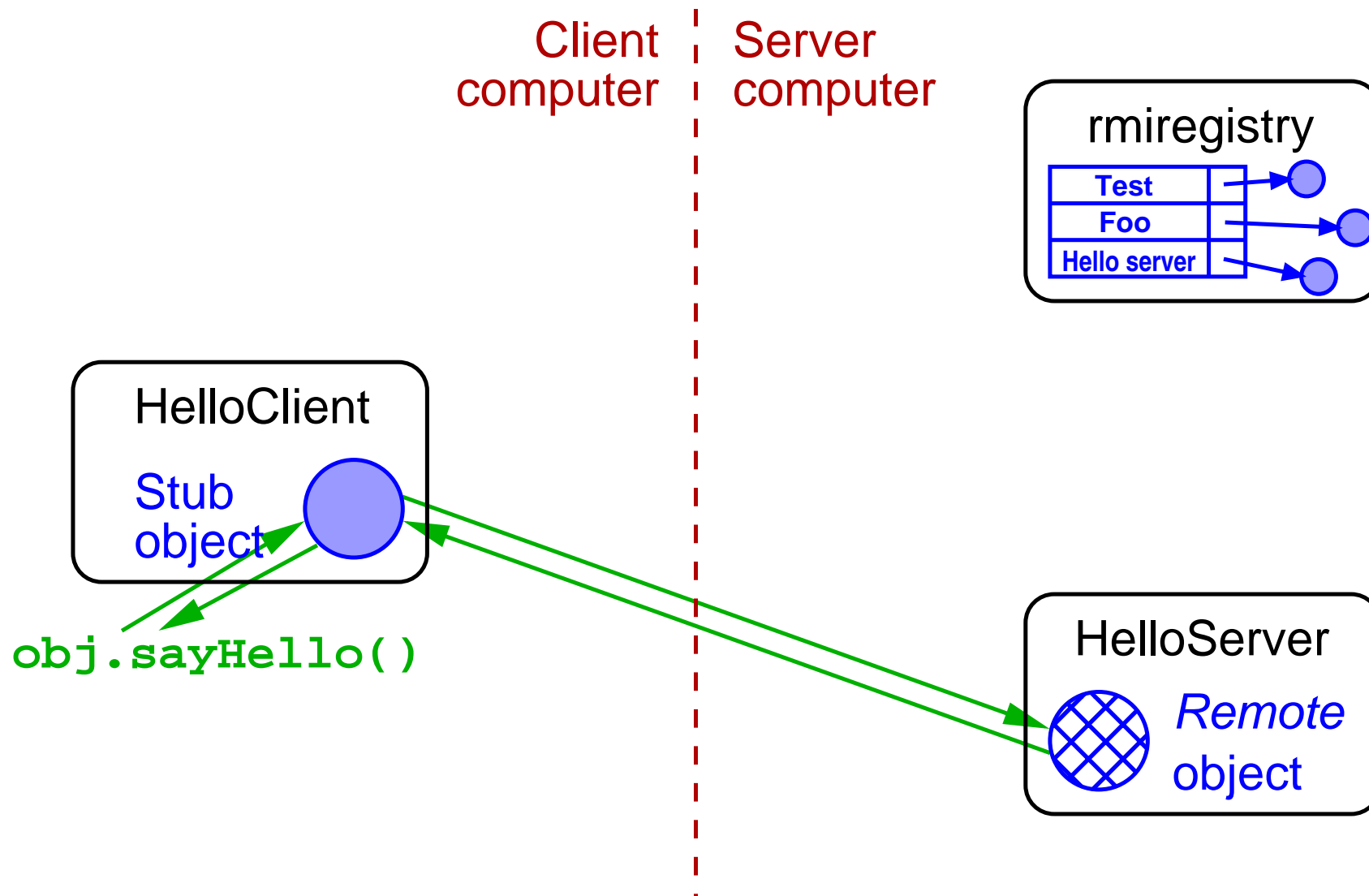
Execution of the Example



3.1.4 Example: *Hello World* with Java RMI ...

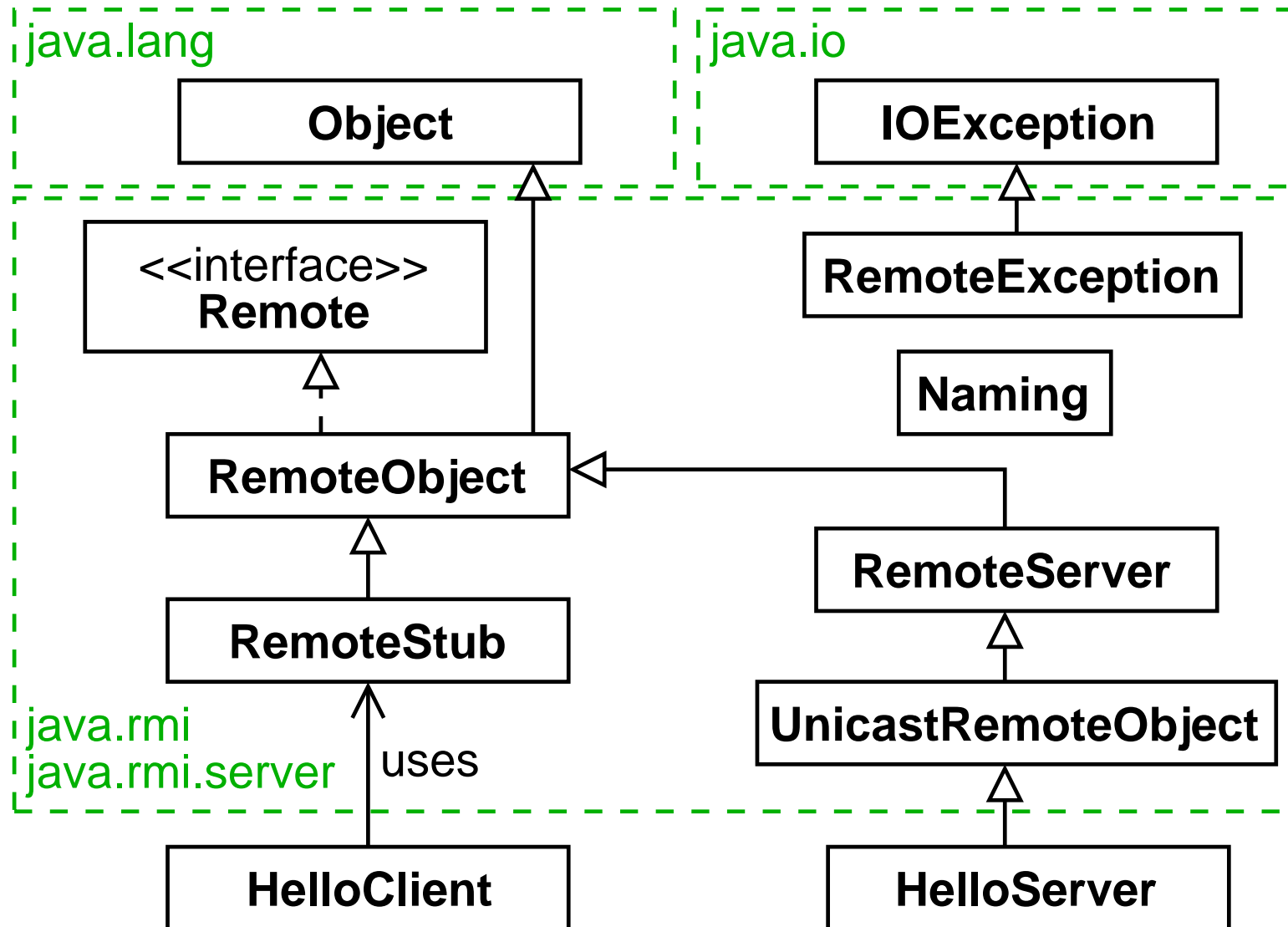


Execution of the Example





3.1.5 Classes and Interfaces





Interface `Remote`

- ➔ Every remote object must implement this interface
- ➔ Does not provide methods, serves only as a marker

Class `RemoteException`

- ➔ Superclass for all exceptions that can be triggered by the RMI system, for example, with
 - ➔ communication errors (server not reachable, ...)
 - ➔ (un-)marshalling errors
 - ➔ protocol errors
- ➔ Each remote method must specify `RemoteException` (or a base class of it) in the `throws` clause



Class `RemoteObject`

- ➔ Base class for all remote objects
- ➔ Redefines the methods `equals`, `hashCode`, and `toString`
- ➔ Static method `toStub()` returns a reference to the stub object

Class `RemoteStub`

- ➔ Base class for all client stubs

Class `RemoteServer`

- ➔ Base class for `UnicastRemoteObject`
- ➔ Method `getClientHost()`: host address of the client of the current RMI call
- ➔ `setLog()` and `getLog()`: logging of RMI calls



Class `UnicastRemoteObject`

- ➔ Implements remote object with the following properties:
 - ➔ references to the object are only valid as long as server process (JVM) is still running
 - ➔ client call is routed to exactly one object (via TCP connection), no replication
- ➔ Constructor allows definition of port and socket factories
 - ➔ so that e.g. connections via TLS/SSL can be realized
- ➔ Static method `exportObject()` makes object available via RMI
- ➔ Static method `unexportObject()` cancels availability



Class Naming

- ➔ Allows easy access to RMI registry
- ➔ Important methods:
 - ➔ `bind() / rebind()`: registers object under given name
 - ➔ `lookup()`: get object reference to a name
- ➔ Names are given in URL format
 - ➔ also define the host and port of the RMI registry.
 - ➔ structure of the URL:

`rmi://bspc02:1234/Hello`

Protocol (always rmi)

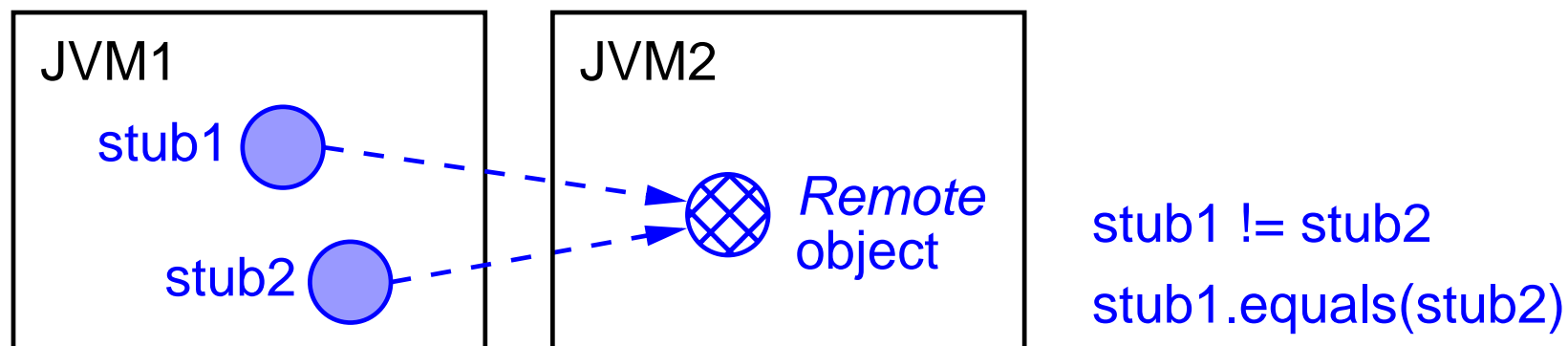
Host of RMI registry

Port of the RMI registry

Name of the registered Object

3.1.6 Special Characteristics of Remote Classes

- ➔ Comparison of remote objects
 - ➔ Comparison with `==` refers only to the stub objects
 - ➔ Result is `false`, even if both stubs refer to the same remote object
 - ➔ comparison with `equals()` returns true if both stubs refer to the same remote object





- ➔ Method `hashCode()`
 - ➔ used by container classes `HashMap`, `HashSet` and others
 - ➔ Hash code is calculated only from the object identifier of the remote object
 - ➔ same remote object \Rightarrow same hash code
 - ➔ but the content of the object is ignored
 - ➔ consistent with behavior of `equals()`
- ➔ Cloning objects
 - ➔ cloning of the remote object is not possible by calling `clone()` on the stub
 - ➔ cloning of stubs neither necessary nor meaningful



3.1.7 Parameter Passing

- ➔ Parameters passed to remote methods
 - ➔ either via *call-by-value*
 - ➔ or via *call-by-reference*
- ➔ The mechanism used depends on the type of the parameter
- ➔ Final decision may only be made at runtime!

- ➔ The return of the result follows the same rules as for parameter passing



Parameter Passing for Local Methods

- ➔ Java supports two kinds of types:
 - ➔ **value types**: simple data types
 - ➔ boolean, byte, char, short, int, long, float, double
 - ➔ are passed to local methods *by value*
 - ➔ that is, the method receives a copy of the value
 - ➔ **reference types**: classes (incl. String and arrays)
 - ➔ are passed to local methods *by reference*
 - ➔ that is, the method works on the original object and can also change object if required



Parameter Passing for Remote Methods

- ➔ Value types: are always passed *by value*
- ➔ Reference types: dependent on the concrete object
 - ➔ object can be serialized: *call-by-value*
 - ➔ object belongs to a class that implements the Remote interface: *call-by-reference*
 - ➔ neither: error (`java.rmi.MarshalException`)
 - ➔ both: `??!` (this case is to be avoided!)
 - ➔ decision is made only at runtime

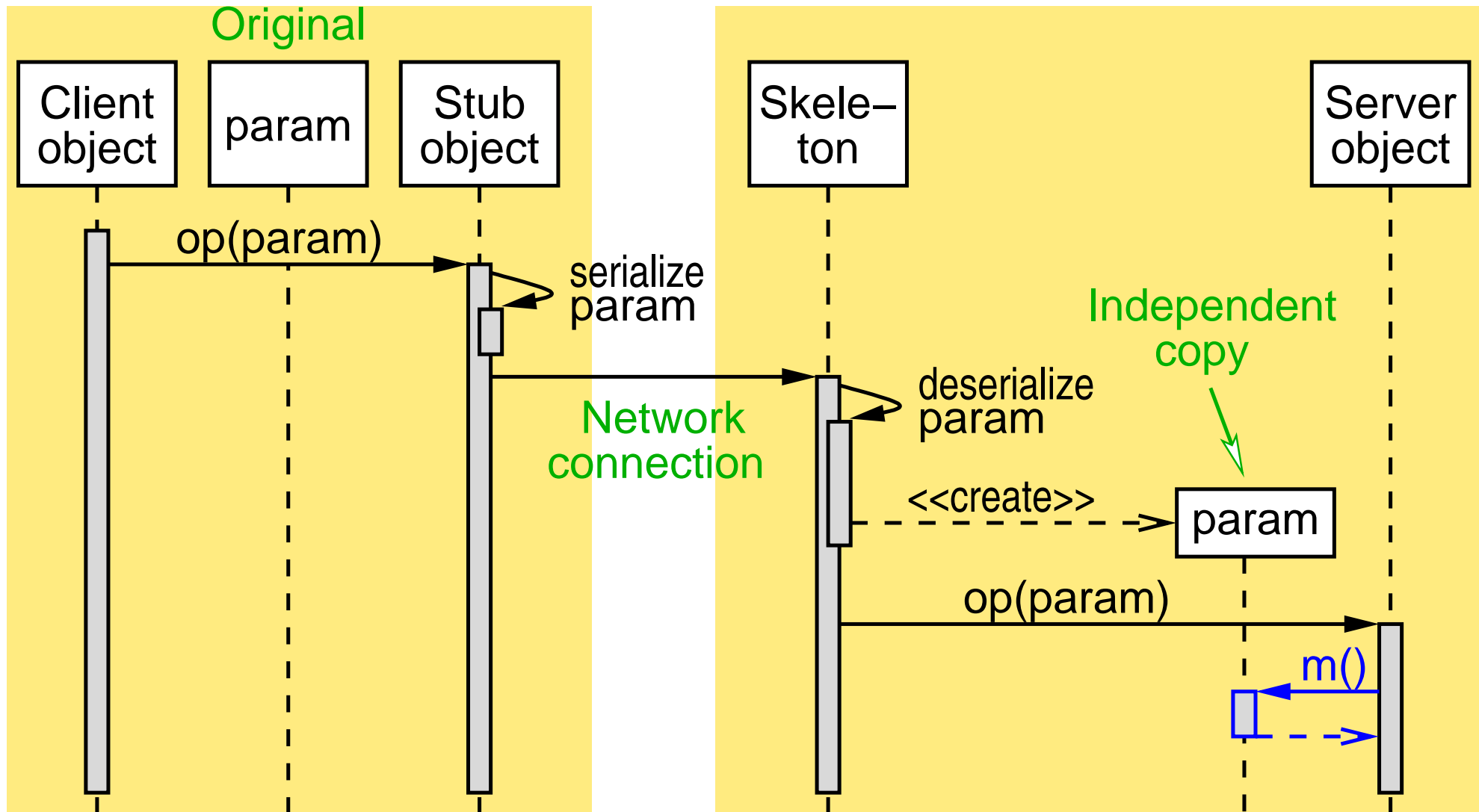


Call-by-Value (Serializable Objects)

- ➔ Class must implement interface `java.io.Serializable`
- ➔ Serializable objects can be transferred over a network
 - ➔ only the data is transferred, the code (`class` file) must be available at the receiver!
- ➔ Default serialization of Java:
 - ➔ all attributes of the object are serialized and transferred
 - ➔ recursive procedure!
 - ➔ prerequisite: all attributes and all base classes can be serialized
- ➔ Application specific serialization is possible:
 - ➔ implement the methods `writeObject` and `readObject`



Passing a Serializable Object



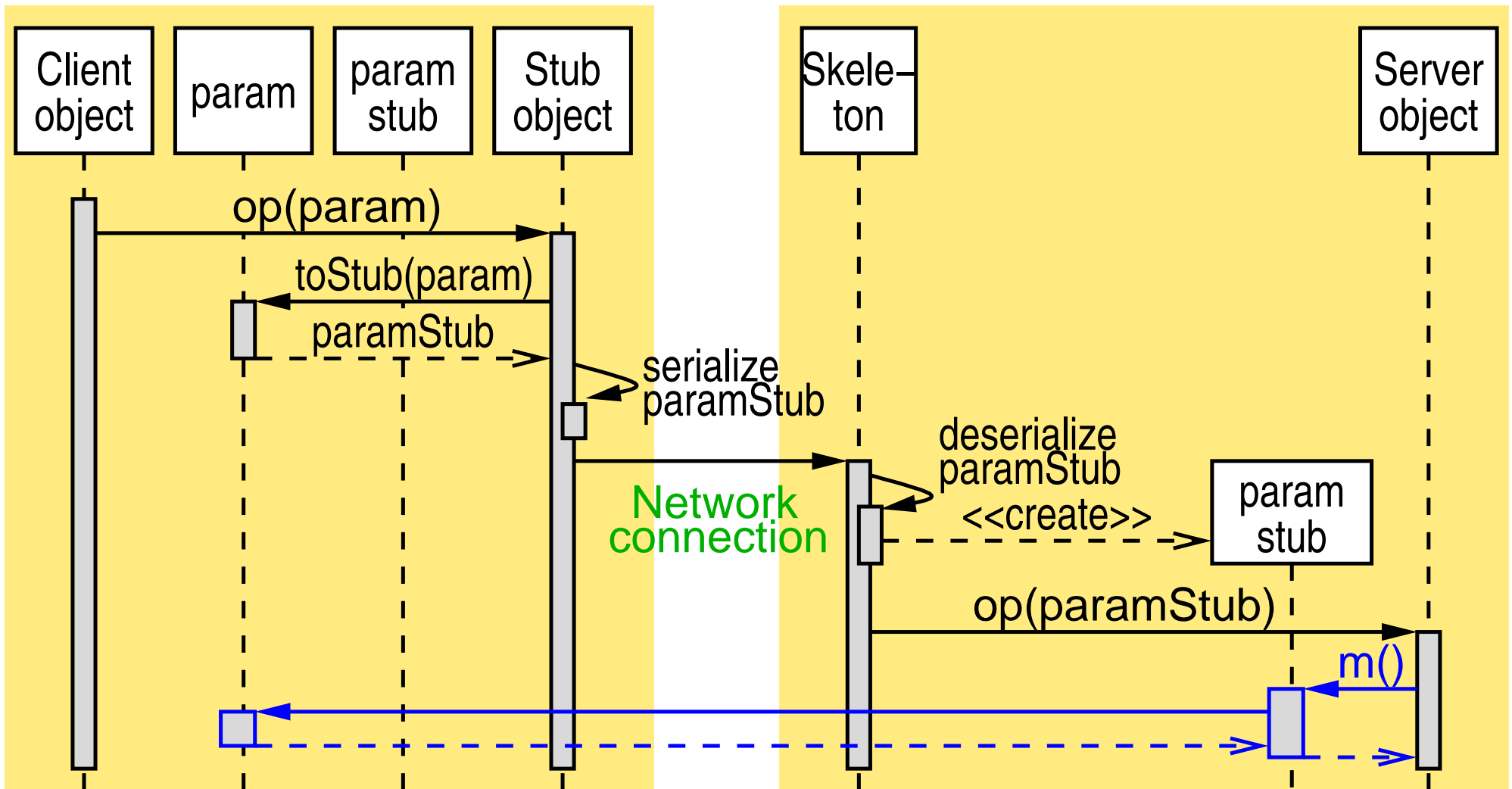


Call-by-Reference (Remote Objects)

- ➔ Class of the parameter object must implement an interface that extends `Remote`
 - ➔ parameter type must be this interface
 - ➔ class is typically derived from `UnicastRemoteObject`
- ➔ A serialized stub object is transferred
 - ➔ stub class is created dynamically
- ➔ If the server calls methods on the parameter object:
 - ➔ calls are routed to the original object using RMI



Passing a Remote Object





Examples

- ➔ See WWW:
 - ➔ *Hello-World* with *call-by-value* parameter
 - ➔ *Hello-World* with *call-by-reference* parameter



Arrays and Container Objects

- ➔ Arrays and container objects (from the Java Collection Framework, `java.util`) can be serialized
 - ➔ i.e., they will be reinstantiated at the receiver
- ➔ To the elements of the array / container the same rules apply as to simple parameters
 - ➔ for mixed content: elements are passed *by value* or *by reference* depending on their actual class



3.1.8 Remote Object References as Results

- ➔ Frequently: via RMI registry, the client receives a reference to a remote object, which provides references to other objects
 - ➔ the remote object may also create these objects on demand (*factory object / factory class*)
- ➔ Example: server for bank accounts
 - ➔ registration of all account objects with RMI registry not useful
 - ➔ instead: registration of a manager object that returns the reference to the account object for a given account number
 - ➔ if necessary, it can create a new object (from a database)
- ➔ Note: RMI does not allow remote object creation
 - ➔ client cannot create objects on a remote host



3.1.9 Client Callbacks

- ➔ Frequently: server wants to make calls in the client
 - ➔ e.g. progress bar, queries, ...
- ➔ For this: client object must be an RMI object
 - ➔ pass `this` reference to the server method
- ➔ In some cases, you cannot inherit from `UnicastRemoteObject`, e.g. for applets
 - ➔ then: export the object using
`UnicastRemoteObject.exportObject(obj, 0);`
- ➔ Example code: see WWW (*Hello-World with callback*)



3.1.10 RMI and Threads

- ➔ RMI does not specify how many threads are provided on the server side for method calls
 - ➔ only one thread, one thread per call, ..
- ➔ This means that several server methods can be active at the same time
 - ➔ requires correct synchronization (synchronized)!
- ➔ Client-side locking of a remote object using a synchronized block is not possible
 - ➔ only local stub is locked
 - ➔ a lock must be implemented using methods of the remote object if necessary

Distributed Systems

Winter Term 2025/26

13.11.2025

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

Stand: January 8, 2026



3.1.11 Deployment

- ➔ **Deployment:** distribution, transfer and installation of the components of a distributed application
 - ➔ specifically for RMI: which `class` file has to go where?
- ➔ Server, RMI registry and client need the `class` files for:
 - ➔ their own implementation
 - ➔ the remote server interface plus all classes / interfaces used therein (recursively)
- ➔ Client and server in addition need the classes of received serializable objects
- ➔ RMI allows remote loading of the required `class` files
 - ➔ security issue, especially since Java Security Manager is deprecated



3.2 gRPC (Google Remote Procedure Call)

- ➔ Open source framework for RPC-based services
- ➔ Support for many languages (C++, Go, Java, Python, Rust, ...)
- ➔ Service is described by a special interface definition language (*protocol buffer language*)
 - ➔ defines structure of messages
 - ➔ defines request and reply message for each procedure
- ➔ *Protocol buffers*: binary serialization format for messages
 - ➔ better efficiency and type safety than JSON
- ➔ Communication is based on HTTP/2
 - ➔ binary format, can multiplex several streams, allows server push messages



Example: Interface Definition

```
syntax = "proto3"; // Use protocol buffers revision 3
```

```
service Greeter {  
    // Procedure with request and reply message  
    rpc SayHello (HelloRequest) returns (HelloReply);  
}
```

```
message HelloRequest { // Request message format  
    string name = 1; // '1' is the field number  
    int32 number = 2;  
}
```

```
message HelloReply { // Reply message format  
    string greeting = 1;  
}
```



Example: Server

```
import io.grpc.Grpc;
import io.grpc.InsecureServerCredentials;
import io.grpc.Server;
import io.grpc.stub.StreamObserver;

public class HelloWorldServer {
    public static void main(String[] args) throws Exception {
        Server server = Grpc.newServerBuilderForPort(12345,
            InsecureServerCredentials.create())
            .addService(new GreeterImpl())
            .build()
            .start();
        server.awaitTermination();
    }
}
```



Example: Server

```
import io.grpc.Grpc;  
import io.grpc.InsecureServerCredentials;  
import io.grpc.Server;  
import io.grpc.stub.StreamObserver  
  
public class HelloWorldServer {  
    public static void main(String [] args) throws Exception {  
        Server server = Grpc.newServerBuilderForPort(12345,  
            InsecureServerCredentials.create())  
            .addService(new GreeterImpl())  
            .build()  
            .start();  
        server.awaitTermination();  
    }  
}
```

Create new server builder
with given port and credentials



Example: Server

```
import io.grpc.Grpc;  
import io.grpc.InsecureServerCredentials;  
import io.grpc.Server;  
import io.grpc.stub.StreamObserver;
```

```
public class Hello {  
    public static void main(String[] args) throws Exception {  
        Server server = ServerBuilderForPort(12345,  
            InsecureServerCredentials.create())  
            .addService(new GreeterImpl())  
            .build()  
            .start();  
        server.awaitTermination();  
    }  
}
```

Add the service implementation



Example: Server

```
import io.grpc.Grpc;  
import io.grpc.InsecureServerCredentials;  
import io.grpc.Server;  
import io.grpc.stub.StreamObserver;
```

```
public class HelloWorldServer {  
    public static void  
        Server server = (  
            .addService(new  
            .build()  
            .start();  
    server.awaitTermination();  
}
```

Create and start
the server

```
s) throws Exception {  
    derForPort(12345,  
    edentials.create())
```



Example: Server ...

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello>HelloRequest req,
                               StreamObserver>HelloReply> resp) {
        String hello = "Hello " + req.getName()
                      + ", " + req.getNumber();
       >HelloReply reply =>HelloReply.newBuilder()
            .setGreeting(hello)
            .build();
        resp.onNext(reply);
        resp.onCompleted();
    }
}
```



Example: Server ...

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {
  @Override
  public void sayHello(HelloRequest req,
                      StreamObserver<HelloReply> resp) {
    String hello = "Hello " + req.getName()
                  + ", " + req.getNumber();
    HelloReply reply = HelloReply.newBuilder()
      .setGreeting(hello)
      .build();
    resp.onNext(reply);
    resp.onCompleted();
  }
}
```

Read fields of request message



Example: Server ...

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(HelloRequest req,
                        StreamObserver<HelloReply> resp) {
        String hello = "Hello " + req.getName()
                    + ", " + req.getAge() + " years old";
        HelloReply reply = HelloReply.newBuilder()
            .setGreeting(hello)
            .build();
        resp.onNext(reply);
        resp.onCompleted();
    }
}
```

Build the reply message



Example: Server ...

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(HelloRequest req,
                        StreamObserver<HelloReply> resp) {
        String hello = "Hello " + req.getName()
                    + ", " + req
        HelloReply reply = HelloReply
            .setGreeting(hello)
            .build();
        resp.onNext(reply);
        resp.onCompleted();
    }
}
```

Push the message to the client
(this invokes the observer's
onNext() method)



Example: Server ...

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(HelloRequest req,
                        StreamObserver<HelloReply> resp) {
        String hello = "Hello " + req.getName()
                    + ", " + req.getNumber();
        HelloReply reply = HelloReply.newBuilder()
            .setGreeting(hello)
            .build();
        resp.onNext(reply);
        resp.onCompleted();
    }
}
```

No more response messages
(this invokes the observer's
onCompleted() method)



Example: Client

```
import io.grpc.Grpc;
import io.grpc.InsecureChannelCredentials;
import io.grpc.ManagedChannel;

public class HelloWorldClient {
    public static void main(String[] args) throws Exception {
        ManagedChannel channel =
            Grpc.newChannelBuilder("localhost:12345",
                InsecureChannelCredentials.create())
                .build();
    }
}
```



Example: Client

```
import io.grpc.Grpc;  
import io.grpc.InsecureChannelCredentials;  
import io.grpc.ManagedChannel;  
  
public class HelloWorld {  
    public static void main(String[] args) throws Exception {  
        ManagedChannel channel =  
            Grpc.newChannelBuilder("localhost:12345",  
                InsecureChannelCredentials.create())  
                .build();  
    }  
}
```

Build a communication channel to the server



Example: Client ...

```
GreeterGrpc.GreeterBlockingStub blockingStub =
    GreeterGrpc.newBlockingStub(channel);
HelloRequest req = HelloRequest.newBuilder()
    .setName("world")
    .setNumber(42)
    .build();
HelloReply resp = blockingStub.sayHello(req);
System.out.println(resp.getGreeting());
channel.shutdown();
}
}
```



Example: Client ...

Create a new stub for synchronous RPCs

```
GreeterGrpc.GreeterBlockingStub blockingStub =
    GreeterGrpc.newBlockingStub(channel);
HelloRequest req = HelloRequest.newBuilder()
    .setName("world")
    .setNumber(42)
    .build();
HelloReply resp = blockingStub.sayHello(req);
System.out.println(resp.getGreeting());
channel.shutdown();
}
```



Example: Client ...

```
GreeterGrpc.GreeterBlockingStub blockingStub = GreeterGrpc.newBlockingStub(channel);
HelloRequest req = HelloRequest.newBuilder()
    .setName("world")
    .setNumber(42)
    .build();
HelloReply resp = blockingStub.sayHello(req);
System.out.println(resp.getGreeting());
channel.shutdown();
}
```

Build the request message



Example: Client ...

```
GreeterGrpc.GreeterBlockingStub blockingStub =
    GreeterGrpc.newBlockingStub(channel);
HelloRequest req = HelloRequest.newBuilder()
    .setName("world")
    .setNumber(42)
    .build();
HelloReply resp = blockingStub.sayHello(req);
System.out.println(resp.getGreeting());
channel.shutdown();
}
```

Do the
RPC



Example: Client ...

```
GreeterGrpc.GreeterBlockingStub blockingStub =
    GreeterGrpc.newBlockingStub(channel);
HelloRequest req = HelloRequest.newBuilder()
    .setName("world")
    .setNumber(42)
    .build();
HelloReply resp = blockingStub.sayHello(req);
System.out.println(resp.getGreeting());
channel.shutdown();
}
```

Read field
from message



Example: Client ...

```
GreeterGrpc.GreeterBlockingStub blockingStub =  
    GreeterGrpc.newBlockingStub(channel);  
HelloRequest req = HelloRequest.newBuilder()  
    .setName("world")  
    .setNumber(42)  
    .build();  
HelloReply resp = blockingStub.sayHello(req);  
System.out.println(resp.getGreeting());  
channel.shutdown();  
}
```

Shutdown the channel
(close the TCP connection)



Details on the Protocol Buffer Language

- ➔ The `protoc` compiler creates stubs, skeletons and interfaces for the implementation language
- ➔ Supported data types:
 - ➔ simple types: `int32`, `uint64`, `double`, `string`, `bool`, ...
 - ➔ enums
 - ➔ arrays, e.g. `repeated int32 val = 1;`
 - ➔ maps, e.g. `map<string, int32> size = 3;`
 - ➔ in addition, messages can be nested
- ➔ Fields can be optional, e.g. `optional int32 val = 1;`
 - ➔ if the field is not set, it gets a fixed default value
- ➔ RPCs may also receive and return *streams* of messages
- ➔ Explicit field numbers support forward and backward compatibility



Example: Replying a Stream of Messages

➔ Interface definition:

```
rpc SayHello (HelloRequest) returns (stream HelloReply);
```

➔ Server: sends several reply messages using `onNext()`

➔ Client: uses an asynchronous stub and a stream observer

```
GreeterGrpc.GreeterStub stub =  
    GreeterGrpc.newStub(channel);  
HelloRequest req = HelloRequest.newBuilder()  
    .setName("world").setNumber(42).build();  
stub.sayHello(req, new StreamObserver<HelloReply>() {  
    @Override public void onNext(HelloReply resp) {  
        System.out.println(resp.getGreeting());  
    }  
    ...  
});
```



Compatibility Issues in Client/Server Interfaces

- ➔ Client and server code evolves over time
 - ➔ message fields may be added or removed
 - ➔ the type of a message field may change
 - ➔ problem when client and server have different versions
- ➔ Safe changes (among others):
 - ➔ adding a field (old receiver code ignores it)
 - ➔ removing a field (old receiver code gets the default value)
 - ➔ `reserved` keyword prevents field number from being reused
- ➔ Unsafe changes (among others):
 - ➔ changing field number of existing field
 - ➔ reusing an old field number for a new field



3.3 Summary

➔ RMI

- ➔ allows transparent access to remote objects via proxy objects
 - ➔ proxy classes are generated automatically at runtime
- ➔ parameter passing semantics
 - ➔ *by value*, if parameter object can be serialized
 - ➔ *by reference*, if parameter object is an RMI object

➔ gRPC

- ➔ service oriented: procedures with request / reply messages
 - ➔ including support for streams
- ➔ stubs / skeletons generated by `protoc` compiler
- ➔ supports encryption and authentication



Distributed Systems

Winter Term 2025/26

4 Name Services



Content

- ➔ Basics
- ➔ Example: JNDI

Literature

- ➔ Tanenbaum, van Steen: Ch. 4.1
- ➔ Farley, Crawford, Flanagan: Ch. 7
- ➔ <http://docs.oracle.com/javase/tutorial/jndi/overview>

Names, Addresses and IDs

- ➔ **Name**: character or bit sequence that refers to a unit
 - ➔ unit: e.g. computer, printer, file, user, website, ...
- ➔ **Address**: name of the entry point of a unit
 - ➔ entry point allows access to the unit
 - ➔ several entry points per unit are possible
 - ➔ entry point may change over time
- ➔ A **position-independent name** identifies a unit independently from its entry point
- ➔ **ID**: name with the following properties:
 - ➔ ID refers to at most one unit, unit has at most one ID
 - ➔ ID always refers to the same unit (not reused)

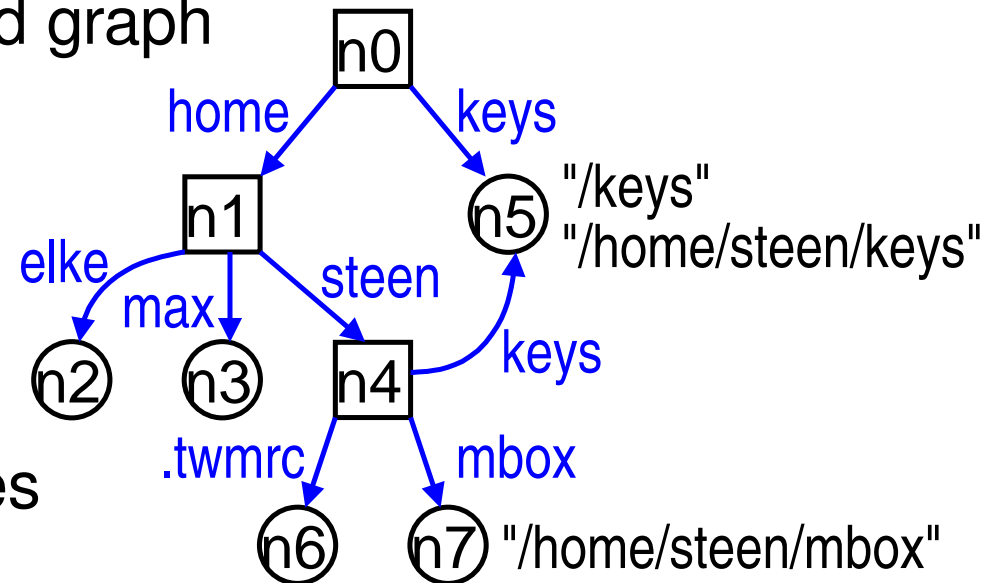
Namespaces

→ represented by directed, labelled graph

→ leaf node: named unit, with information / status if required

→ inner node: directory node

→ edges are labeled with names



→ Units are named by paths in the graph:

Start node: < Label-1, Label-2, ... >

→ absolute path: starting from root (of namespace)

→ relative path: starting from any node

→ Example: names in the UNIX file system



Aliasing and Linking

- ➔ **Alias**: alternative name for the same unit
- ➔ Possibilities for the realization of aliases:
 - ➔ allow several absolute pathnames for one unit
 - ➔ e.g. *hard link* in Unix
 - ➔ a (special) leaf node stores pathname of the unit
 - ➔ e.g. *symbolic link* in Unix
- ➔ Transparent linking of different namespaces:
 - ➔ a (special) directory node stores the ID of a directory node in another namespace
 - ➔ e.g. *mounted* file system in Unix



Name Resolution

- ➔ Finding the node (or information) that corresponds to a name
 - ➔ start at the start node
 - ➔ look up first label in directory table
 - ⇒ ID of the next node
 - ➔ etc., until the path is completely processed
- ➔ **Conclusion mechanism**: determination of the start node
 - ➔ usually implicit
- ➔ **Global names**: resolution independent of specific context
- ➔ **Local names**: resolution is context-dependent
 - ➔ e.g. pathname relative to working directory in Unix



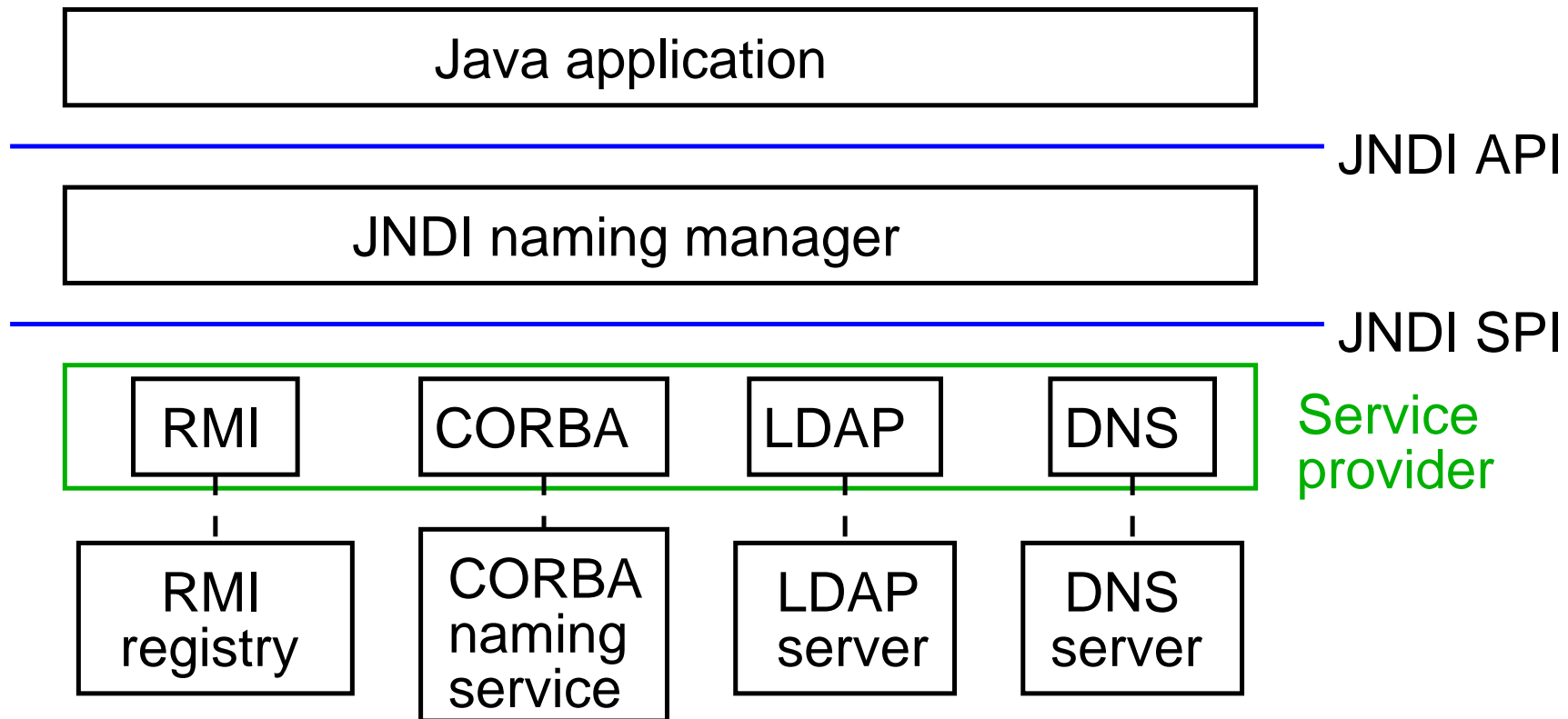
Implementation of Naming Services

- ➔ Typical operations:
 - ➔ bind(name, address, attributes)
 - ➔ lookup(name, attributes) \Rightarrow address, attributes
 - ➔ unbind(name, address)
- ➔ In distributed systems:
 - ➔ namespace is stored distributed (usually hierarchically)
 - ➔ for high availability: additionally replicated storage
- ➔ Name resolution can be iterative or recursive
 - ➔ iterative: Server responds with address of next server
 - ➔ recursive: server requests even at next server
- ➔ Example: *Domain Name Service* (👉 **RN_I, 9.3**)

4.2 Example: JNDI



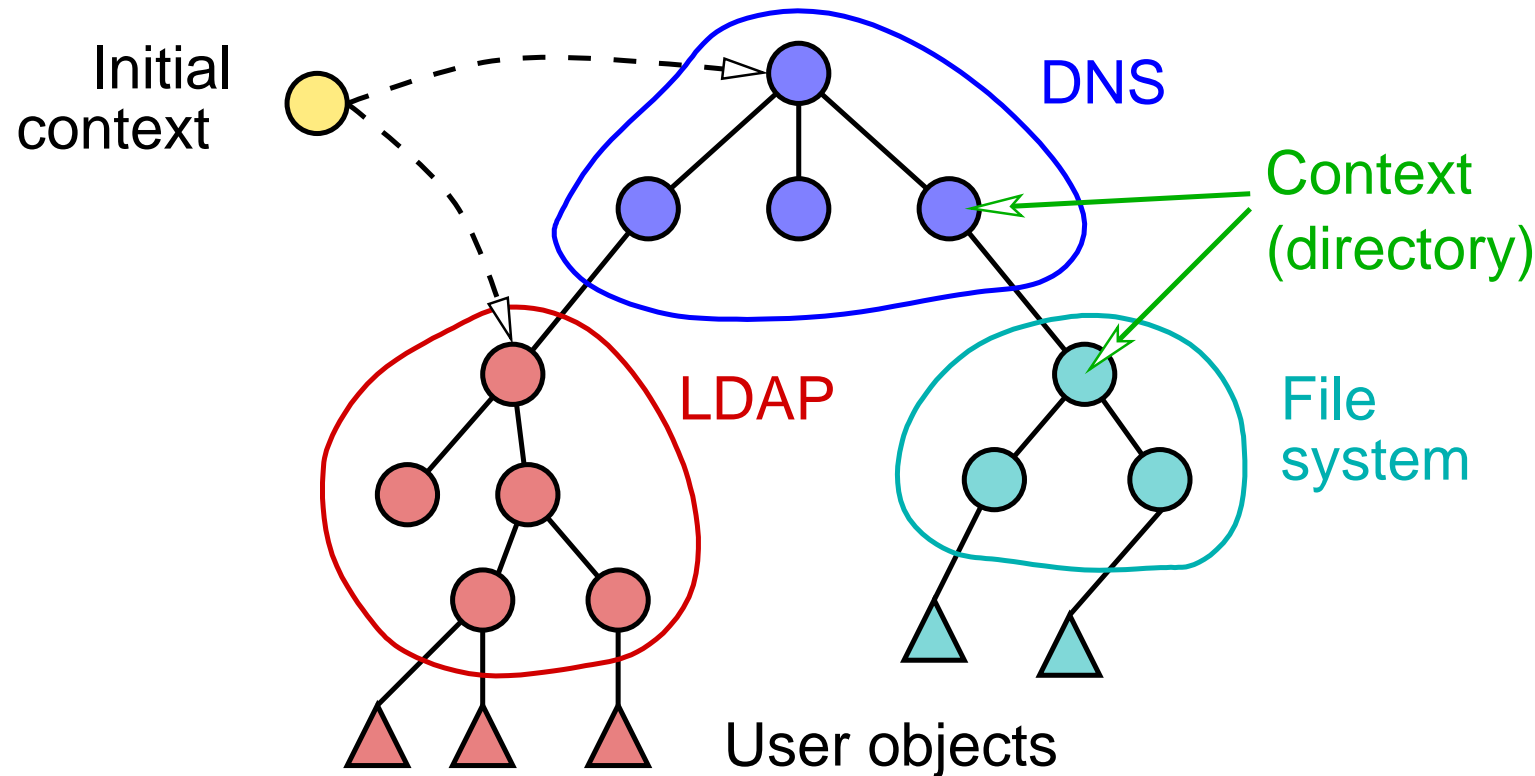
- ➔ JNDI: *Java Naming and Directory Interface*
- ➔ API for access to different name and directory services
 - ➔ directory service also stores attributes of objects



4.2 Example: JNDI ...



- ➔ JNDI supports compound namespaces
 - ➔ managed by various name or directory services



- ➔ Directories are called “contexts”
 - ➔ objects are bound to names within a context



The Interface `javax.naming.Context` for Naming Contexts

➔ Important methods:

- ➔ `bind()`, `rebind()` : bind objects to names
 - ➔ `bind()` throws exception if name already exists
- ➔ `unbind()` : remove names
- ➔ `rename()` : rename
- ➔ `lookup()` : resolve name to object
- ➔ `listBindings()` : list of all bindings
- ➔ `createSubcontext()` : create sub-context
- ➔ `destroySubcontext()` : delete sub-context



The Interface `javax.naming.Context` for Naming Contexts ...

- ➔ Implementation class `InitialContext`
 - ➔ for initial context (depending on the concrete name service)
 - ➔ `Context iC = new InitialContext(properties);`
 - ➔ configuration via `Properties` object (`Hashtable`), among others:
 - ➔ `"java.naming.factory.initial"`
 - ➔ factory for `InitialContext`
 - ➔ `"java.naming.provider.url"`
 - ➔ contact information for service provider
 - ➔ `"java.naming.security.principal"` and `"java.naming.security.credentials"`
 - ➔ user name and password for authentication



Example: Accessing the RMI Registry

```
import javax.naming.*;
```

```
...
```

```
Properties props = new Properties();  
props.put("java.naming.factory.initial",  
    "com.sun.jndi.rmi.registry.RegistryContextFactory");  
props.put("java.naming.provider.url",  
    "rmi://localhost:1099");  
Context ctx = new InitialContext(props);  
  
obj = (Hello)ctx.lookup("Hello-Server");  
  
message = obj.sayHello();
```



Example: Accessing a Local File System

```
import javax.naming.*;
```

```
...
```

```
Properties props = new Properties();  
props.put("java.naming.factory.initial",  
         "com.sun.jndi.fscontext.RefFSContextFactory");  
Context ctx = new InitialContext(props);
```

```
for (int i=0; i<args.length-1; i++)  
    ctx = (Context)ctx.lookup(args[i]);  
NamingEnumeration<Binding> list  
    = ctx.listBindings(args[args.length-1]);  
while (list.hasMore()) {  
    Binding b = list.next();  
    System.out.println(b.getName()+" : "+b.getClassName());  
}
```

Distributed Systems

Winter Term 2025/26

5 Process Management



Contents

- ➔ Distributed process scheduling
- ➔ Code migration

Literature

- ➔ Tanenbaum, van Steen: Ch. 3
- ➔ Stallings: Ch. 14.1

5.1 Distributed Process Scheduling

- ➔ Typical: middleware component that
 - ➔ decides on which node a process is executed
 - ➔ and probably migrates processes between nodes
- ➔ Goals:
 - ➔ balance the load between nodes
 - ➔ maximize the system performance (average response time)
 - ➔ also: minimize the communication between nodes
 - ➔ meet special hardware / resource requirements
- ➔ Load: typically the length of the process queue (ready queue)
 - ➔ sometimes resource consumption and communication volume are considered, too



Approaches to distributed scheduling

- ➔ Static scheduling
 - ➔ mapping of processes to nodes is defined before execution
 - ➔ NP-complete, therefore heuristic methods
- ➔ Dynamic load balancing, two variants:
 - ➔ execution location of a process is defined during creation and is not changed later
 - ➔ execution location of a process can be changed at runtime (several times, if necessary)
 - ➔ preemptive dynamic load balancing, **process migration**



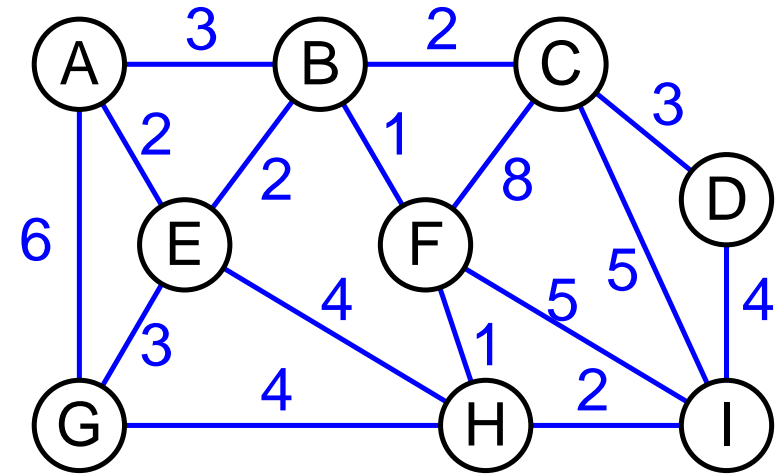
5.1.1 Static Scheduling

- ➔ Procedure dependent on the structure / the modelling of a job
 - ➔ jobs always consist of several processes
 - ➔ differences in communication structure
- ➔ Examples:
 - ➔ communicating processes: graph partitioning
 - ➔ non-communicating tasks with dependencies: list scheduling



Scheduling through graph partitioning

- ➔ Given: process system with
 - ➔ CPU / memory requirements
 - ➔ specification of communication load between each pair of processesusually represented as a graph



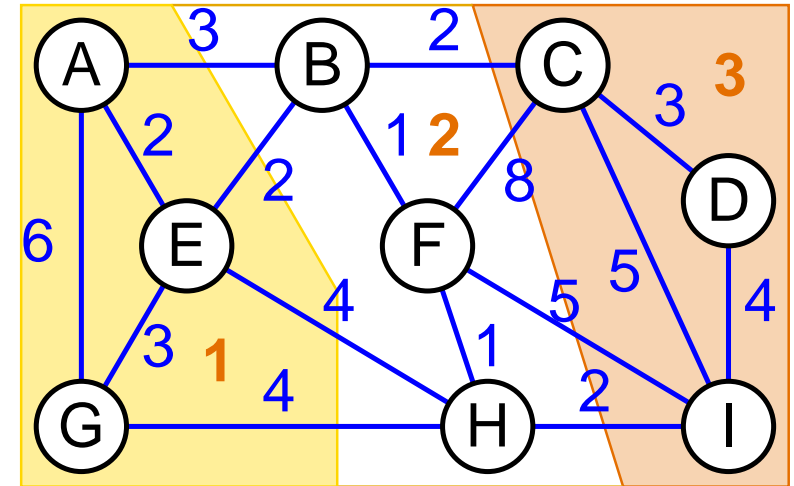
- ➔ Wanted: partitioning of the graph in such a way that
 - ➔ CPU and memory requirements are met for each node
 - ➔ partitions are about the same size (load balancing)
 - ➔ weighted sum of cut edges is minimal
 - ➔ i.e. as little communication as possible between nodes
- ➔ NP-complete, therefore many heuristic procedures



Scheduling through graph partitioning

$$\Sigma = 30$$

- ➔ Given: process system with
 - ➔ CPU / memory requirements
 - ➔ specification of communication load between each pair of processesusually represented as a graph



- ➔ Wanted: partitioning of the graph in such a way that
 - ➔ CPU and memory requirements are met for each node
 - ➔ partitions are about the same size (load balancing)
 - ➔ weighted sum of cut edges is minimal
 - ➔ i.e. as little communication as possible between nodes
- ➔ NP-complete, therefore many heuristic procedures

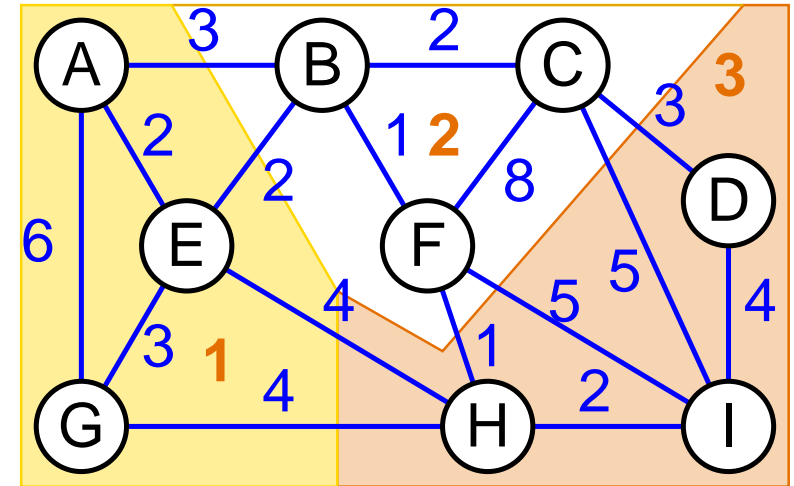
5.1.1 Static Scheduling ...



Scheduling through graph partitioning

$$\Sigma = 27$$

- ➔ Given: process system with
 - ➔ CPU / memory requirements
 - ➔ specification of communication load between each pair of processesusually represented as a graph



- ➔ Wanted: partitioning of the graph in such a way that
 - ➔ CPU and memory requirements are met for each node
 - ➔ partitions are about the same size (load balancing)
 - ➔ weighted sum of cut edges is minimal
 - ➔ i.e. as little communication as possible between nodes
- ➔ NP-complete, therefore many heuristic procedures



Distributed Systems

Winter Term 2025/26

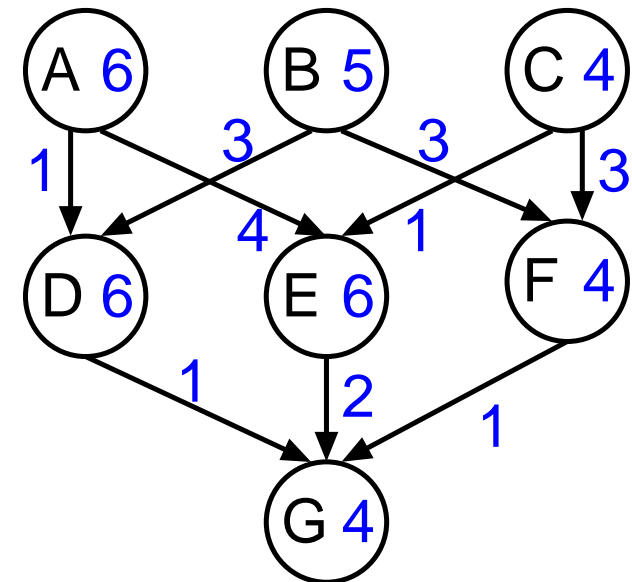
20.11.2025

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

Stand: January 8, 2026

List scheduling

- ➔ Tasks with dependencies, but without communication during execution
 - ➔ tasks work on results of other tasks
- ➔ Modelling
 - ➔ program represented as a DAG
 - ➔ nodes: tasks with execution times
 - ➔ edges: communication with transfer time (if sender and receiver are on different hosts)

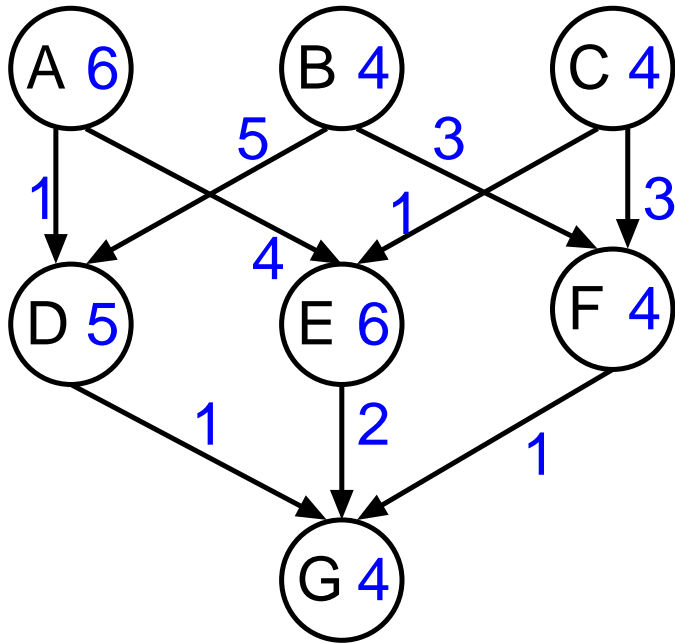




Method

- ➔ Create prioritized list of all tasks
 - ➔ many different heuristics to determine the priorities, e.g. according to:
 - ➔ length of the longest path (without communication) from the node to the end of the DAG (*High Level First with Estimated Time*, HLFET).
 - ➔ earliest possible start time (*Earliest Task First*, ETF)
- ➔ Process the list as follows:
 - ➔ assign the first task to the host that allows the earliest start time
 - ➔ remove the task from the list
- ➔ Creation and processing of the list can also be interleaved

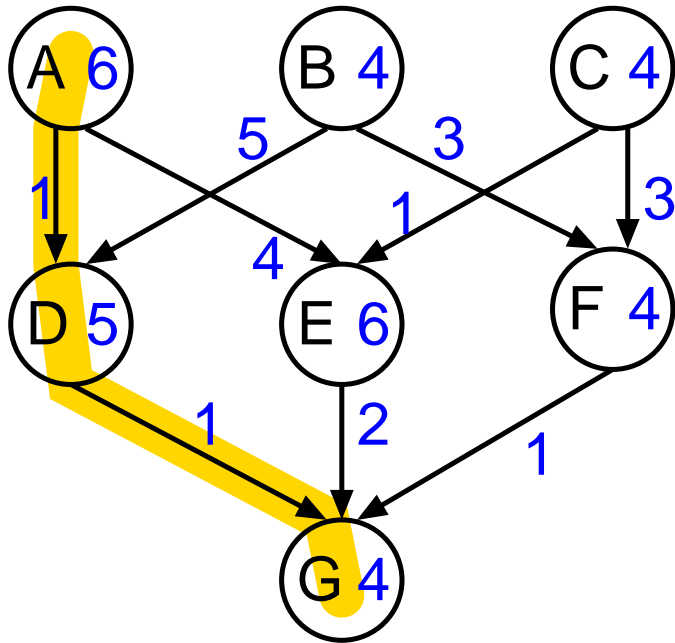
Example: List Scheduling with HLFET



5.1.1 Static Scheduling ...



Example: List Scheduling with HLFET



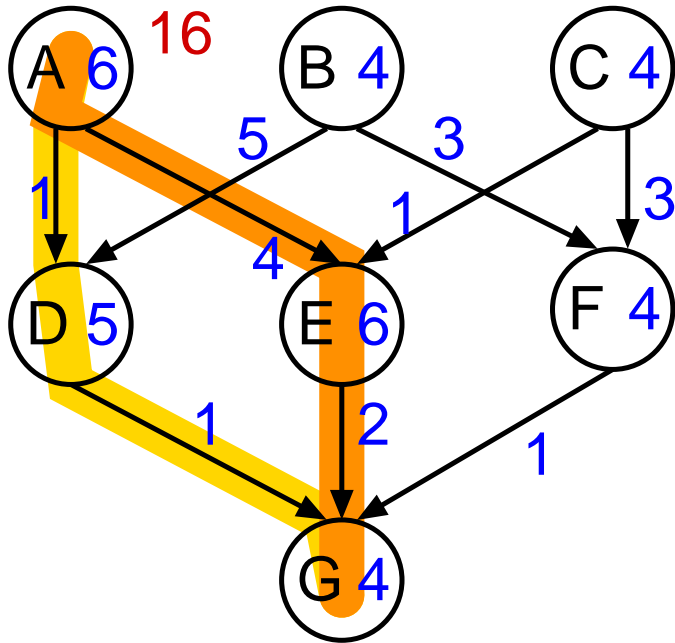
Static level (without comm.):

$$6+5+4 = 15$$

5.1.1 Static Scheduling ...



Example: List Scheduling with HLFET

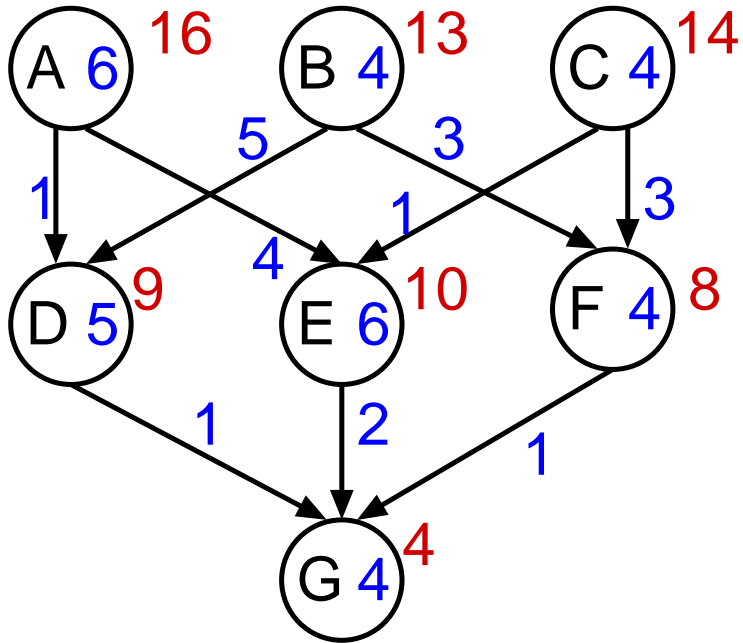


Static level (without comm.):

$$6+5+4 = 15 \quad 6+6+4 = 16$$



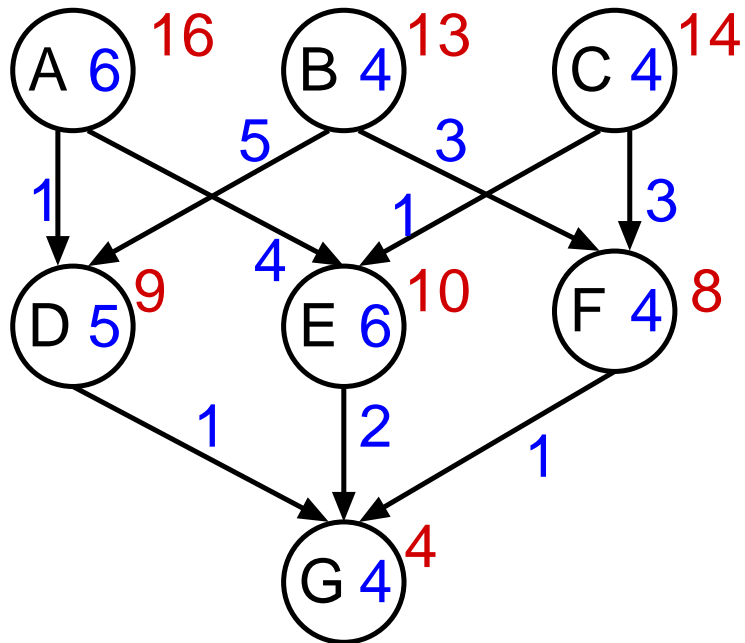
Example: List Scheduling with HLFET



5.1.1 Static Scheduling ...



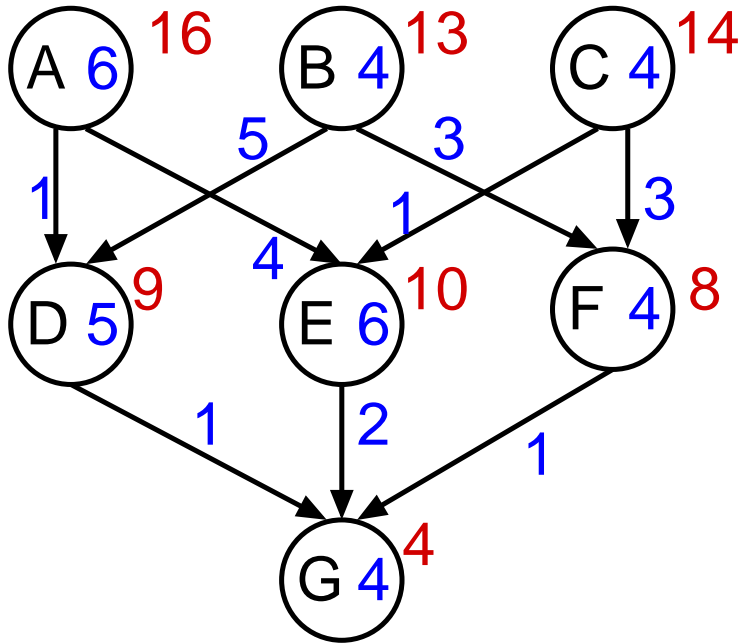
Example: List Scheduling with HLFET



5.1.1 Static Scheduling ...



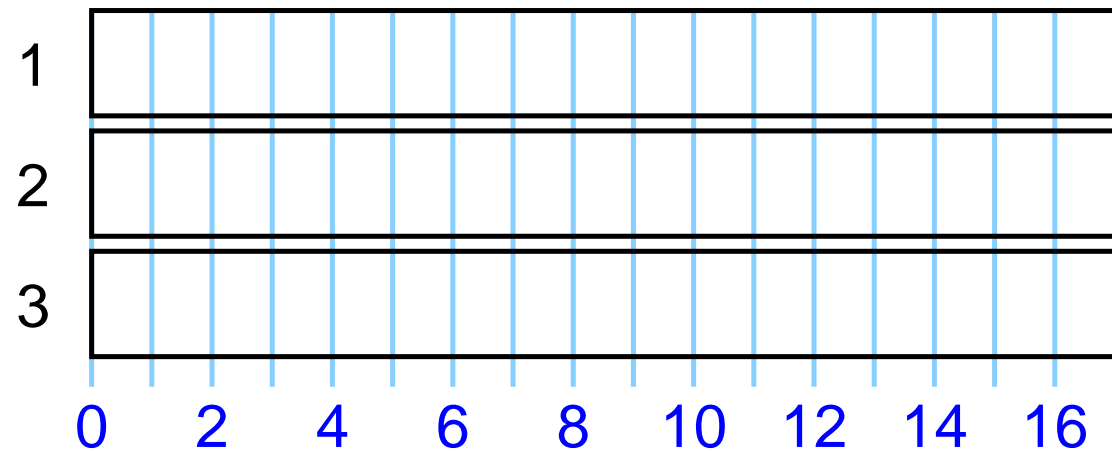
Example: List Scheduling with HLFET



List:

A	C	B	E	D	F	G
---	---	---	---	---	---	---

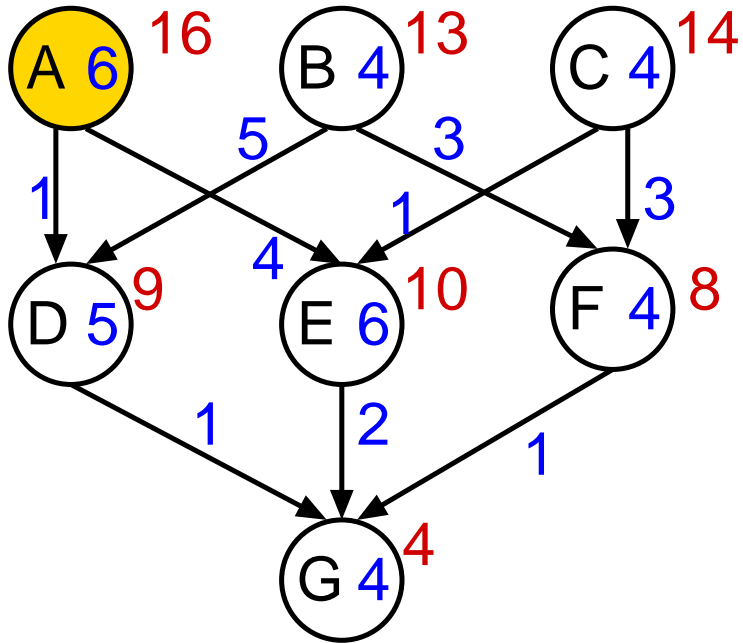
Schedule with 3 hosts:



5.1.1 Static Scheduling ...



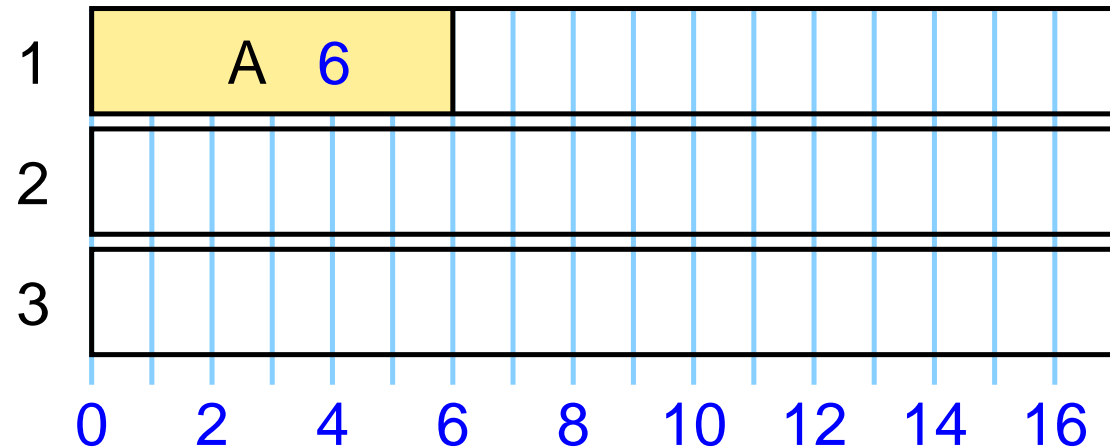
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

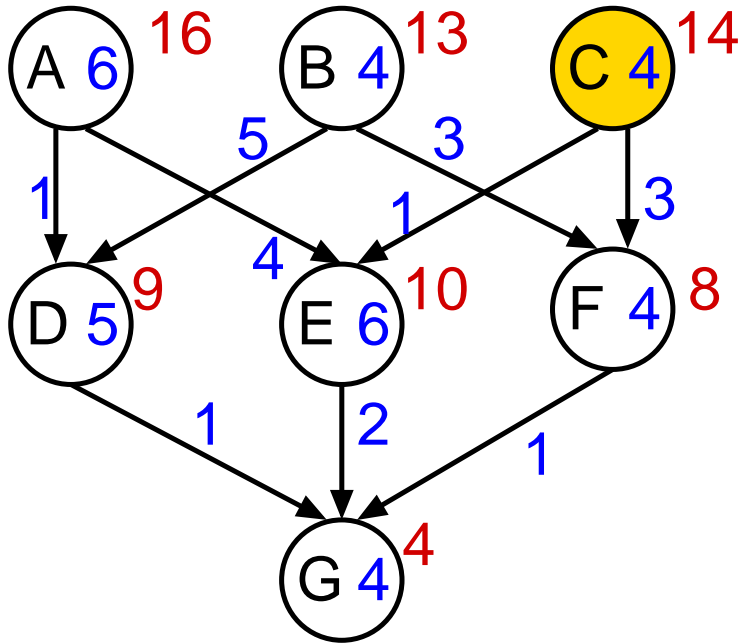


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



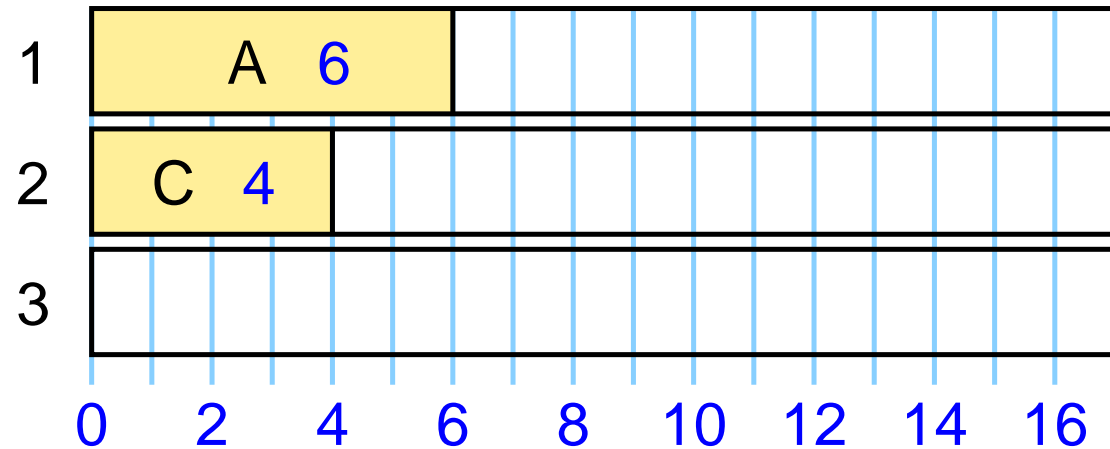
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

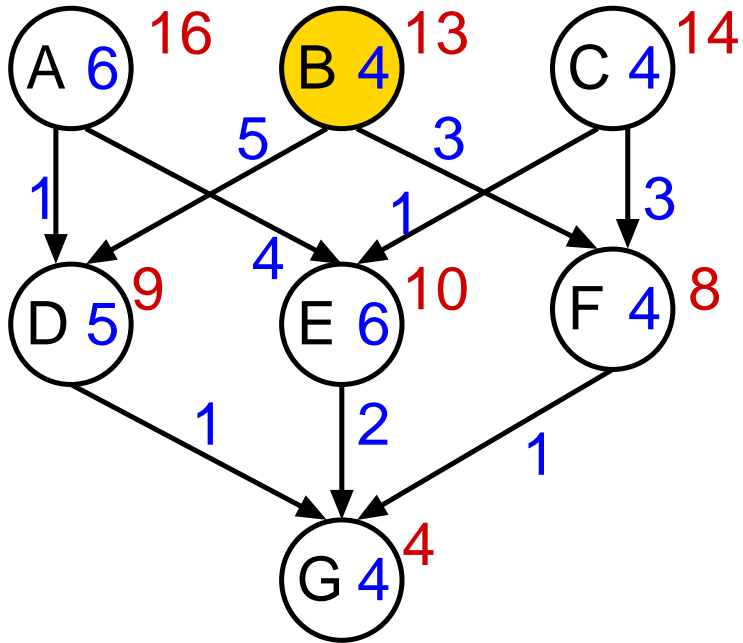


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



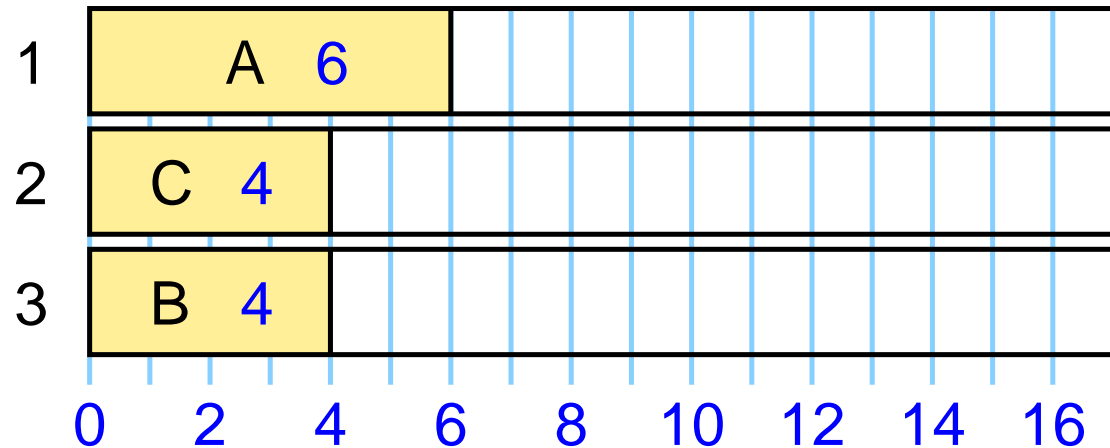
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

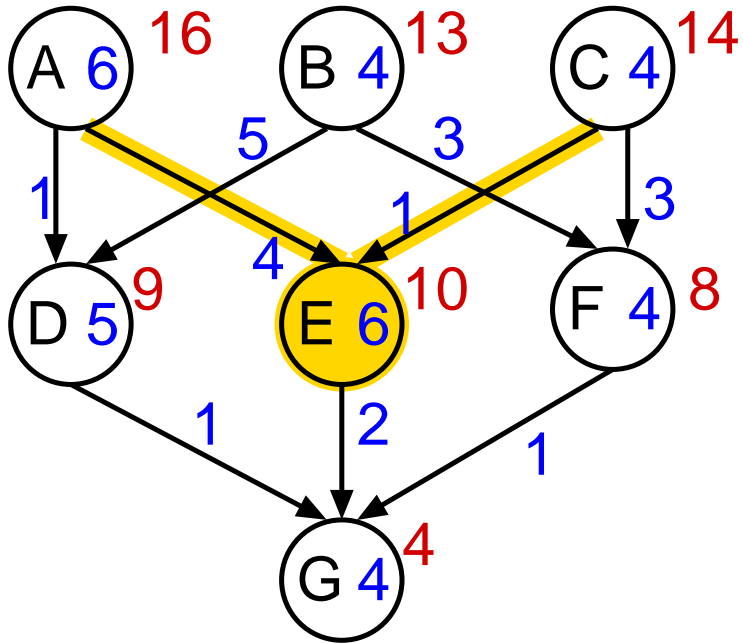


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



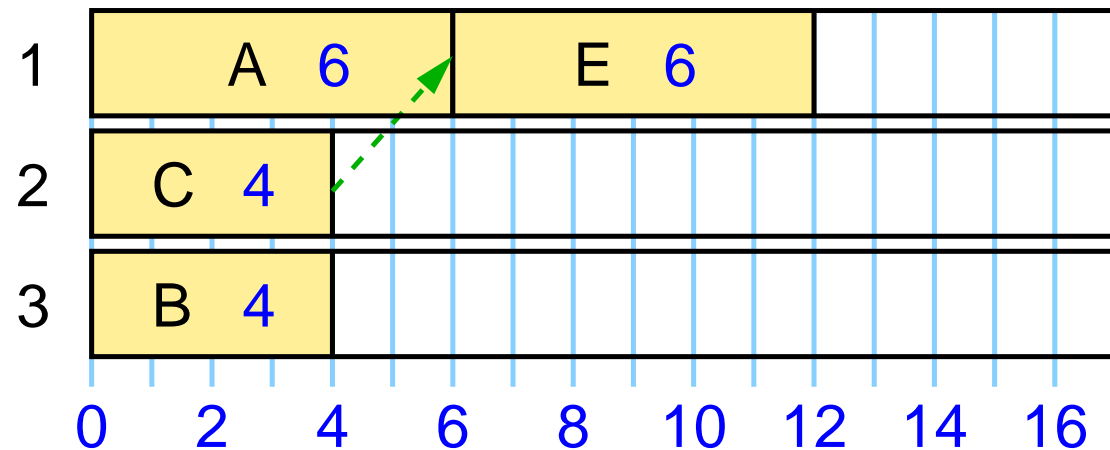
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

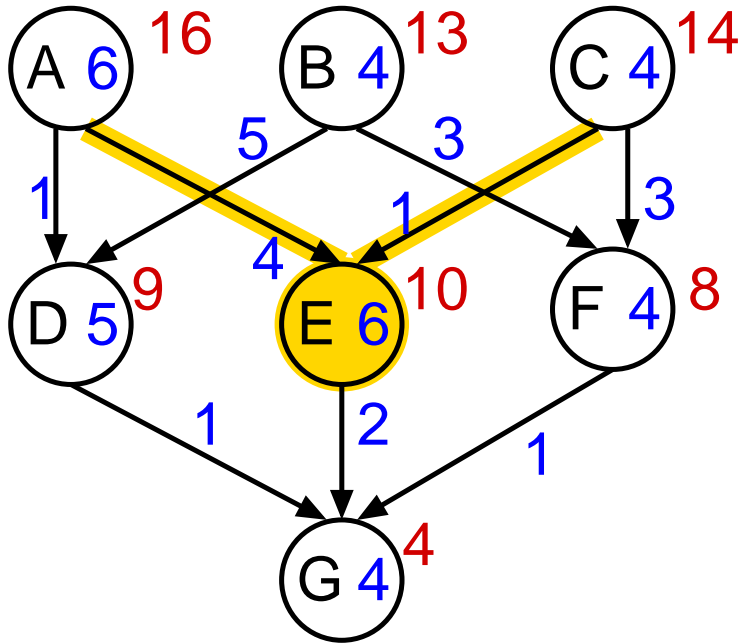


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



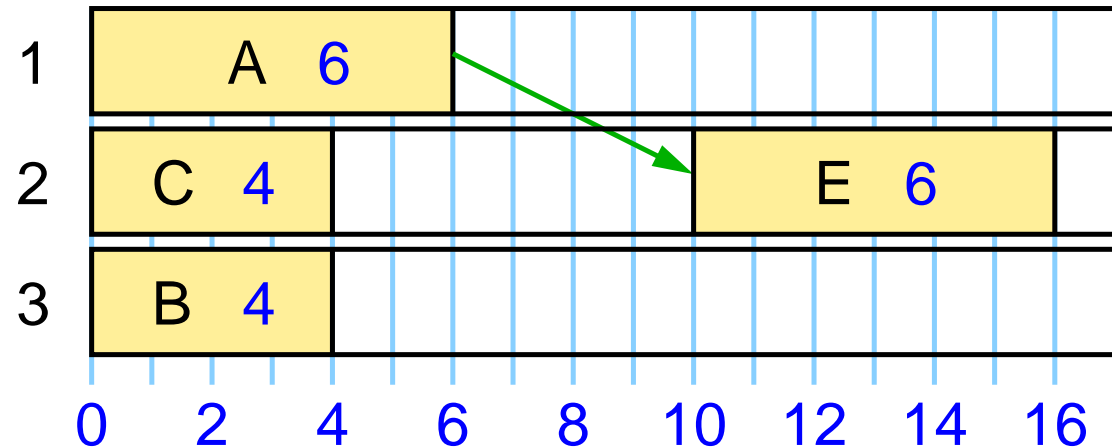
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

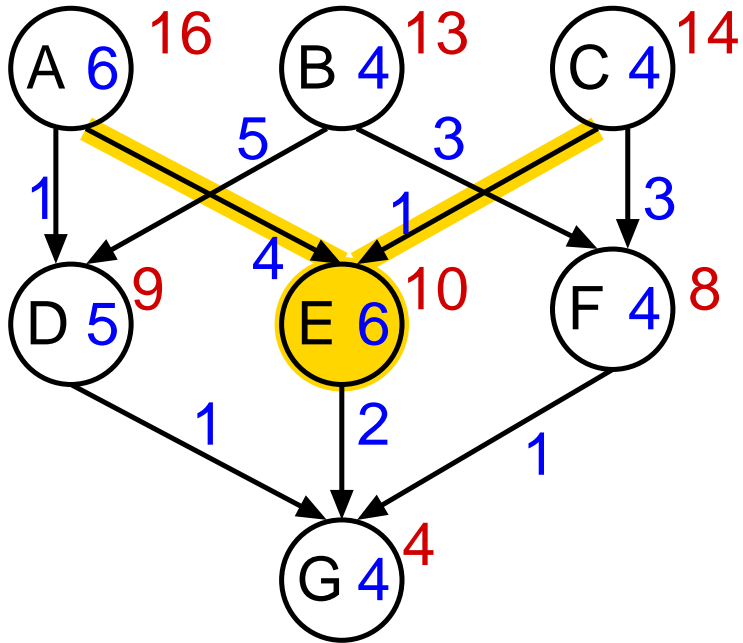


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



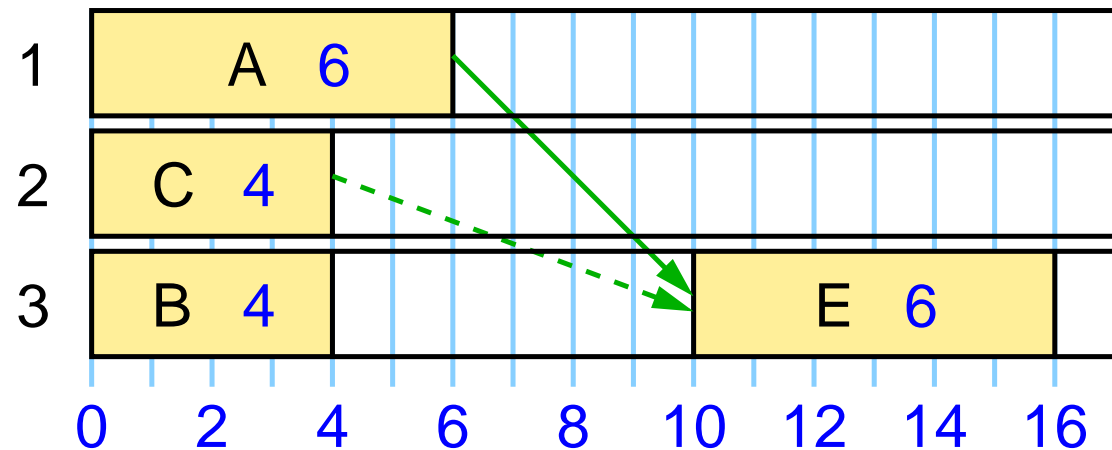
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

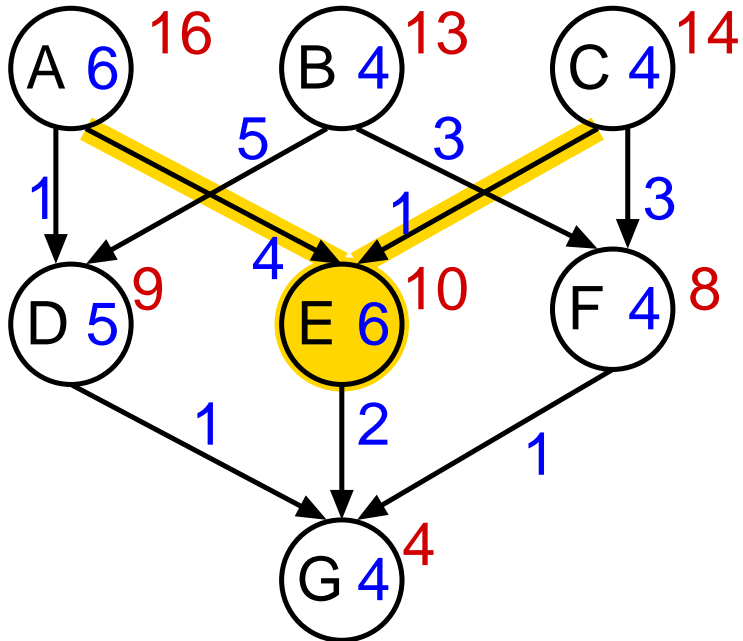


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



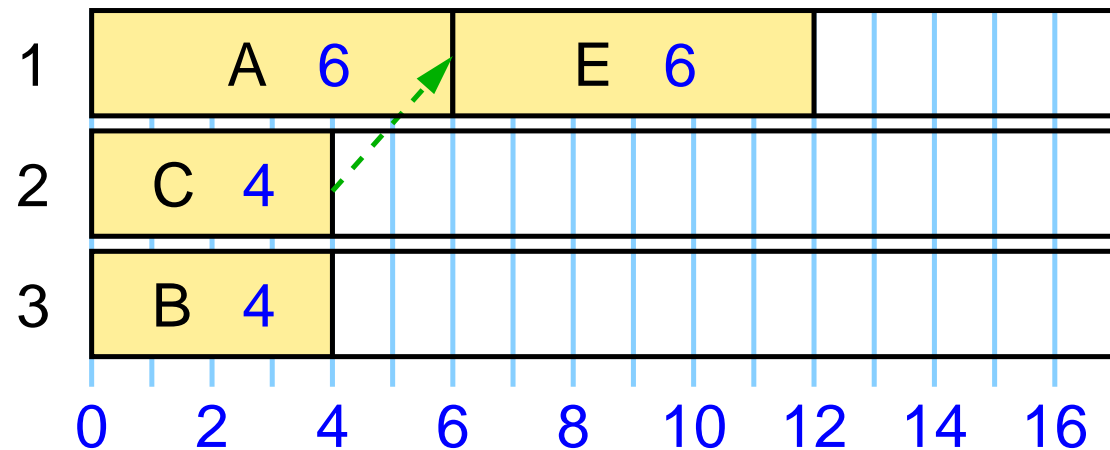
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

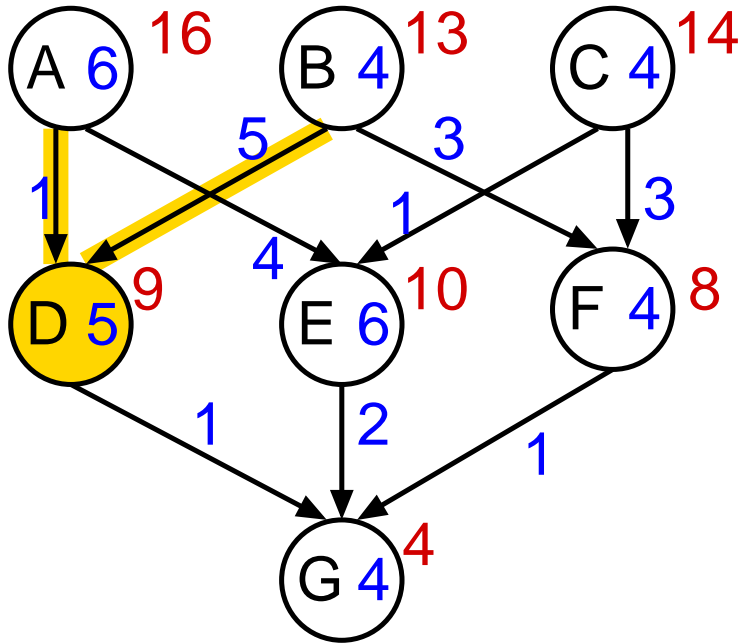


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



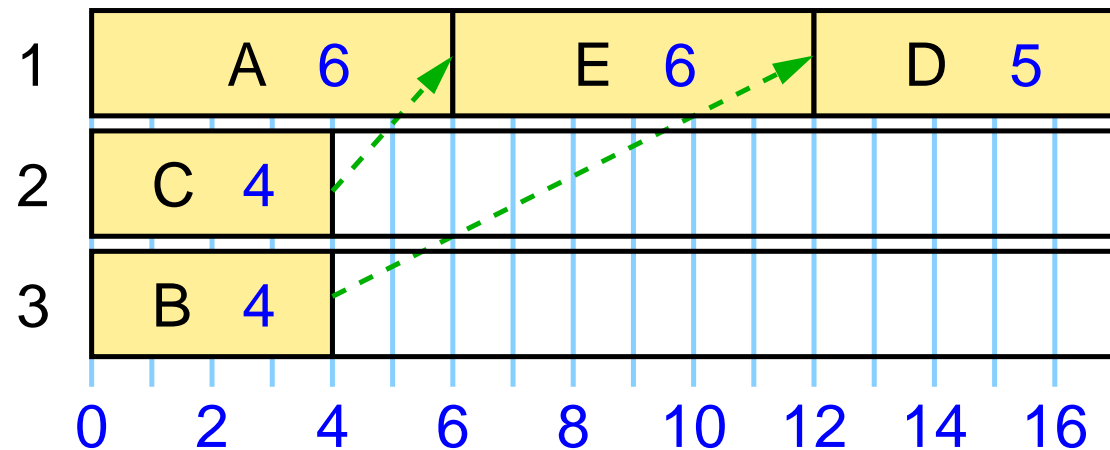
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

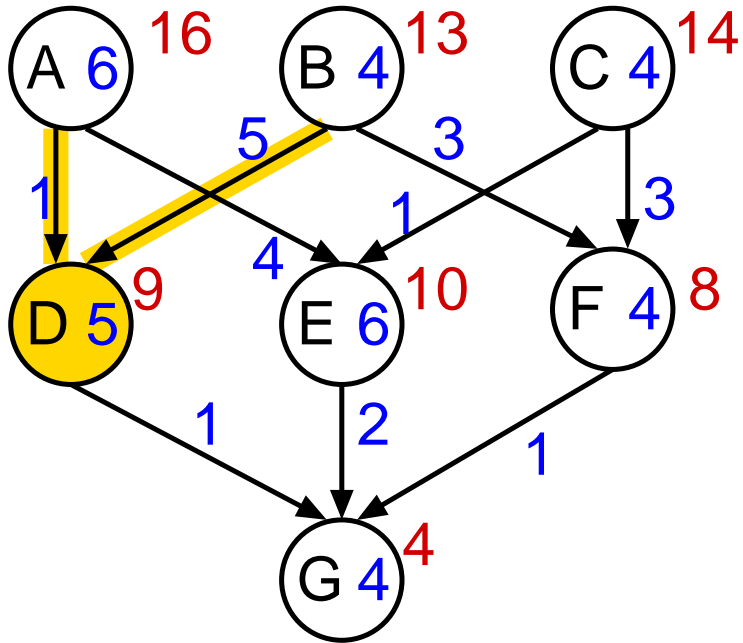


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



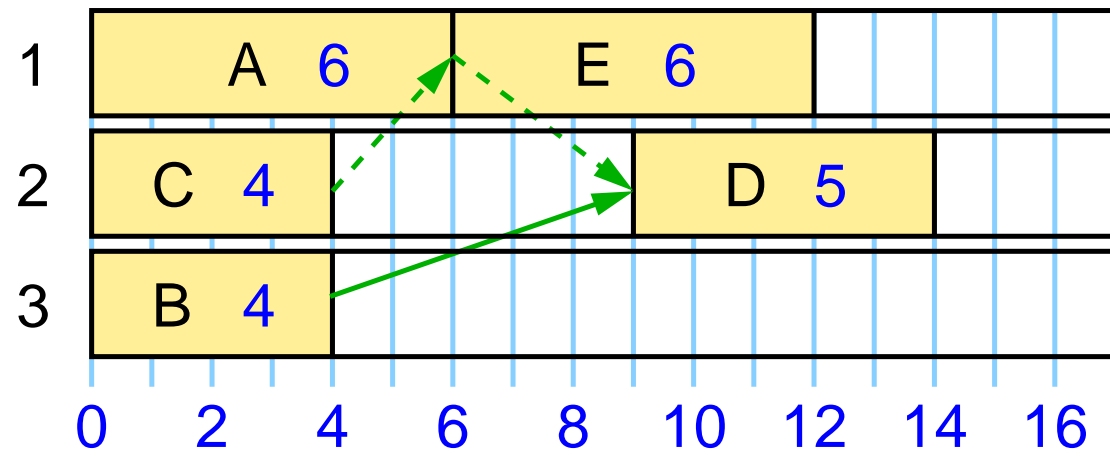
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

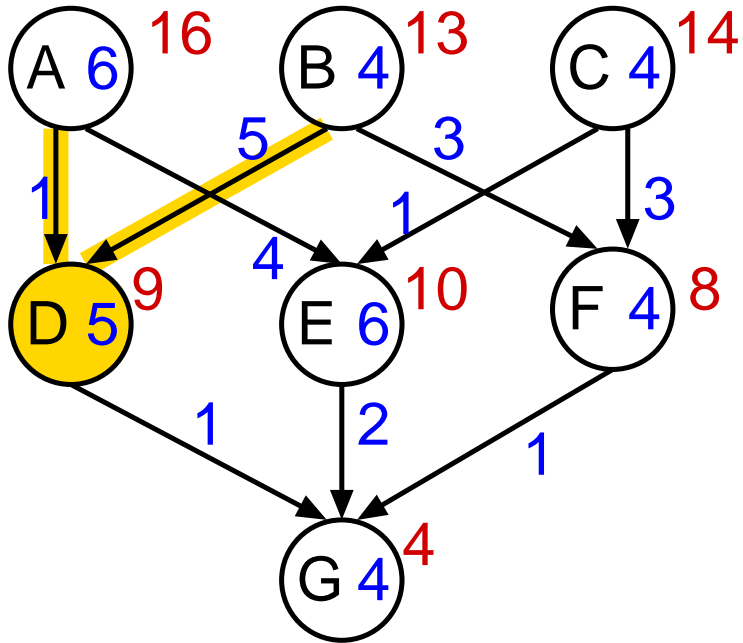


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



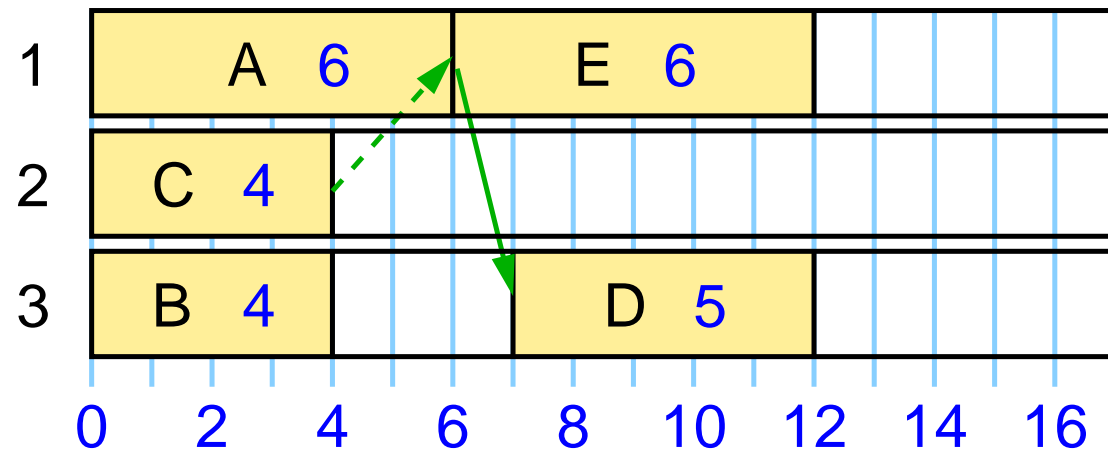
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

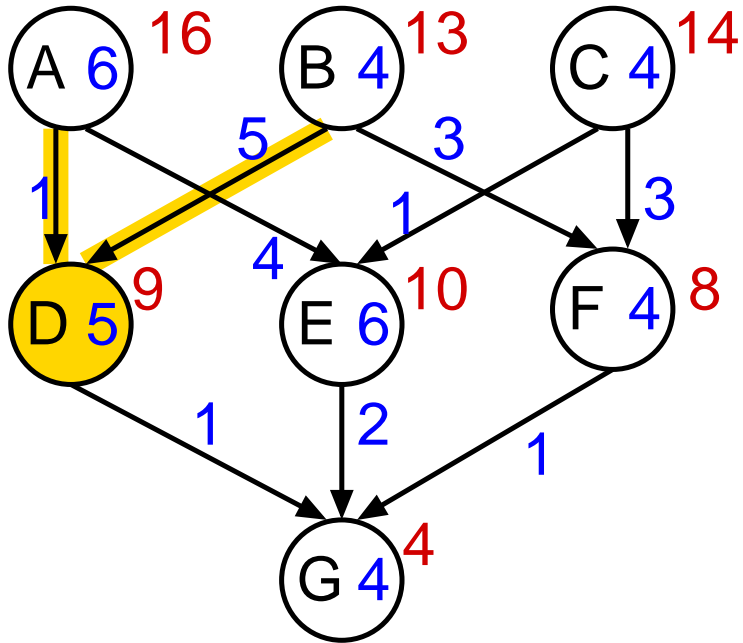


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



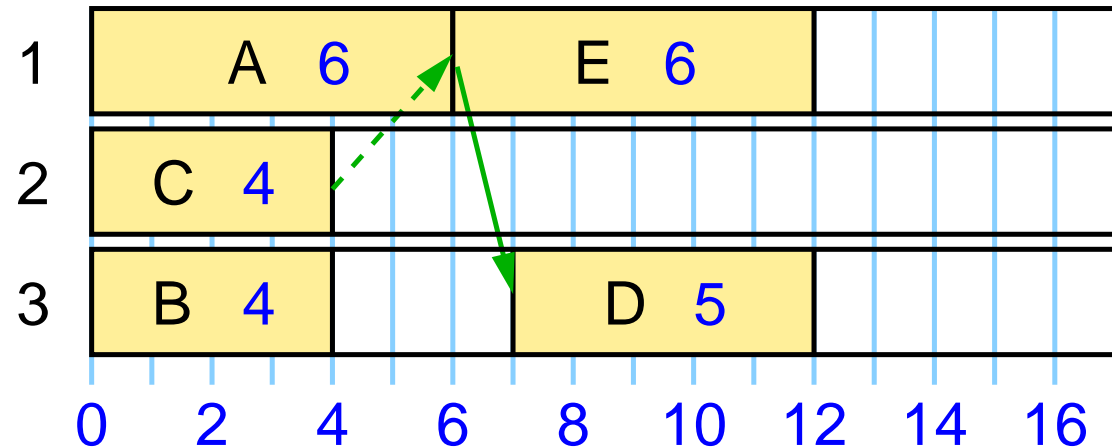
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

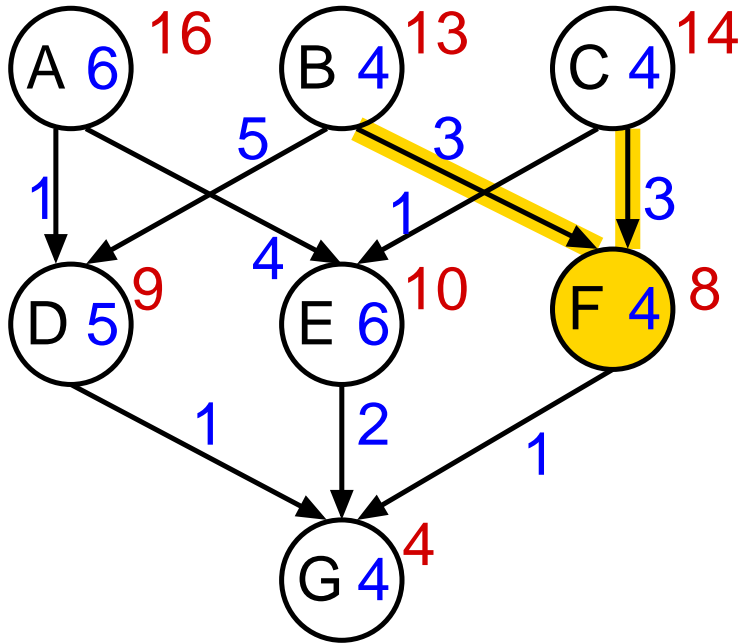


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...



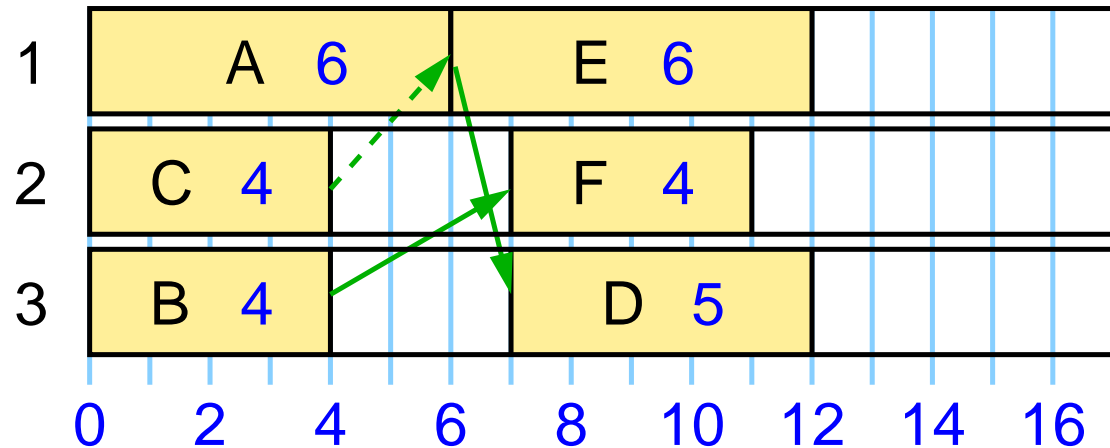
Example: List Scheduling with HLFET



List:



Schedule with 3 hosts:

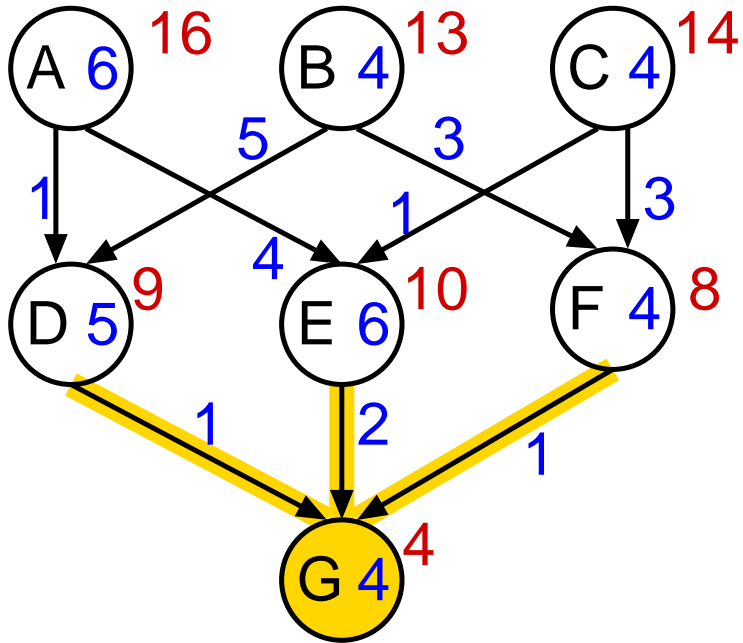


➔ Assumption: local communication does not cost any time

5.1.1 Static Scheduling ...

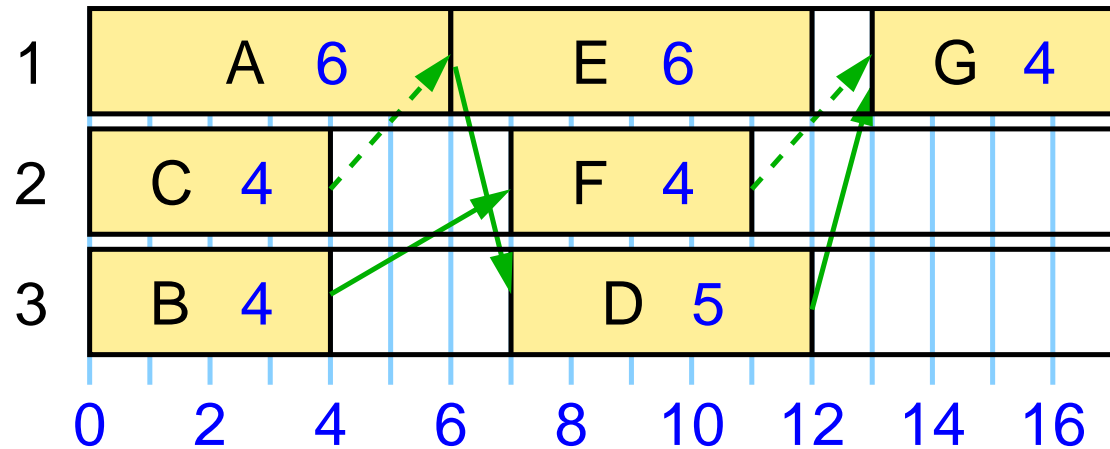


Example: List Scheduling with HLFET



List:

Schedule with 3 hosts:



➔ Assumption: local communication does not cost any time



5.1.2 Dynamic Load Balancing

- ➔ Components of a load balancing system
 - ➔ *Information policy* – when is load balancing triggered?
 - ➔ on demand, periodically, in case of state changes, ...
 - ➔ *Transfer policy* – under which condition is load shifted?
 - ➔ often: decision with the help of threshold values
 - ➔ *Location policy* – how is the receiver (or sender) found?
 - ➔ polling of some nodes, broadcast, ...
 - ➔ *Selection policy* – which tasks are moved?
 - ➔ new tasks, long tasks, location-independent tasks, ...



Typical approaches to dynamic load balancing

- ➔ Sender initiated load balancing
 - ➔ new process usually start on the local node
 - ➔ if node is overloaded: determine load of other nodes and start process on low-loaded node
 - ➔ e.g. ask randomly selected nodes for their load, send process if load \leq threshold, otherwise: next node
 - ➔ disadvantage: additional work for already overloaded node!
- ➔ Receiver initiated load balancing
 - ➔ when scheduling a process: check whether the node has still enough work (processes)
 - ➔ if not: ask other nodes for work
- ➔ Similar also for preemptive dynamic load balancing



5.2 Code Migration

- ➔ In distributed systems, in addition to data also programs are transferred between nodes
 - ➔ partly also during their execution
- ➔ Motivation: performance and flexibility
 - ➔ preemptive dynamic load balancing
 - ➔ optimization of communication (move code to data or highly interactive code to client)
 - ➔ increased availability (migration before system maintenance)
 - ➔ use of special HW or SW resources
 - ➔ use / evacuation of unused workstation computers
 - ➔ avoid code installation on client machines (dynamic loading of code from server)



Models for Code Migration

- ➔ Conceptual model: a process consists of three “segments”:
 - ➔ code segment
 - ➔ the executable program code of the process
 - ➔ execution segment
 - ➔ complete execution status of the process
 - ➔ virtual address space (data, heap, stack)
 - ➔ processor register (incl. instruction counter)
 - ➔ process / thread control block
 - ➔ resource segment
 - ➔ contains references to external resources required by the process
 - ➔ e.g. files, devices, other processes, mailboxes, ...



Models for Code Migration ...

➔ Weak mobility

- ➔ only the code segment is transferred
 - ➔ including initialization data if necessary
- ➔ program is always started from initial state
- ➔ examples: remotely loaded classes in Java, Java Script

➔ Strong mobility

- ➔ code and execution segment are transferred
- ➔ migration of a process in execution
- ➔ examples: process migration, agents

➔ Sender- or receiver-initiated migration



Code Migration Issues and Solutions

- ➔ Security: target computer executes unknown code
 - ➔ restricted environment (sandbox)
 - ➔ signed code
- ➔ Heterogeneity: code and execution segment depend on CPU and operating system
 - ➔ use of virtual machines (e.g. JVM, XEN)
 - ➔ migration points at which state can be stored and read in a portable way (possibly supported by compiler)
- ➔ Access to (local) resources
 - ➔ remote access with a global reference
 - ➔ move or copy the resource
 - ➔ new binding to resource of the same type



Process migration

- ➔ Migration of a process that is already running
 - ➔ triggered by OS or the process itself
 - ➔ mostly for dynamic load balancing
- ➔ Sometimes combined with *checkpoint/restart* function
 - ➔ instead of transferring the status of the process, it can also be stored persistently
- ➔ Design goals of migration procedures:
 - ➔ low communication effort
 - ➔ only short blocking of the migrated process
 - ➔ no dependency on source computer after migration



Process Flow of a Process Migration

- ➔ Creating a new process on the target system
- ➔ Transfer the code and execution segment (process address space, process control block), initialization of the target process
 - ➔ required: identical CPU and OS or virtual machine
- ➔ Update all connections to other processes
 - ➔ communication links, signals, ...
 - ➔ during migration: buffering at source
 - ➔ then: forwarding to target computer
- ➔ Delete the original process
 - ➔ if necessary, retain a “shadow process” for redirected system calls, e.g. file accesses



Transferring the process address space

- ➔ **Eager (all)**: transfer the entire address space
 - ➔ no traces of the process remain on source nodes
 - ➔ very expensive for large address space (especially if not all pages are used)
 - ➔ often together with checkpoint/restart function
- ➔ **Precopy**: process continues to run on source node during transfer
 - ➔ to minimize time in which the process is blocked
 - ➔ pages modified while the migration is in progress must be sent again



Transferring the process address space ...

- ➔ **Eager (dirty)**: transfer only modified pages that are in main memory
 - ➔ all other pages are only transferred when accessed
 - ➔ integration with virtual memory management
 - ➔ motivation: quickly “flush” main memory of the source node
 - ➔ source node may remain involved until the end of the process
- ➔ **Copy-on-reference**: transfer each page only when accessed
 - ➔ variation of *eager (dirty)*
 - ➔ lowest initial costs
- ➔ **Flushing**: move all pages to disk before migration
 - ➔ after that: *copy-on-reference*
 - ➔ advantage: main memory of the source node is relieved



Distributed Systems

Winter Term 2025/26

6 Time and Global State



- ➔ Synchronization of physical clocks
- ➔ Lamport's happened before relation
- ➔ Logical clocks
- ➔ Global state

Literature

- ➔ Tanenbaum, van Steen: Kap. 5.1-5.3
- ➔ Colouris, Dollimore, Kindberg: Kap. 10
- ➔ Stallings: Kap 14.2



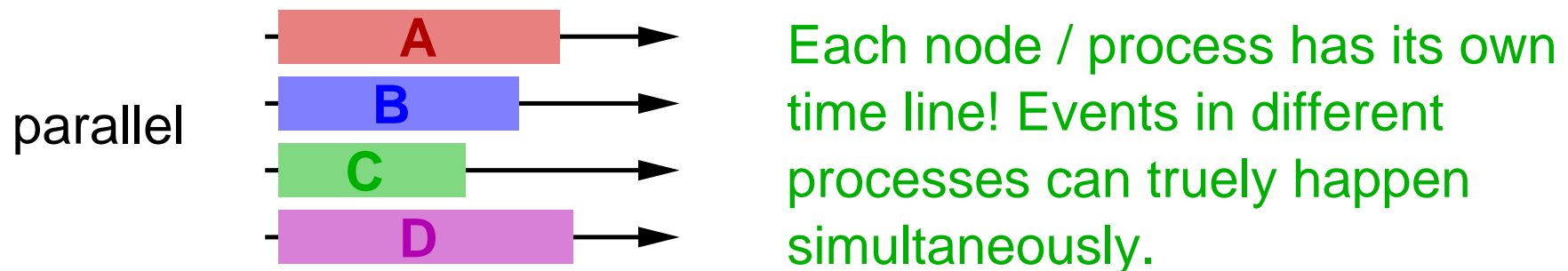
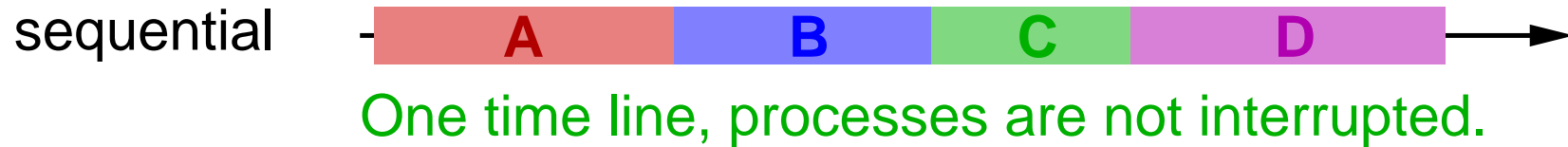
What is the difference between a distributed system and a single/multiprocessor system?

- ➔ Single or multiprocessor system:
 - ➔ concurrent processes: pseudo-parallel by time sharing or truly parallel
 - ➔ global time: all events in the processes can be ordered unambiguously in terms of time
 - ➔ global state: at any time a unique state of the system can be determined
- ➔ Distributed system
 - ➔ true parallelism
 - ➔ no global time
 - ➔ no unique global state



Concurrency vs. (true) parallelism

Example: 4 processes





Global Time

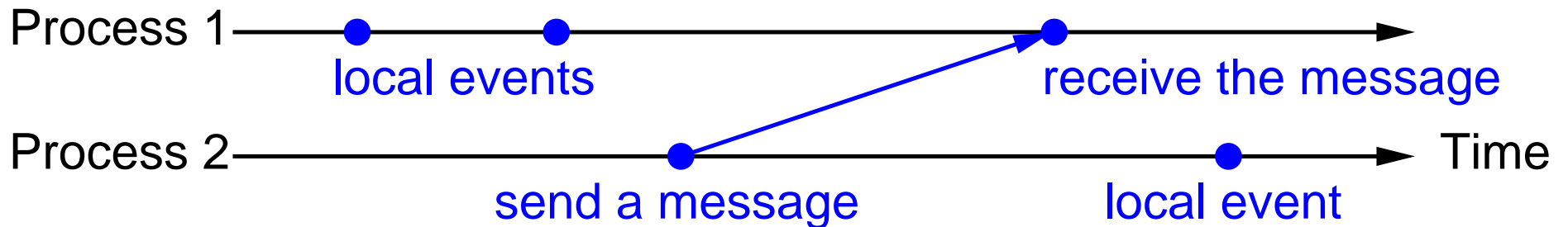
- ➔ In a single/multiprocessor system
 - ➔ each event can (at least theoretically) be assigned a unique time stamp of the same local clock
 - ➔ for multiprocessor systems: synchronization at the shared memory

- ➔ In distributed systems:
 - ➔ many local clocks (one per node)
 - ➔ exact synchronization of clocks is (on principle!) not possible
 - ➔ \Rightarrow the sequence of events on different nodes can not (always) be determined uniquely
 - ➔ (cf. special theory of relativity)

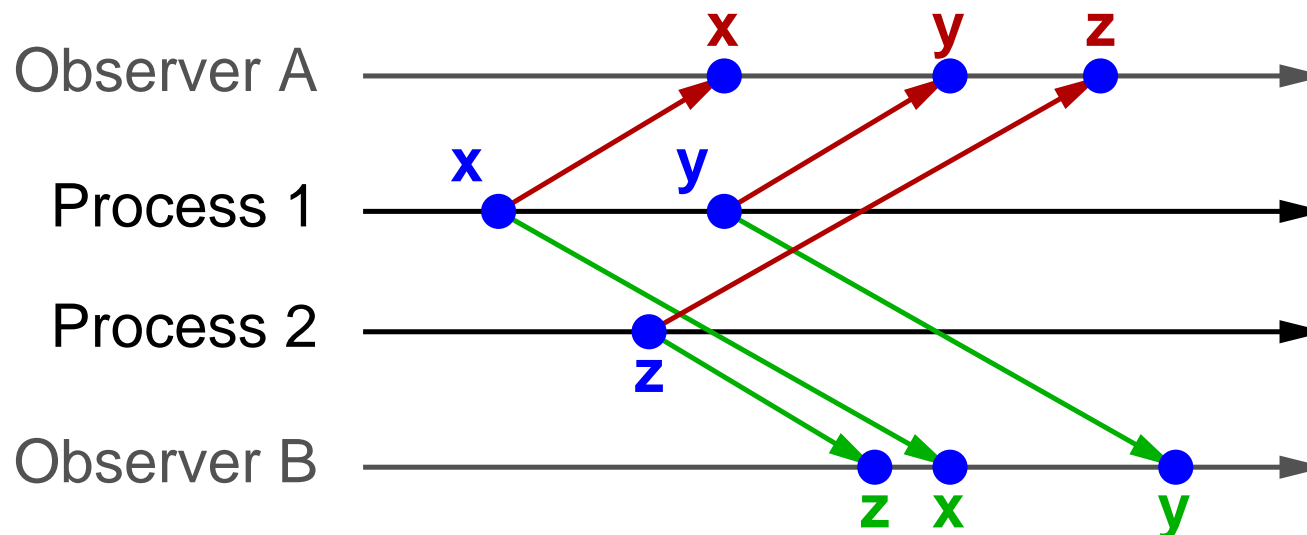


An effect of distribution

➔ Visualization of events in distributed systems: **Lamport diagram**



➔ Scenario: two processes observe two other processes





An effect of distribution ...

- ➔ The observers may see the events in different order!
- ➔ Problem e.g., if the observers are replicated databases and the events are database updates
 - ➔ replicas are no longer consistent!
- ➔ Even from time stamps of (local) clocks it is not possible to determine the order of events in a meaningful way
- ➔ Hence, in such cases:
 - ➔ events with timestamps of **logical clocks** (👉 6.3)
 - ➔ logical clocks allow conclusions to be made about causal order



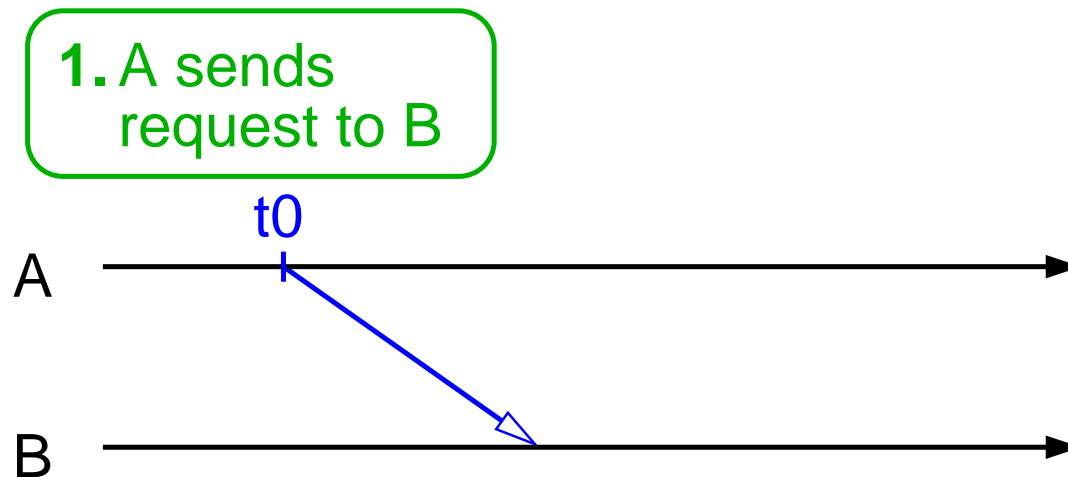
6.1 Synchronizing Physical Clocks

- ➔ Physical clock shows 'real' time
 - ➔ based on UTC (Universal Time Coordinated)
- ➔ Each computer has its own (physical) clock
 - ➔ quartz oscillator with counter in HW and if necessary in SW
- ➔ Clocks usually differ from each other (**offset**)
 - ➔ Offset changes over time: **clock drift**
 - ➔ typ. 10^{-6} for quartz crystals, 10^{-13} for atomic clocks
- ➔ Goal of clock synchronization:
 - ➔ keep the offset of the clocks under a given limit
 - ➔ **clock skew**: maximum allowed deviation



Cristian's Method

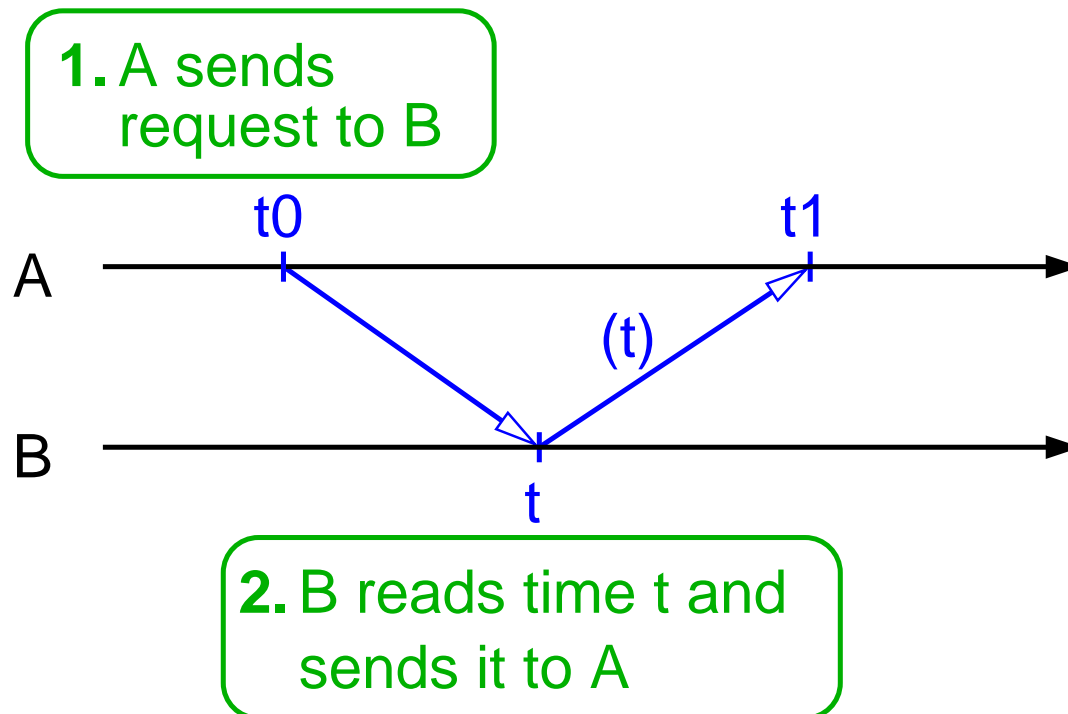
- ➔ Assumption: A and B want to synchronize their clocks with each other
 - ➔ B can also be a time server (e.g. with GPS clock)
- ➔ Protocol:





Cristian's Method

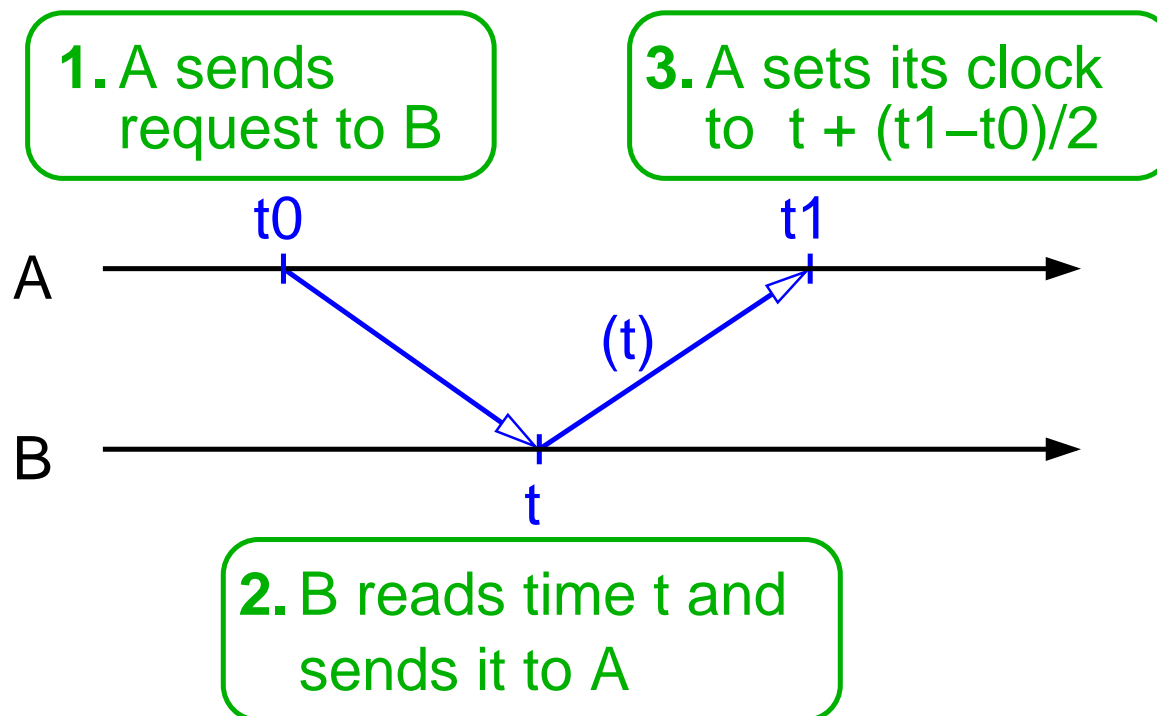
- ➔ Assumption: A and B want to synchronize their clocks with each other
 - ➔ B can also be a time server (e.g. with GPS clock)
- ➔ Protocol:





Cristian's Method

- ➔ Assumption: *A* and *B* want to synchronize their clocks with each other
 - ➔ *B* can also be a time server (e.g. with GPS clock)
- ➔ Protocol:

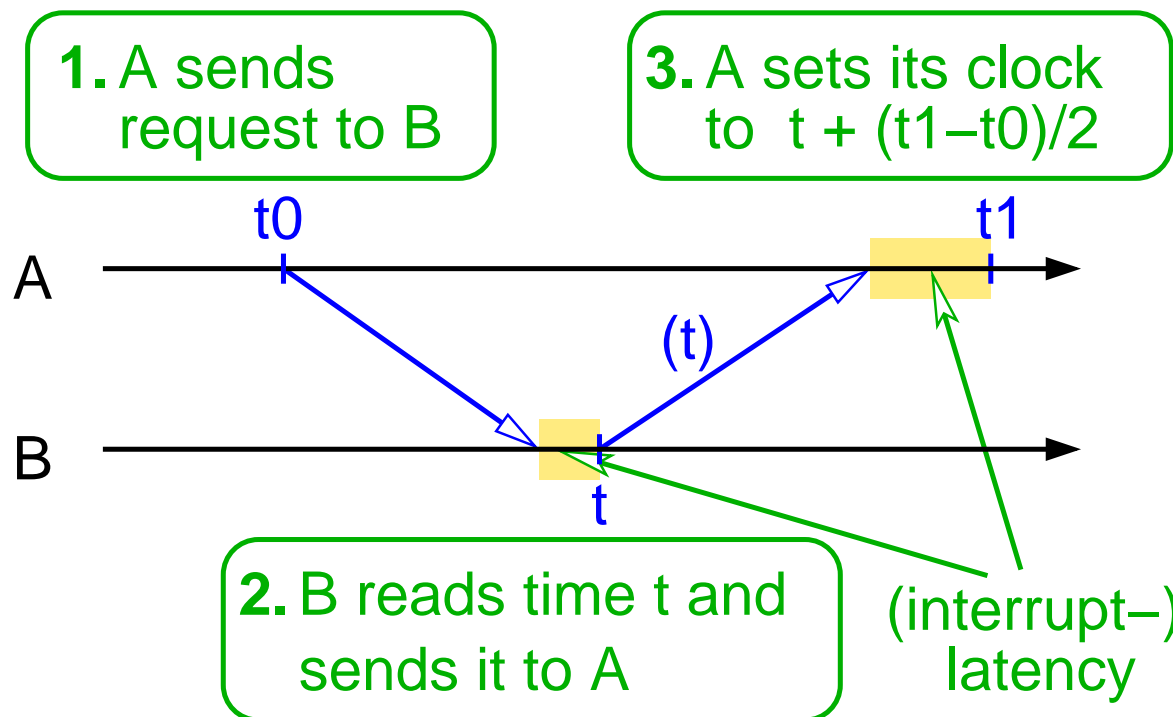


- ➔ *A* must take the runtime of the reply message into account
- ➔ estimate: runtime = half the round trip time = $(t_1 - t_0)/2$



Cristian's Method

- ➔ Assumption: *A* and *B* want to synchronize their clocks with each other
 - ➔ *B* can also be a time server (e.g. with GPS clock)
- ➔ Protocol:

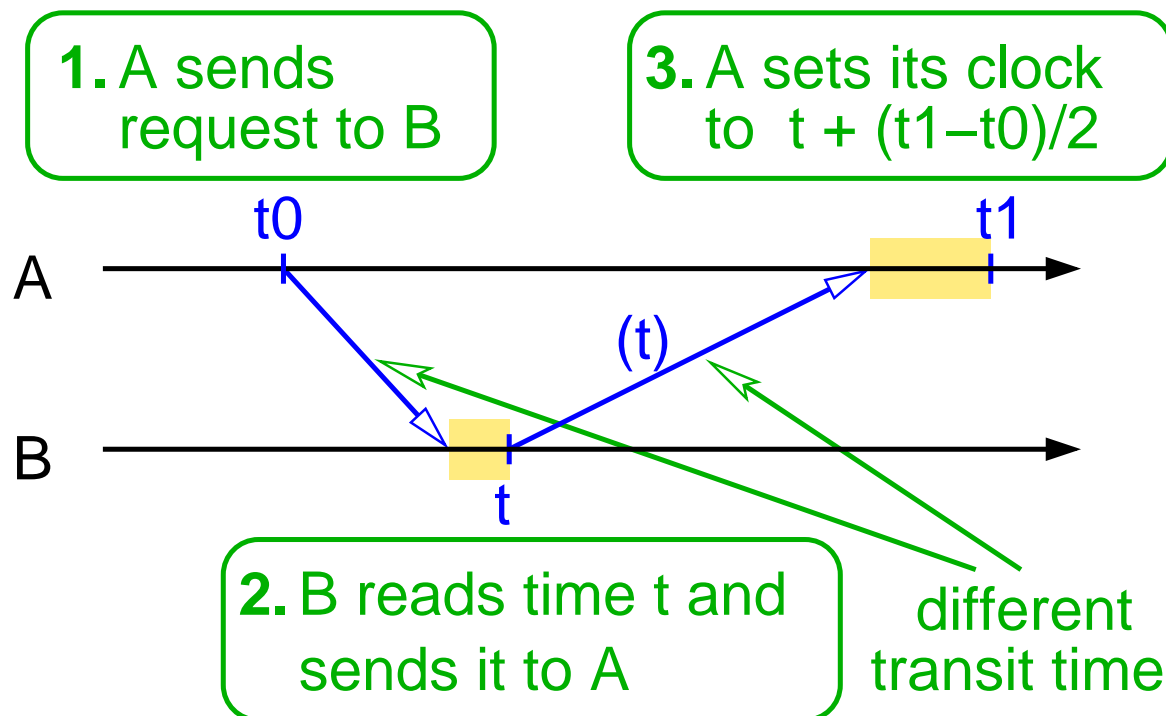


- ➔ *A* must take the runtime of the reply message into account
- ➔ estimate: runtime = half the round trip time = $(t_1 - t_0)/2$



Cristian's Method

- ➔ Assumption: *A* and *B* want to synchronize their clocks with each other
 - ➔ *B* can also be a time server (e.g. with GPS clock)
- ➔ Protocol:



- ➔ *A* must take the runtime of the reply message into account
- ➔ estimate: runtime = half the round trip time = $(t_1 - t_0)/2$



Cristian's Method: Discussion

- ➔ Problem: runtimes of both messages may be different
 - ➔ systematic differences (different paths / latencies)
 - ➔ statistical fluctuations of the transit time
- ➔ Accuracy estimate, if minimum transit time (min) is known:
 - ➔ B can have determined t at the earliest at time $t_0 + min$, at the latest at time $t_1 - min$ (measured with A 's clock)
 - ➔ thus accuracy $\pm ((t_1 - t_0)/2 - min)$
- ➔ To improve accuracy:
 - ➔ execute the message exchange multiple times
 - ➔ use the one with minimum round trip time



Adjusting the clock

- ➔ Turning back is problematic
 - ➔ order / uniqueness of time stamps
- ➔ Non-monotonous “jumping” of the time also problematic
- ➔ Therefore: clock is generally adjusted slowly
 - ➔ runs faster / slower, until clock skew has been compensated

Further protocols

- ➔ Berkeley algorithm: server calculates mean value of all clocks
- ➔ NTP (Network Time Protocol): hierarchy of time servers in the Internet with periodic synchronization
- ➔ IEEE 1588: clock synchronization for automation systems

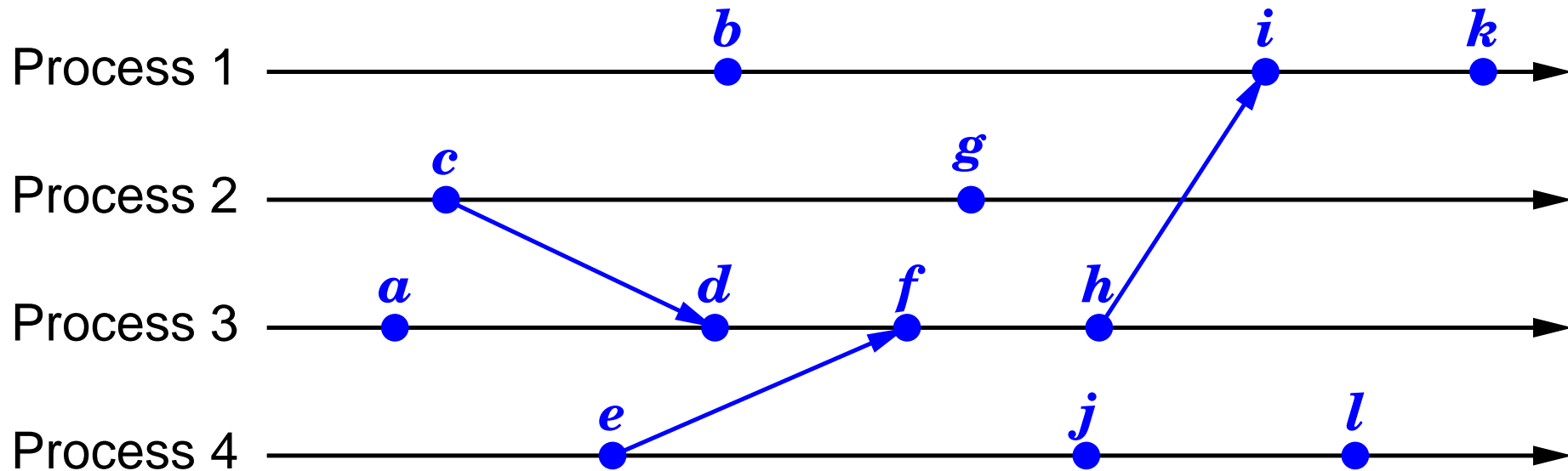


6.2 Lamport's Happened-Before Relation

- ➔ In two cases, the order of events can also be determined without a global clock:
 - ➔ if the events are in the same process, local clock is sufficient
 - ➔ the sending of a message is always before its reception
- ➔ Definition of the happened-before causality relation \rightarrow (*causality relation*)
 - ➔ if events a, b are in the same process i and $t_i(a) < t_i(b)$ (t_i : time stamp with i 's clock), then $a \rightarrow b$
 - ➔ if a is the sending of a message and b its receipt, then $a \rightarrow b$
 - ➔ if $a \rightarrow b$ and $b \rightarrow c$, then also $a \rightarrow c$ (transitivity)
- ➔ $a \rightarrow b$ means, that b **may** causally depend on a



Examples



➡ Among others, we have here:

➡ $b \rightarrow i$ and $a \rightarrow h$ (events in the same process)

➡ $c \rightarrow d$ and $e \rightarrow f$ (sending / receiving a message)

➡ $c \rightarrow k$ and $a \rightarrow i$ (transitivity)

➡ $g \not\rightarrow l$ and $l \not\rightarrow g$: l and g are **concurrent** (*nebenläufig*)

Distributed Systems

Winter Term 2025/26

27.11.2025

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

Stand: January 8, 2026



6.3 Logical Clocks

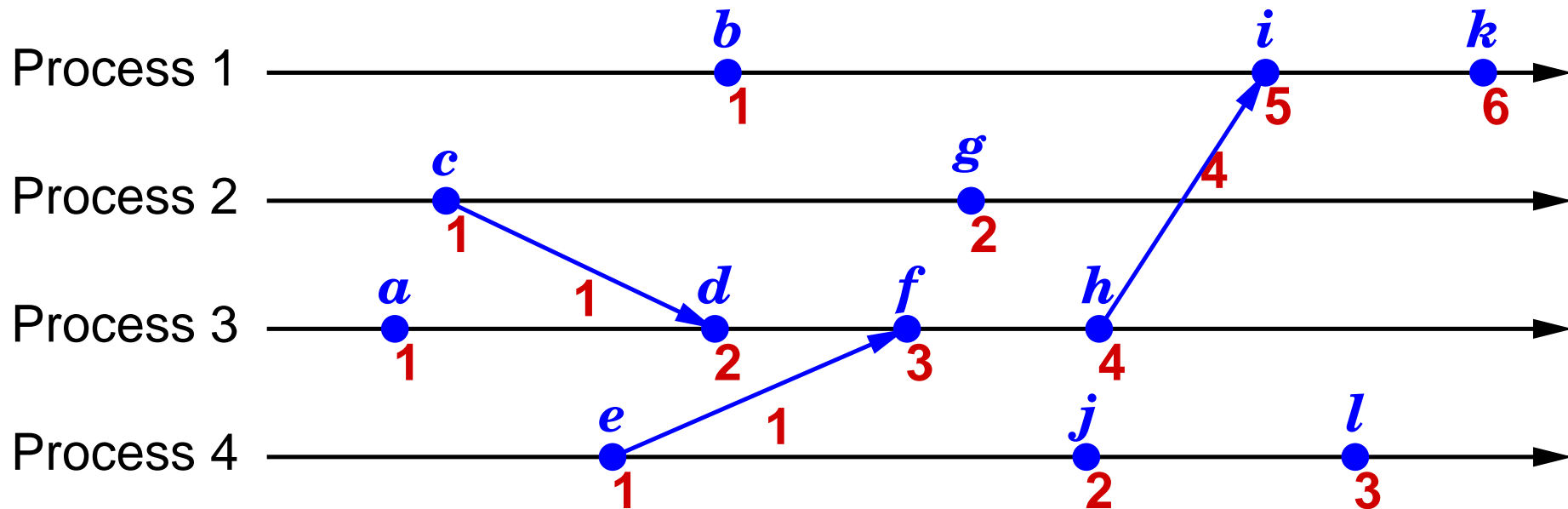
- ➔ Physical clocks cannot be synchronized exactly
 - ➔ therefore: unsuitable for determining the **order** in which events occurred
- ➔ Logical clocks
 - ➔ refer to the causal order of events (happened-before relation)
 - ➔ no fixed relationship to real time
- ➔ In the following:
 - ➔ Lamport timestamps
 - ➔ are consistent with the happened-before relation
 - ➔ vector timestamps
 - ➔ allow sorting of events according to causality (i.e. happened-before relation)

Lamport Timestamps

- ➔ Lamport timestamps are natural numbers
- ➔ Each process i has a local counter $L_i \in \mathbb{N}_0$
- ➔ L_i is updated as follows:
 - ➔ at (more precisely: before) each local event: $L_i := L_i + 1$
 - ➔ in each message, the time stamp L_i of the send event is also sent
 - ➔ at receipt of a message with time stamp t :
 $L_i := \max(L_i, t + 1)$
- ➔ Lamport time stamps are consistent with the causality:
 - ➔ $a \rightarrow b \Rightarrow L(a) < L(b)$, where L is the Lamport timestamp in the respective process
 - ➔ but the reversal does not apply!

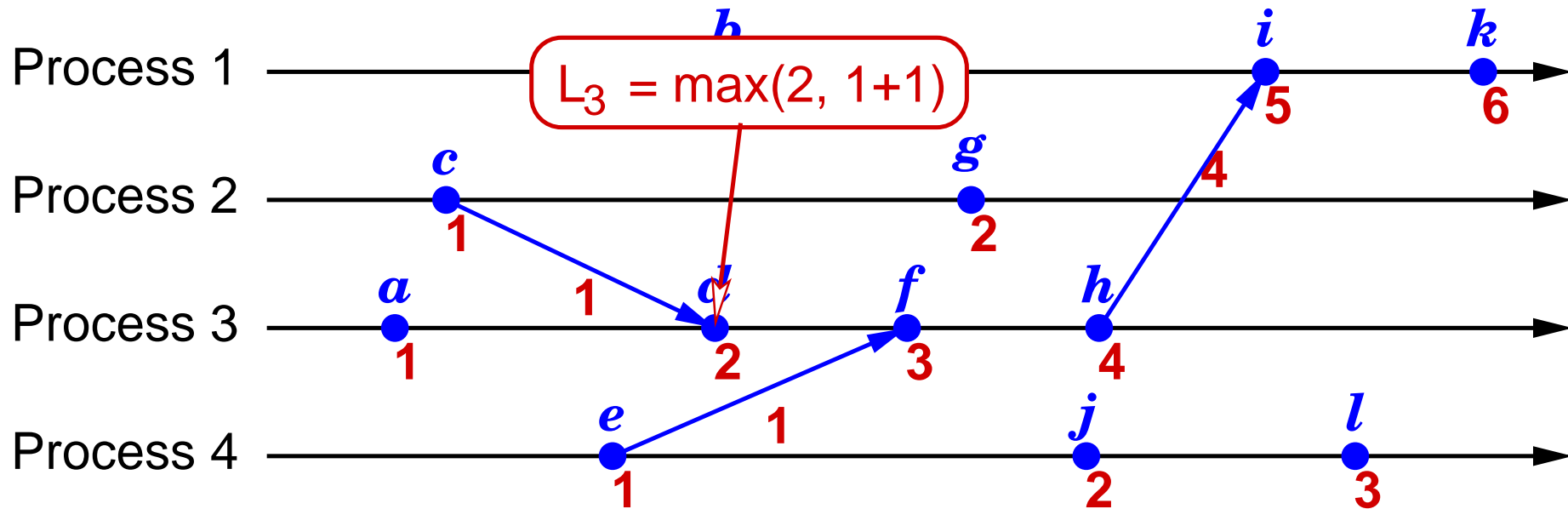


Lamport Timestamps: Example



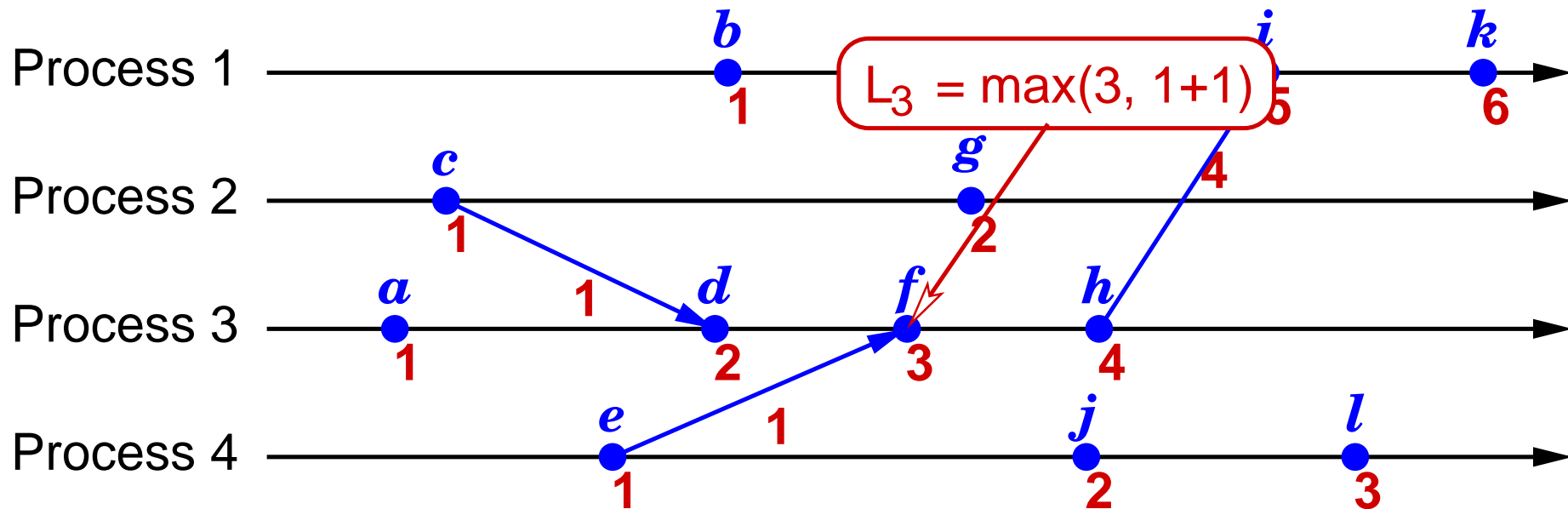


Lamport Timestamps: Example



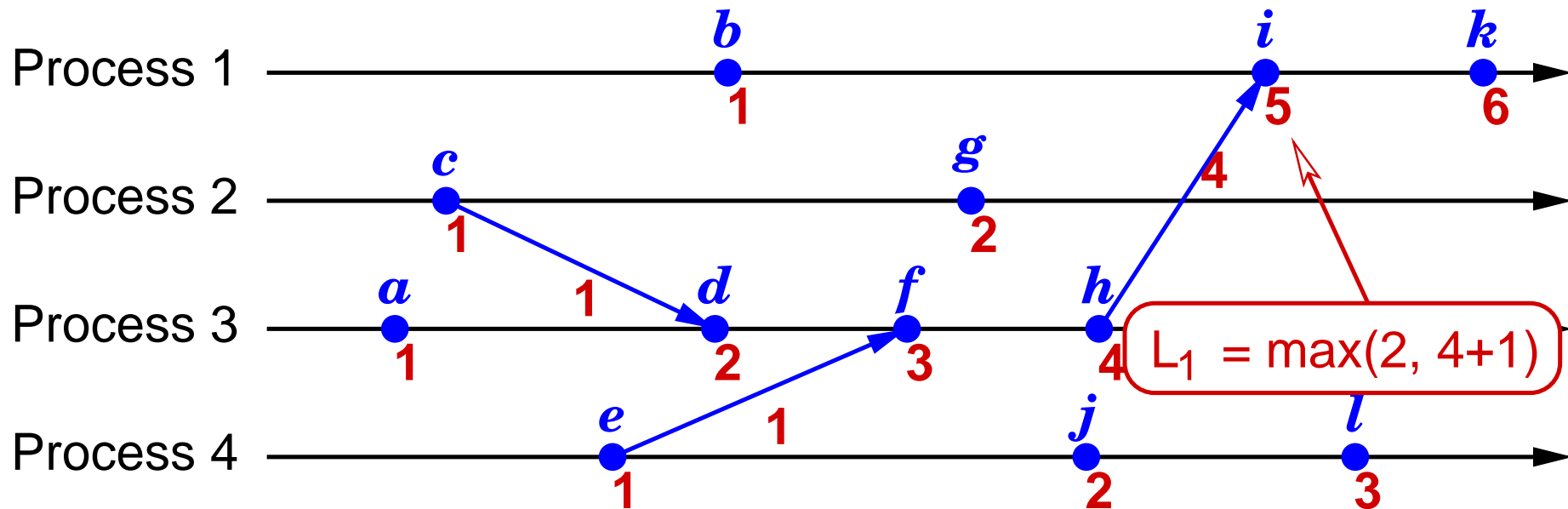


Lamport Timestamps: Example

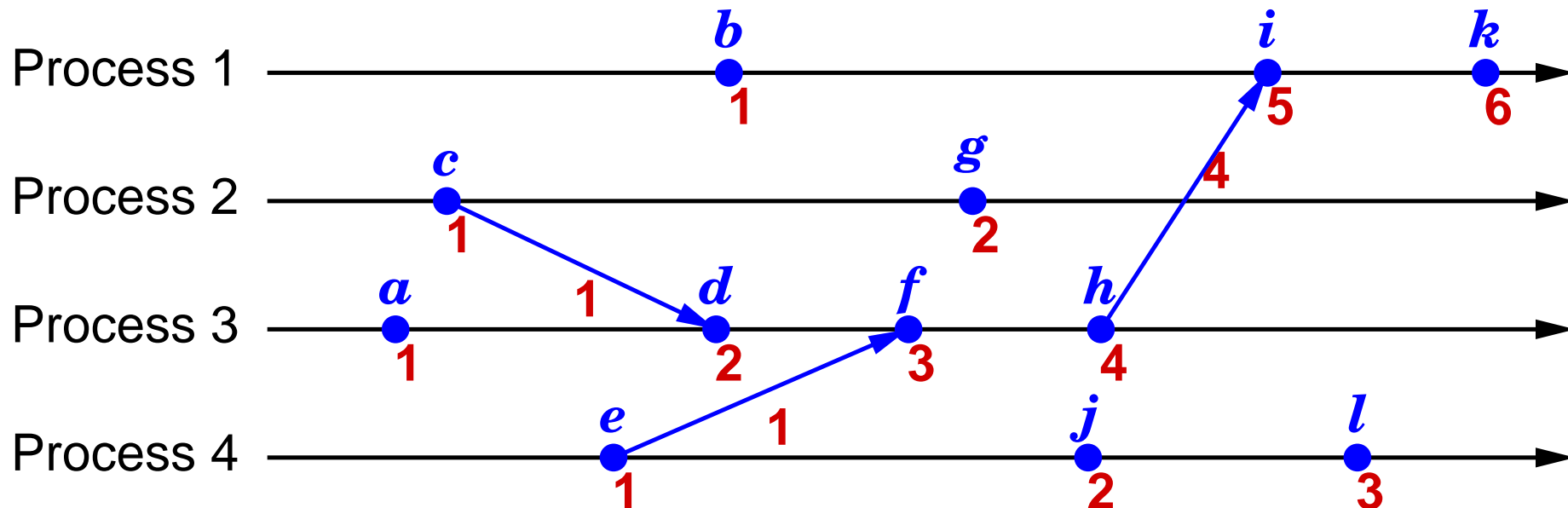




Lamport Timestamps: Example



Lamport Timestamps: Example



➡ Among others, we have here:

➡ $c \rightarrow k$ and $L(c) < L(k)$

➡ $g \not\rightarrow j$ and $L(g) \not< L(j)$

➡ $g \not\rightarrow l$, but still $L(g) < L(l)$



Vector Timestamps

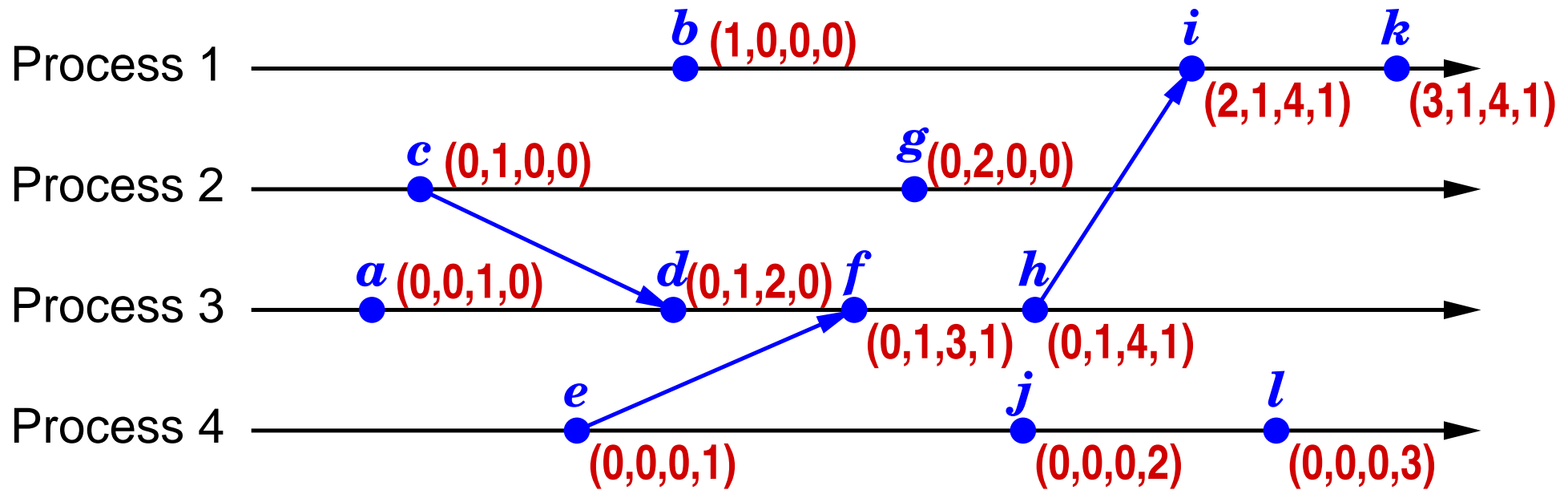
- ➔ Objective: timestamps that characterize causality
 - ➔ $a \rightarrow b \Leftrightarrow V(a) < V(b)$, where V is the vector timestamp in the respective process
- ➔ Vector timestamps
 - ➔ in a system with N processes, each process has its own vector $V_i \in (\mathbb{N}_0)^N$ of N integers
 - ➔ $V_i[i]$: number of events that have occurred so far in process i
 - ➔ $V_i[j], j \neq i$: number of events in process j , of which i knows
 - ➔ i.e. by which it could have been causally influenced

Vector Timestamps ...

- ➔ Update of V_i in process i :
 - ➔ before any local event: $V_i[i] := V_i[i] + 1$
 - ➔ V_i is included in every message sent
 - ➔ when receiving a message with timestamp t :
 $V_i[j] := \max(V_i[j], t[j])$ for all $j = 1, 2, \dots, N$
- ➔ Comparison of vector timestamps:
 - ➔ $V = V' \Leftrightarrow V[j] = V'[j]$ for all $j = 1, 2, \dots, N$
 - ➔ $V \leq V' \Leftrightarrow V[j] \leq V'[j]$ for all $j = 1, 2, \dots, N$
 - ➔ $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
 - ➔ the relation $<$ defines a **partial** order

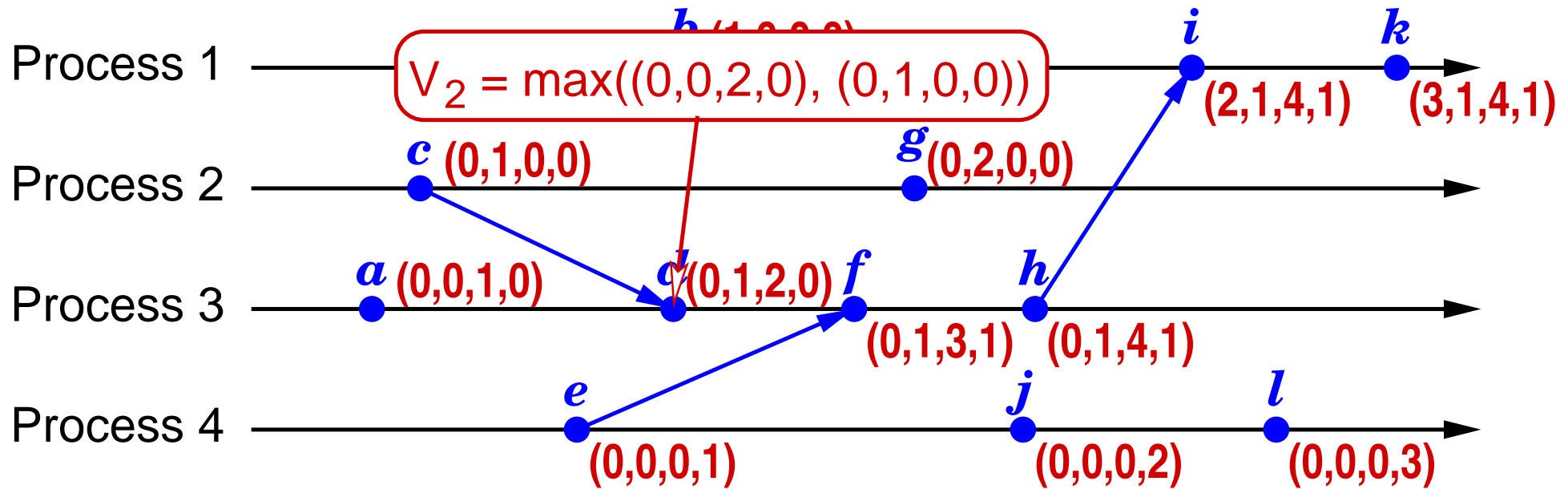


Vector Timestamps: Example



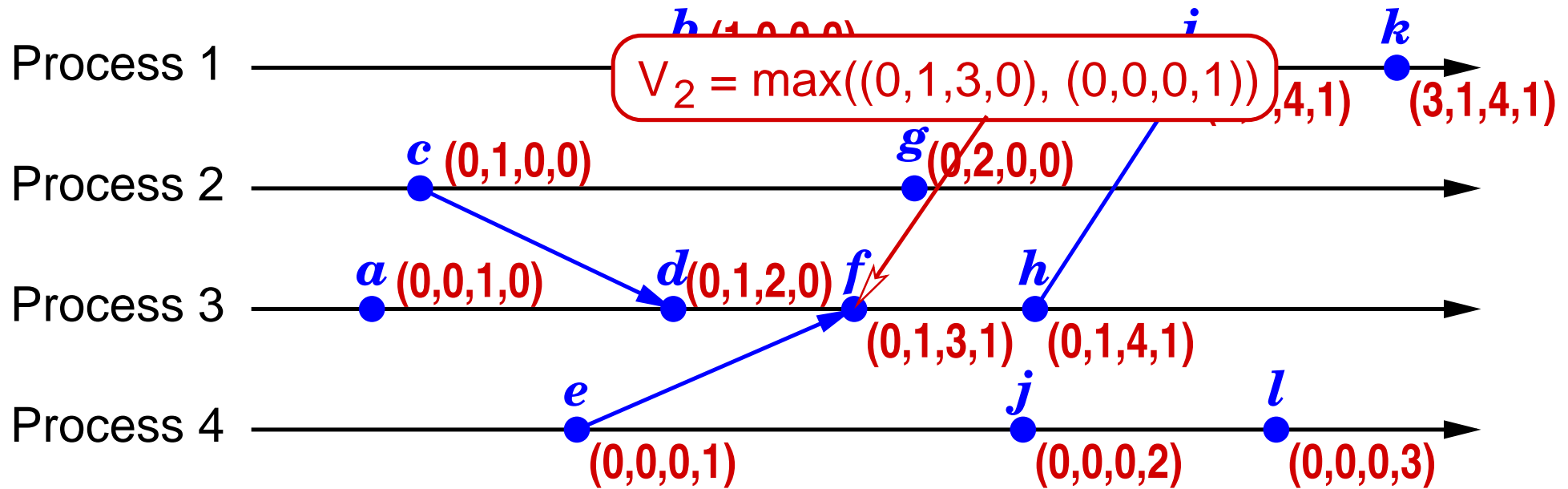


Vector Timestamps: Example



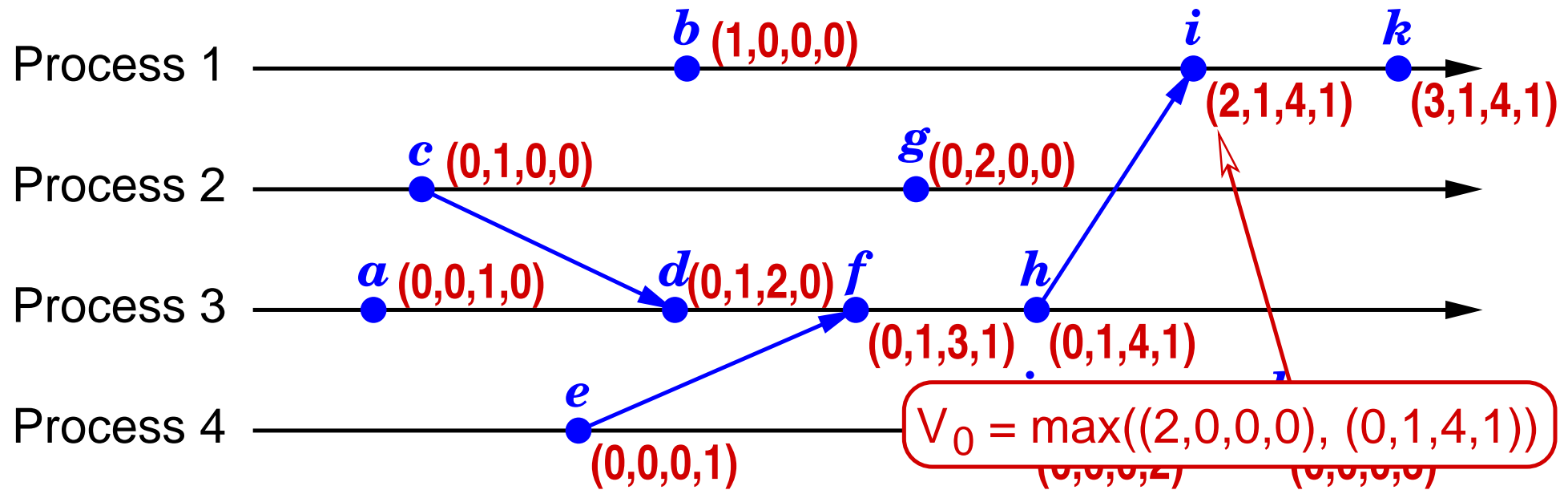


Vector Timestamps: Example

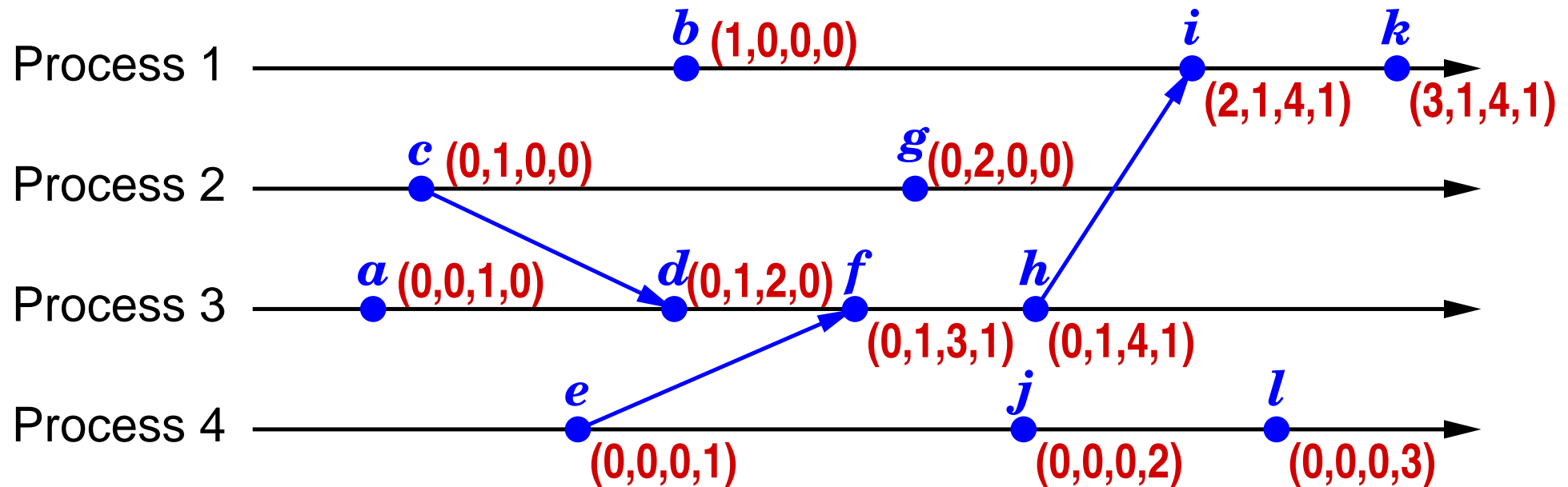




Vector Timestamps: Example



Vector Timestamps: Example



➔ Among others, we have here:

➔ $c \rightarrow k$ and $V(c) < V(k)$

➔ $g \not\rightarrow l$ and $V(g) \not\leq V(l)$, as well as $l \not\rightarrow g$ and $V(l) \not\leq V(g)$

➔ $V(l)$ and $V(g)$ not comparable $\Leftrightarrow l$ and g concurrent



A Motivating Example

- ➔ Scenario: peer-to-peer application, processes send requests to each other
- ➔ Question: when can the application terminate?
- ➔ Answer: when no process is processing a request

A Motivating Example

- ➔ Scenario: peer-to-peer application, processes send requests to each other
- ➔ Question: when can the application terminate?
- ➔ **Wrong** answer: when no process is processing a request
 - ➔ reason: requests can still be on the way in messages!



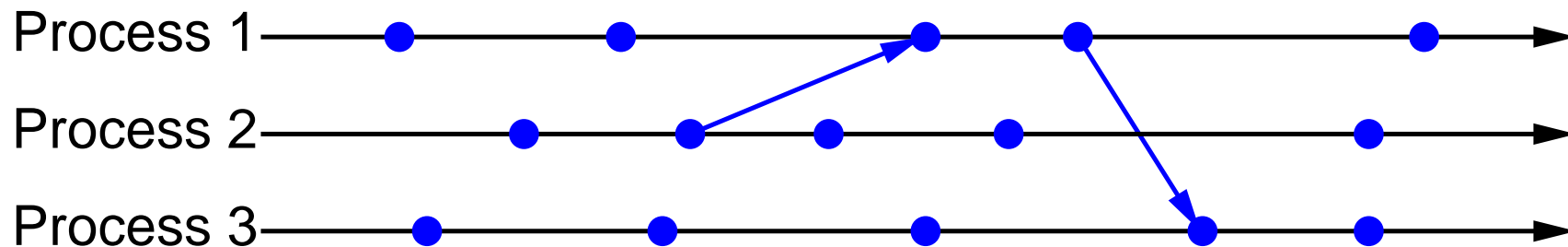
- ➔ Other applications: distributed garbage collection, distributed deadlock detection, ...



- ➔ How can we determine the overall state of a distributed process system?
 - ➔ naïvely: union of the states of all processes (**wrong!**)
- ➔ Two aspects have to be considered:
 - ➔ messages that are still in transit
 - ➔ must be included in the state
 - ➔ lack of global time
 - ➔ a global state at time t cannot be defined!
 - ➔ process states always refer to local (and thus different) times
 - ➔ question: condition on local times? \Rightarrow **consistent cuts**

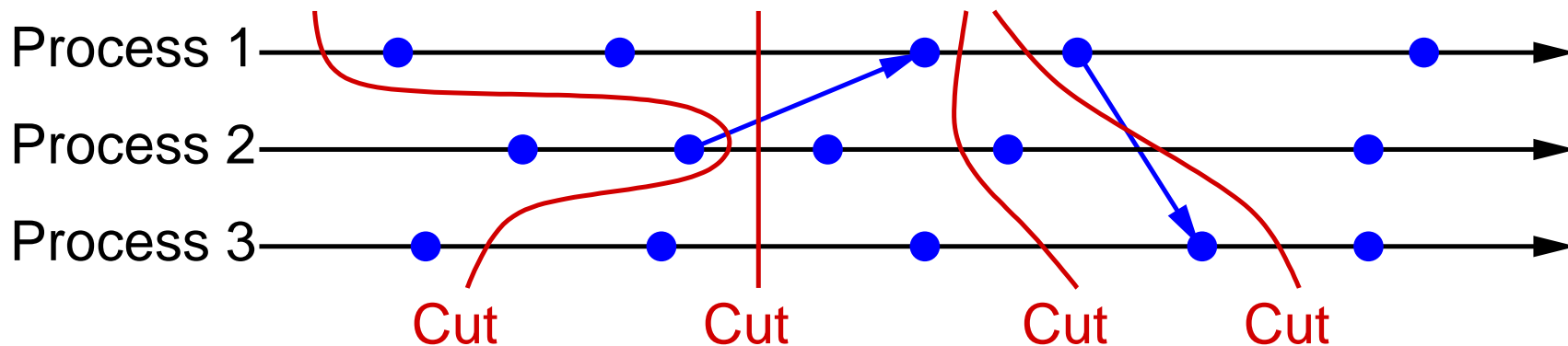
Consistent Cuts

- ➔ Objective: build a meaningful global state from local states (which are not determined simultaneously)
- ➔ Processes are modeled by sequences of events:



Consistent Cuts

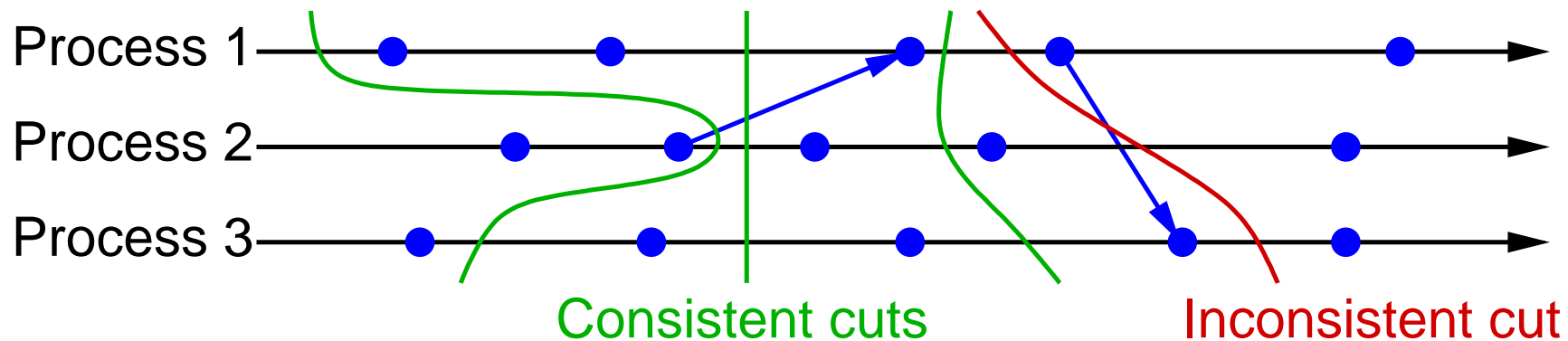
- ➔ Objective: build a meaningful global state from local states (which are not determined simultaneously)
- ➔ Processes are modeled by sequences of events:



- ➔ **Cut**: consider a **prefix** of the event sequence in each process

Consistent Cuts

- ➔ Objective: build a meaningful global state from local states (which are not determined simultaneously)
- ➔ Processes are modeled by sequences of events:



- ➔ **Cut**: consider a **prefix** of the event sequence in each process
- ➔ **Consistent cut**:
 - ➔ if the cut contains the reception of a message, it also contains the sending of this message



The Snapshot Algorithm of Chandy and Lamport

- ➔ Determines online a “snapshot” of the global state
 - ➔ i.e.: a consistent cut
- ➔ The global state consists of:
 - ➔ the local states of all processes
 - ➔ the status of all communication connections
 - ➔ i.e. the messages in transmission
- ➔ Assumptions / properties:
 - ➔ reliable message channels with sequence retention
 - ➔ process graph is strongly connected
 - ➔ each process can trigger a snapshot at any time
 - ➔ the processes are not blocked during the algorithm



The Snapshot Algorithm of Chandy and Lamport ...

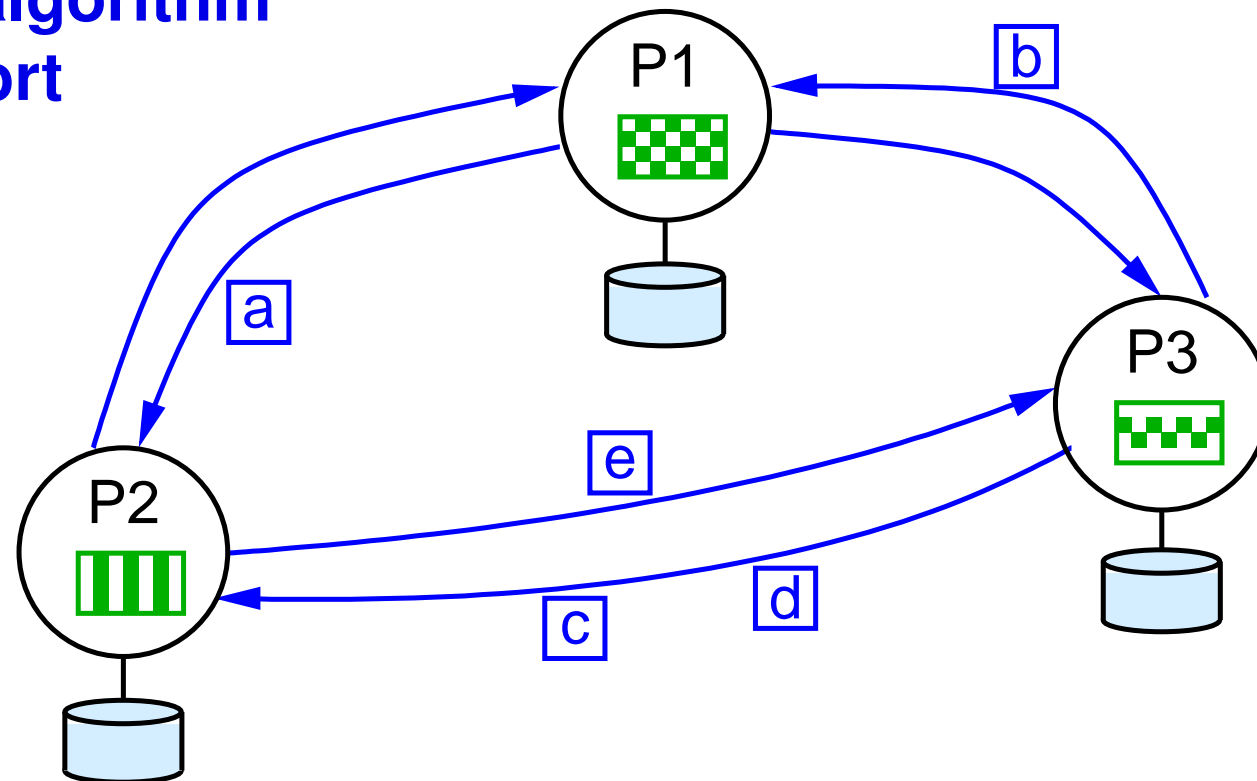
- ➔ When a process wants to initiate a snapshot:
 - ➔ process first saves its local state
 - ➔ then it sends a marker message over each outgoing channel
- ➔ When a process receives a marker message:
 - ➔ if it has not yet saved its local state:
 - ➔ it saves its local state
 - ➔ and sends a marker over each outgoing channel
 - ➔ else:
 - ➔ for the channel where the marker was received, it saves all messages that have been received since the local state was saved
 - ➔ i.e., it records the status of the channel



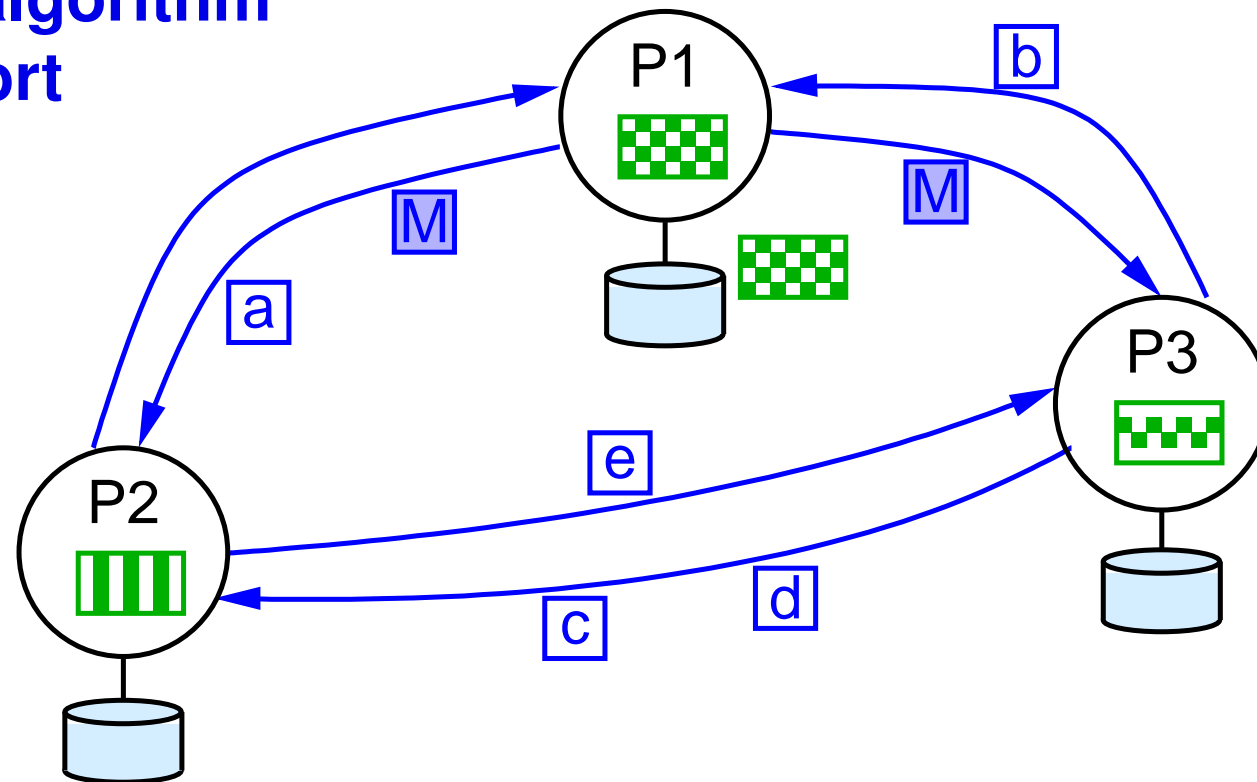
The Snapshot Algorithm of Chandy and Lamport ...

- ➔ The algorithm terminates when each process has received a marker message on each channel
- ➔ the determined consistent section is then (initially) stored in a distributed way

Example for the algorithm of Chandy/Lamport

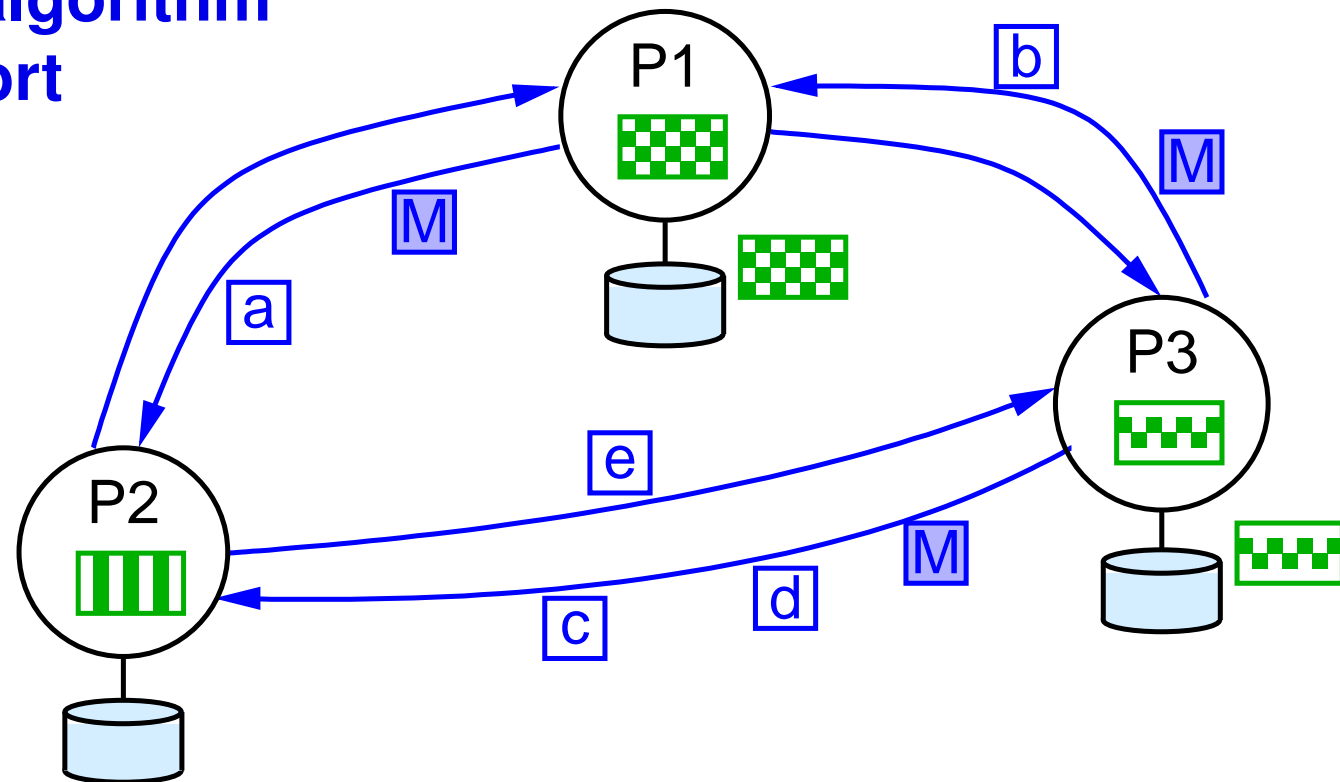


Example for the algorithm of Chandy/Lampert



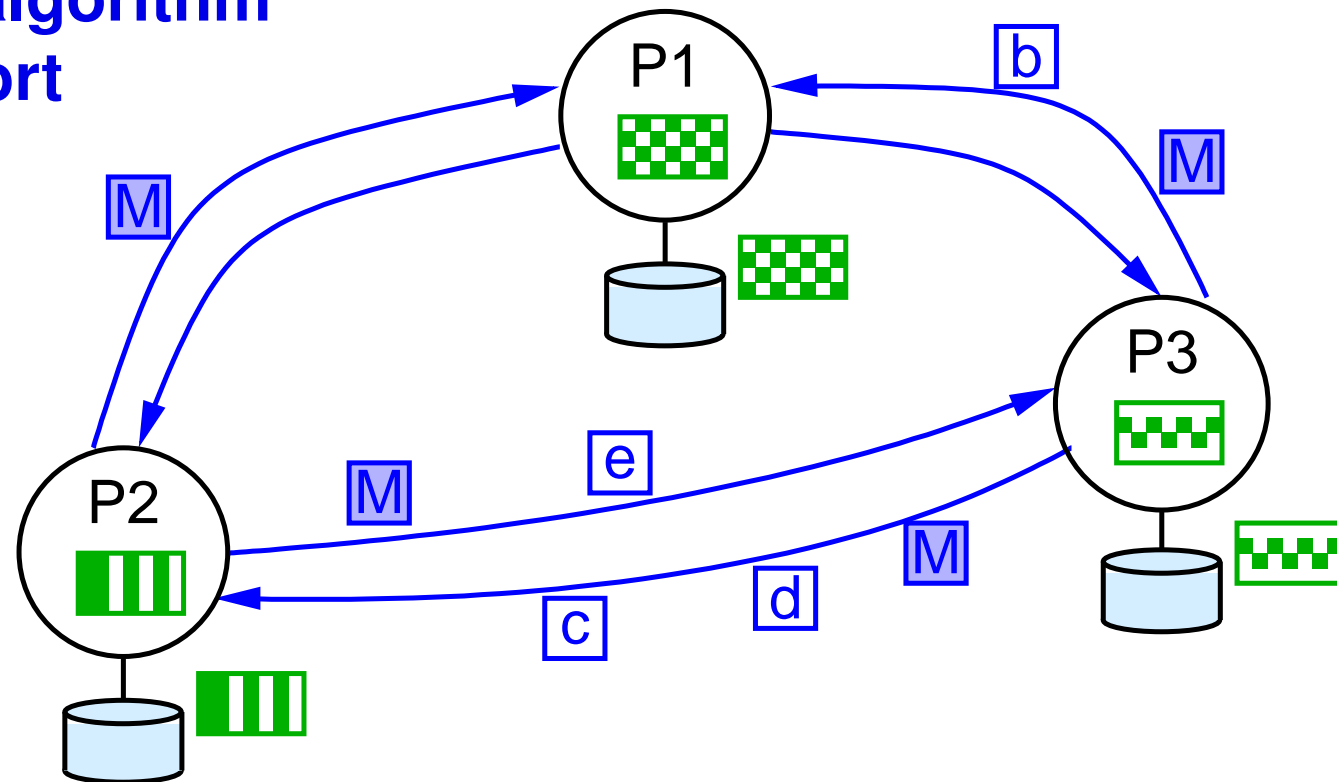
1. P1 initiates a snapshot, saves its state, and sends markers

Example for the algorithm of Chandy/Lampert



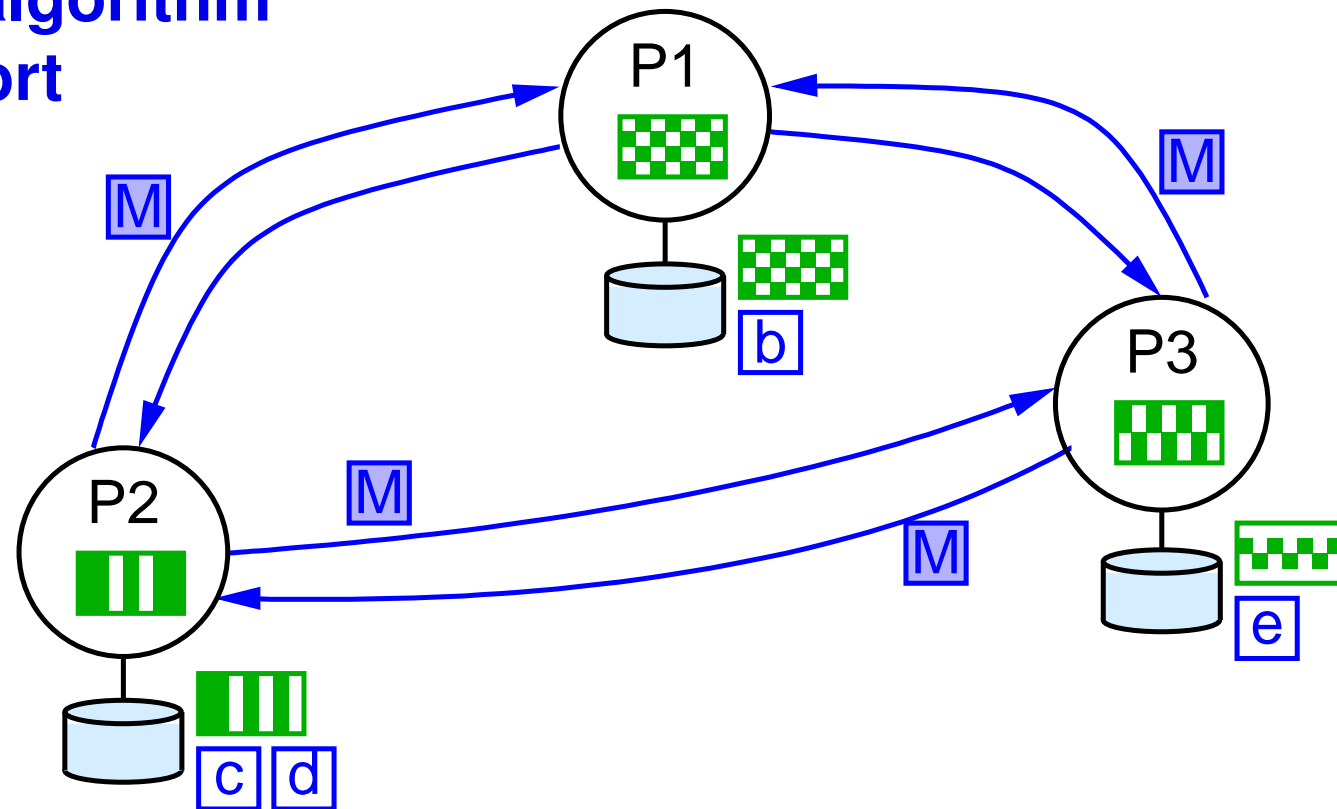
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers

Example for the algorithm of Chandy/Lamport



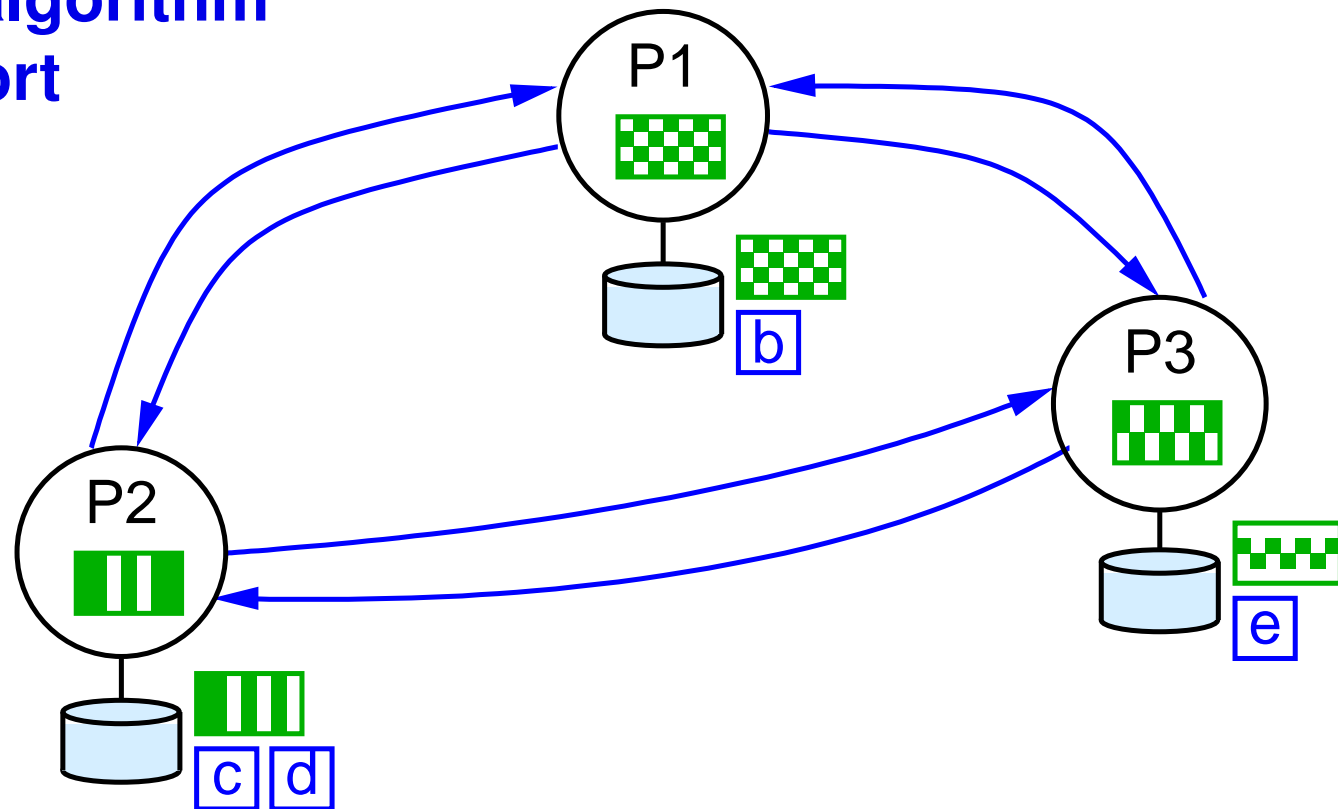
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a
P2 receives the marker from P1, saves its state, and sends markers

Example for the algorithm of Chandy/Lampert



1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a
P2 receives the marker from P1, saves its state, and sends markers
4. P1, P2, P3 save the incoming messages, until all markers are received

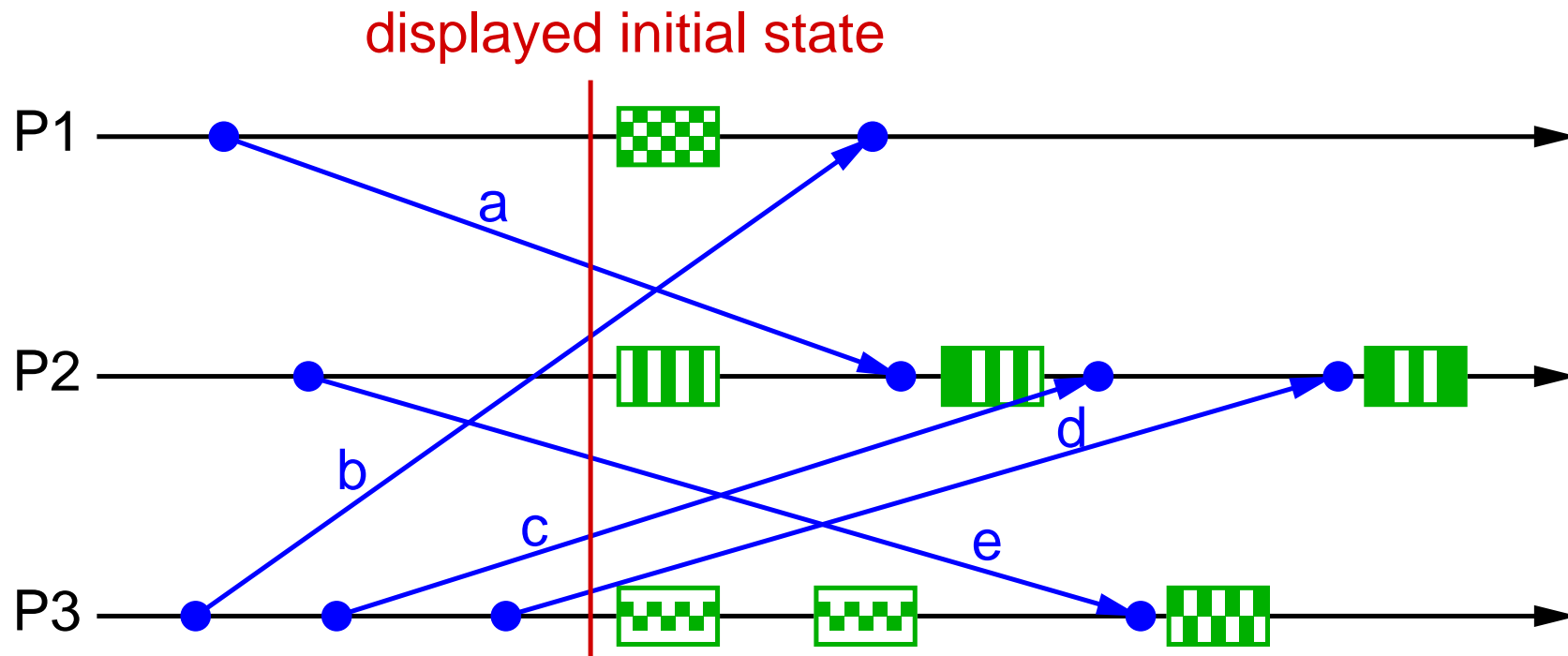
Example for the algorithm of Chandy/Lamport



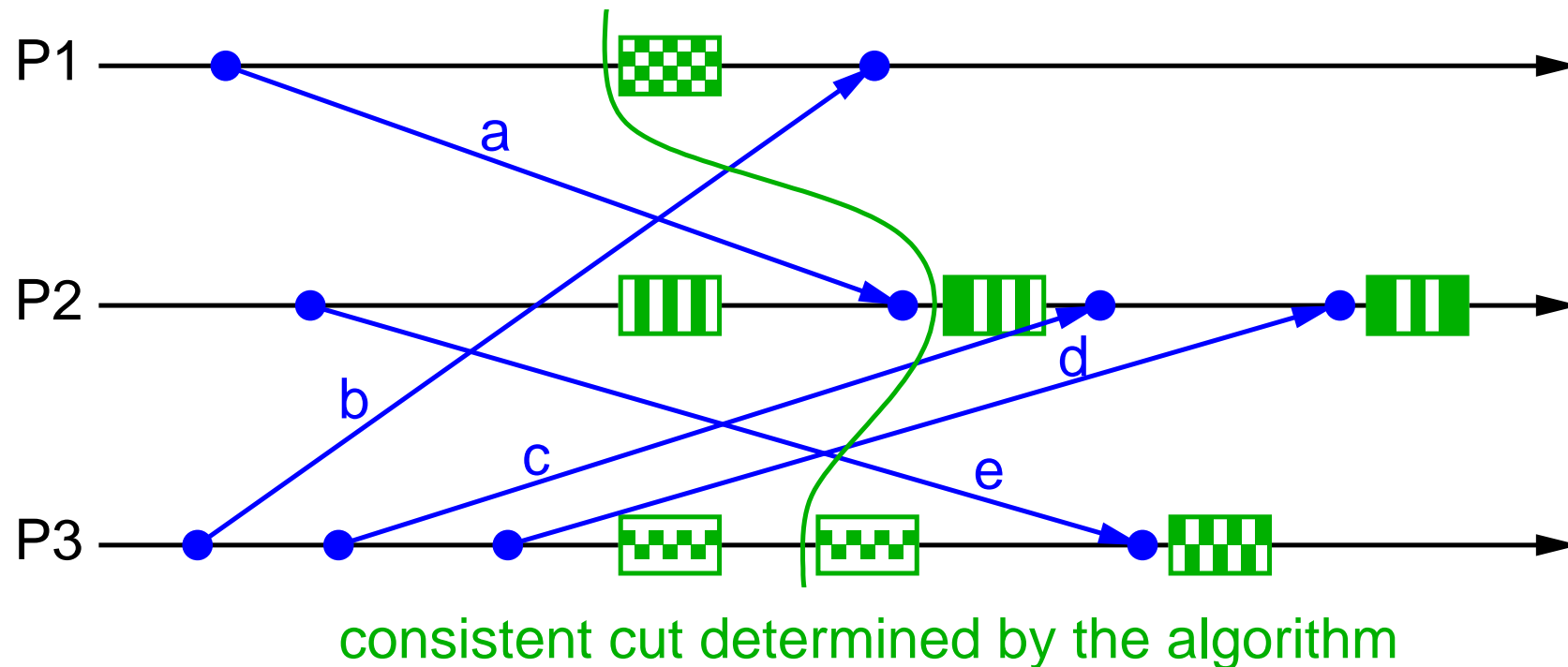
1. P1 initiates a snapshot, saves its state, and sends markers
2. P3 receives a marker from P1, saves its state, and sends markers
3. P2 receives and processes a
P2 receives the marker from P1, saves its state, and sends markers
4. P1, P2, P3 save the incoming messages, until all markers are received



Sequence in the Example and Selected Cut



Sequence in the Example and Selected Cut



- ➔ The cut consists of the local states of P1, P2, P3 and the messages b, c, d, e

Distributed Systems

Winter Term 2025/26

7 Fault Tolerance



Contents

- ➔ Introduction
- ➔ Process elasticity
- ➔ Reliable communication
- ➔ Recovery

Literature

- ➔ Tanenbaum, van Steen: Ch. 7



Concepts

- ➔ **Failure**: external incorrect behavior (system no longer keeps its promises)
- ➔ **Error**: (unobserved) incorrect internal state
- ➔ **Fault**: physical defect (in HW or SW) causing the error
 - ➔ fault can be transient, periodic or permanent
- ➔ **Fault tolerance**: system does not fail despite a fault
- ➔ Requirement for reliable systems:
 - ➔ **availability**: $p(\text{system is working at time } t)$
 - ➔ **reliability**: $p(\text{system is working in time interval } \Delta t)$
 - ➔ **safety**: no major damage if system fails
 - ➔ **maintainability**: effort for “repair” after a failure



Failure models

Crash failure	Server halts
Omission failure Receive omission Send omission	Server is not responding to requests Server doesn't receive incoming requests Server doesn't send messages
Timing failure	Response time is outside the specification
Response failure Value failure State transition f.	Server's response is incorrect Only the value of the answer is wrong Incorrect control flow in server
Byzantine failure	Random answers at arbitrary time

➔ Further distinction: can the client detect the failure or not?



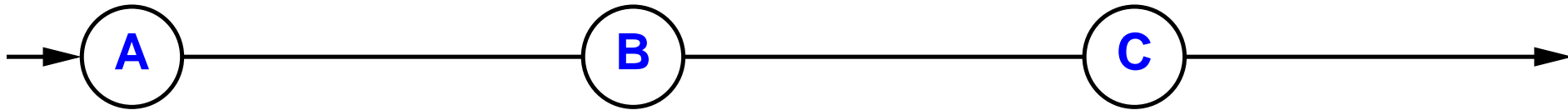
Failure masking through redundancy

- ➔ Fault tolerant system must hide faults from other processes
- ➔ Most important technique: **redundancy**
 - ➔ **information redundancy**: additional “check bits” (e.g., CRC)
 - ➔ **time redundancy**: repetition of faulty actions
 - ➔ **physical redundancy**: important components are provided multiple times
- ➔ Example: **TMR, *triple modular redundancy***
 - ➔ components are replicated three times
 - ➔ majority decision for the results
 - ➔ protects against (Byzantine) failure of a *single* replicated component

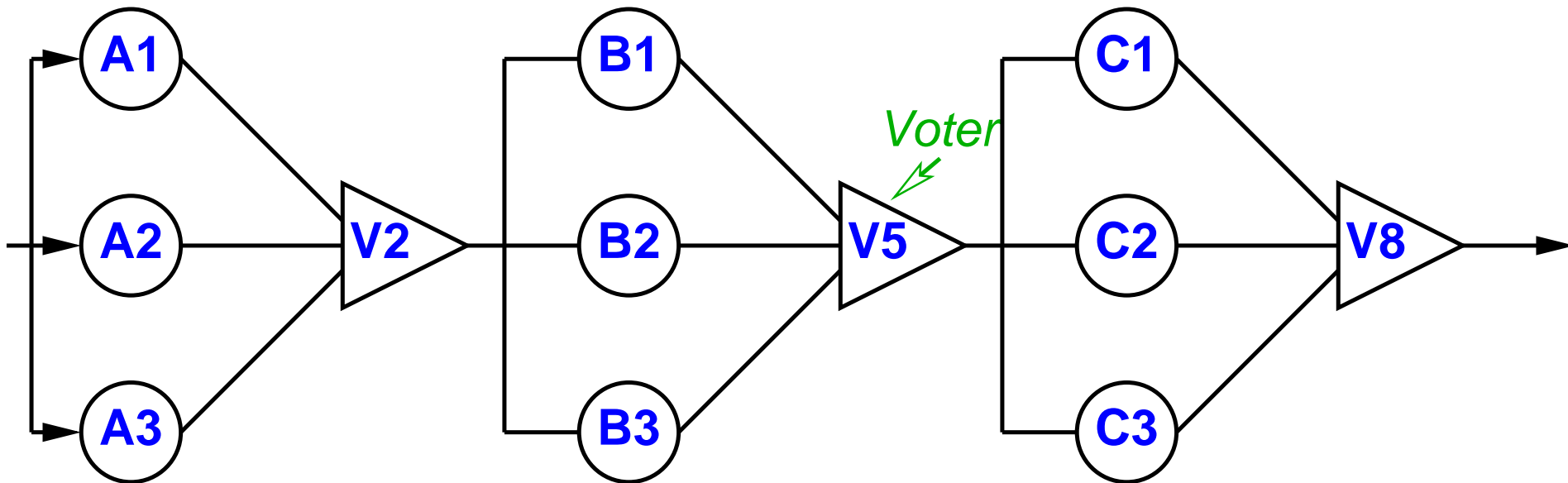


Example for TMR

Without redundancy



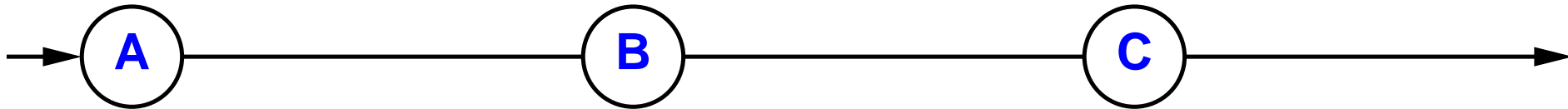
With TMR



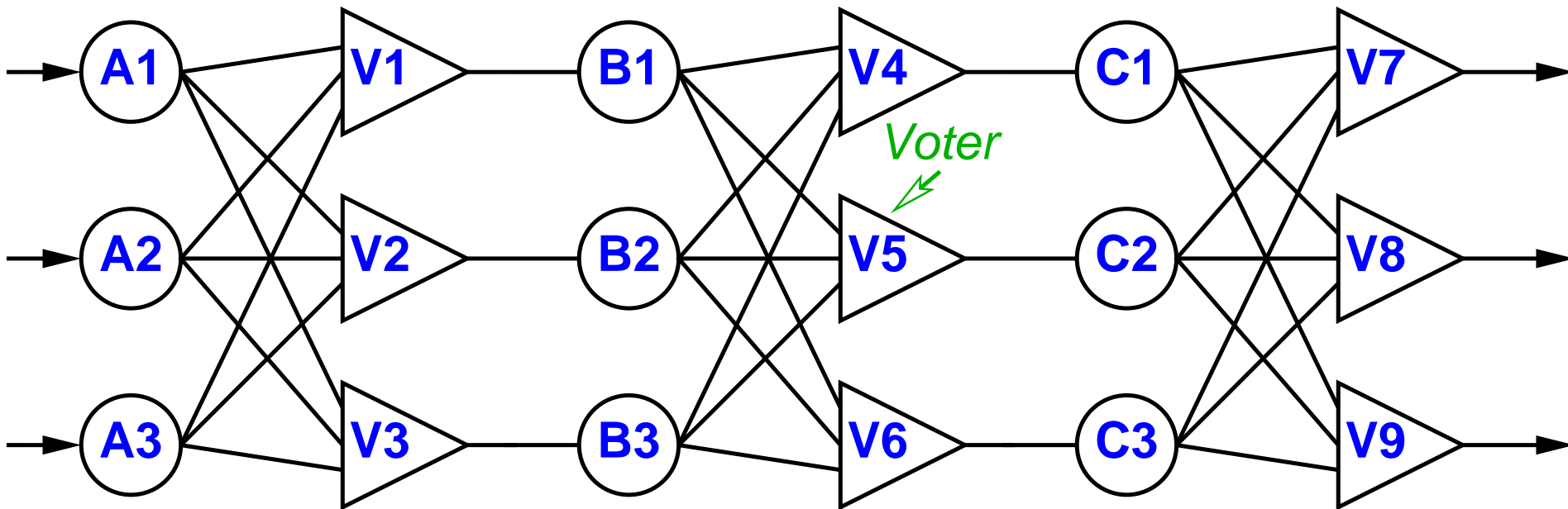


Example for TMR

Without redundancy



With TMR





Objective: Protection Against Process Failure

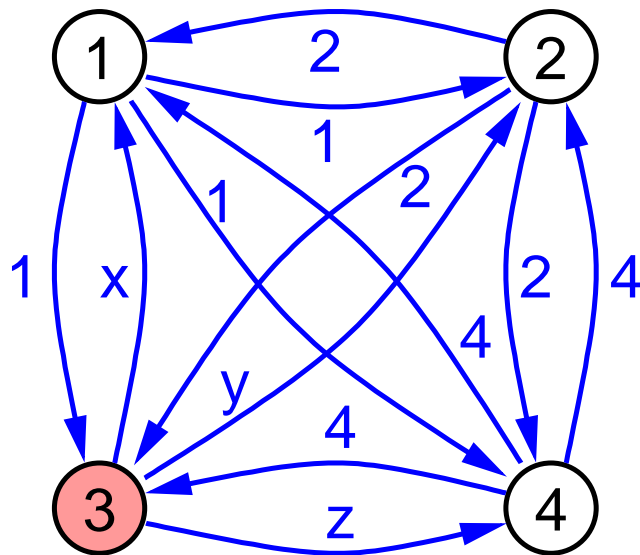
- ➔ By replicating processes in groups
 - ➔ message to the group is received by all members
 - ➔ usually with total ordered multicast (☞ 8.3)
- ➔ Questions:
 - ➔ organization of the groups?
 - ➔ flat (symmetrical) vs. hierarchical (central coordinator)
 - ➔ group administration, synchronous join / exit
 - ➔ necessary number of replicas?
 - ➔ **k fault tolerant**: failure of k processes can be tolerated
 - ➔ for silent failures: $\geq k + 1$ Processes
 - ➔ for Byzantine failures: $\geq 2k + 1$ processes
 - ➔ agreement in faulty systems?



Agreement in faulty systems

- ➔ Agreement is impossible with unreliable communication
 - ➔ two army problem
- ➔ Agreement of faulty processes with reliable communication
 - ➔ Byzantine agreement problem (*byzantinische Generäle*)
 - ➔ agreement only possible if $> \frac{2}{3}$ of the processes work correctly

1. Send information



2. Received information

- 1: (1, 2, x, 4)
- 2: (1, 2, y, 4)
- 3: (t, u, v, w)
- 4: (1, 2, z, 4)

3. Send received information to all other processes

1 gets:	2 gets:	4 gets:
from 1:	(1, 2, x, 4)	(1, 2, x, 4)
from 2:	(1, 2, y, 4)	(1, 2, y, 4)
from 3:	(a, b, c, d)	(e, f, g, h)
from 4:	(1, 2, z, 4)	(1, 2, z, 4)

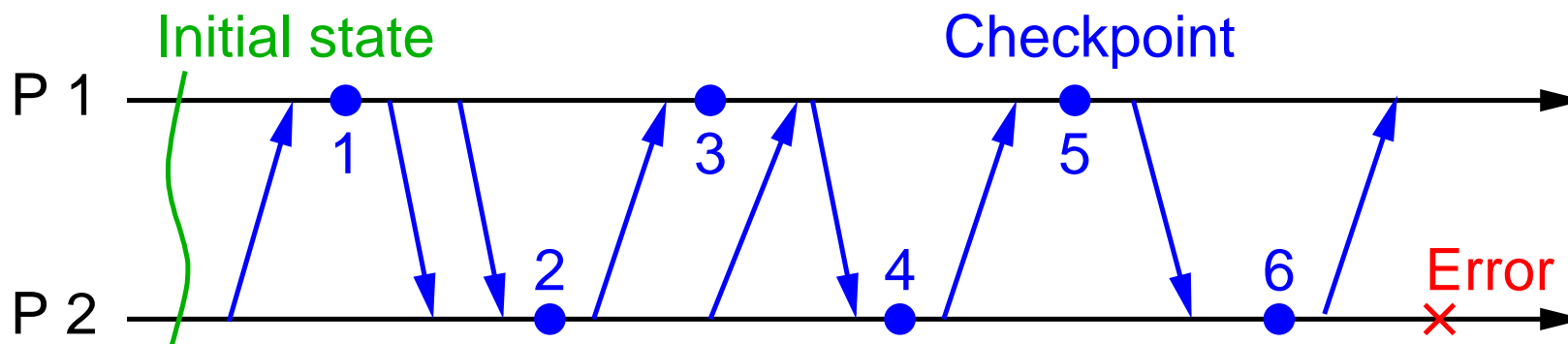


Objective: Protection Against Communication Failures

- ➔ Point-to-point communication (☞ **RN_I**)
 - ➔ TCP masks omission failures, but not crash failures
- ➔ Client/server communication (☞ **2.1**)
 - ➔ possible failures:
 - ➔ server not found
 - ➔ lost request
 - ➔ server crash while processing the request
 - ➔ lost reply
 - ➔ client crash after sending the request
- ➔ Group communication (☞ **8.3**)
- ➔ Distributed commit (☞ **8.4**)

Objective: System Recovery After an Error

- ➔ Forward error recovery: go to a correct new state
- ➔ Backward error recovery: go to a correct earlier state
 - ➔ i.e. reset to a consistent cut
 - ➔ regular backup to stable storage (*checkpointing*)
- ➔ Independent checkpointing
 - ➔ processes save their state independently of each other
 - ➔ problem: domino effect



Distributed Systems

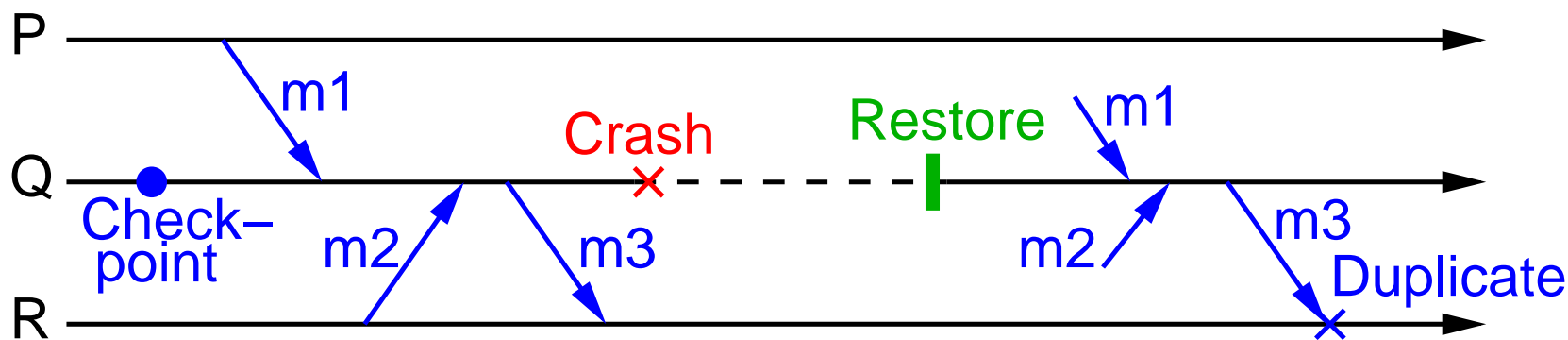
Winter Term 2025/26

04.12.2025

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

Stand: January 8, 2026

- ➔ Coordinated checkpoints
 - ➔ Chandy/Lamport algorithm (👉 6.4)
 - ➔ alternatively: blocking 2 phase protocol
 - ➔ problem: requires to reset all processes
- ➔ Local checkpoints with message logging
 - ➔ goal: restore the crashed process to a state consistent with the current state of the other processes
 - ➔ reset to last checkpoint and restore the received messages





Distributed Systems

Winter Term 2025/26

8 Coordination



Contents

- ➔ Election algorithms
- ➔ Mutual exclusion
- ➔ Group communication (multicast)
- ➔ Transactions

Literature

- ➔ Tanenbaum, van Steen: Kap. 5.4-5.6
- ➔ Colouris, Dollimore, Kindberg: Kap. 11, 12
- ➔ Stallings: Kap 14.3



8.1 Election Algorithms

- ➔ In many distributed algorithms **one** arbitrary process must play an exceptional role
 - ➔ e.g. central coordinator, initiator, ...
- ➔ Question: how to choose this process unambiguously?
 - ➔ processes must be distinguishable, e.g. via a unique ID.
 - ➔ then select e.g. the process with the highest ID
- ➔ Prerequisites / requirements:
 - ➔ election can be initiated by multiple processes concurrently
 - ➔ e.g. after failure or recovery of a process
 - ➔ after the election all processes must have the same result
 - ➔ in the following: each process knows the IDs of all other processes, but does not know whether they are running or not

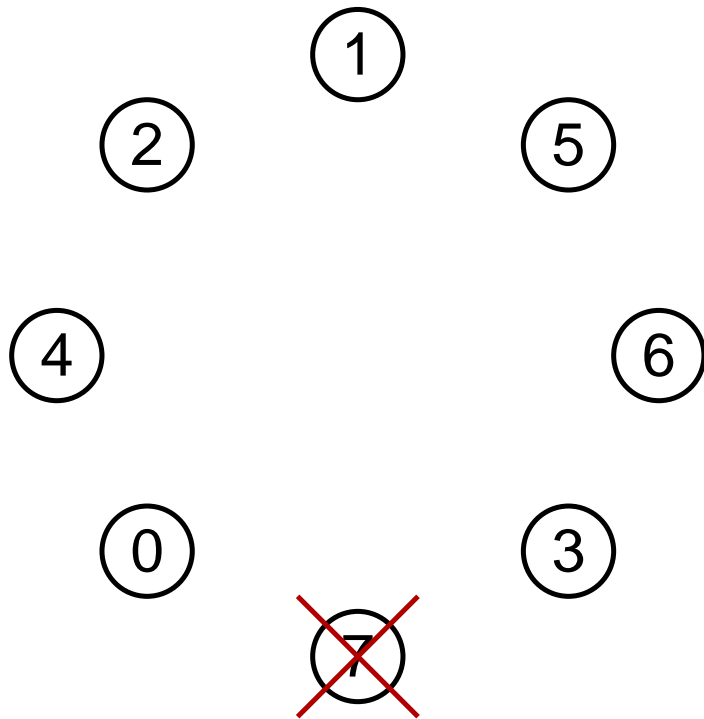


The Bully Algorithm

- ➔ A process P holds an election as follows:
 - ➔ P sends an ELECTION message to all processes with a larger ID
 - ➔ if none of the processes reacts, P wins the election
 - ➔ if a process responds: P loses the election
- ➔ When a process receives an ELECTION message:
 - ➔ (message comes from a process with a lower ID)
 - ➔ return an OK message
 - ➔ hold an election of your own
- ➔ At some point, there is only one process left
 - ➔ this wins the election and sends the result to all others



Bully Algorithm: Example

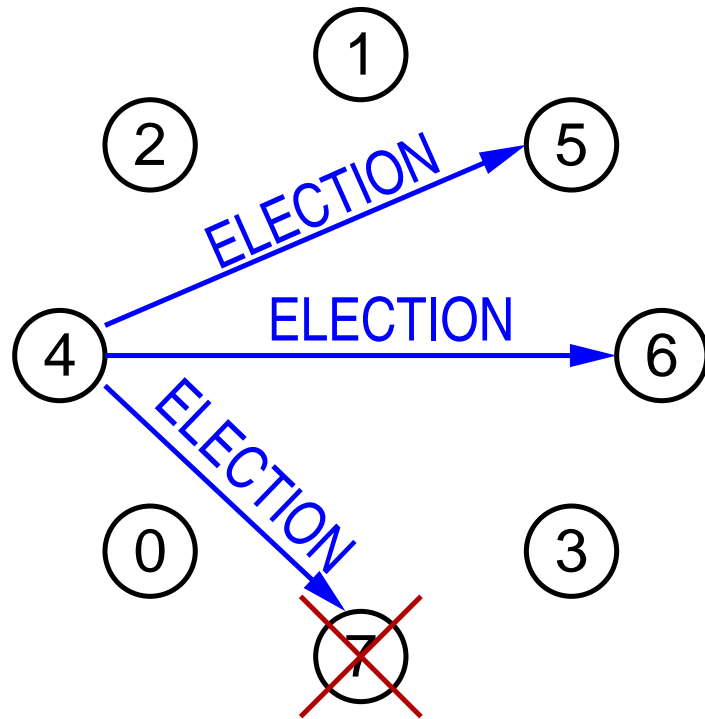


Previous Coordinator
has crashed



Bully Algorithm: Example

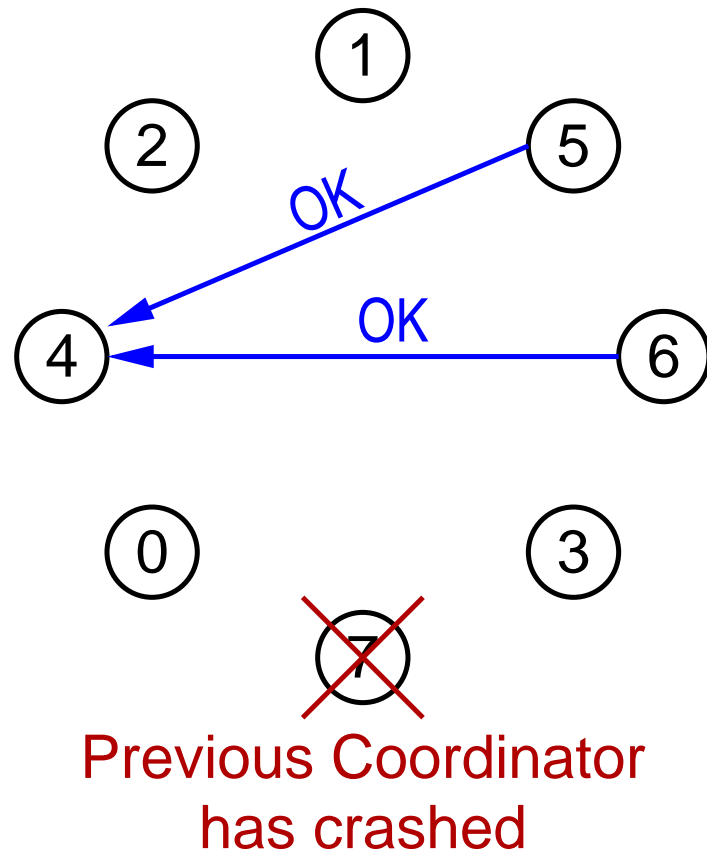
Process 4 holds an election



Previous Coordinator
has crashed



Bully Algorithm: Example



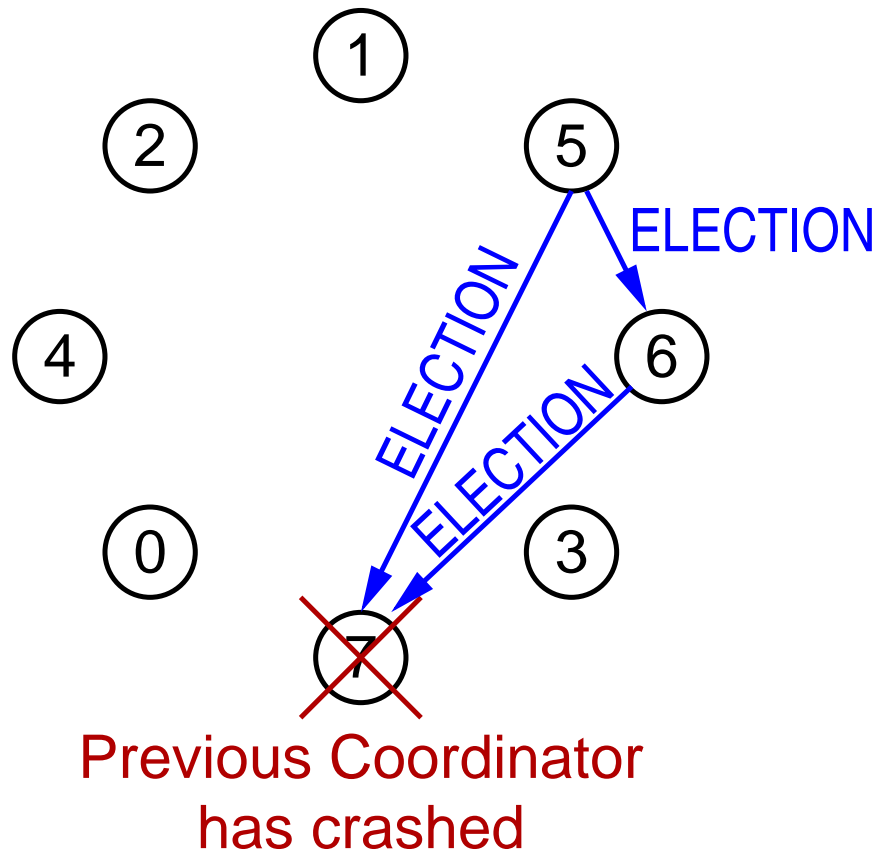
Process 4 holds an election

Processes 5 and 6 reply,

Process 4 terminates its election



Bully Algorithm: Example



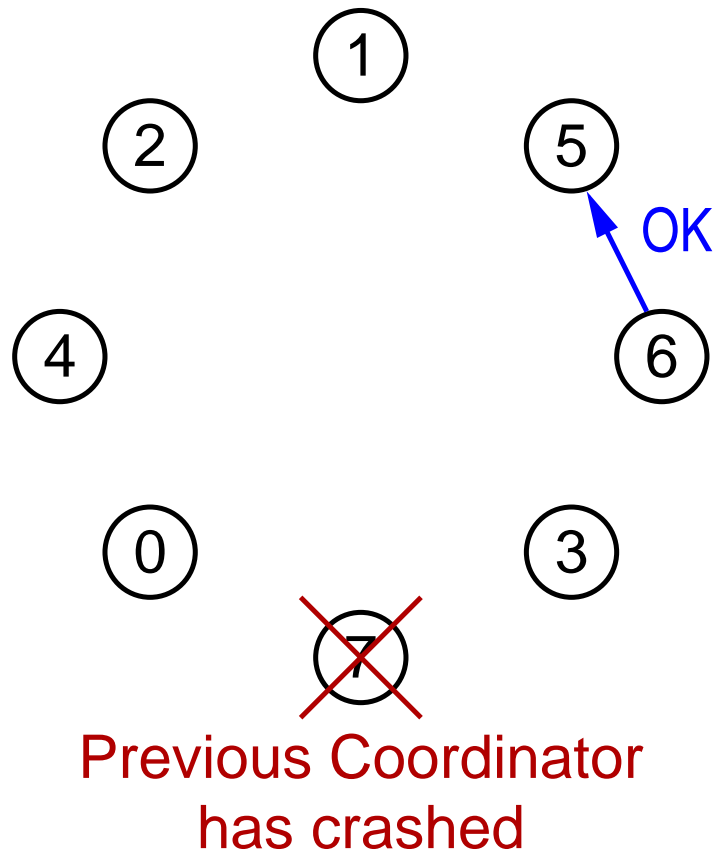
Process 4 holds an election

Processes 5 and 6 reply,
Process 4 terminates its election

Processes 5 and 6 simultaneously
hold an election



Bully Algorithm: Example



Process 4 holds an election

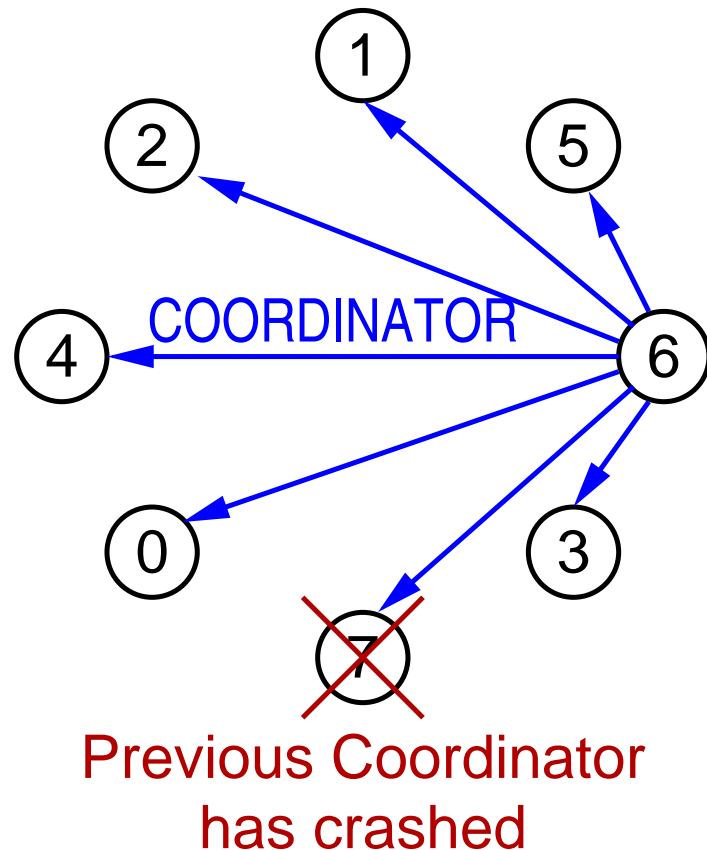
Processes 5 and 6 reply,
Process 4 terminates its election

Processes 5 and 6 simultaneously
hold an election

Process 6 replies to 5

Process 5 terminates its election

Bully Algorithm: Example



Process 4 holds an election

Processes 5 and 6 reply,
Process 4 terminates its election

Processes 5 and 6 simultaneously
hold an election

Process 6 replies to 5

Process 5 terminates its election

Noone replied to the election of
process 6, thus, this process wins
the election and communicates the
result to all others



A Ring Algorithm

- ➔ Assumption: processes form a logical ring, i.e. each process knows its successors in the ring
- ➔ Messages are sent along the ring as follows:
 - ➔ a process tries to send the message to its direct successor
 - ➔ if this process is not active, the message will be sent to the next process in the ring, etc.
- ➔ ELECTION messages contain a list of process IDs

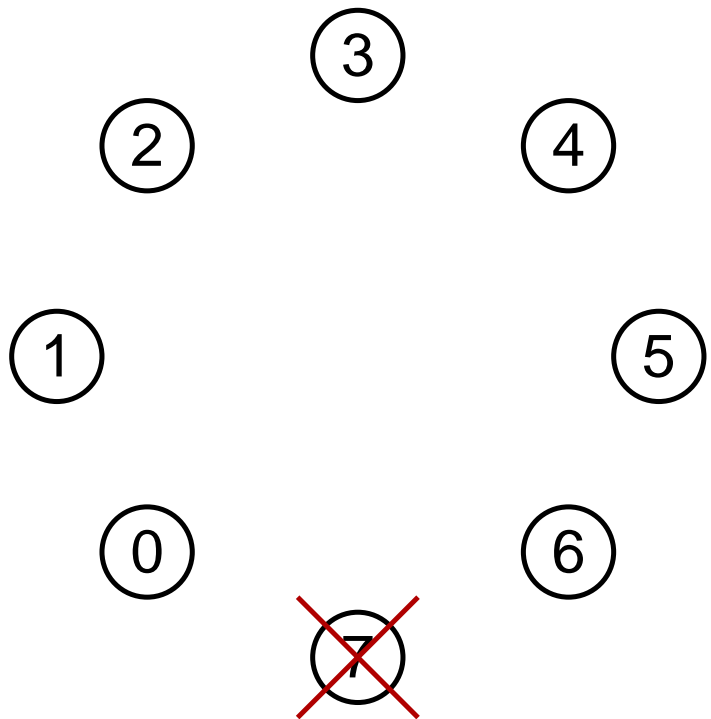


A Ring Algorithm ...

- ➔ A process that initiates the election sends an ELECTION message with its own ID along the ring
- ➔ When an ELECTION message is received by a process:
 - ➔ if its own ID is not in the list of IDs:
 - ➔ append the own ID to the list
 - ➔ continue sending message along the ring
 - ➔ else (message came back to the initiator):
 - ➔ determine highest ID in the list
 - ➔ send this ID in a COORDINATOR message along the ring



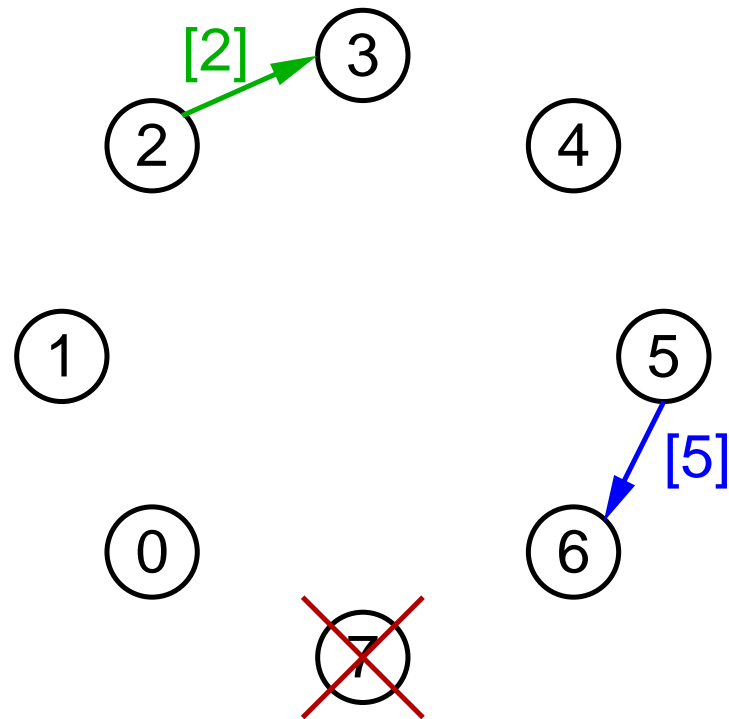
Ring Algorithm: Example



Previous coordinator
has crashed



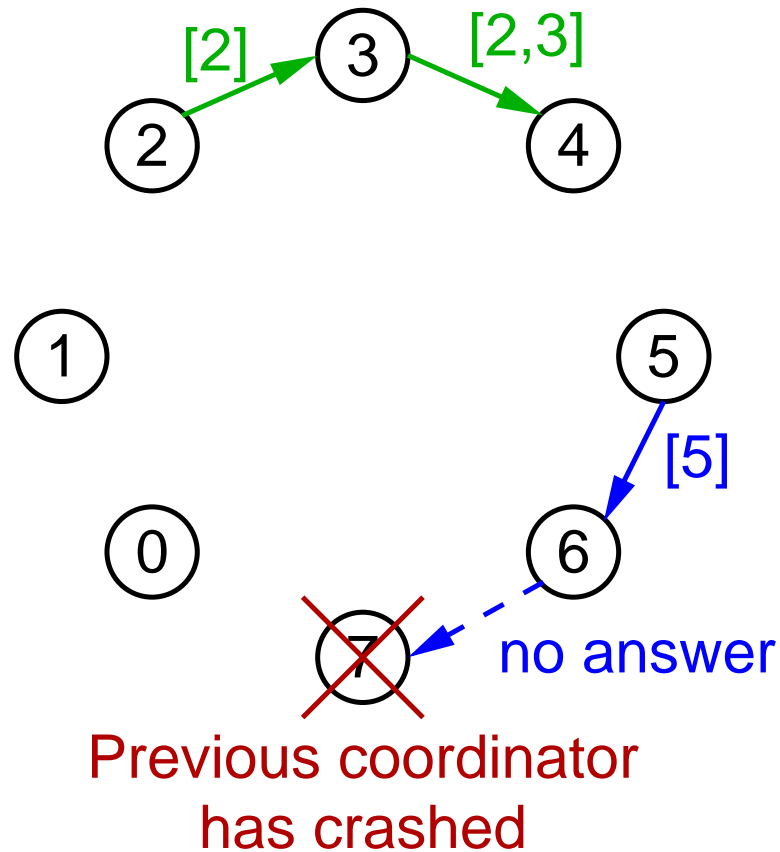
Ring Algorithm: Example



Processes 2 and 5 concurrently initiate an election



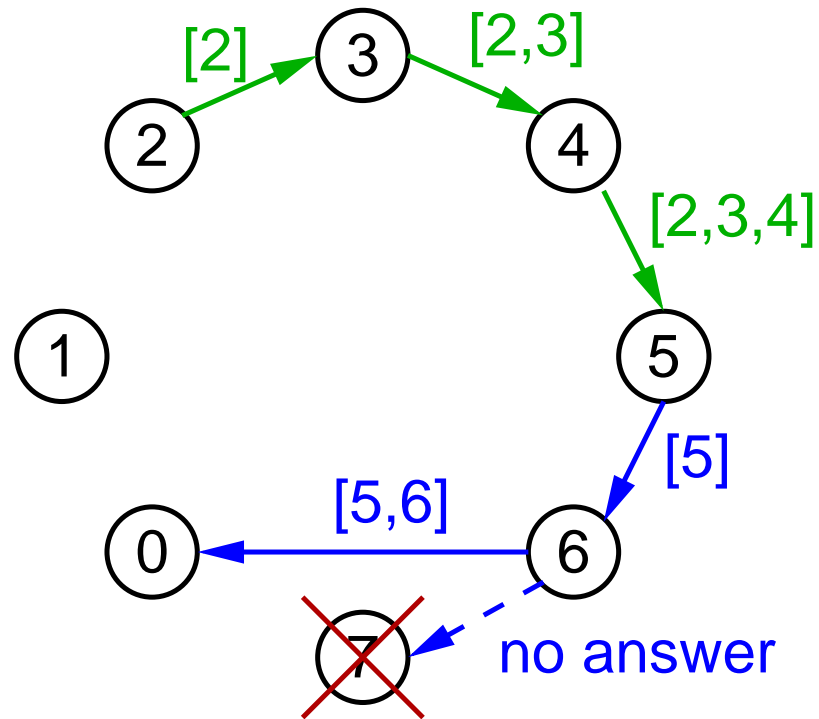
Ring Algorithm: Example



Processes 2 and 5 concurrently initiate an election



Ring Algorithm: Example

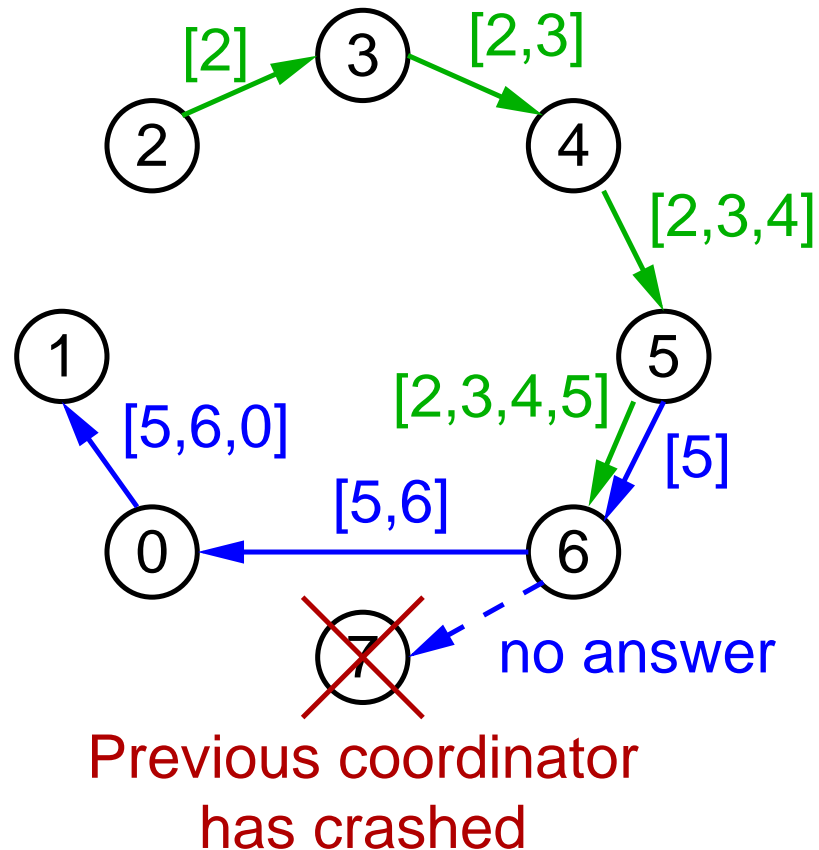


Previous coordinator
has crashed

Processes 2 and 5 concurrently
initiate an election



Ring Algorithm: Example



Processes 2 and 5 concurrently initiate an election

Eventually both processes get their ELECTION messages back and send a COORDINATOR message (with identical contents!)



8.2 Mutual Exclusion

- ➔ Here mainly: use / allocation of exclusive resources
- ➔ Requirements:
 - ➔ safety: the resource is not used concurrently by more than one process
 - ➔ liveness: any process that requests the resource will eventually get it
 - ➔ fairness: access to resources in 'FIFO' order
- ➔ Solution approaches:
 - ➔ centralized server
 - ➔ distributed algorithm with Lamport clock
 - ➔ token ring algorithm



Centralized Server

- ➔ An special coordinator process manages the resource and a queue for waiting processes
 - ➔ determined e.g. via an election algorithm
- ➔ Resource is requested by sending a message to the coordinator
 - ➔ if resource is free: coordinator answers with OK
 - ➔ otherwise: coordinator does not answer
 - ➔ requesting process is blocked (waiting for reply)
- ➔ Resource is released by sending a message to the coordinator
 - ➔ if processes wait: coordinator sends an OK to one of them
- ➔ Problem: processes cannot detect failure of the coordinator
 - ➔ this could be done using negative replies and polling

A Distributed Algorithm (Ricart / Agrawala)

- ➔ Idea: a process that wants to have a resource asks all other processes for their OK
 - ➔ a process replies with OK, if
 - ➔ it does not want the resource, or
 - ➔ it wants the resource, but the other process has requested it “earlier”
- ➔ Requires **total** order of request events
 - ➔ order must be consistent with causality
 - ➔ realizable e.g. via a time stamp (Lamport time, process ID) with lexicographic order
 - ➔ in the example of slide 195 this results in the event order:
b, c, a, e, g, d, j, f, l, h, i, k



A Distributed Algorithm (Ricart / Agrawala) ...

- ➔ To request a resource, a process sends the following message to all other processes:
 - ➔ resource ID
 - ➔ time stamp T of the request
 - ➔ pair: (current Lamport time, own process ID)(the message must be delivered reliably)
- ➔ The process then waits until it receives an OK message from all other processes
- ➔ After that it can use the resource (exclusively)



A Distributed Algorithm (Ricart / Agrawala) ...

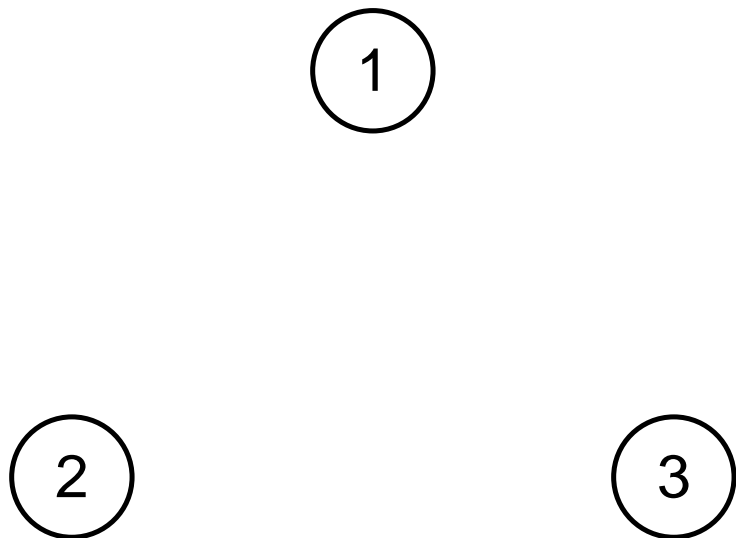
- ➔ Each process responds to request messages as follows:
 - ➔ resource is not used and not requested by the process:
 - ➔ return OK message
 - ➔ resource is used by the process:
 - ➔ do not send a reply
 - ➔ put the request in a queue
 - ➔ Resource id not used, but requested by the process:
 - ➔ if $T(\text{incoming message}) < T(\text{own request})$:
 - ➔ return OK message
 - ➔ or else:
 - ➔ do not send a reply
 - ➔ put the request in a queue



A Distributed Algorithm (Ricart / Agrawala) ...

- ➔ When a process releases the resource:
 - ➔ send an OK message to **all** processes in the queue
 - ➔ delete the queue

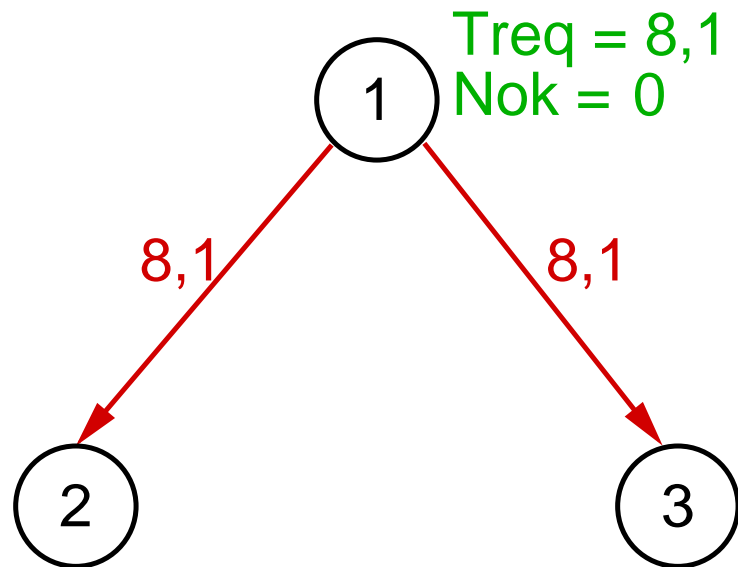
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want
the resource

Example for the Algorithm of Ricart / Agrawala

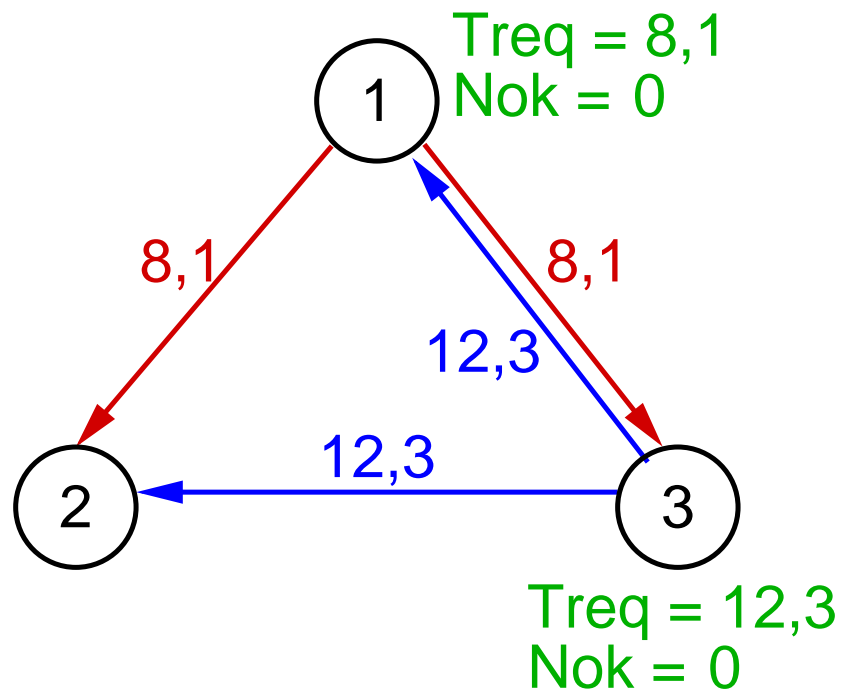
1. P1 sends request to all others



Both P1 and P3 want
the resource

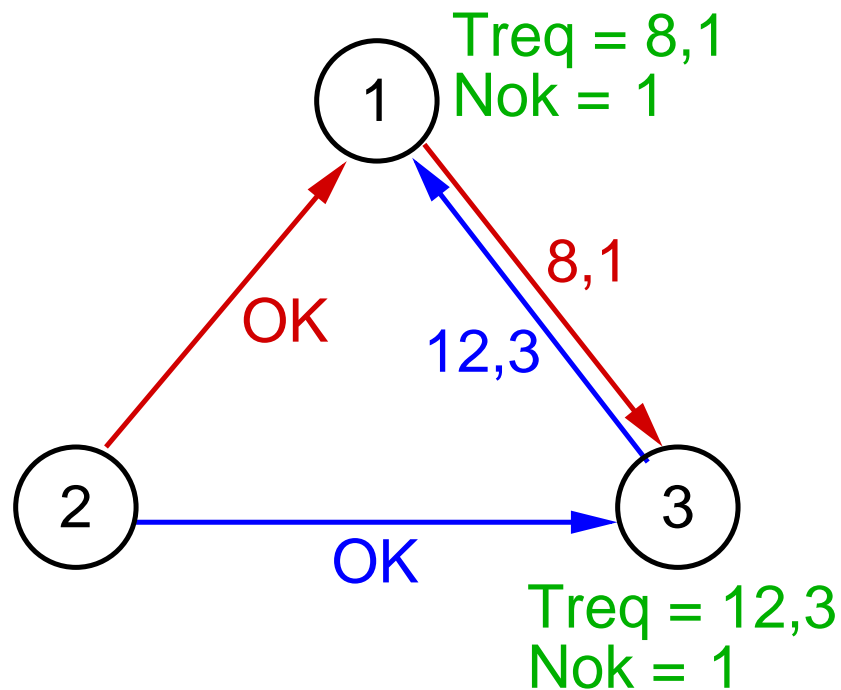
Example for the Algorithm of Ricart / Agrawala

1. P1 sends request to all others
2. P3 sends request to all others



Both P1 and P3 want
the resource

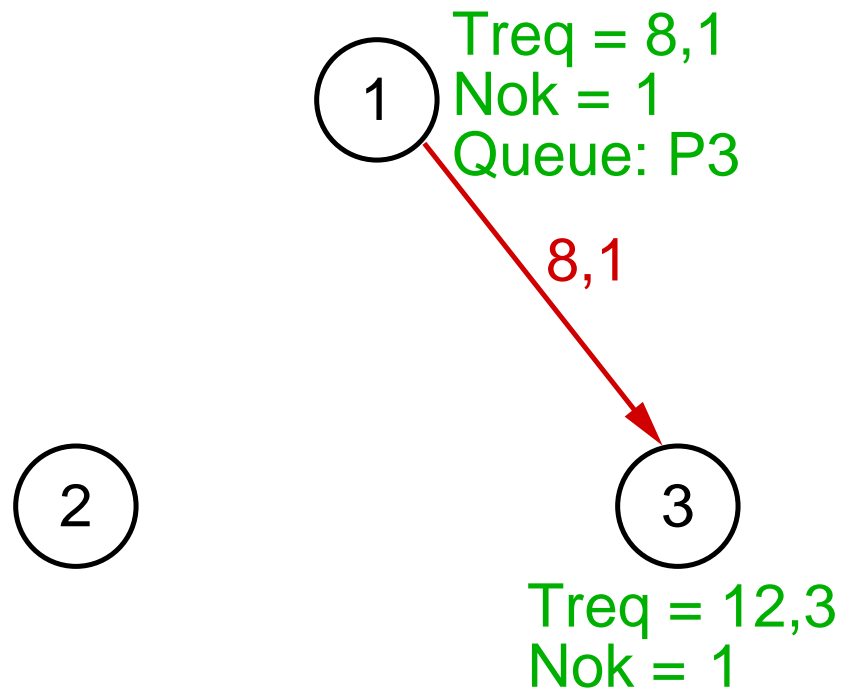
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want
the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3,
since it doesn't want the resource

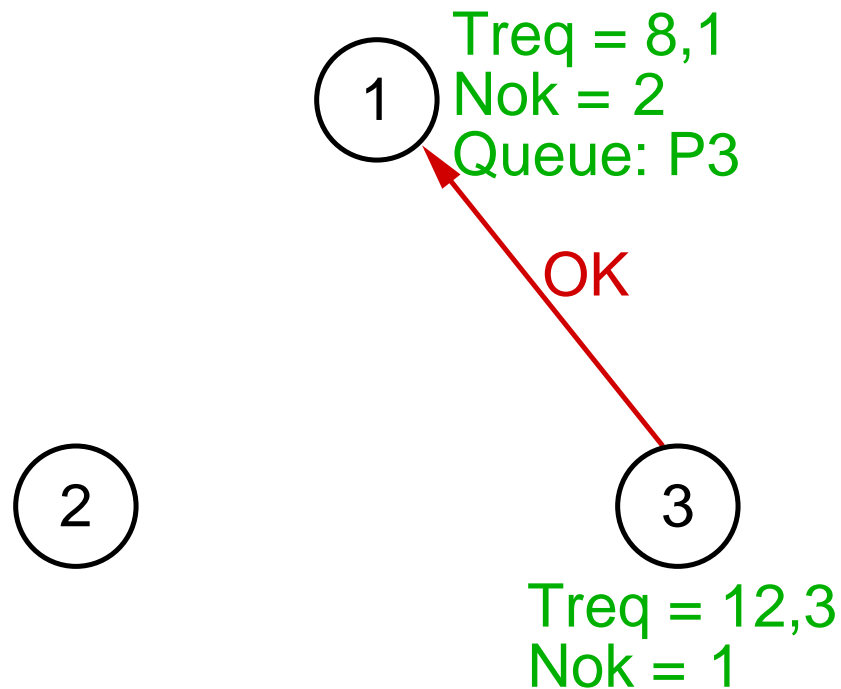
Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want
the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3,
since it doesn't want the resource
4. P1 doesn't send an OK to P3,
since $(12,3) > (8,1)$.
P1 adds P3 to its queue

Example for the Algorithm of Ricart / Agrawala



Both P1 and P3 want
the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3,
since it doesn't want the resource
4. P1 doesn't send an OK to P3,
since $(12,3) > (8,1)$.
P1 adds P3 to its queue
5. P3 sends OK to P1, since
 $(8,1) < (12,3)$

Example for the Algorithm of Ricart / Agrawala

P1 owns
the resource

1 $T_{req} = 8,1$
 $N_{ok} = 2$
Queue: P3

2

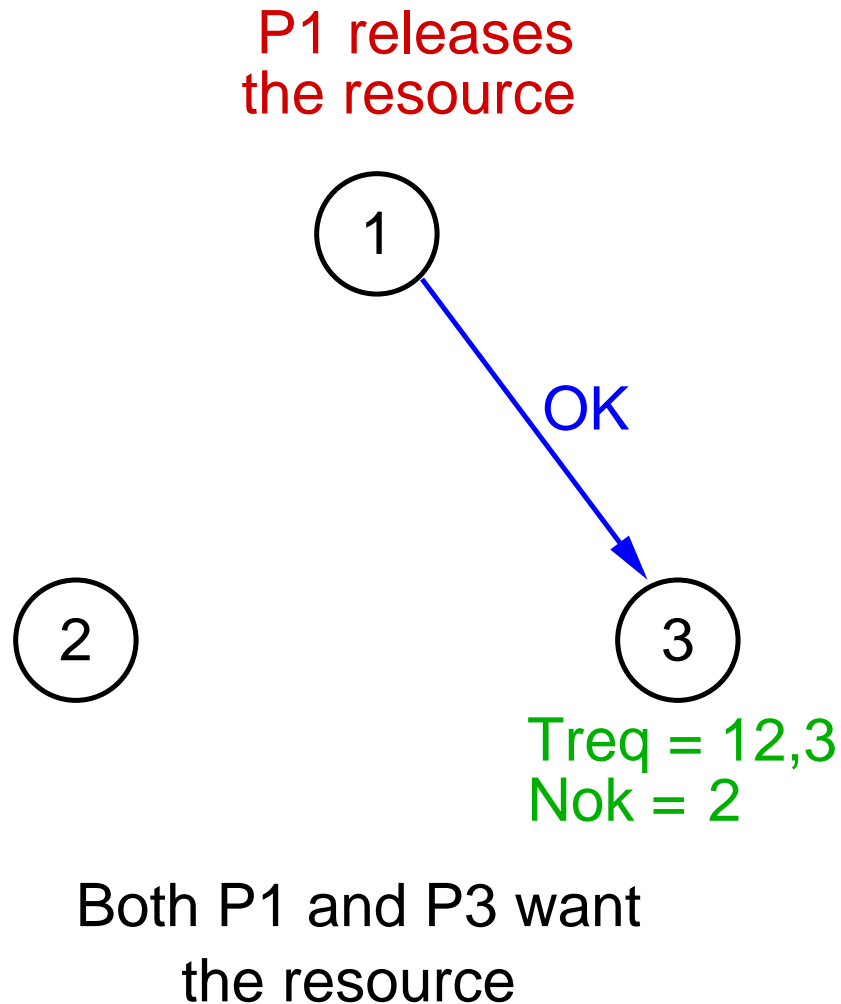
3

$T_{req} = 12,3$
 $N_{ok} = 1$

Both P1 and P3 want
the resource

1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3,
since it doesn't want the resource
4. P1 doesn't send an OK to P3,
since $(12,3) > (8,1)$.
P1 adds P3 to its queue
5. P3 sends OK to P1, since
 $(8,1) < (12,3)$
6. P1 received all OKs
and uses the resource

Example for the Algorithm of Ricart / Agrawala



1. P1 sends request to all others
2. P3 sends request to all others
3. P2 sends OK to P1 and P3, since it doesn't want the resource
4. P1 doesn't send an OK to P3, since $(12,3) > (8,1)$. P1 adds P3 to its queue
5. P3 sends OK to P1, since $(8,1) < (12,3)$
6. P1 received all OKs and uses the resource
7. P1 releases the resource and sends an OK to P3



A Token Ring Algorithm

- ➔ The processes form a logical ring
- ➔ A **token** circles in the ring
 - ➔ authorization for (exclusive) use of the resource
 - ➔ token is initially generated by one of the processes
- ➔ On arrival of the token: process checks whether it wants the resource
 - ➔ if so:
 - ➔ use the resource
 - ➔ after releasing the resource:
 - ➔ pass token to successor in the ring
 - ➔ else:
 - ➔ pass token immediately to successor in the ring



Comparison of algorithms

- ➔ Centralized server:
 - ➔ server is *single point of failure* and may be a performance bottleneck
 - ➔ clients cannot distinguish (without additional measures) between server failure and occupied resource
 - ➔ only little communication necessary
- ➔ Distributed algorithm:
 - ➔ failure of **any** node is problematic
 - ➔ any node can become a performance bottleneck
 - ➔ high communication effort
 - ➔ just a proof that a distributed, symmetrical algorithm is possible

Comparison of algorithms ...

- ➔ Token ring algorithm:
 - ➔ problem: loss of the token (detection, re-creation)
 - ➔ failure of nodes is problematic
 - ➔ communication, even if resource is not used

Algorithm	Messages per allocation	Delay before allocation	Problems
centralized	3	2	server failure
distributed	$2(n - 1)$	$2(n - 1)$	failure of any process
token ring	$1 \dots \infty$	$0 \dots n - 1$	lost token, failure of any process



8.3 Group Communication (Multicast)

- ➔ In distributed systems, communication with a **group** of processes (**multicast**) is often also important, e.g. for:
 - ➔ fault tolerance based on replicated services
 - ➔ service realized by group of servers
 - ➔ all servers receive and process the requests
 - ➔ finding of services (especially discovery / name services)
 - ➔ multicast is a possible approach for this
 - ➔ better performance through replicated data
 - ➔ changes must be sent to all copies
 - ➔ sending event notifications
 - ➔ all subscribers receive the event



Questions / Problems

- ➔ Addressing the recipients
 - ➔ explicit list of all recipients
 - ➔ addressing a process group
 - ➔ static / dynamic groups
- ➔ Reliability
 - ➔ reasonable guarantees that messages will reach their recipients
- ➔ Order
 - ➔ adequate guarantees as to the order in which multicast messages arrive at the various recipients



Reliability

➔ Unreliable multicast:

- ➔ some processes may not receive the message (e.g. due to packet loss)

➔ Reliable multicast:

- ➔ apart from network and process failures, the message is delivered to all processes in the group

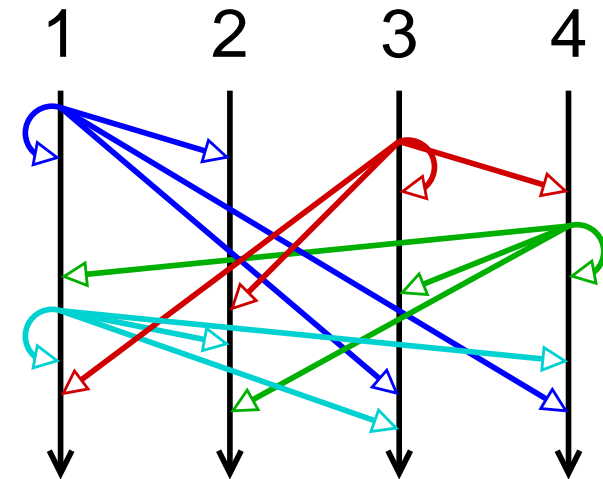
➔ Atomic multicast:

- ➔ the message is (under all circumstances) received either by **all** processes of the group or by **none** of them
- ➔ required if all processes in the group must be kept consistent (e.g., operations on replicated data)

Order

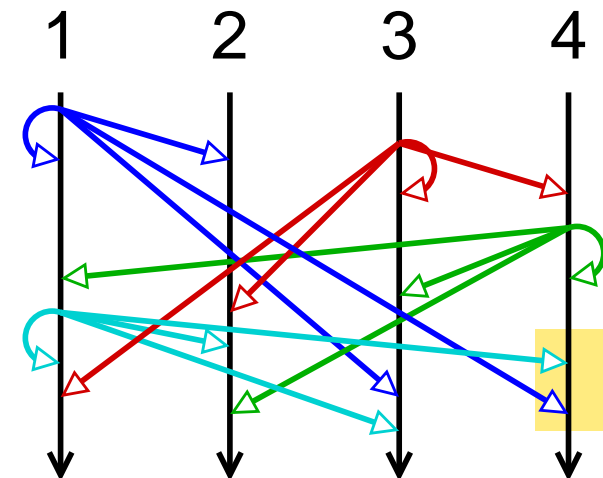
➔ Unordered

- ➔ receiving order is undefined and can be different in different processes



➔ FIFO order

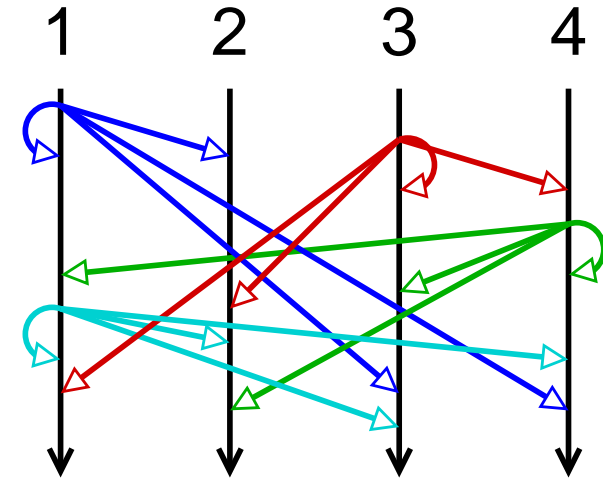
- ➔ messages from the **same** sender are received by all processes in FIFO order
- ➔ i.e. introduction of sequence numbers **local to the sender**



Order

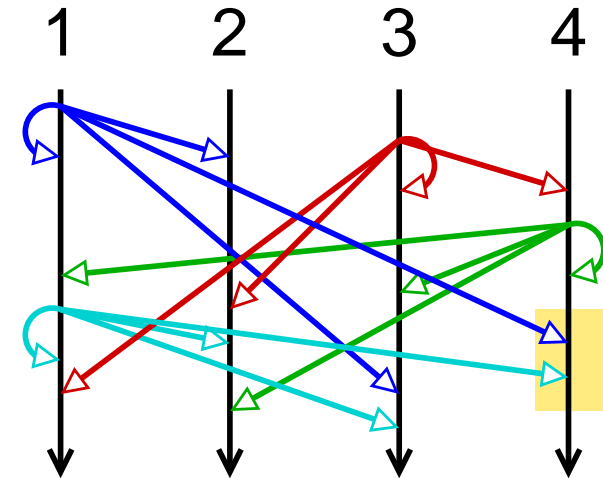
➔ Unordered

- ➔ receiving order is undefined and can be different in different processes



➔ FIFO order

- ➔ messages from the **same** sender are received by all processes in FIFO order
- ➔ i.e. introduction of sequence numbers **local to the sender**



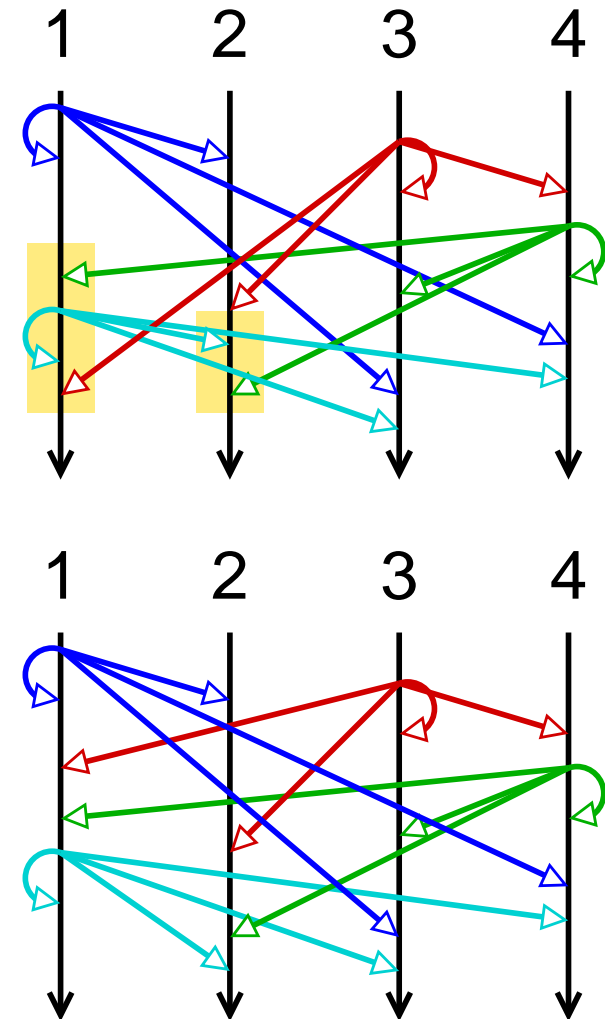
Order ...

→ Causal order

- if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
- i.e. introduction of vector time stamps

→ Total order

- **all** messages are received by all processes in the same order
- i.e. introduction of **global** sequence numbers



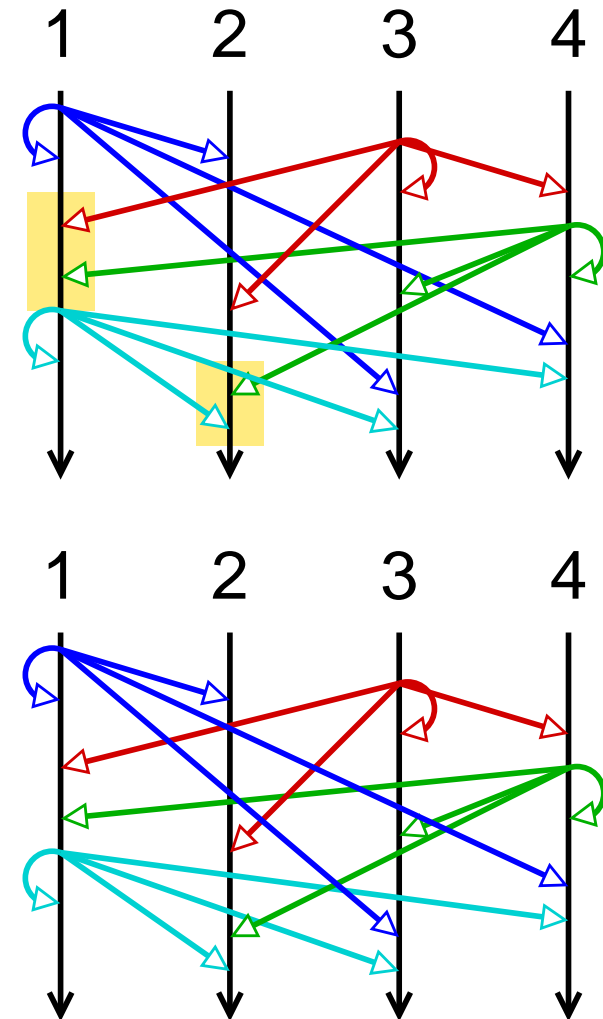
Order ...

➔ Causal order

- ➔ if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
- ➔ i.e. introduction of vector time stamps

➔ Total order

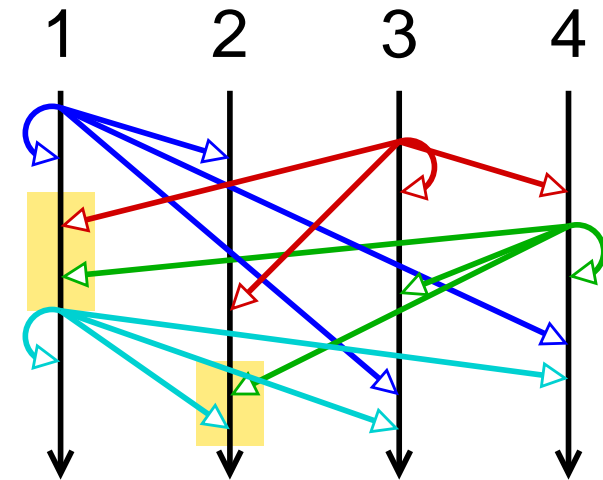
- ➔ **all** messages are received by all processes in the same order
- ➔ i.e. introduction of **global** sequence numbers



Order ...

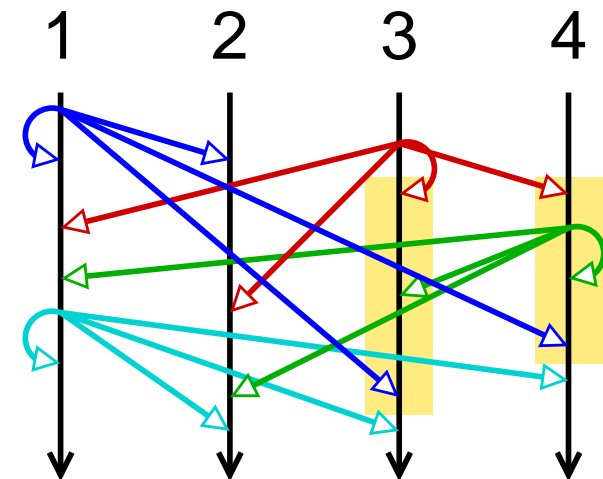
→ Causal order

- if message m' can causally depend on m ($m \rightarrow m'$), then all processes receive m before m'
- i.e. introduction of vector time stamps



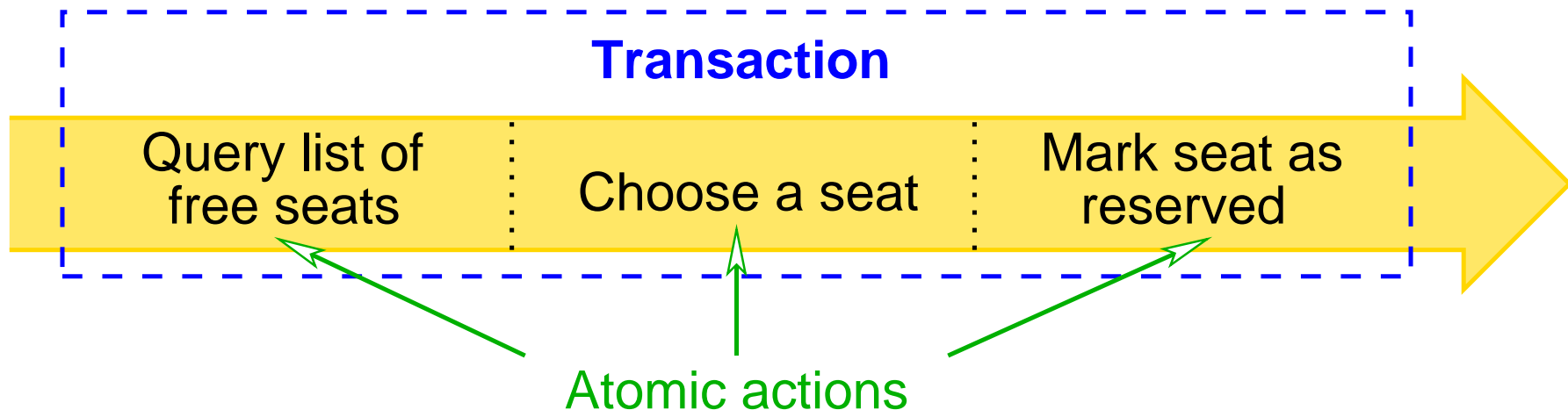
→ Total order

- **all** messages are received by all processes in the same order
- i.e. introduction of **global** sequence numbers



8.4 Transactions

- ➔ Combining a sequence of atomic actions into a single unit
 - ➔ atomic actions: read, change, write data
- ➔ Example: seat reservation



- ➔ Used not only in database systems



Properties of Transactions: ACID

➔ Atomicity

- ➔ all-or-nothing principle: either all atomic actions are executed (correctly) or none at all

➔ Consistency

- ➔ a transaction always transfers a consistent state back to consistent state

➔ Isolation

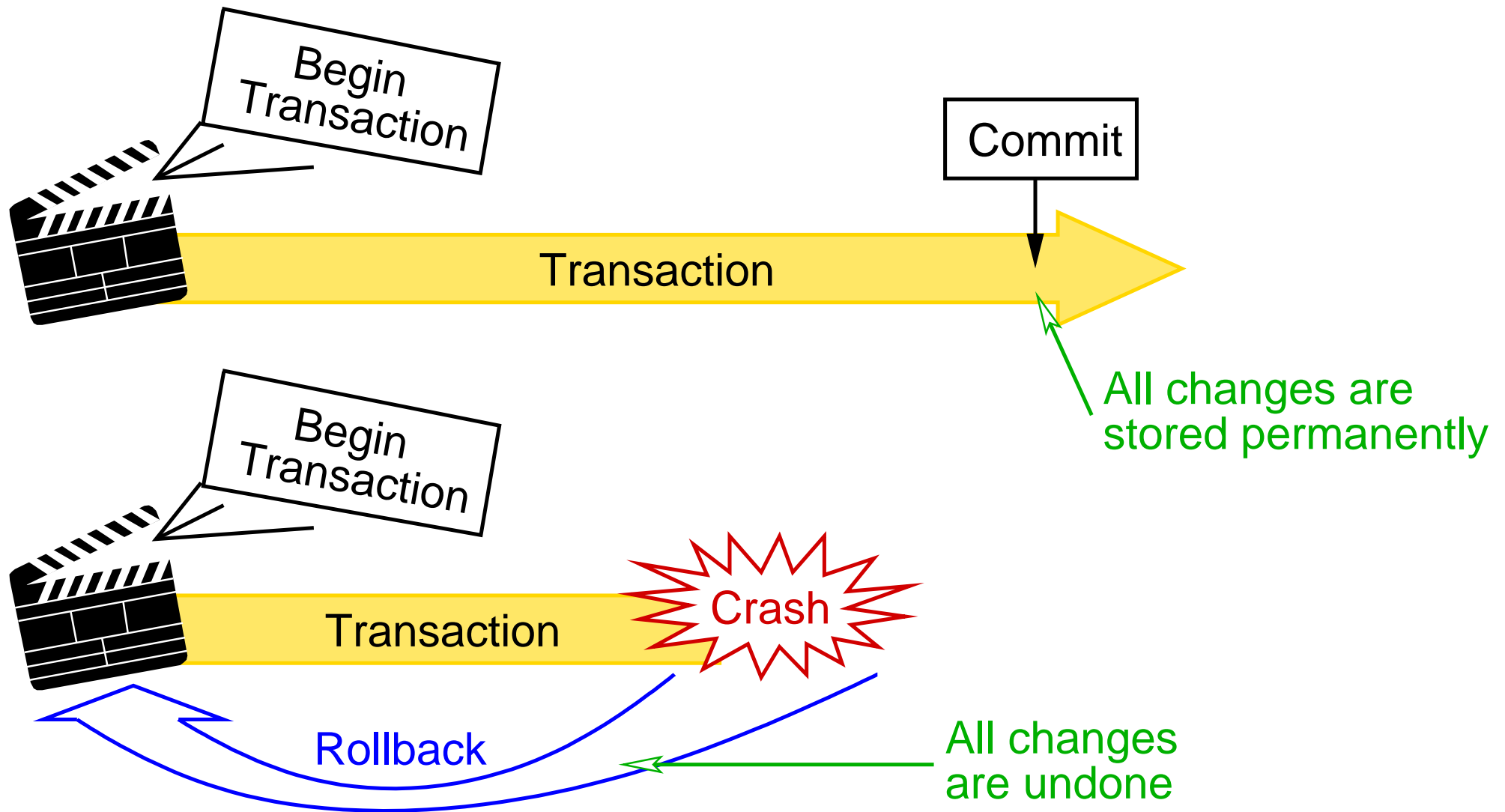
- ➔ concurrent transactions do not affect each other; the result is the same as with sequential execution

➔ Durability

- ➔ at the (successful) end of the transaction all changes are stored permanently

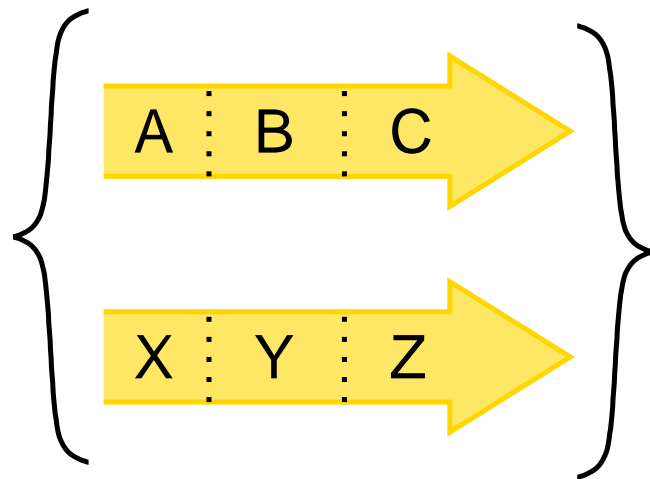


Atomicity

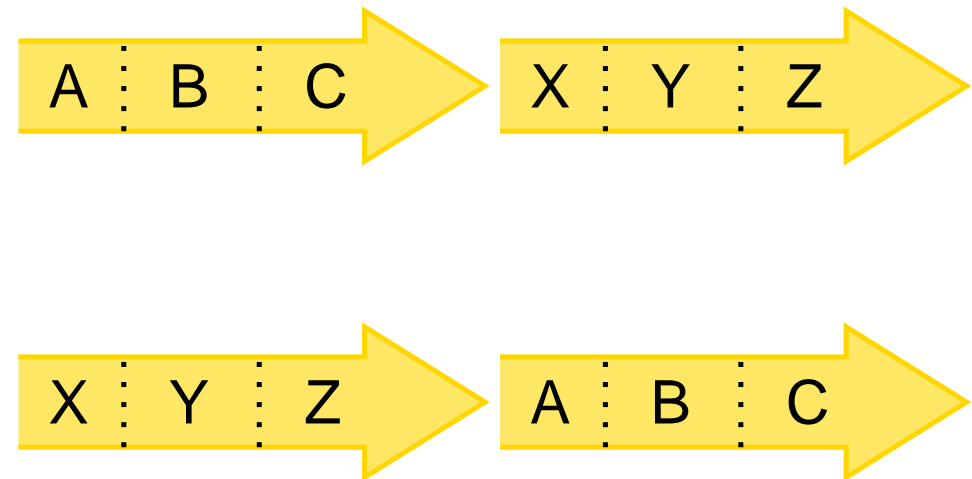


Isolation

Two concurrent transactions



Permitted serializations



➔ The result of the concurrent transactions corresponds to one of the two serializations



Isolation Levels

- ➔ Complete isolation of (database) transactions often is too restrictive / too little performant
- ➔ Therefore: SQL99 standard defines four isolation levels
- ➔ Goal: avoidance of unwanted phenomena
 - ➔ **dirty reads:** a transaction can read data of another transaction before they have been committed
 - ➔ **unrepeatable reads:** when reading repeatedly, a transaction can see committed changes of other transactions
 - ➔ **phantom reads:** when reading repeatedly, a transaction can see that other transactions have added or deleted records

Isolation Level According to ANSI/ISO-SQL99

Isolation level \ Phenomenon	Dirty Reads	Unrepeatable Reads	Phantom Reads
Read Uncommitted	possible	possible	possible
Read Committed	not possible	possible	possible
Repeatable Read	not possible	not possible	possible
Serializable	not possible	not possible	not possible

➔ *Serializable* corresponds to complete isolation

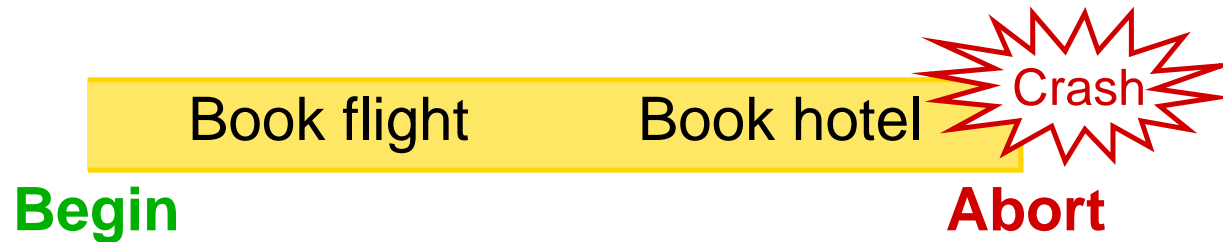


Nested Transactions

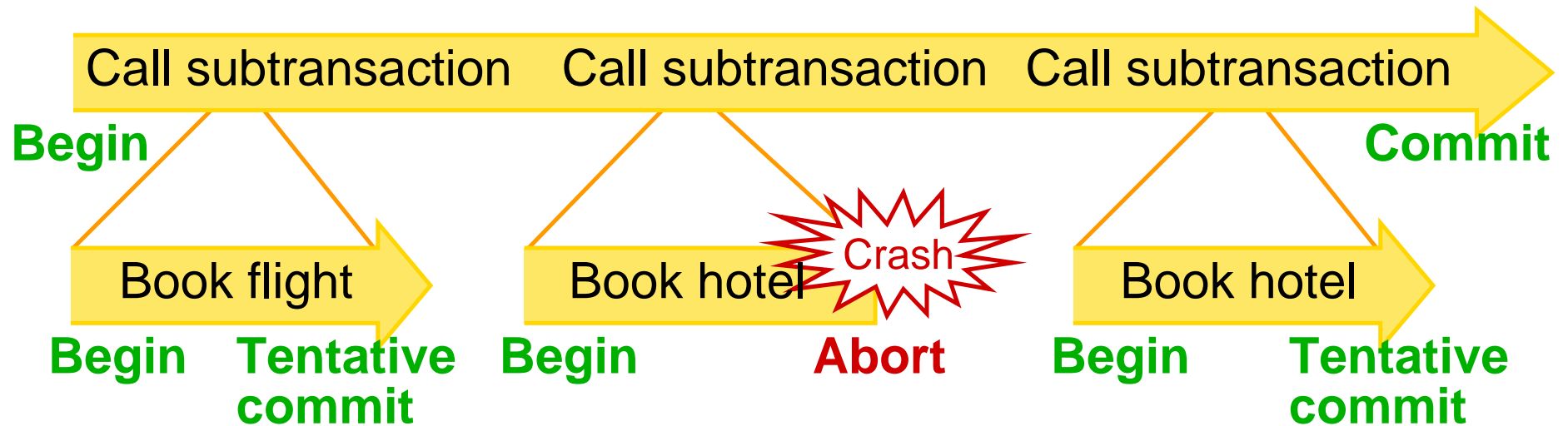
- ➔ Within a transaction, several subtransactions take place
- ➔ Higher-level transaction can run successfully to completion, even if subtransaction was terminated with an error
- ➔ Abort of the higher-level transaction results in aborting all subtransactions
- ➔ Example: booking of flight and hotel
 - ➔ booking of the flight should be maintained, even if hotel booking (in the first attempt) fails
- ➔ Nested transactions are supported by only a few transaction services



Flat transaction



Nested transactions





Distributed Systems

Winter Term 2025/26

11.12.2025

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

Stand: January 8, 2026

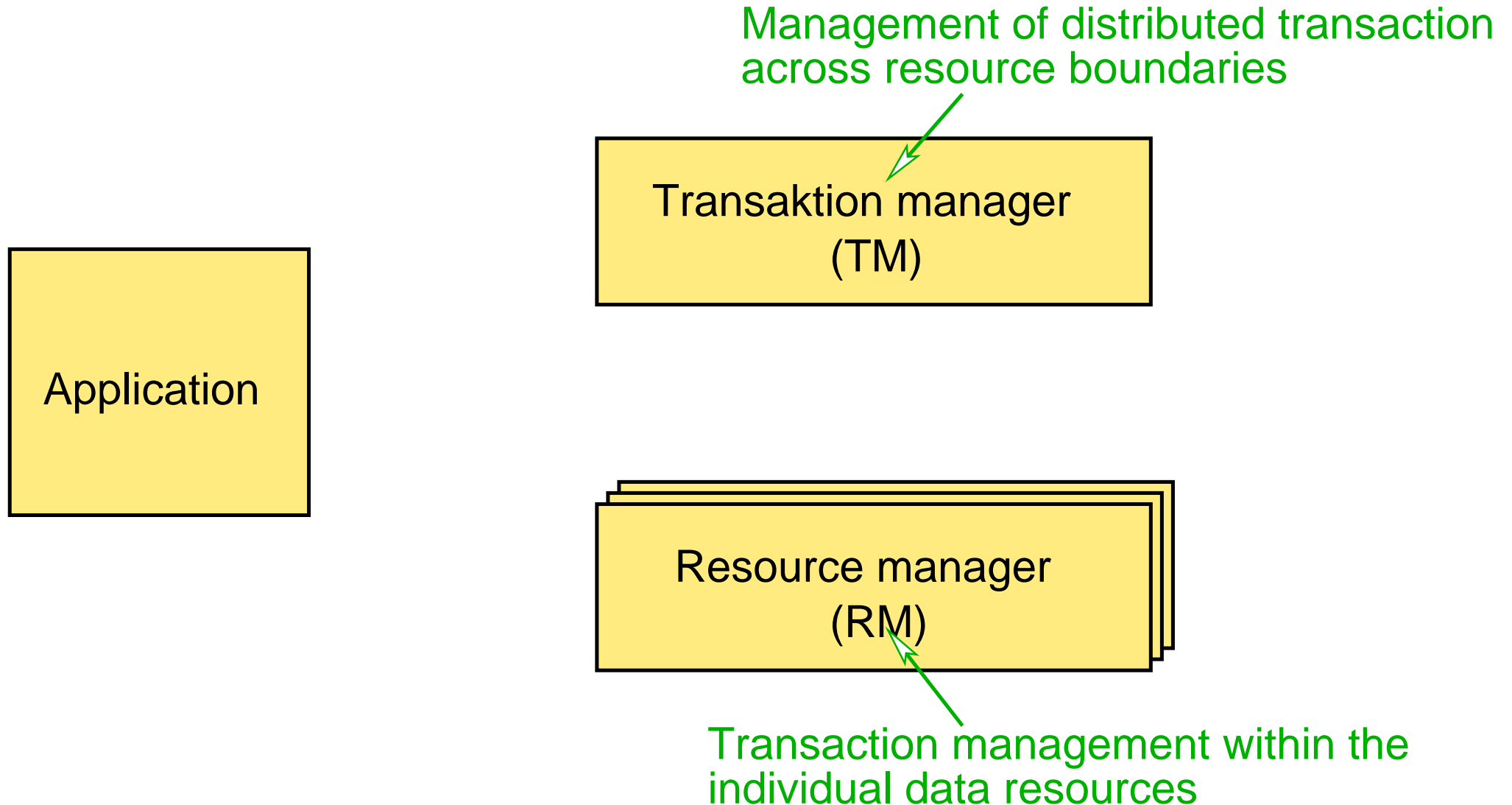


Distributed Transactions

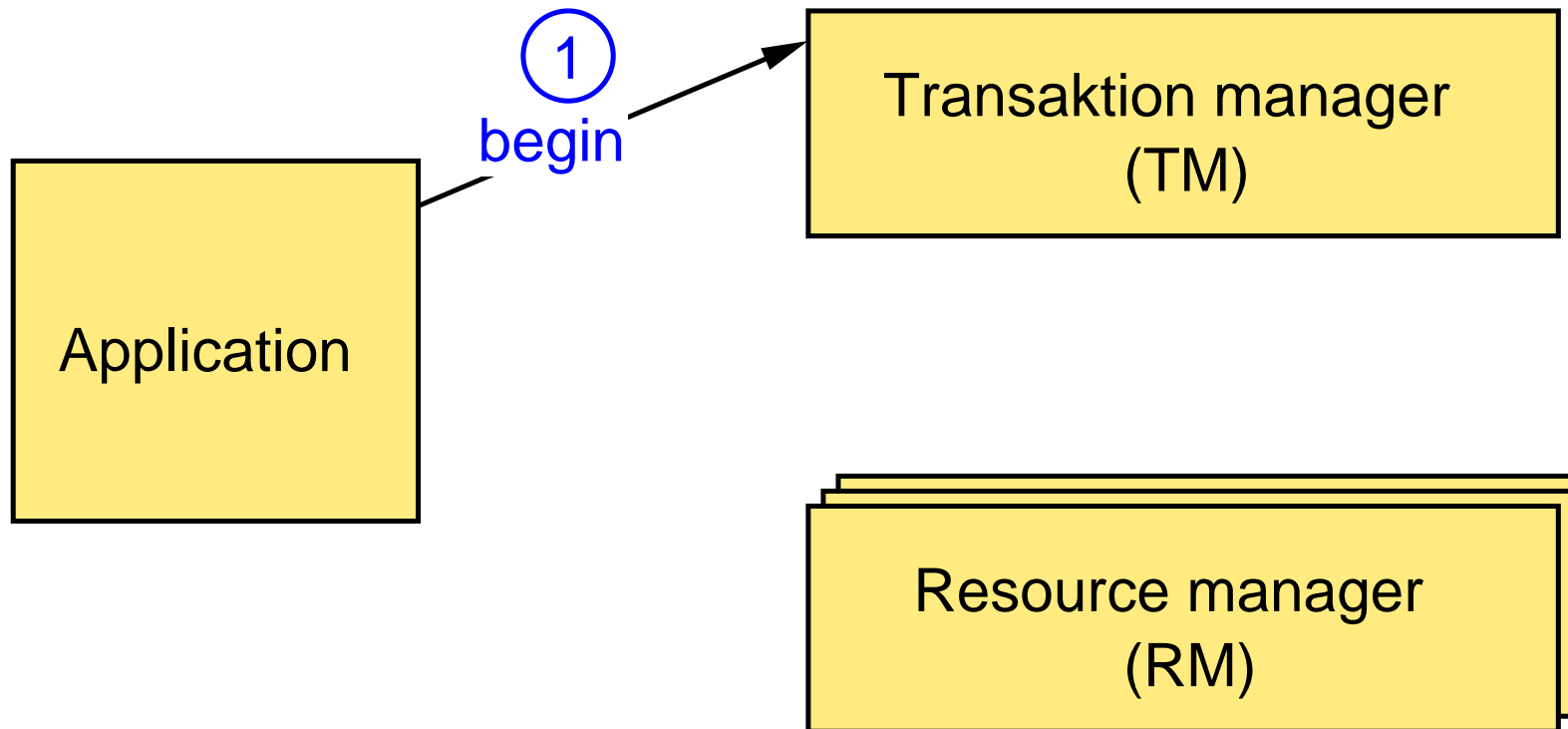
- ➔ So far: data is stored at exactly one location
- ➔ Distributed transactions: data is stored distributed
- ➔ Realization of transactions on the individual data resources (databases) is no longer sufficient
 - ➔ distributed transaction management becomes necessary
- ➔ There is a generally accepted Open Group model for the management of distributed transactions
 - ➔ is implemented by most transaction services
 - ➔ most important feature: 2-Phase-Commit



Model for Managing Distributed Transactions

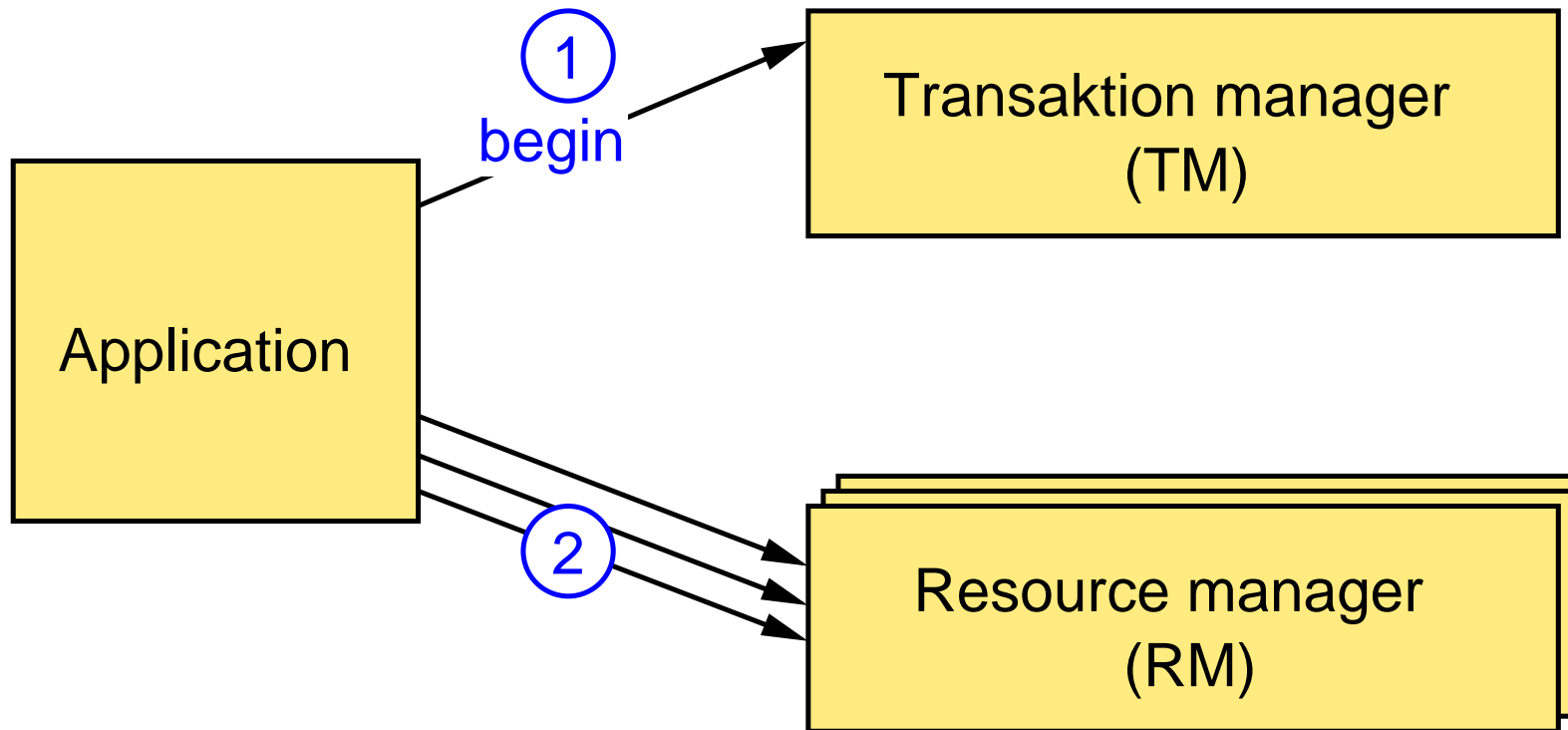


Model for Managing Distributed Transactions



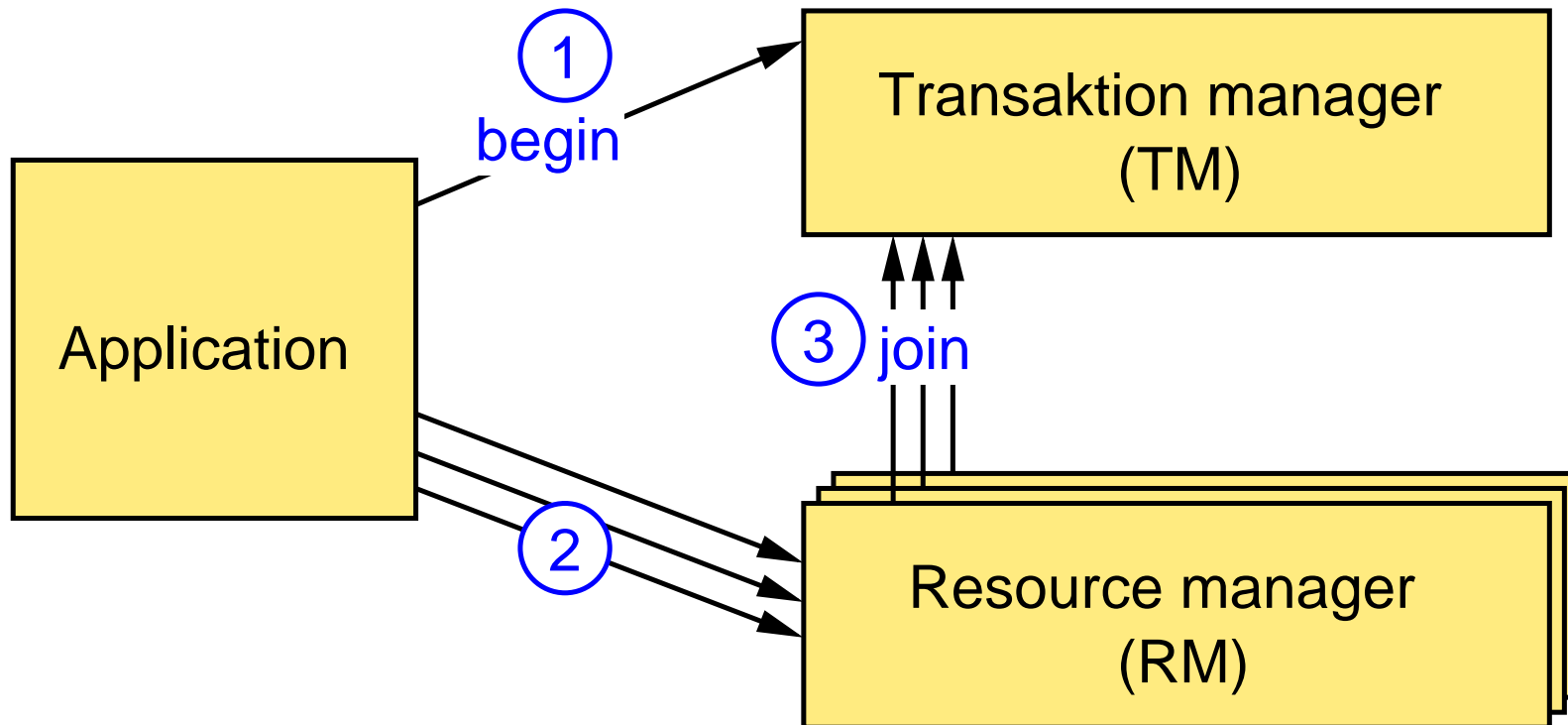
1. Application requests start of a new transaction.
TM internally initializes a new transaction.

Model for Managing Distributed Transactions



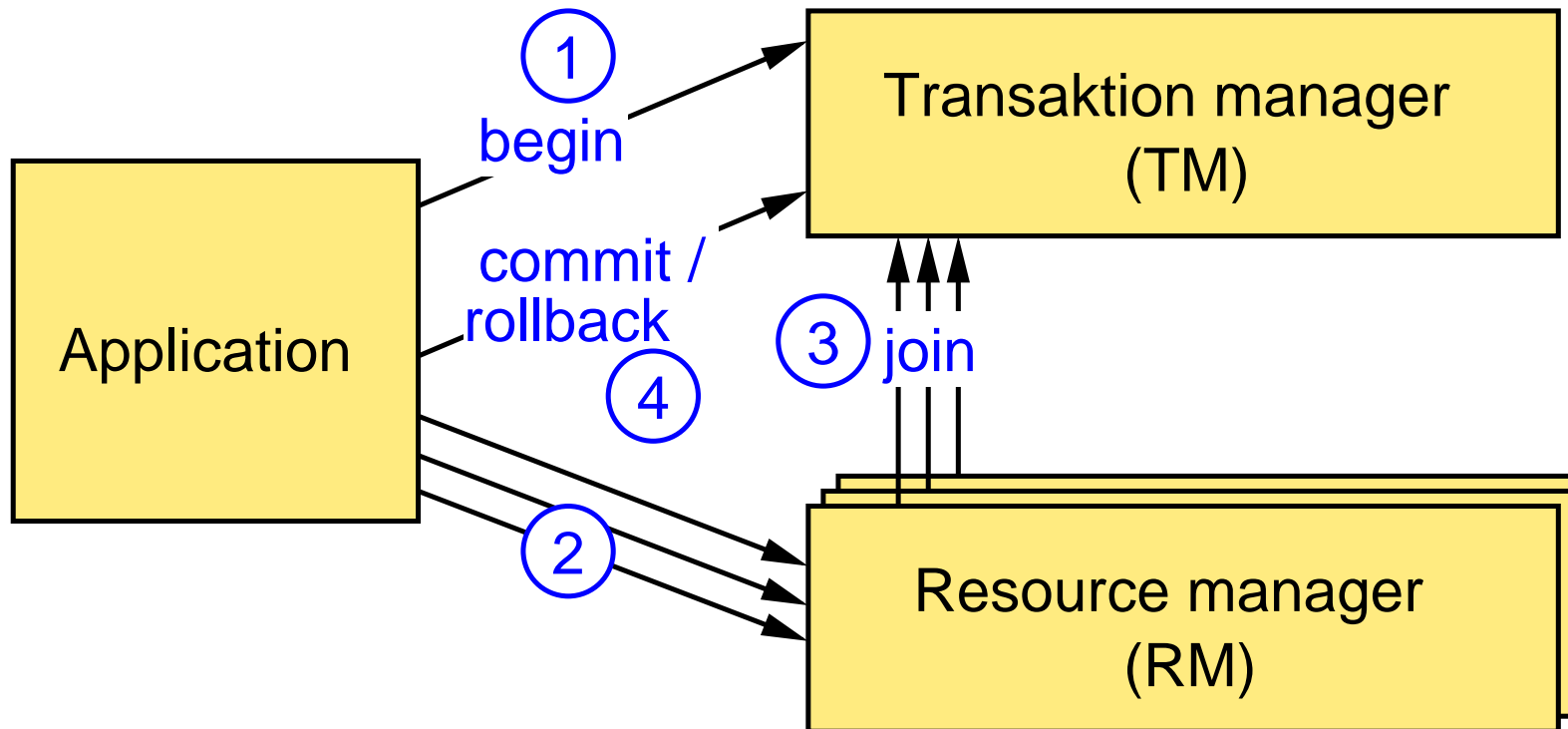
2. Transaction is active.
Application can use the resources.

Model for Managing Distributed Transactions



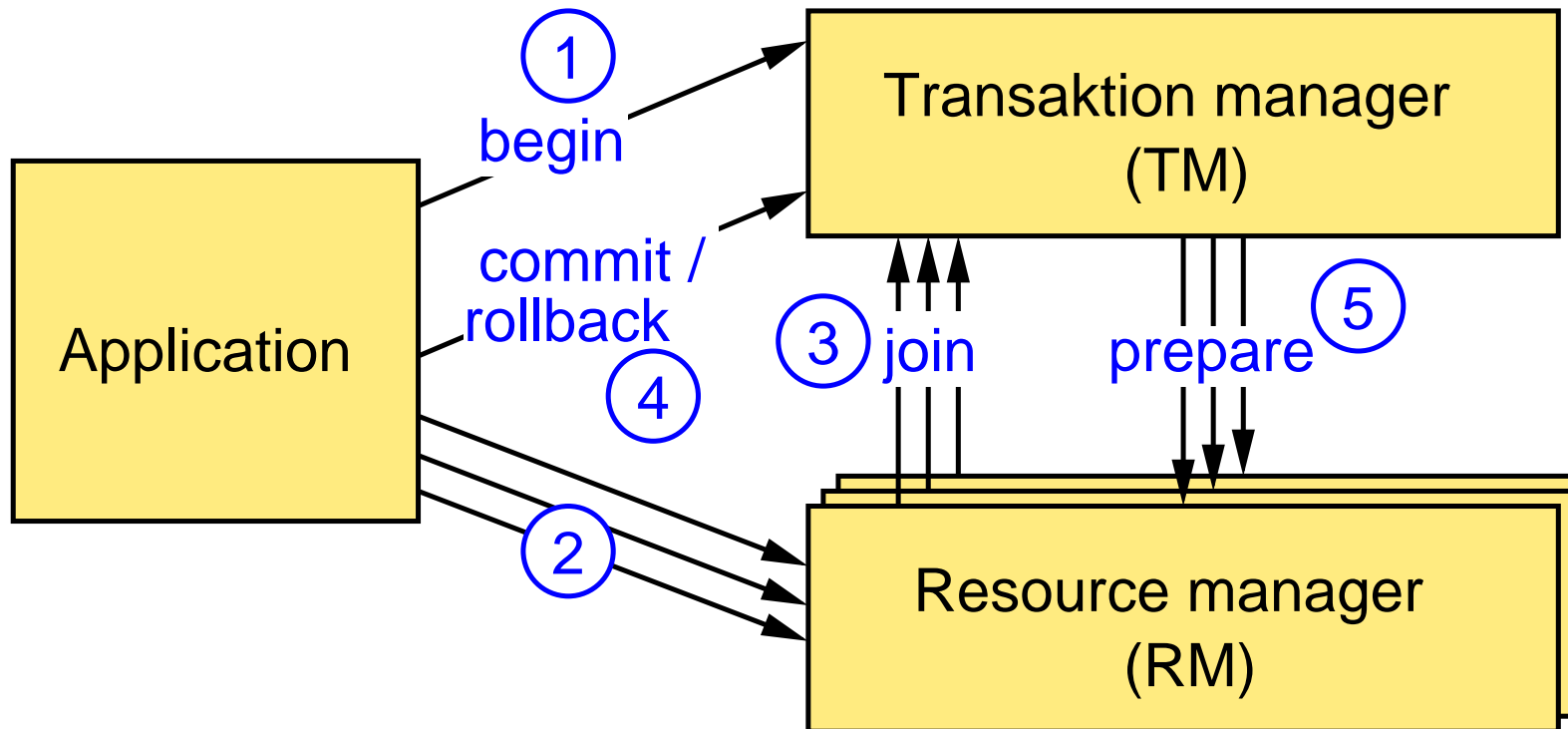
3. Each RM used by the application registers with the TM for the transaction.

Model for Managing Distributed Transactions



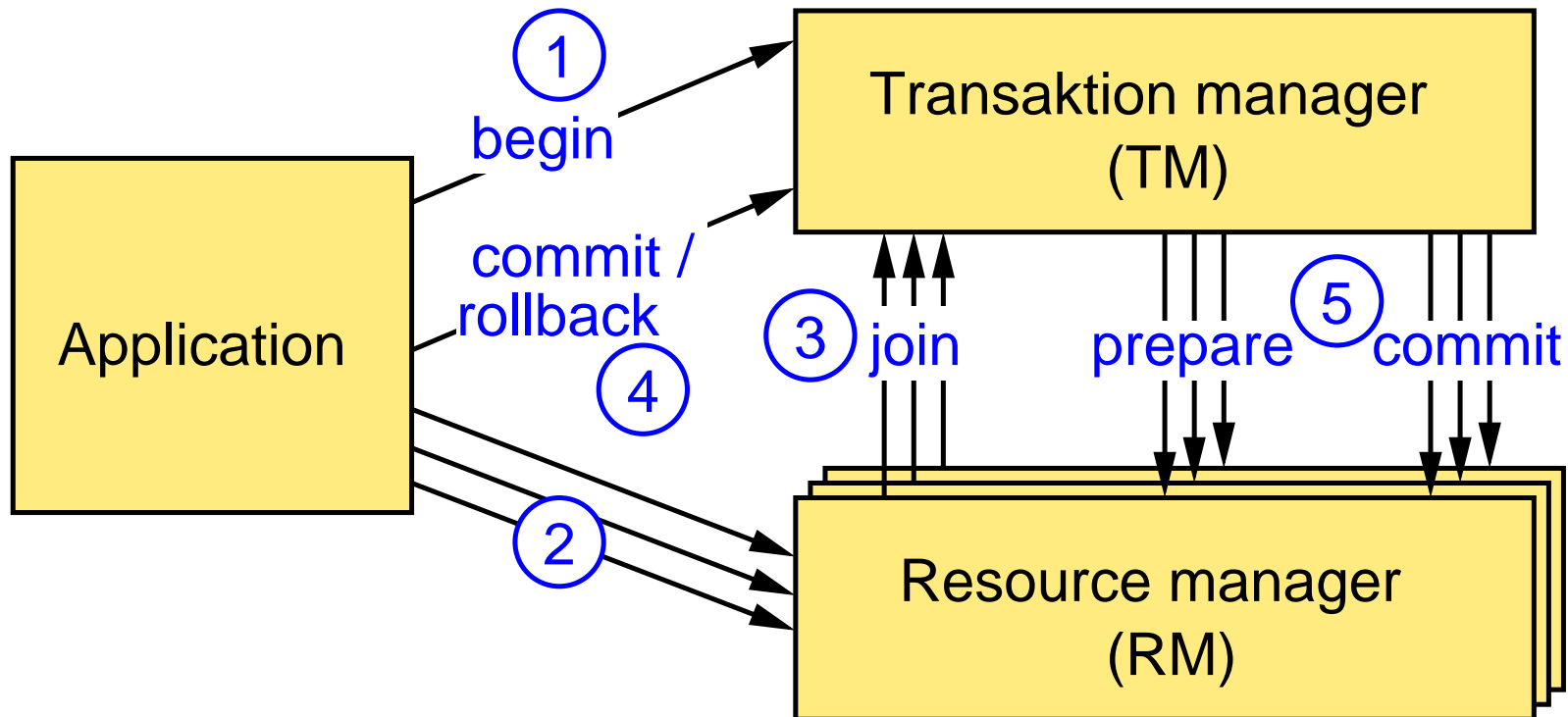
4. Application requires to commit or abort the transaction.

Model for Managing Distributed Transactions



5. TM requires RM to commit the changes:
2-phase commit

Model for Managing Distributed Transactions



5. TM requires RM to commit the changes:
2-phase commit



2-Phase Commit

- ➔ Phase 1 (voting phase)
 - ➔ TM asks all involved RM, if the commit would be successful (“*prepare*”)
 - ➔ each RM that answers “yes” prepares for the commit
- ➔ Phase 2 (finalization)
 - ➔ if all RMs answered with “yes”:
 - ➔ TM sends *commit* command to all RMs
 - ➔ RM ultimately commits the data and sends an acknowledgement to TM
 - ➔ else:
 - ➔ TM sends an *abort* command to all RMs
 - ➔ RMs acknowledge the receipt of *commit/abort*



Distributed Systems

Winter Term 2025/26

9 Replication and Consistency



Contents

- ➔ Introduction, motivation
- ➔ Distribution protocols
- ➔ Data centric consistency
- ➔ (Strong) eventual consistency
- ➔ Client centric consistency

Literature

- ➔ Tanenbaum, van Steen: Kap. 6



9.1 Introduction and Motivation

- ➔ **Replication**: several (identical) copies of data objects are stored in the distributed system
 - ➔ processes can access an arbitrary copy
- ➔ Reasons for the replication:
 - ➔ increase in availability and reliability
 - ➔ if a replica is not available, use another one
 - ➔ reading multiple replicas with majority vote
 - ➔ increase in read performance
 - ➔ for large systems: concurrent read access can be serviced by different replicas
 - ➔ with systems spread over a large area: access request is sent to a replica in the vicinity



Central Problem of Replication: Consistency

- ➔ When data is changed, **all** replicas must be kept consistent
- ➔ e.g.: send all updates to all replicas via totally ordered atomic multicast
 - ➔ high overhead when frequent updates occur
 - ➔ in some replicas these may actually never be read
 - ➔ totally ordered atomic multicast is very expensive with many / widely dispersed replicas
- ➔ Strict consistency maintenance of replicas always deteriorates performance and scalability
- ➔ Solution: weakened consistency requirements
 - ➔ often only very weak demands, e.g. News, Web, ...



Consistency Models

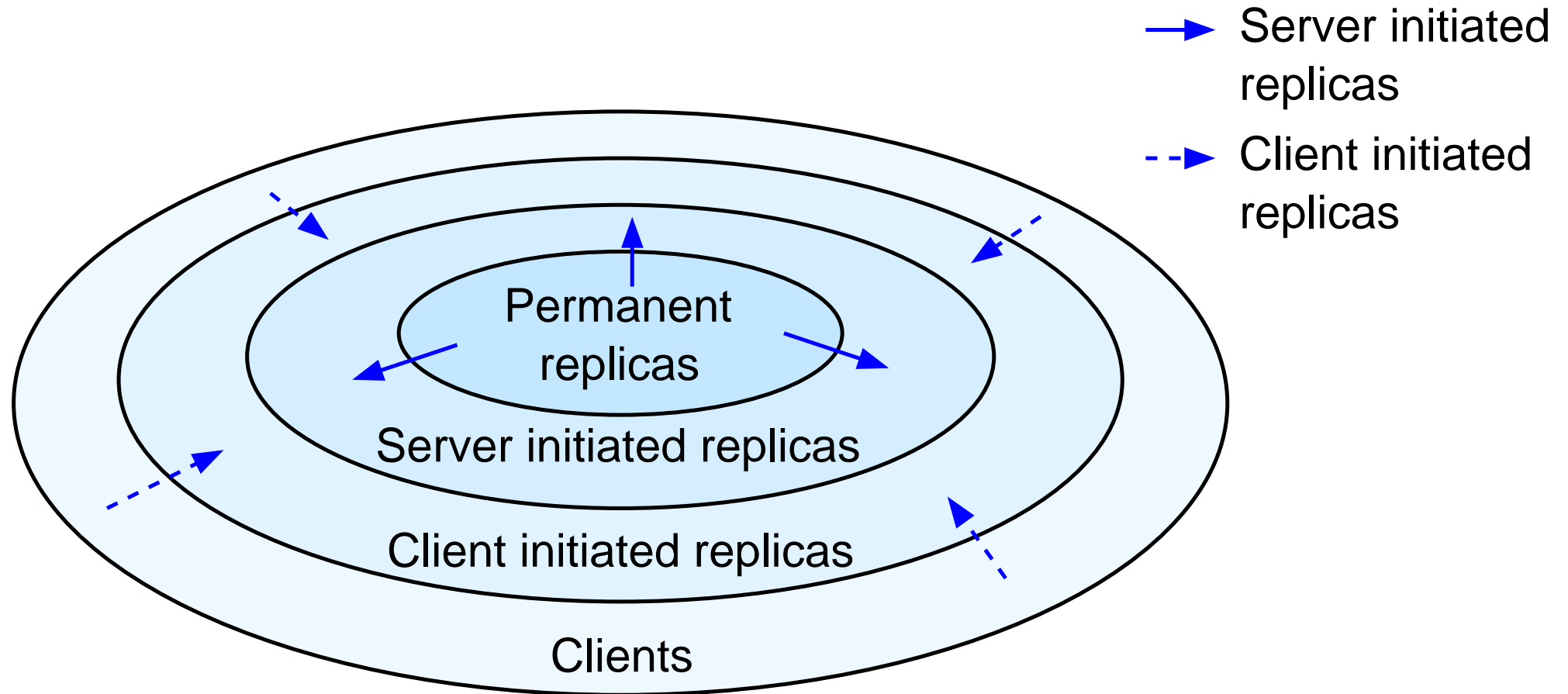
- ➔ A consistency model determines the order in which the write operations (updates) of the processes are “seen” by the other processes
- ➔ Intuitive expectation: a read operation always returns the result of the last write operation (**strict consistency**)
 - ➔ problem: there is no global time
 - ➔ pointless to speak of the “last” write operation
 - ➔ therefore: other consistency models necessary
- ➔ **Data-centric consistency models**: view of the data storage
- ➔ **Client-centric consistency models**: view of **one** process
 - ➔ assumption: (essentially) no update by multiple processes



9.2 Distribution Protocols

- ➔ Question: where, when and by whom are replicas placed?
 - ➔ permanent replicas
 - ➔ server initiated replicas
 - ➔ client initiated replicas
- ➔ Question: how are updates distributed (regardless of consistency model)
 - ➔ sending invalidations, status or operations
 - ➔ pull or push protocols
 - ➔ unicast or multicast

Placing the Replicas



→ All three types can occur simultaneously



Permanent Replicas

- ➔ Initial set of replicas, static, mostly small
- ➔ Examples:
 - ➔ replicated web site (transparent to client)
 - ➔ mirroring (client deliberately chooses a replica)

Server Initiated Replicas

- ➔ Server creates additional replicas on demand (*Push-Cache*)
 - ➔ e.g., for web hosting services
- ➔ Difficult: deciding when and where replicas will be created
 - ➔ usually access counter for each file, additional information about the origin of the requests (→ nearest server)



Client initiated Replicas

- ➔ Other term: *Client Cache*
- ➔ Client cache locally stores (frequently) used data
- ➔ Goal: improving access time
- ➔ Management of the cache is completely left to the client
 - ➔ server doesn't care about consistency
- ➔ Data is usually kept in the cache for a limited time only
 - ➔ prevents use of extremely obsolete data
- ➔ Cache usually placed on client machines, or shared cache for multiple clients in their proximity
 - ➔ e.g., web proxy caches



Forwarding Updates: What's Being Sent?

- ➔ The new value of the data object
 - ➔ good with high read/update ratio
- ➔ The update operation (active replication)
 - ➔ saves bandwidth (operation with parameters is usually small)
 - ➔ but more computing power required
- ➔ Just a notification (invalidation protocols)
 - ➔ notification makes the copy of the data object invalid
 - ➔ on next access a new copy will be requested
 - ➔ requires very little network bandwidth
 - ➔ good at low read/update ratio



Pull and Push Protocols

- ➔ **Push**: updates are distributed on the initiative of the server that made the change
 - ➔ replicas don't have to request updates
 - ➔ common in permanent and server-initiated replicas
 - ➔ when a relatively high degree of consistency is required
 - ➔ at high read/update ratio
 - ➔ problem: server must know all replicas
- ➔ **Pull**: replicas actively request data updates
 - ➔ common with client caches
 - ➔ at low read/update ratio
 - ➔ disadvantage: higher response time for cache access
- ➔ **Leases**: mixed form: first push for some time, then pull later



Unicast vs. Multicast

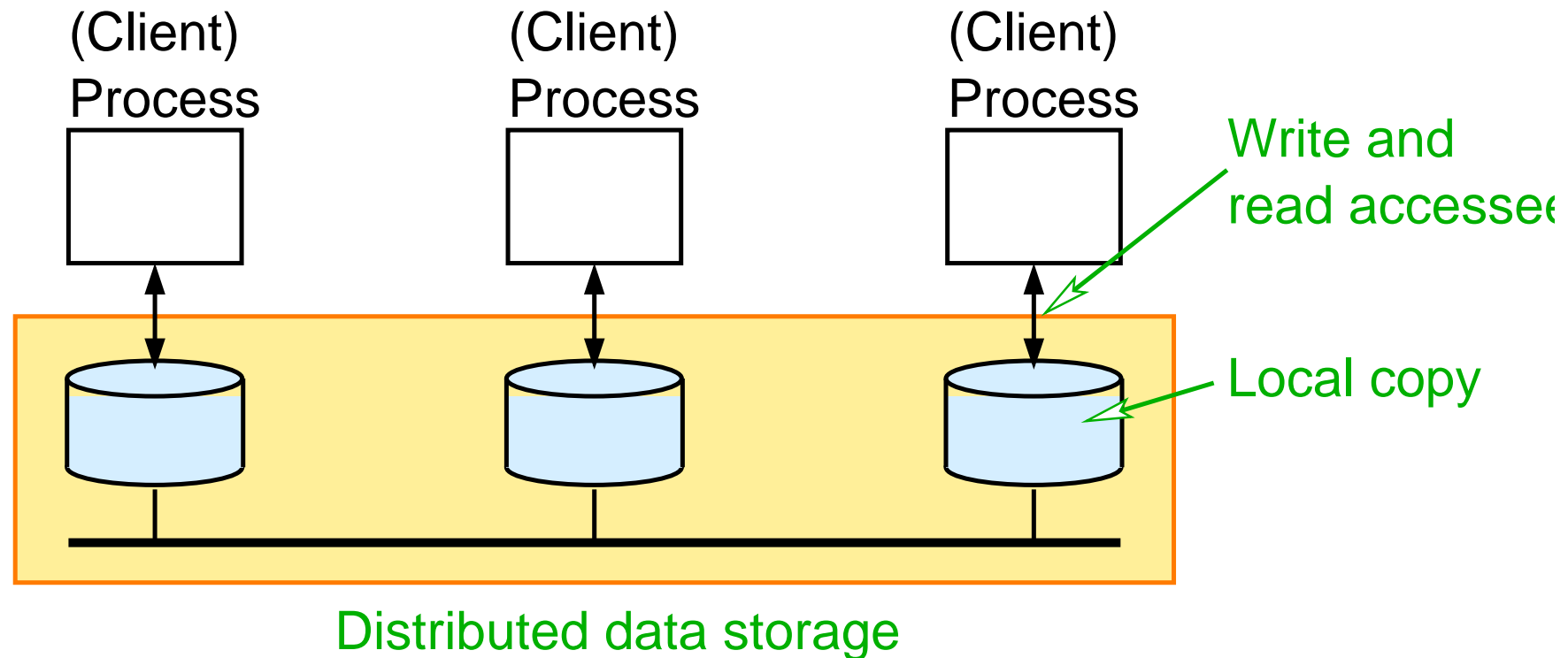
- ➔ Unicast: send update individually to each replica server
- ➔ Multicast: send one message and leave the distribution to the network (e.g. IP multicast)
 - ➔ often much more efficient
 - ➔ especially in LANs: hardware broadcast possible
- ➔ Multicast is useful for push protocols
- ➔ Unicast is better with pull protocols
 - ➔ only a single client/server requests an update

9.3 Data Centric Consistency



9.3.1 Consistency Models

Basis: model of a distributed data store



- ➔ logical, shared data memory
- ➔ physically distributed and replicated across multiple nodes

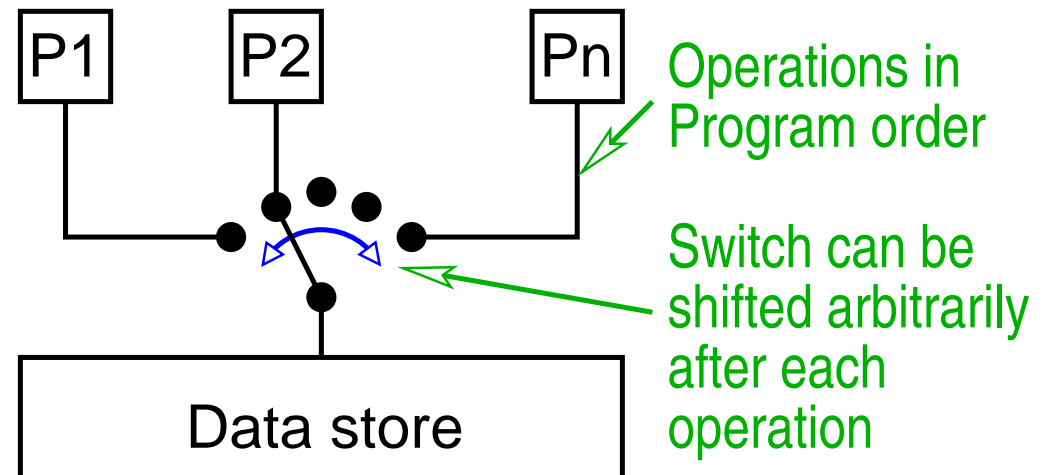
Sequential Consistency

- ➔ A data store is **sequentially consistent** if the result of each program execution is as if:
 - ➔ the (read/write) operations of all processes are executed in a (random) sequential order,
 - ➔ in which the operations of each individual process appear in the order specified by the program.

➔ I.e. the execution of the operations of the individual processes can be interleaved arbitrarily

➔ Independent of time or clocks W

➔ All processes see the (write) accesses in the same order



Sequential Consistency: Examples

Allowed sequence:

P1:	W(x)a		
P2:	W(x)b		
P3:	R(x)b	R(x)a	
P4:	R(x)b	R(x)a	

Forbidden Sequence:

P1:	W(x)a		
P2:	W(x)b		
P3:	R(x)b	R(x)a	
P4:		R(x)a	R(x)b

➔ Notation:

➔ $W(x)a$: the value 'a' is written into the variable 'x'

➔ $R(x)a$: variable 'x' will be read, result is 'a'

➔ A possible sequential order of the left sequence:

➔ $W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a$



Sequential Consistency: Examples

Allowed sequence:

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b R(x)a
P4:	R(x)b R(x)a

Forbidden Sequence:

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b R(x)a
P4:	R(x)a R(x)b

➔ Notation:

➔ $W(x)a$: the value 'a' is written into the variable 'x'

➔ $R(x)a$: variable 'x' will be read, result is 'a'

➔ A possible sequential order of the left sequence:

➔ $W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a$



Linearizability

- ➔ Stronger than sequential consistency
- ➔ Assumption: the nodes (processes) have synchronized clocks
 - ➔ i.e. an approximation of a global time
- ➔ Operations have time stamps based on these clocks
- ➔ In comparison with sequential consistency additionally required:
 - ➔ the sequential order of operations is consistent with their timestamps
- ➔ Complex implementation
- ➔ Used for formal verification of concurrent algorithms

Causal Consistency

- ➔ Weakening of sequential consistency
- ➔ (Only) write operations that are potentially causally dependent must be visible to all processes in the same order

Causally, but not seq. consistent:

P1:	W(x)a	W(x)c
P2:	R(x)a W(x)b	
P3:	R(x)a	R(x)c R(x)b
P4:	R(x)a	R(x)b R(x)c

Not causally consistent:

P1:	W(x)a	
P2:	R(x)a W(x)b	
P3:		R(x)b R(x)a
P4:		R(x)a R(x)b



Weak Consistency

- ➔ In practice: access to shared resources is coordinated via synchronization variables (SV)
- ➔ Then: weaker consistency requirements are sufficient:
 - ➔ accesses to SVs are sequentially consistent
 - ➔ an operation on a SV is not allowed until all previous write accesses to data have been completed everywhere
 - ➔ no operation on data is allowed before all previous operations on SVs have been completed

Allowed event sequence:

P1:	W(x)a	W(x)b	S	
P2:		S	R(x)b	
P3:		R(x)a	R(x)b	S
P4:		R(x)b	R(x)a	S

Invalid event sequence:

P1:	W(x)a	W(x)b	S
P2:		S	R(x)a



Release Consistency (*Freigabe-Konsistenz*)

- ➔ Idea as with weak consistency, but distinction between *acquire* and *release* operations (mutual exclusion!)
- ➔ before an operation on the data is performed all *acquire*-operations of the process must be completed
- ➔ before the end of a *release* operation all operations of the process on the data must be completed
- ➔ *acquire* / *release* operations of a process are seen everywhere in the same order

Allowed event sequence:

P1:	acq(L)	W(x)a	W(x)b	rel(L)		
P2:			acq(L)	R(x)b	rel(L)	
P3:						R(x)a



Comparison of models

Strict	Absolute time sequence of all shared accesses (physically not useful!)
Linearization	All processes see all (write) accesses in the same order. Accesses are sorted by a (non-unique) global timestamp.
Sequential	All processes see all (write) accesses in the same order. Accesses are not sorted by time.
Causal	All processes see causally linked (write) accesses in the same order.
Weak	Data is only reliably consistent after a synchronization has been performed.
Release	Data is made consistent when leaving the critical region.



9.3.2 Consistency Protocols

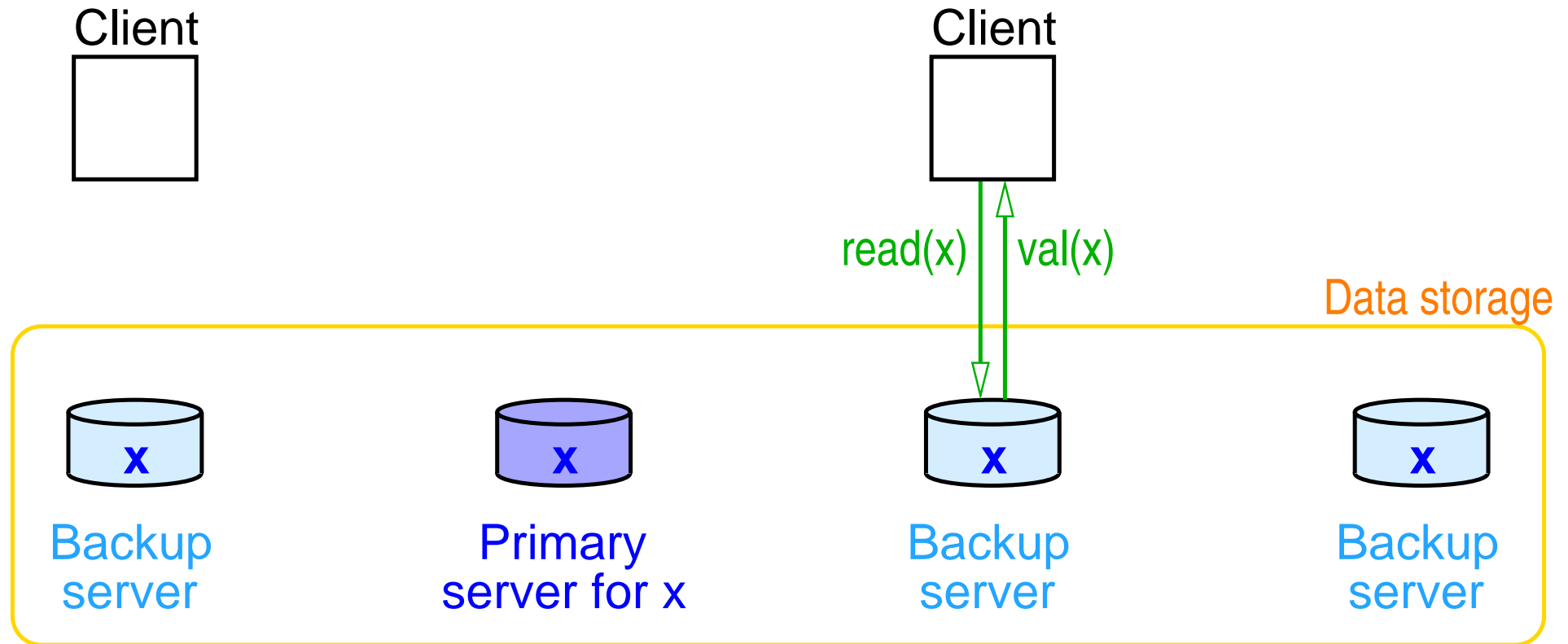
- ➔ Describe how replica servers coordinate with each other to implement a specific consistency model
- ➔ Focus in the following:
 - ➔ consistency models that globally serialize operations
 - ➔ e.g., sequential, weak and release consistency
- ➔ Two basic approaches:
 - ➔ **primary-based** (*primärbasierte*) **protocols**
 - ➔ write operations are always coordinated by a special copy (**primary copy**)
 - ➔ **replicated-write protocols**
 - ➔ write operations go to multiple copies



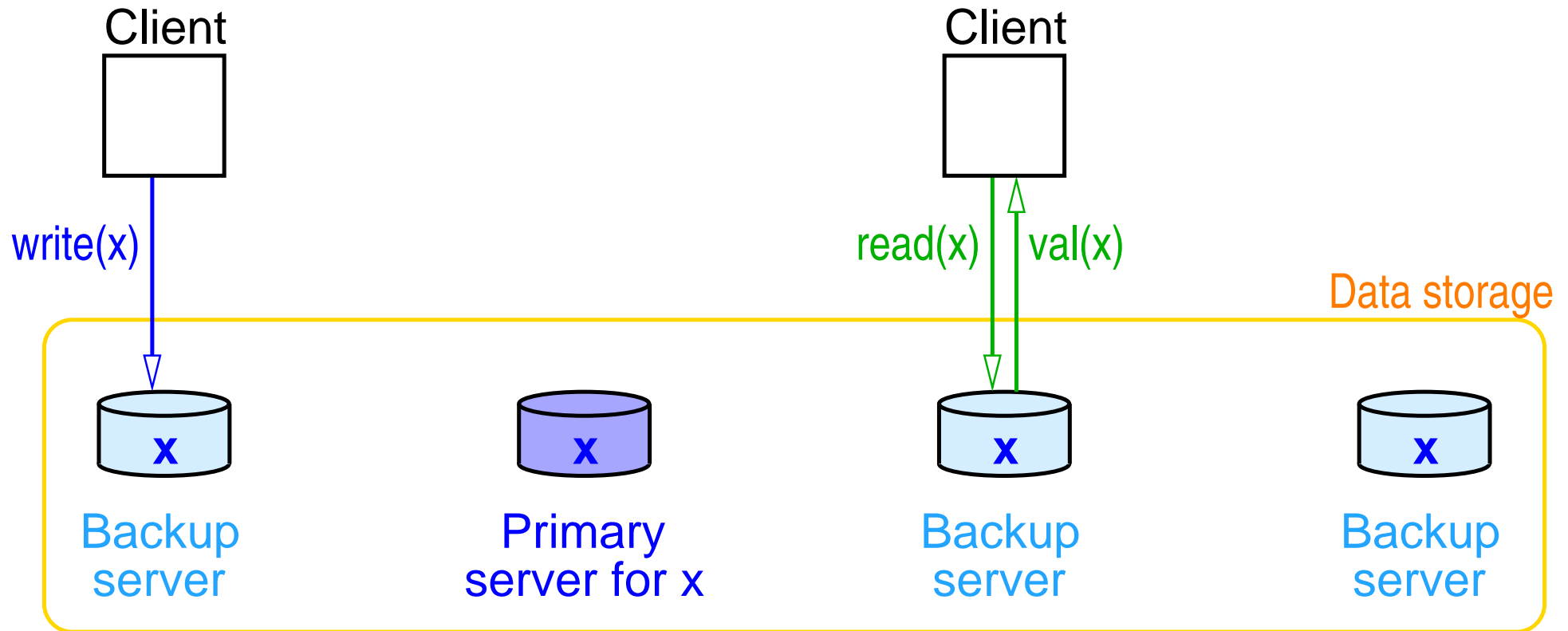
Primary-Based Protocols

- ➔ Read operations are possible on arbitrary (local) copies
- ➔ Write operations must be handled by the primary copy
 - ➔ e.g., to realize a sequential consistency:
 - ➔ the primary copy updates all other copies and waits for acknowledgements, only then it replies to the client
 - ➔ problem: performance
- ➔ **Remote-write protocols**
 - ➔ the writer forwards the operation to a fixed primary copy
- ➔ **Local-write protocols**
 - ➔ writer must become primary copy before it can do the update
 - ➔ i.e., the primary copy is migrated between servers
 - ➔ good model also for mobile users

Remote Write Protocol: Workflow (Sequential Consistency)

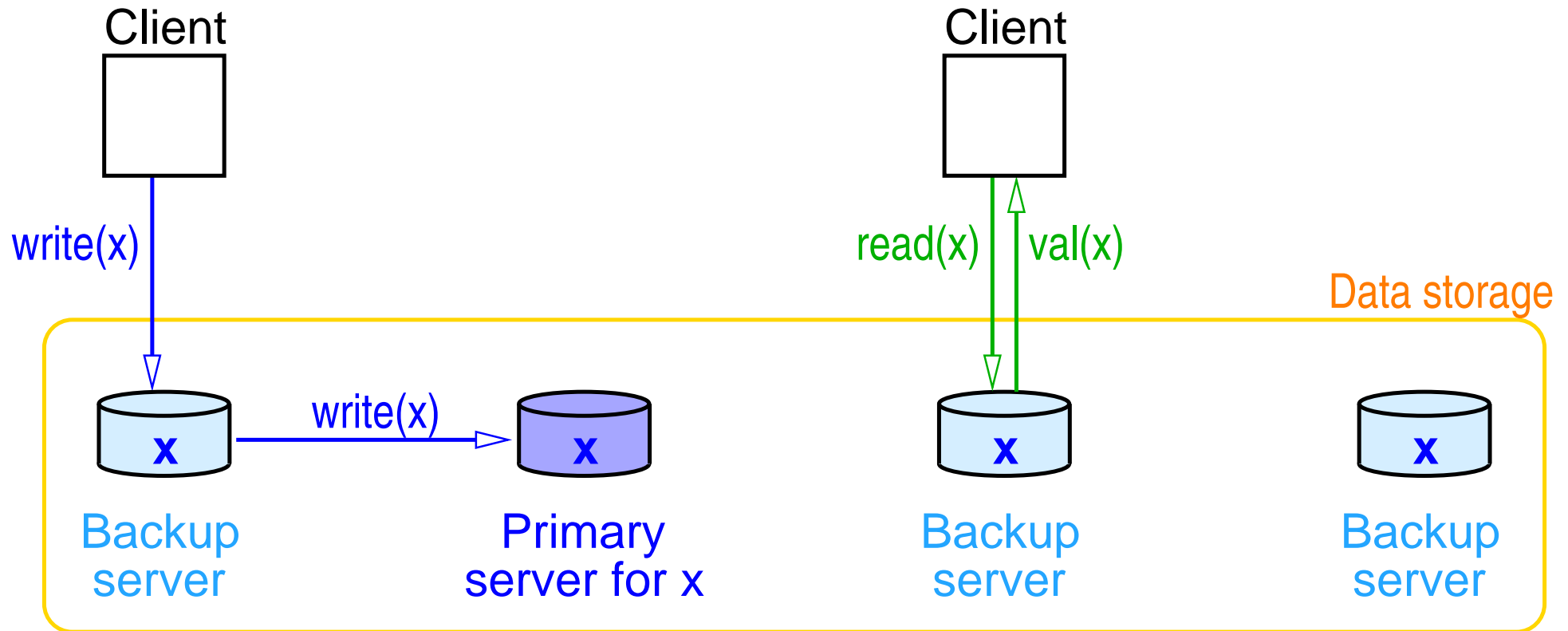


Remote Write Protocol: Workflow (Sequential Consistency)



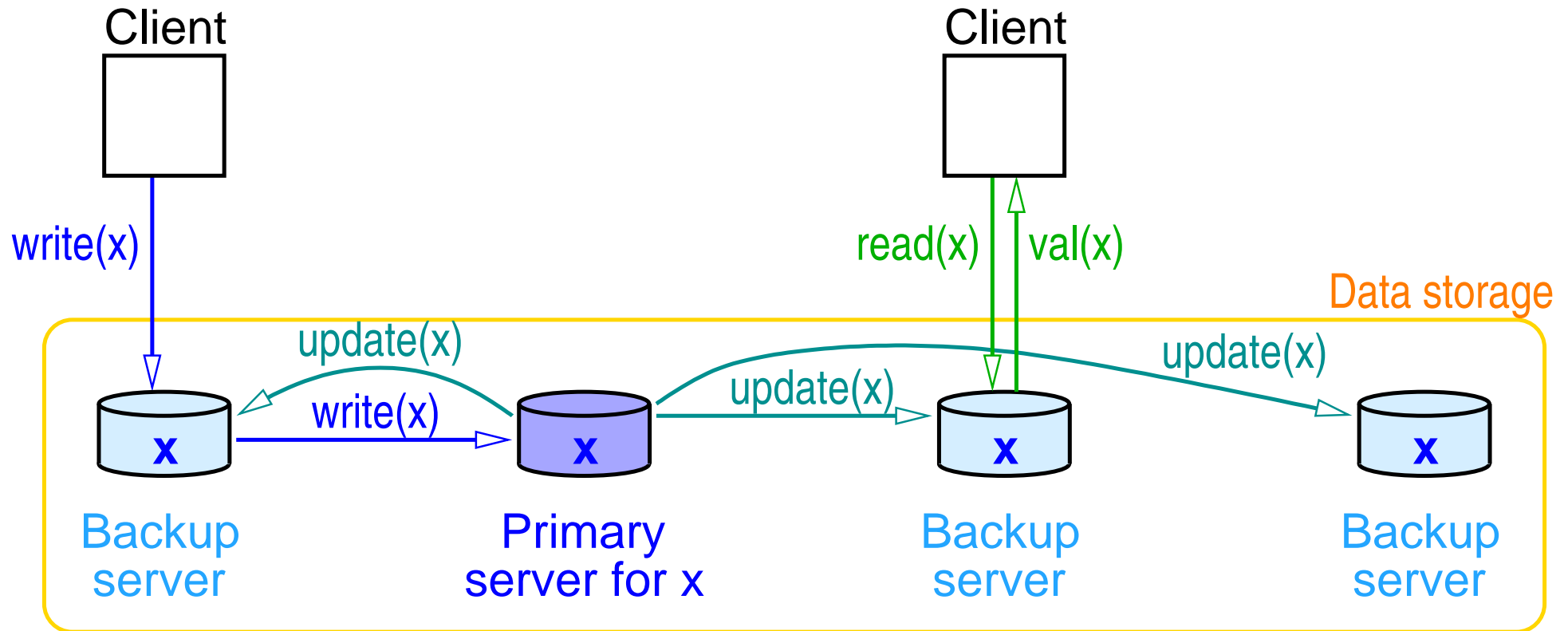
(1) Write request

Remote Write Protocol: Workflow (Sequential Consistency)



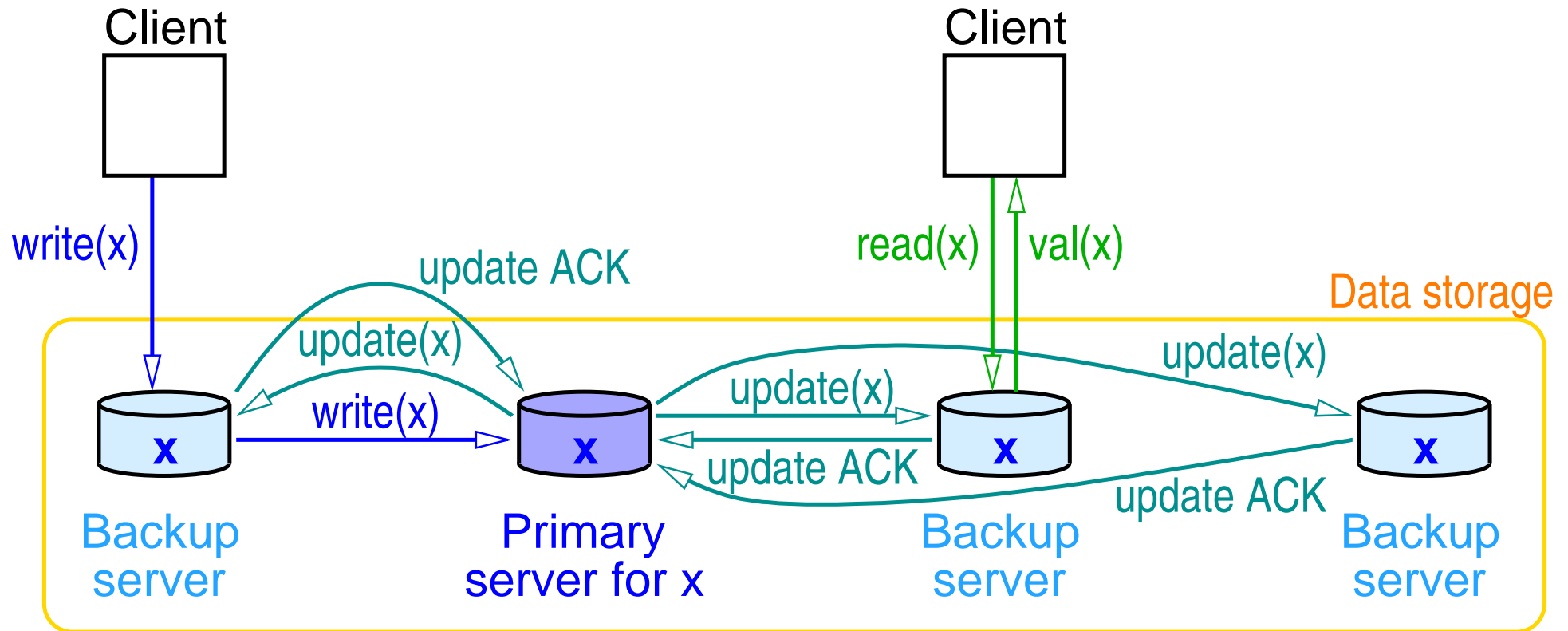
(1) Write request is forwarded to primary server

Remote Write Protocol: Workflow (Sequential Consistency)



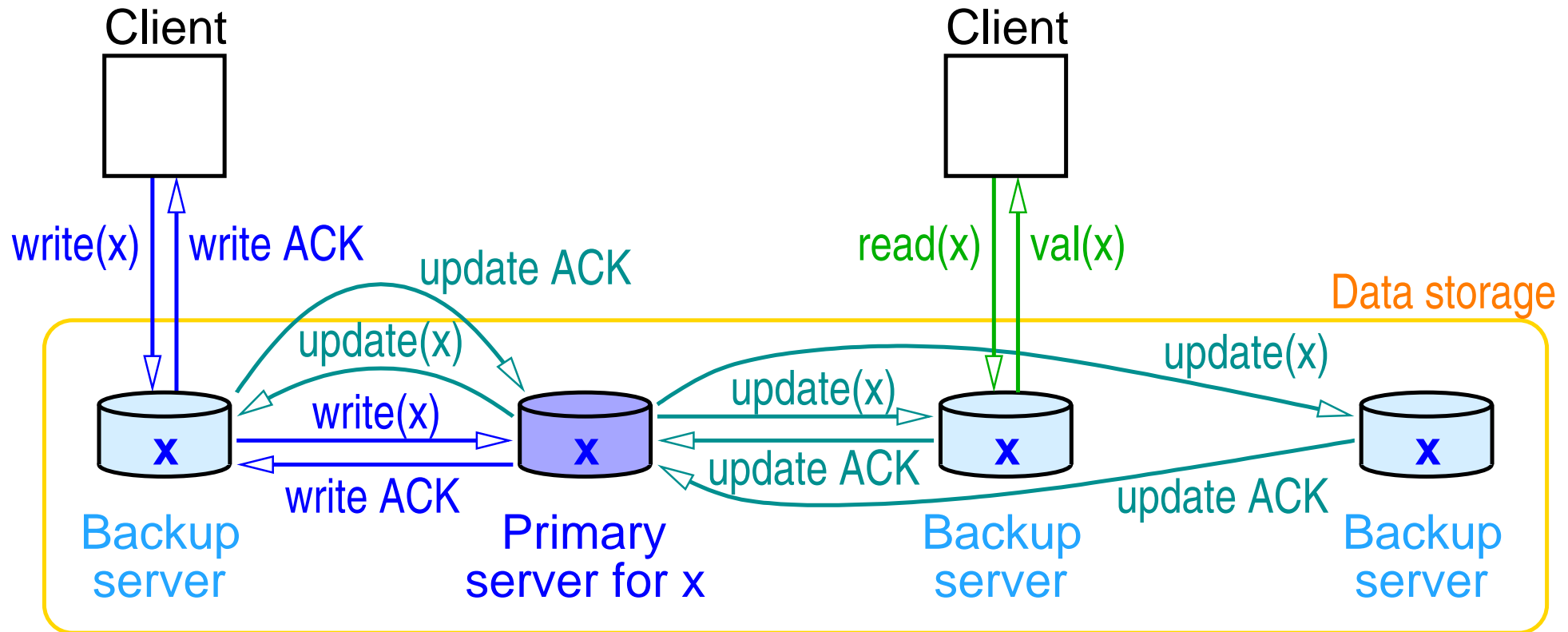
- (1) Write request is forwarded to primary server
- (2) Primary server updates all backups

Remote Write Protocol: Workflow (Sequential Consistency)



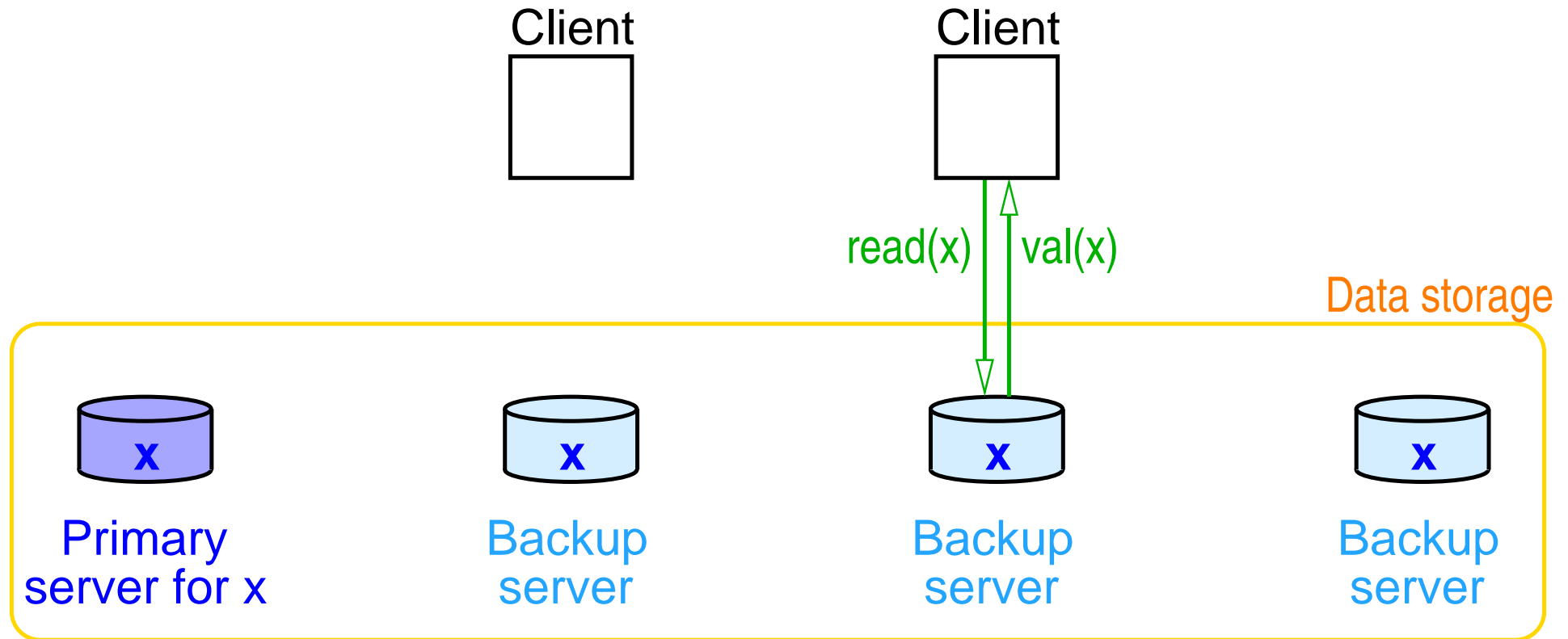
- (1) Write request is forwarded to primary server
- (2) Primary server updates all backups and waits for acknowledgements

Remote Write Protocol: Workflow (Sequential Consistency)

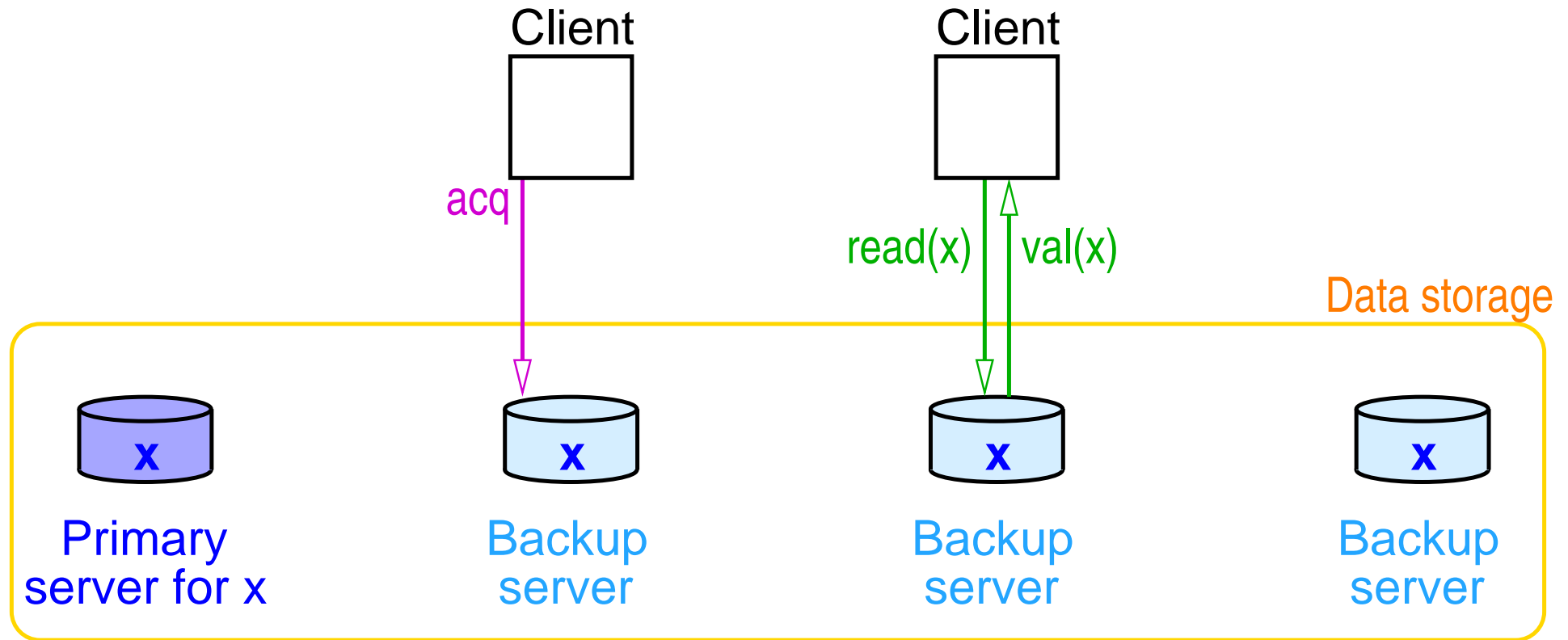


- (1) Write request is forwarded to primary server
- (2) Primary server updates all backups and waits for acknowledgements
- (3) Acknowledge the end of the write operation

Local Write Protocol: Workflow (Release Consistency)

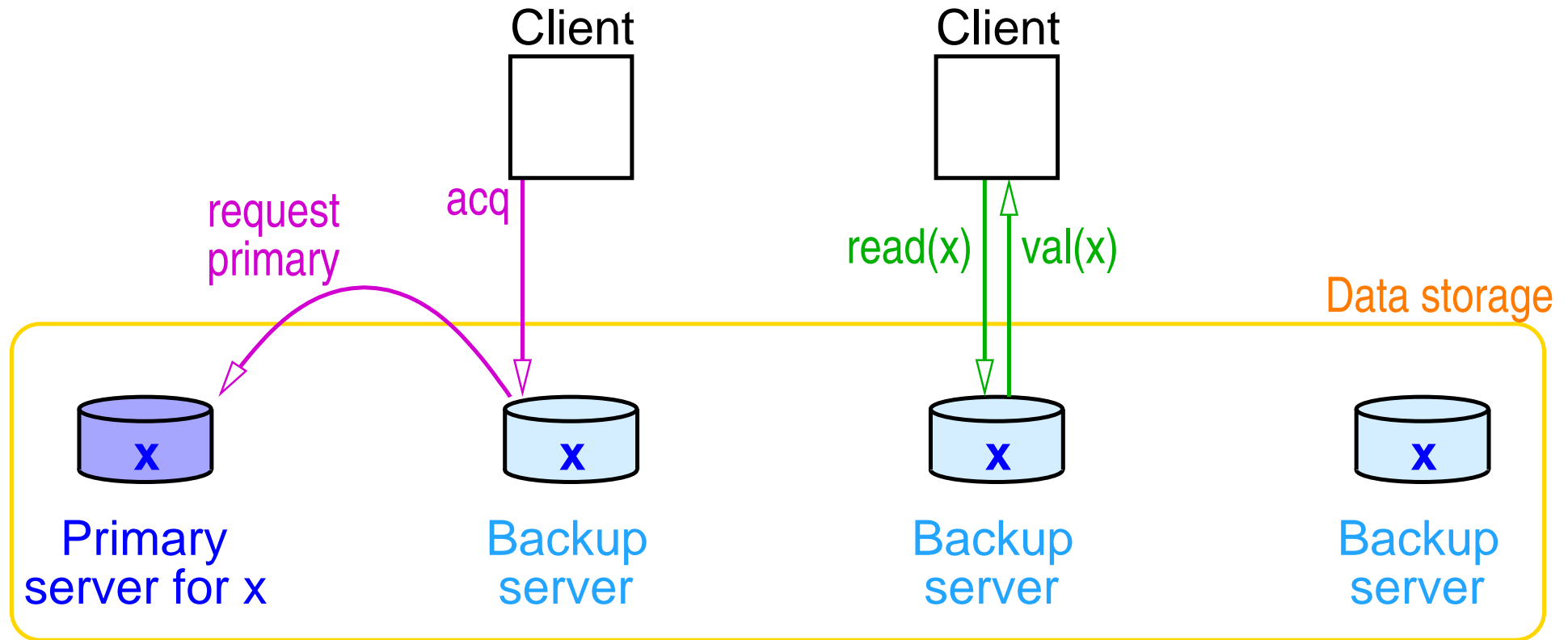


Local Write Protocol: Workflow (Release Consistency)



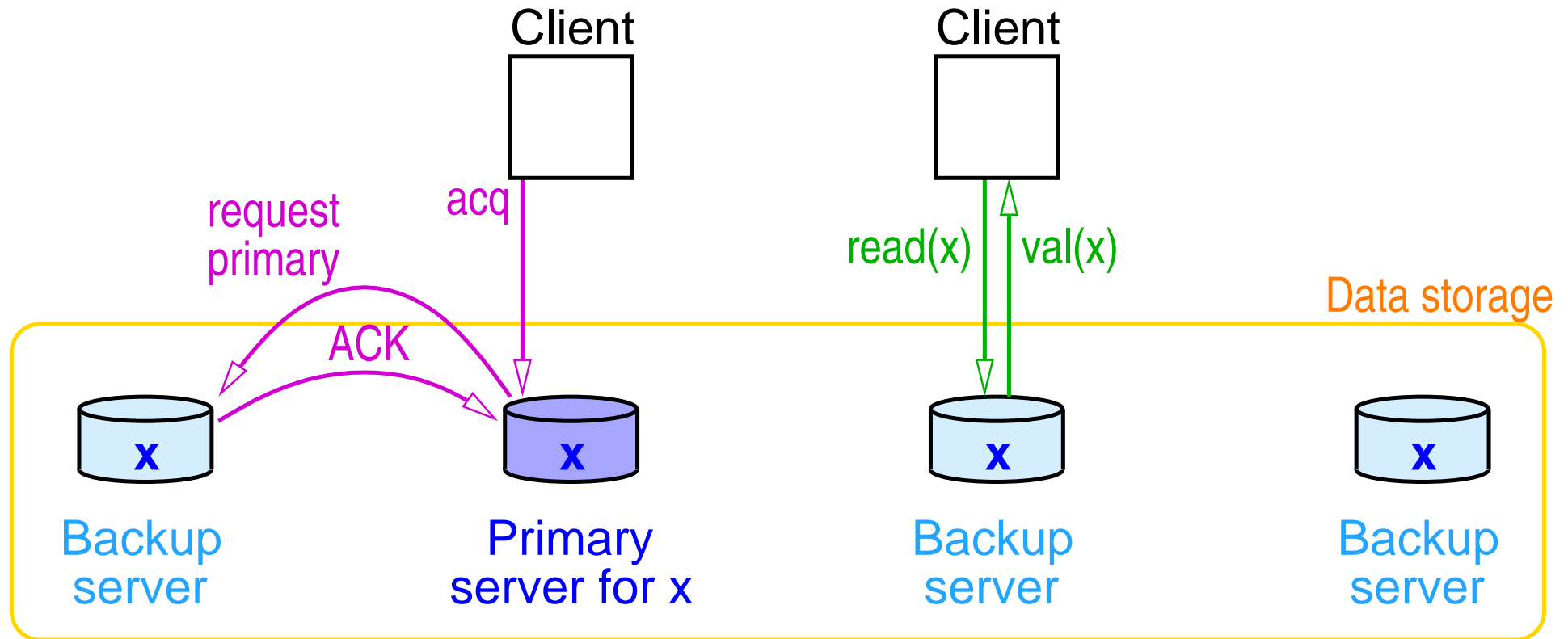
(1) Acquire lock;

Local Write Protocol: Workflow (Release Consistency)



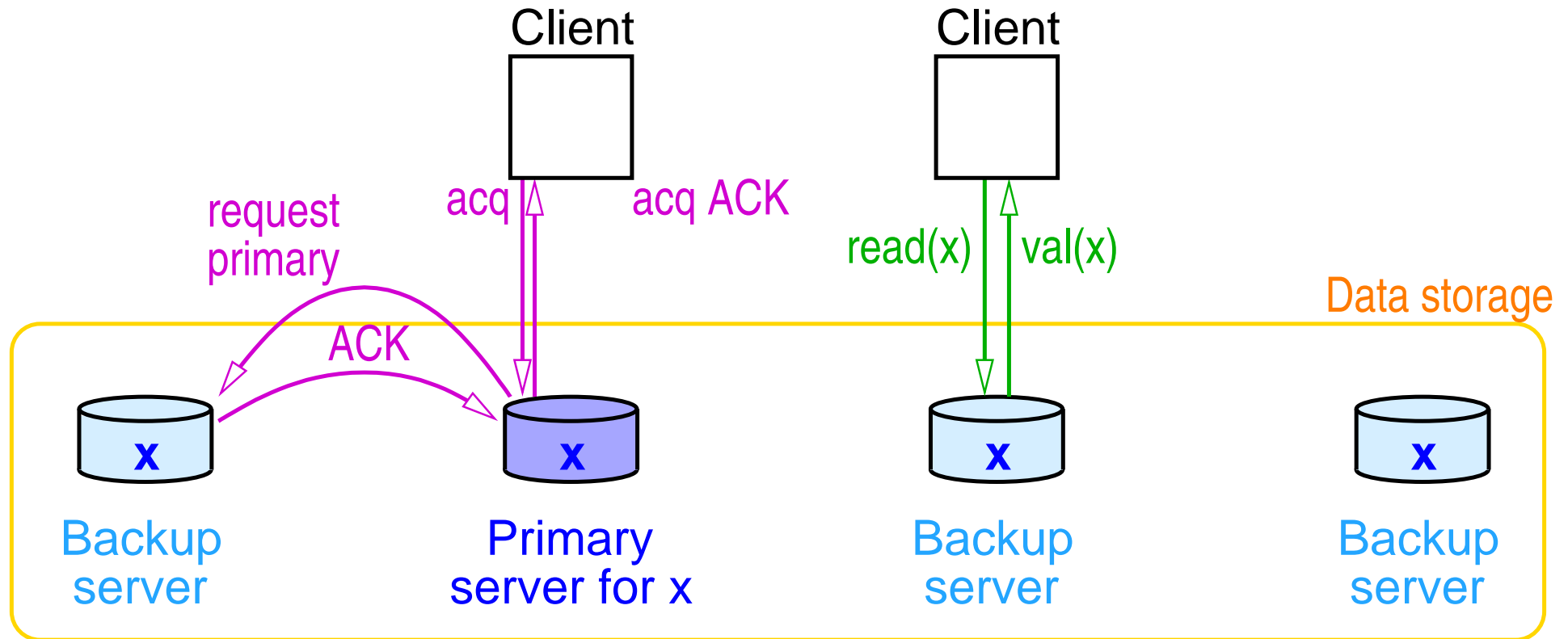
(1) Acquire lock; Move primary copy to new server

Local Write Protocol: Workflow (Release Consistency)



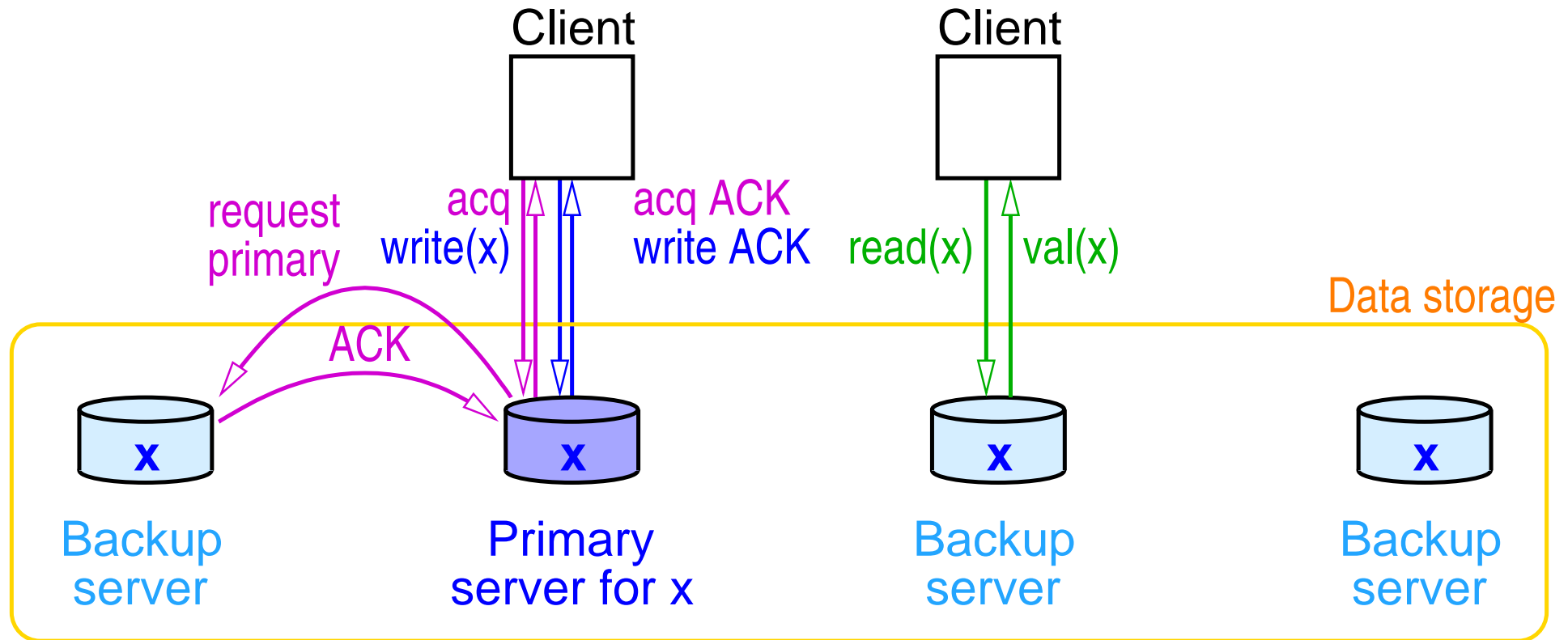
(1) Acquire lock; Move primary copy to new server

Local Write Protocol: Workflow (Release Consistency)



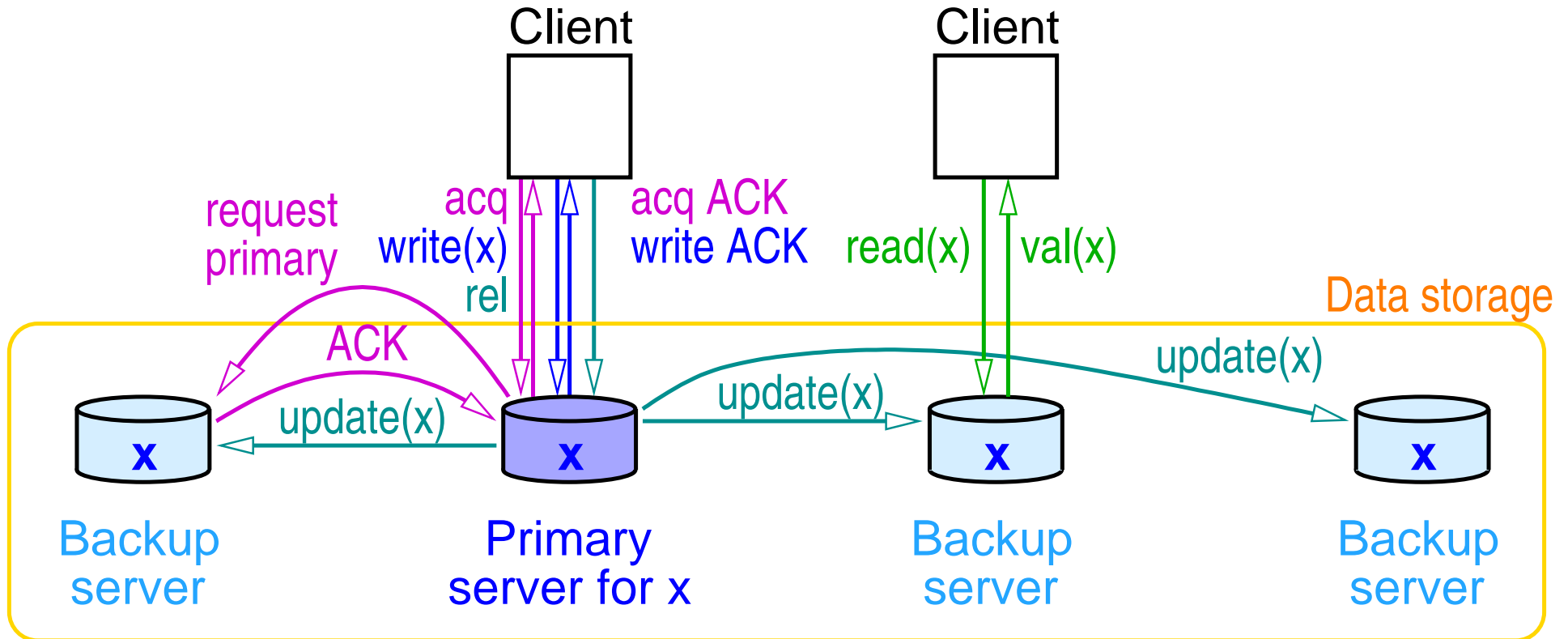
- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation

Local Write Protocol: Workflow (Release Consistency)



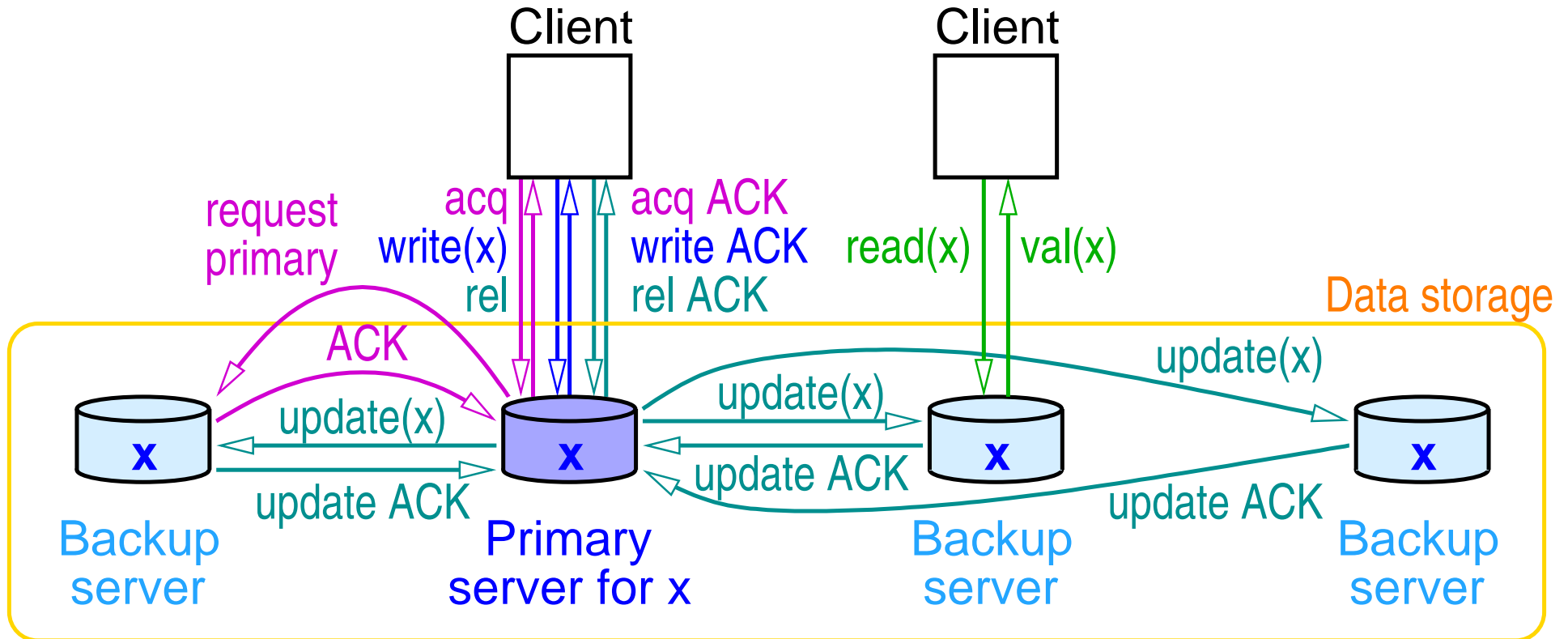
- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation
- (3) Write operations are executed (only) on the local server

Local Write Protocol: Workflow (Release Consistency)



- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation
- (3) Write operations are executed (only) on the local server
- (4) New primary server updates backups

Local Write Protocol: Workflow (Release Consistency)



- (1) Acquire lock; Move primary copy to new server
- (2) Acknowledge the end of the write operation
- (3) Write operations are executed (only) on the local server
- (4) New primary server updates backups and waits for acknowledgements

Distributed Systems

Winter Term 2025/26

18.12.2025

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

Stand: January 8, 2026

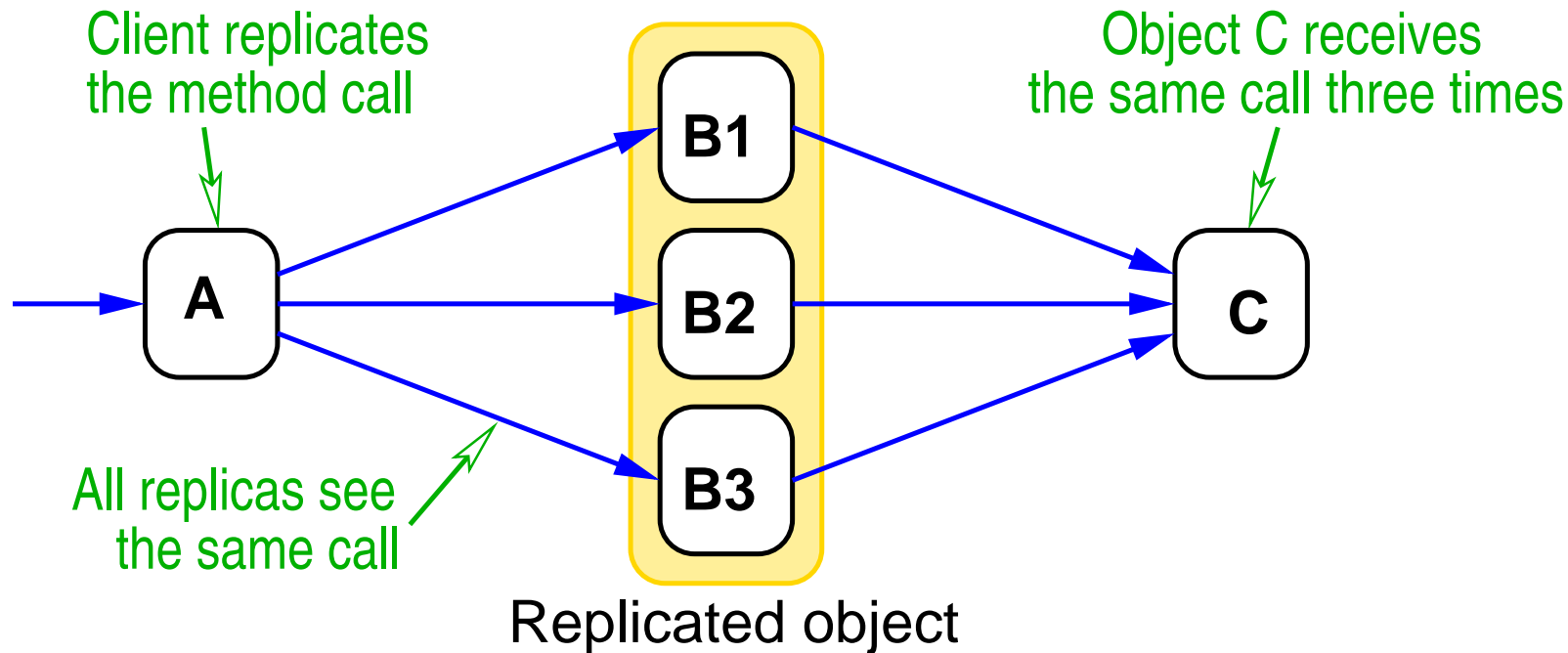


Replicated Write Protocols

- ➔ Allow execution of write operations on (multiple) arbitrary replicas
- ➔ In the following, two approaches (for sequential consistency):
 - ➔ active replication
 - ➔ update operations are passed on to all copies
 - ➔ requirement: globally unique sequence of operations
 - ➔ using totally ordered multicast
 - ➔ or via central sequencer process
 - ➔ quorum-based protocols
 - ➔ only a portion of the replicas needs to be modified
 - ➔ however, also multiple copies need to be read

Problem With Replicated Object Calls

➔ What happens when a replicated object calls another?



➔ Solution: middleware that is aware of replication

➔ coordinator of B makes sure that only one call is sent to C and its result is distributed to all replicas of B

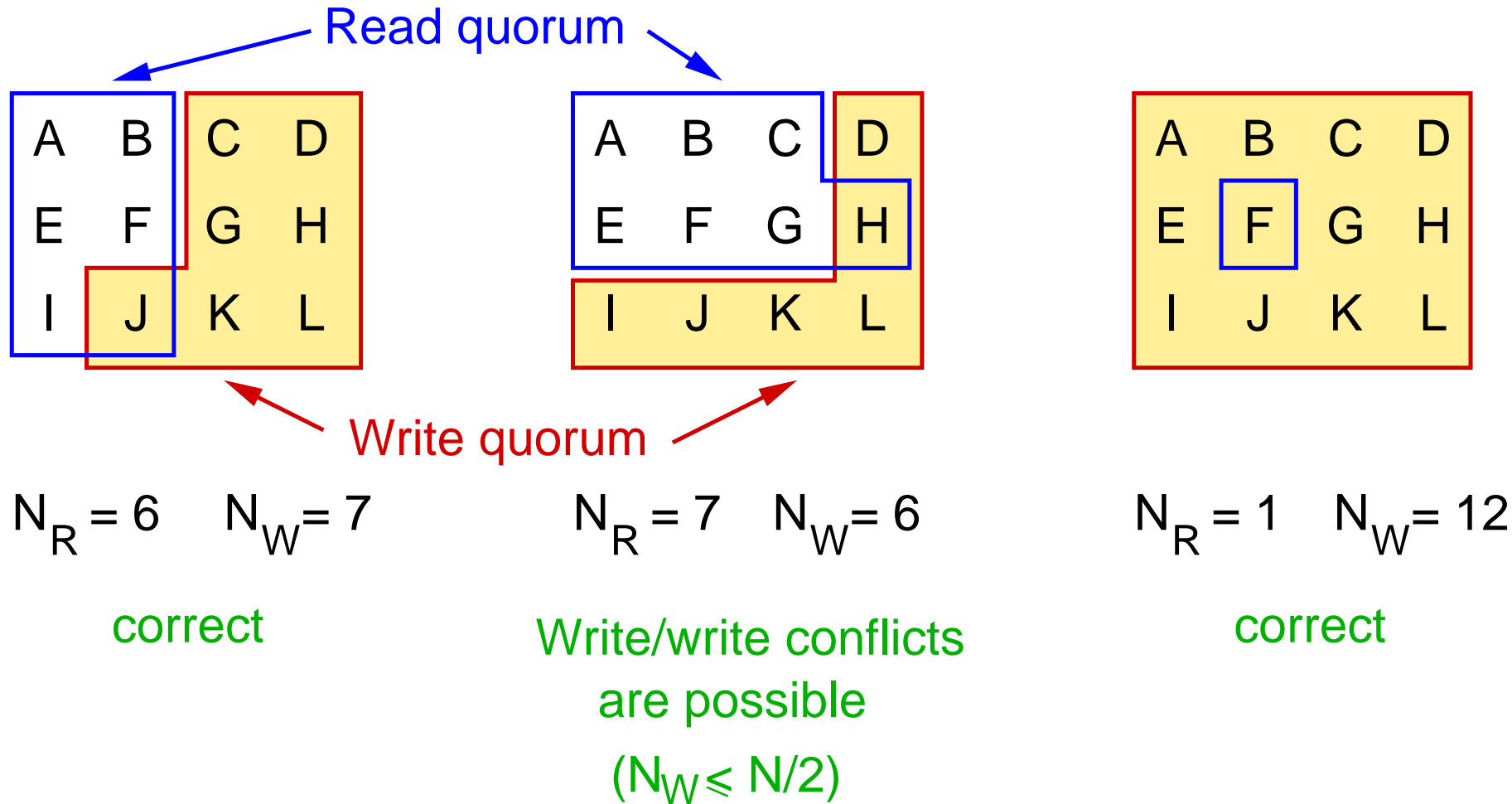


Quorum-based Protocols

- ➔ Clients need to communicate with multiple servers for writing **and** for reading
- ➔ When writing: send the request to (at least) N_W copies
 - ➔ their servers must agree to the change
 - ➔ data gets a new version number when changed
 - ➔ condition: $N_W > N/2$ (N = total number of copies)
 - ➔ prevents write/write conflicts
- ➔ When reading: send the request to (at least) N_R copies
 - ➔ client selects the latest version (highest version number)
 - ➔ condition: $N_R + N_W > N$
 - ➔ ensures that in any case the latest version is read



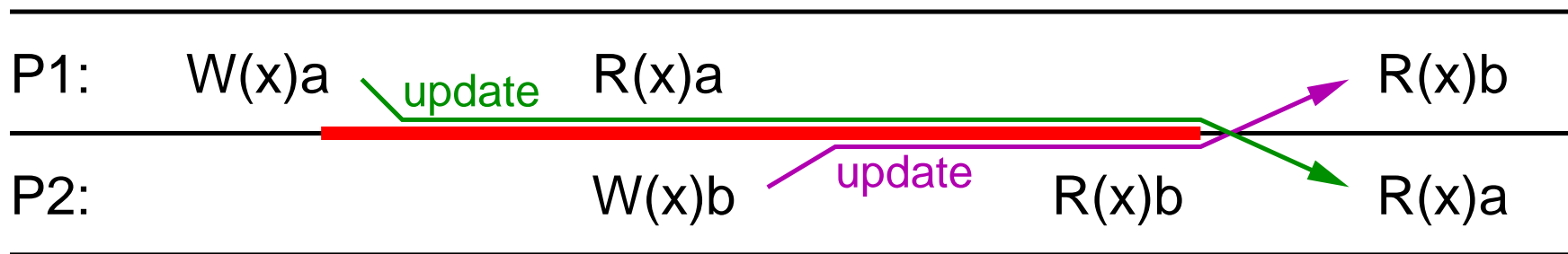
Quorum-based Protocols: Examples



9.4 (Strong) Eventual Consistency



- ➔ Problem with protocols that globally serialize operations: fault tolerance
 - ➔ esp. loss of communication / network partitioning
- ➔ Example (assuming sequential consistency):



- ➔ while P1 and P2 cannot communicate, write operations are not allowed to become visible, not even locally
- ➔ i.e. while network is partitioned, at least one partition can no longer perform read/write operations!
 - ➔ c.f. quorum based protocols!



Common situation in practice:

- ➔ Conflicting (concurrent) updates to the data are relatively rare
 - ➔ shared data store contains independent data elements
- ➔ Clients are usually independent from each other
 - ➔ i.e., they don't compare their results

Eventual Consistency

- ➔ If no new updates are made, **eventually** all correct replicas will have the same value
 - ➔ a replica is correct, if it did not (permanently) crash
- ➔ In case of conflicting updates, conflict resolution is needed
 - ➔ i.e., updates must be 'merged'
 - ➔ may require consensus and/or rollback



Strong Eventual Consistency

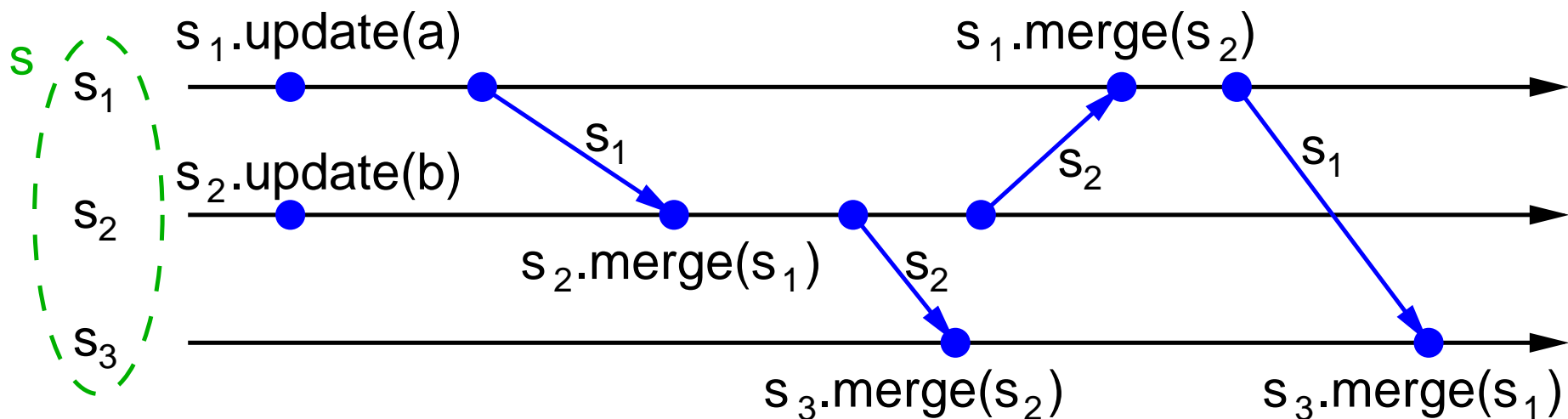
- ➔ Adds safety guarantee: if two replicas have seen the same set of updates, they are identical
- ➔ Does not require conflict resolution or rollbacks
- ➔ Ensures safety and liveness despite any number of non-byzantine failures

Conflict-Free Replicated Datatypes (CRDTs)

- ➔ Pattern for distributed data structures that guarantees strong eventual consistency by design
- ➔ No synchronisation required
- ➔ Examples: counters, lists, sets, editable text documents, ...
- ➔ Important application: collaborative editing of documents

State-based (Convergent) Replicated Datatypes

- ➔ Object with the following methods (executing on a single replica):
 - ➔ *init* (constructor): sets initial object state
 - ➔ *value*: returns the value of the object
 - ➔ *update*: modifies the object
 - ➔ *merge*: merges state from a remote replica into the local state
- ➔ Each replica occasionally send its state to the other replicas





State-based (Convergent) Replicated Datatypes ...

- ➔ Sufficient conditions for strong eventual consistency:
 - ➔ all methods executions terminate
 - ➔ all updates are eventually delivered (directly or indirectly) to all correct replicas
 - ➔ the set S of possible object values is a join semilattice
 - ➔ i.e., there is a partial order \leq on S , such that any two elements $x, y \in S$ have a least upper bound $x \sqcup y$
 - ➔ $x.merge(y) = x \sqcup y$
 - ➔ state is monotonically non-decreasing across updates, i.e., $s \leq s.update(a)$



Example: State-based Grow-only Counter

```
class GCounter:  
    def __init__(self, n):  
        self.val = [0] * n # Vector: one element per replica  
    def value(self):  
        return sum(self.val)  
    def increment(self):  
        self.val[myReplicaID()] += 1  
    def merge(self, other):  
        # component-wise maximum  
        self.val = [max(x,y) for x,y in zip(self.val, other.val)]
```

➔ The (partial) order on vectors is defined component-wise:

$$a \leq b \Leftrightarrow \forall i \in [0, n - 1] : a_i \leq b_i$$

➔ Very similar to a vector timestamp (👉 6.3)



Example: State-based Positive-Negative Counter

- ➔ Problem: we cannot decrement the local counter, as this violates the monotonicity requirement
- ➔ Solution: compose two grow-only counters

```
class PNCounter:
    def __init__(self, n):
        self.inc, self.dec = GCounter(n), GCounter(n)
    def value(self):
        return self.inc.value() - self.dec.value()
    def increment(self):
        self.inc.increment()
    def decrement(self):
        self.dec.increment()
    def merge(self, other):
        self.inc, self.dec = self.inc.merge(other.inc), \
            self.dec.merge(other.dec)
```



Operation-based (Commutative) Replicated Datatypes

➔ Object with the following methods:

➔ *init* (constructor): sets initial object state

➔ *value*: returns the value of the object

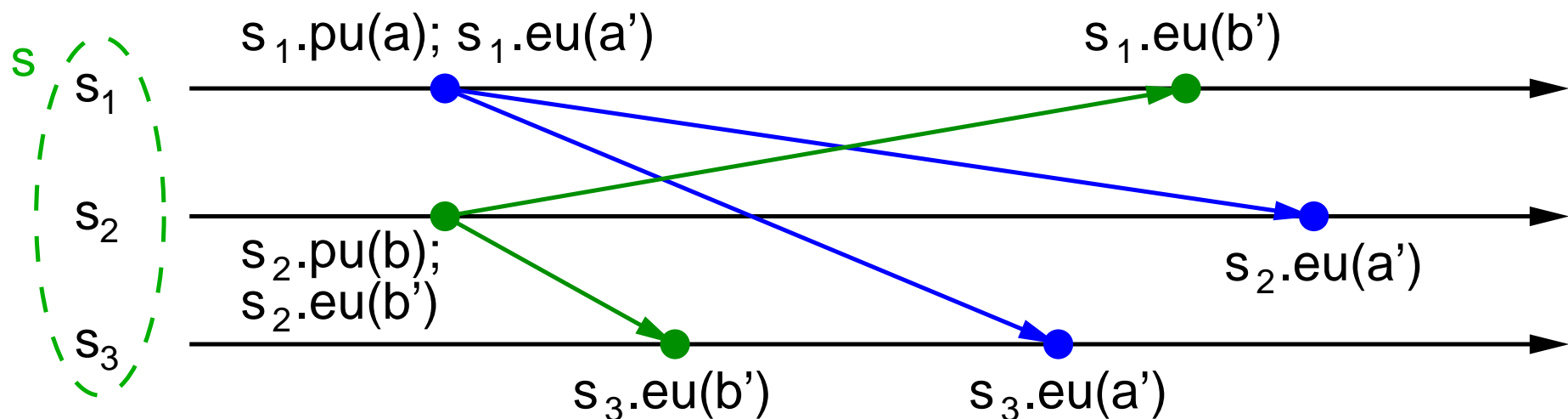
➔ *prepare_update*: prepare an update

➔ is immediately followed by *effect_update*

➔ *effect_update*: performs the update

execute on
single replica

executes on all replicas





Operation-based (Commutative) Replicated Datatypes ...

- ➔ Sufficient conditions for strong eventual consistency:
 - ➔ all methods executions terminate
 - ➔ communication uses a reliable, causally ordered multicast
 - ➔ allows concurrent updates to be delivered in any order
 - ➔ concurrent updates commute
 - ➔ i.e., the order of execution doesn't matter

- ➔ Note: if *all* updates commute, a reliable, unordered multicast is sufficient



Example: Operation-based Positive-Negative Counter

```
class Counter:
    def __init__():
        counter = 0
    def value():
        return counter
    def effect_increment():      # no prepare method necessary
        counter += 1
    def effect_decrement():    # no prepare method necessary
        counter -= 1
```

- ➔ Easier than state-based counter, but requires a multicast whenever counter is incremented / decremented
- ➔ Reliable, unordered multicast is sufficient, as updates always commute



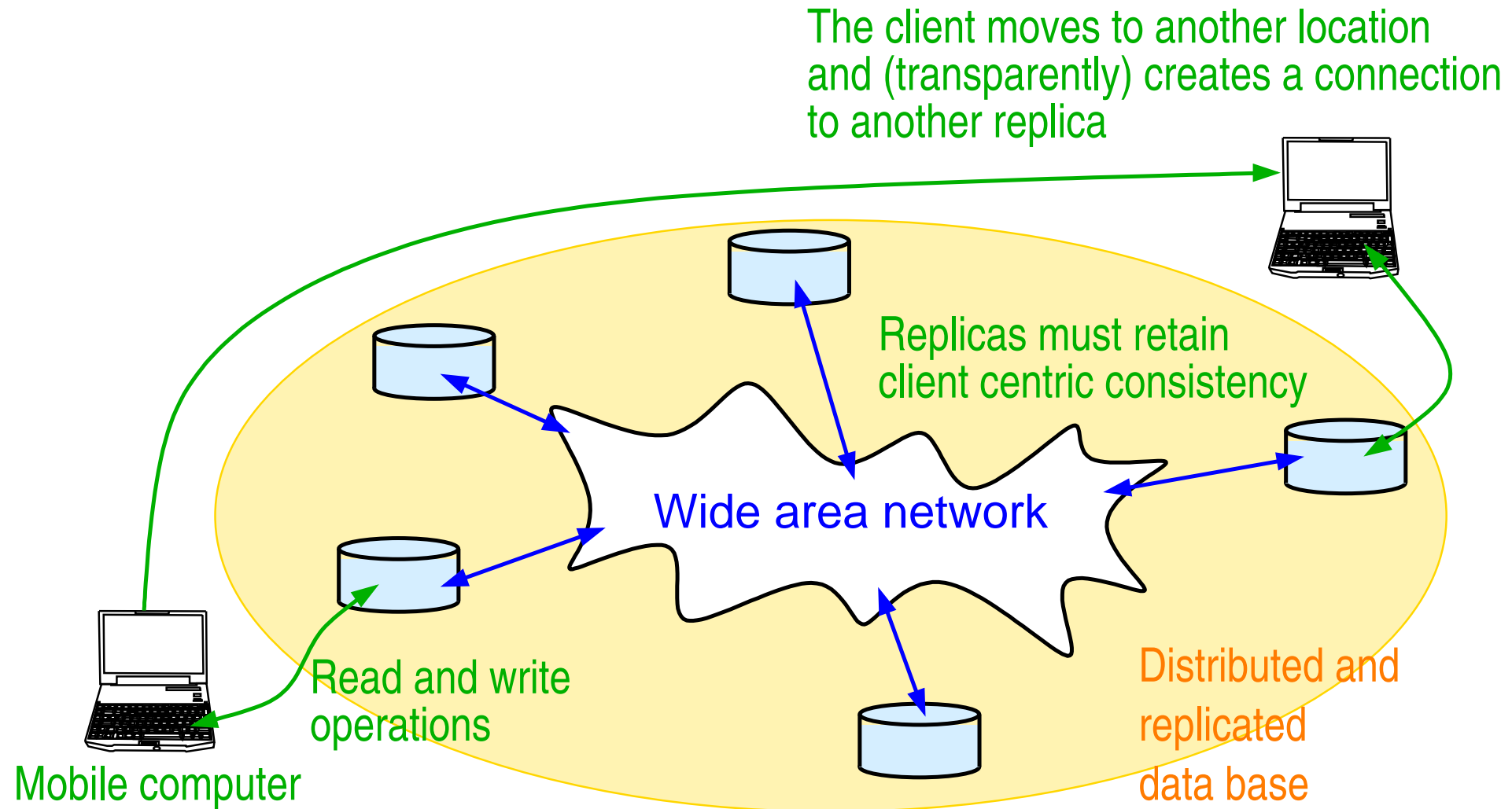
Some more CRDTs

- ➔ Grow-only set
 - ➔ similar to grow-only counter
- ➔ Two-phase set: supports both add and remove operations
 - ➔ similar to positive-negative counter: removed objects are collected in a *tombstone* set
- ➔ Sequence CRDTs
 - ➔ allow distributed, collaborative real-time editing of documents
 - ➔ basic idea:
 - ➔ each element has a (globally) unique identifier
 - ➔ identifier space is dense and totally ordered
 - ➔ order of identifiers determines element sequence
 - ➔ ensures that concurrent insert / remove operations commute



- ➔ Eventual consistency may create problems, when a client changes the replica it is accessing
 - ➔ updates may not have arrived there yet
 - ➔ client experiences inconsistent behavior
- ➔ Solution: client-centric consistency models
 - ➔ guarantee consistency for an **individual** client
 - ➔ but not for concurrent accesses by multiple clients
- ➔ Models (see Tanenbaum / van Steen, Ch. 6.3):
 - ➔ Monotonic Read
 - ➔ when a client reads a variable x , every subsequent read of x returns the same or a more recent value
 - ➔ Monotonic Write
 - ➔ Read Your Writes
 - ➔ Writes Follow Reads

Illustration of the problem





9.6 Summary

- ➔ Replication due to availability and performance
- ➔ Problem: consistency of copies
 - ➔ strictest model: sequential consistency
 - ➔ weakenings: causal consistency, weak \sim , release \sim
 - ➔ (strong) eventual consistency
 - ➔ client-centric consistency models
- ➔ Implementation of replication and consistency:
 - ➔ replication scheme: static, server initiated, client initiated
 - ➔ distribution protocols
 - ➔ type of update, push / pull, unicast / multicast
 - ➔ consistency protocols
 - ➔ primary based / replicated write / CRDTs



Distributed Systems

Winter Term 2025/26

10 Distributed File Systems



Contents

- ➔ General
- ➔ Case study: NFS

Literature

- ➔ Tanenbaum, van Steen: Ch. 10
- ➔ Colouris, Dollimore, Kindberg: Ch. 8



10.1 General

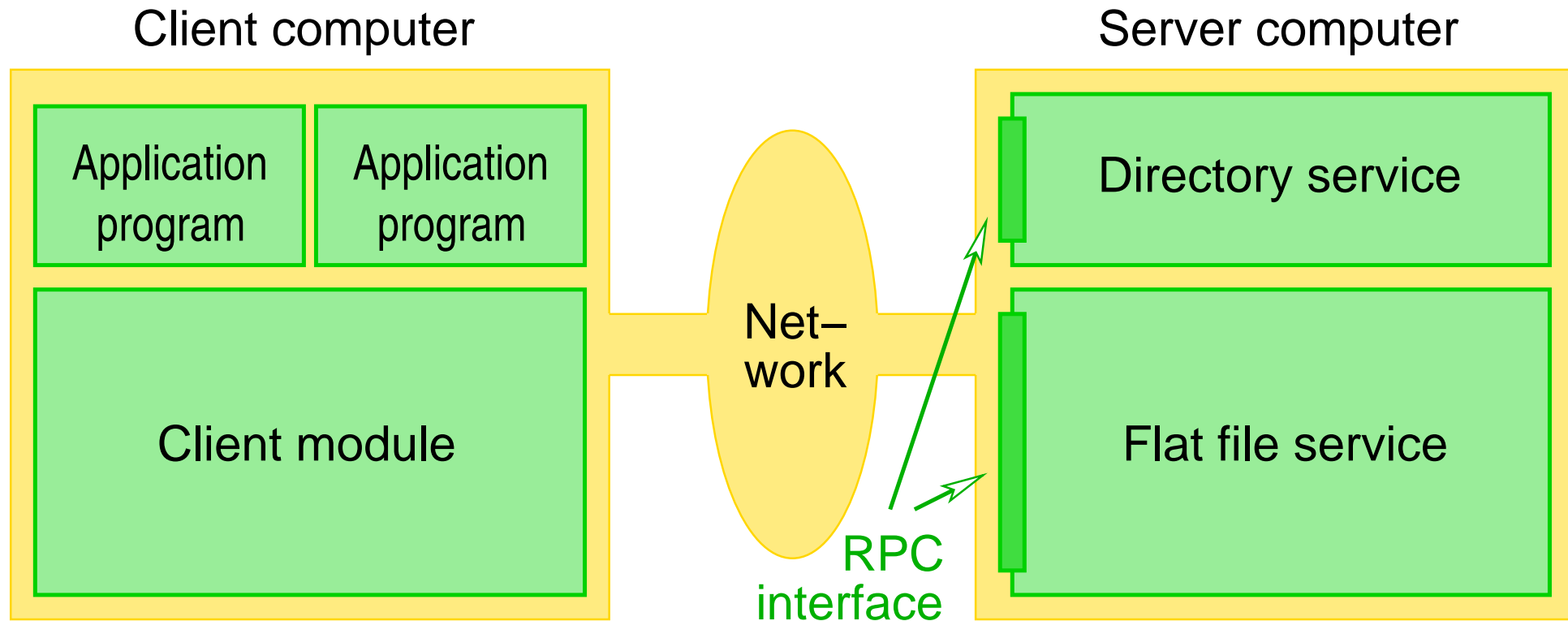
- ➔ Objective: support the sharing of information (files) in an **intranet**
 - ➔ in the Internet: WWW
- ➔ Allows applications to access remote files in the same way as local files
 - ➔ similar (or even better) performance and reliability
- ➔ Allows operation of diskless nodes
- ➔ Examples:
 - ➔ NFS (standard in the UNIX area)
 - ➔ AFS (goal: scalability), CIFS (Windows), CODA, xFS, ...



Requirements

- ➔ Transparency: access, location, mobility, performance and scaling transparency
- ➔ Concurrent file updates (e.g., locks)
- ➔ File replication (often: local caching)
- ➔ Heterogeneity of hardware and operating system
- ➔ Fault tolerance (especially in case of server failure)
 - ➔ often: at-least-once semantics + idempotent operations
 - ➔ advantageous: stateless server (easy reboot)
- ➔ Consistency (👉 9)
- ➔ Security (access control, authentication, encryption)
- ➔ Efficiency

Model Architecture of a Distributed File System



- ➔ **Tasks of the client module:**
 - ➔ emulation of the file interface of the local OS
 - ➔ if necessary, caching of files or file sections

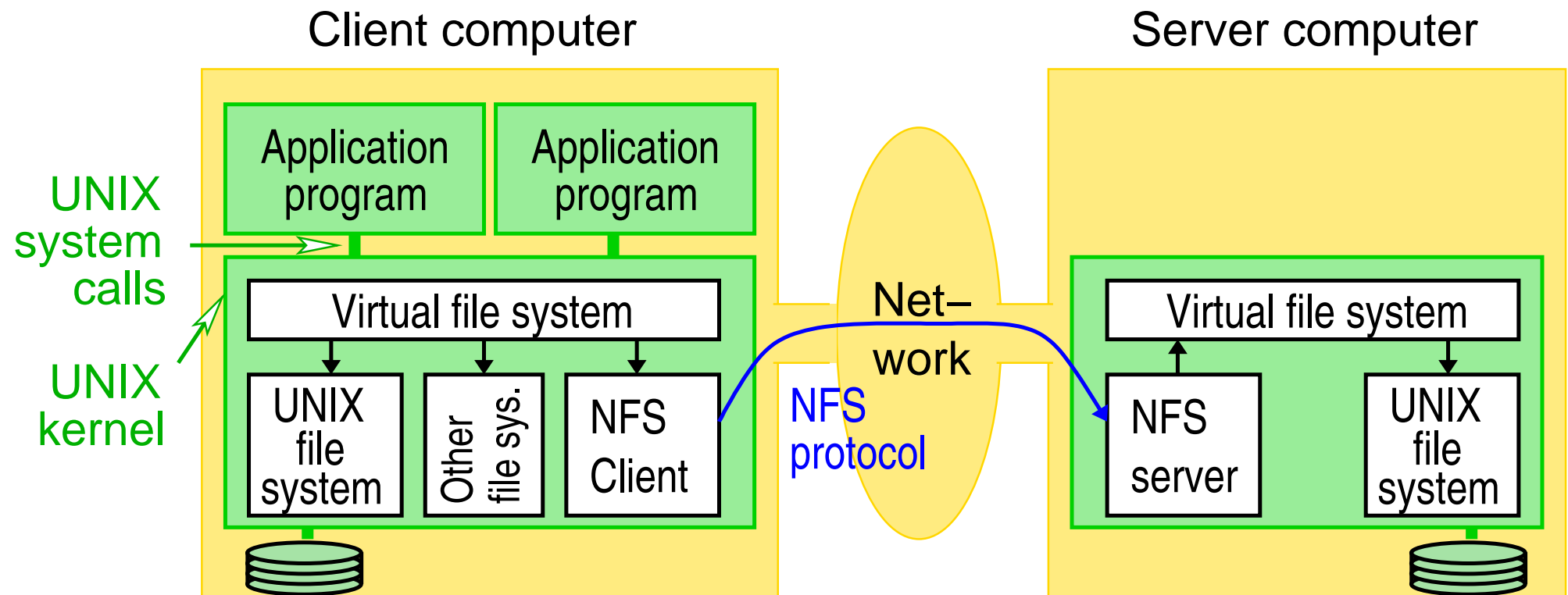


Model Architecture of a Distributed File System ...

- ➔ Flat file service:
 - ➔ provides idempotent access operations to files
 - ➔ e.g., *read*, *write*, *create*, *remove*, *getAttributes*, *setAttributes*
 - ➔ no *open* / *close*, no implicit file pointer
 - ➔ files are identified by UFIDs (Unique File IDs)
 - ➔ (long) integer IDs, can serve as capabilities
- ➔ Directory service:
 - ➔ maps file or path names to UFIDs
 - ➔ if necessary first authenticates the client and verifies its access rights
 - ➔ services for creating, deleting and modifying directories

10.2 Case Study: NFS

- ➔ Introduced in 1984 by Sun
- ➔ Open, OS independent protocol
- ➔ Architecture:





Access Control and Authentication

- ➔ NFS server is stateless (up to and including NFS3)
- ➔ UFID (file handle): essentially just the file system ID and i-node
 - ➔ not a capability
- ➔ Thus, access rights are checked with each request
 - ➔ by the RPC protocol
- ➔ Authentication usually only via user and group ID
 - ➔ extremely insecure!
- ➔ More possibilities in NFS3:
 - ➔ Diffie-Hellman key exchange (insecure)
 - ➔ Kerberos
- ➔ NFS4: secure RPC (RPCSEC_GSS)



Distributed Systems

Winter Term 2025/26

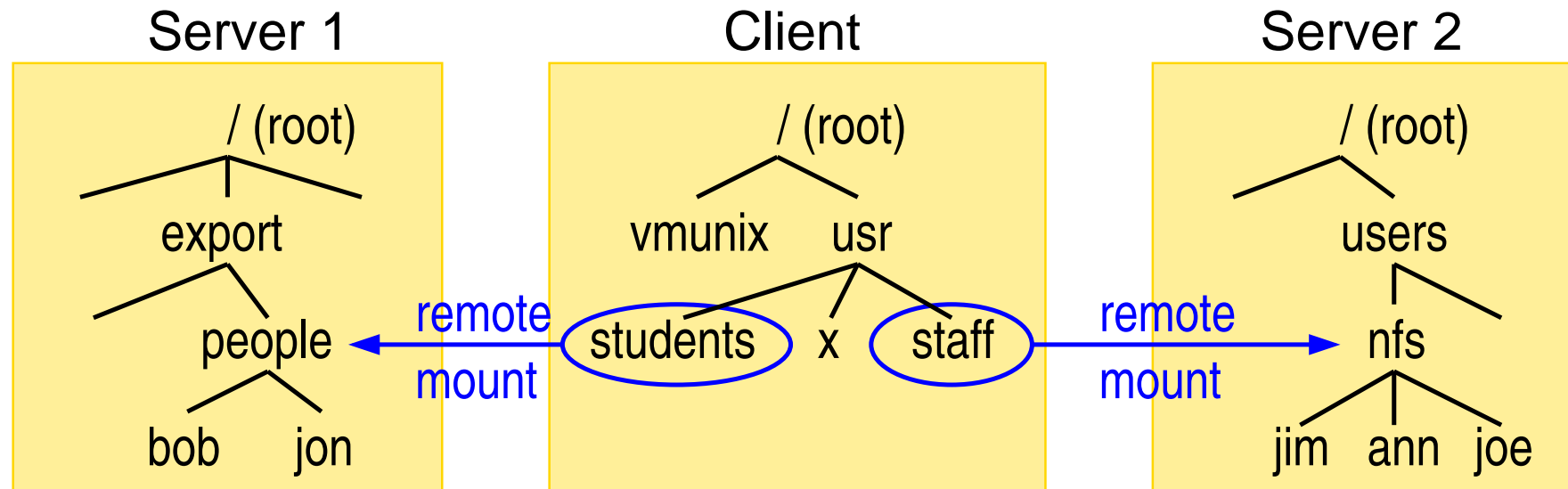
08.01.2026

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

Stand: January 8, 2026

Mount Service

- ➔ An NFS file system can be mounted in the local directory tree



- ➔ Collaboration of `mount` command in the client with the mount service of the NFS server
 - ➔ on request, the mount service provides file handles of the exported directories (for name resolution)



Translation of Pathnames

- ➔ Iteratively (NFS3): for each directory one request to NFS server
 - ➔ necessary because path can cross mount points
 - ➔ inefficiency is mitigated by client caching

Automounter

- ➔ Goal: set up an NFS mount only when it is accessed
 - ➔ better fault tolerance, load balancing is possible
- ➔ Automounter is local NFS server
 - ➔ thereby it sees the *lookup()*-requests of the client
- ➔ On request: set up the NFS mount and create a symbolic link to the mount point
- ➔ After prolonged inactivity: release the mount



Server Caching

- ➔ Traditional file caching in UNIX:
 - ➔ buffer in main memory for most recently used disk blocks
 - ➔ *read ahead*: sequential blocks are loaded into cache beforehand
 - ➔ *delayed write*: modified blocks only written back when space is needed; additionally every 30s by `sync`
- ➔ Server caching in NFS: two modes
 - ➔ *write through*: write requests are executed in the server cache and immediately also on disk
 - ➔ advantage: no data loss in case of server crash
 - ➔ *delayed write*: modified data will remain in the cache until a *commit* operation is executed (i.e. file is closed)
 - ➔ advantage: better performance if many write operations



Client Caching

- ➔ NFS client buffers the results of (among other things) *read* / *write* and *lookup* operations in a local cache
 - ➔ leads to consistency issues, since now multiple copies
- ➔ Client is responsible for maintaining consistency
- ➔ Timeliness of the cache entry is checked with each access
 - ➔ for that: compare whether the modification timestamp in the cache matches the modification timestamp on the server
 - ➔ in case of negative validation: cache entry is deleted
 - ➔ if validation is successful: cache entry is considered current for a certain time (3 - 30 s) without further checks
 - ➔ i.e. changes only become visible after a few seconds
 - ➔ compromise between consistency and efficiency



Client Caching ...

- ➔ Treatment of write operations:
 - ➔ file block is marked as *dirty* in the cache
 - ➔ marked blocks are sent asynchronously to the server:
 - ➔ when closing the file
 - ➔ at a `sync` operation on client machine
 - ➔ possibly more often by block-input/output-demons
- ➔ Demons also realize asynchronous operations for *read ahead* and *delayed write*
 - ➔ for performance optimization
- ➔ NFS does not guarantee real consistency of client caches



Distributed Systems

Winter Term 2025/26

11 Distributed Shared Memory



Contents

- ➔ Introduction
- ➔ Design alternatives

Literature

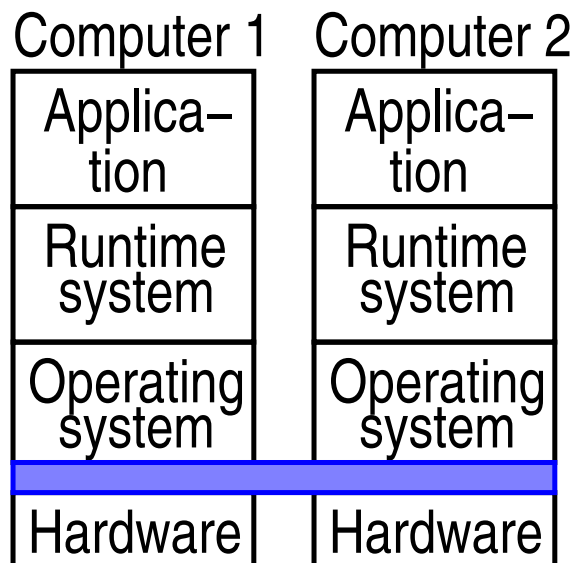
- ➔ Colouris, Dollimore, Kindberg: Kap. 16.1-16.3

11 Distributed Shared Memory ...

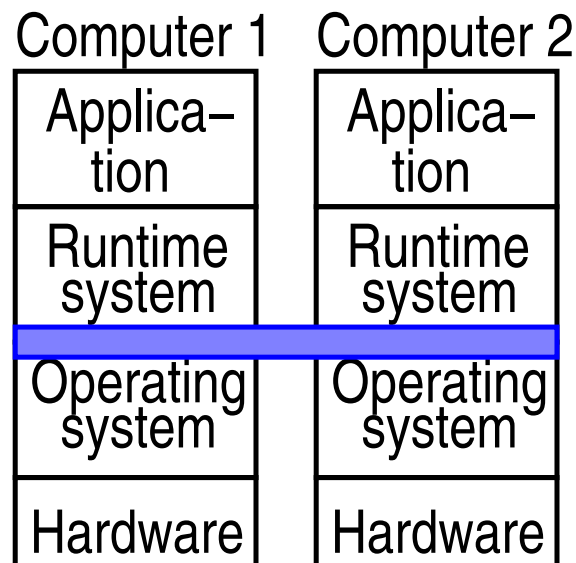


- ➔ Goal: shared memory in distributed systems
- ➔ Basic technique considered here:
 - ➔ page-based memory management on the nodes
 - ➔ on demand: loading pages over the network
 - ➔ if necessary replication of pages to increase performance
- ➔ Differentiation:

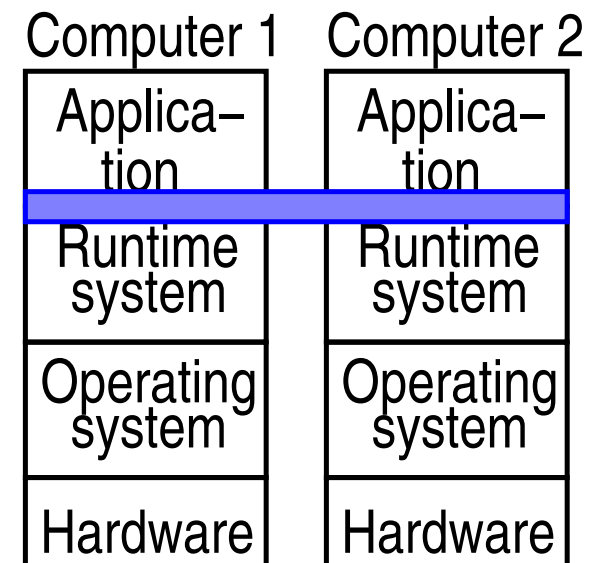
Hardware DSM: NUMA



Shared Virtual Memory



Middleware





Design alternatives

- ➔ Structure of the shared memory:
 - ➔ byte-oriented (distributed shared memory pages)
 - ➔ object-oriented (distributed shared objects)
 - ➔ e.g., Orca
 - ➔ immutable data (distributed shared container)
 - ➔ operations: read, add, remove
 - ➔ e.g., Linda Tuple Space, JavaSpaces
- ➔ Granularity (for page-based methods):
 - ➔ when changing a byte: transmission of entire page
 - ➔ with large pages: more efficient communication, less administrative effort, more false sharing



Design alternatives ...

- ➔ Consistency model: mostly sequential or release consistency
- ➔ Consistency protocol: usually local write protocol
 - ➔ i.e., writable memory page migrated to accessing process
 - ➔ with or without replication for read accesses
 - ➔ client initiated replication, i.e., reader requests copy
 - ➔ usually only one writer per page
 - ➔ mostly invalidation protocols (with push model)
 - ➔ update protocols only if write accesses can be buffered (e.g. with release consistency)



Design alternatives ...

- ➔ Management of copies
 - ➔ mostly: at any time either multiple readers or one writer
 - ➔ each page has an owner
 - ➔ writer or one of the readers (last writer)
 - ➔ manages a list of processes with copies of the page
 - ➔ before write access: process requests current copy
- ➔ Finding the owner of a page:
 - ➔ central manager
 - ➔ manages owners, forwards requests
 - ➔ distributed manager with fixed distribution
 - ➔ fixed mapping: page → manager



Design alternatives ...

- ➔ Finding the owner of a page ...:
 - ➔ multicast instead of manager
 - ➔ problem: concurrent requests
 - ➔ solution: totally ordered multicast, vector time stamps
 - ➔ dynamically distributed manager
 - ➔ every process knows a likely owner
 - ➔ this node forwards the request if necessary
 - ➔ the likely owner is updated,
 - ➔ when a process transfers the ownership property
 - ➔ upon receipt of an invalidation message
 - ➔ upon receipt of a requested read-only page
 - ➔ when a request is forwarded (to the requestor)



Design alternatives ...

- ➔ Problems: e.g., thrashing, especially due to false sharing
 - ➔ simple remedy:
 - ➔ a page can be migrated again only after a certain period of time
 - ➔ TreadMarks: multiple writer protocol
 - ➔ release consistency; when released, only the changed parts of the page are transferred
 - ➔ changes are then “merged”
 - ➔ in case of conflicts: result is non-deterministic



Distributed Systems

Winter Term 2025/26

12 Summary, Important Topics



1. Introduction

- ➔ Definition of a distributed system
- ➔ **Features / challenges of distributed systems**
- ➔ Architecture models: client/server, n-tier

2. Middleware

- ➔ Tasks of the middleware
- ➔ Communication-oriented and application-oriented middleware
- ➔ **Implementation of remote calls (proxy pattern)**

3. Distributed Programming

- ➔ **Approach to create a Java RMI application**
- ➔ **Programming of a Java RMI server and client**



4. Name Services

- ➔ Task and purpose of a name service

5. Process Management

- ➔ Graph partitioning, list scheduling, code migration

6. Time and Global State

- ➔ **Synchronization of physical clocks**
- ➔ **Lamport's happened-before relation (causality relation)**
- ➔ **Lamport and vector clocks**
- ➔ **Consistent cuts, Chandy/Lamport algorithm**



7. Fault Tolerance

- ➔ **Failure models**
- ➔ Physical redundancy, agreement
- ➔ Recovery

8. Coordination

- ➔ Election algorithms
- ➔ **Mutual exclusion (centralized, Ricart/Agrawala, ring)**
- ➔ **Multicast (reliability, order)**
- ➔ Transactions



9. Replication and Consistency

- ➔ **Sequential consistency, release consistency**
- ➔ **Distribution protocols**
- ➔ **Consistency protocols (primary-based, quorum-based)**

10. Distributed File Systems

11. Distributed Shared Memory