

I. Introduction

Distributed Systems (DS) – what's it all about?

[Umar 1995]: A Distributed Computing System (DCS) is a collection of autonomous computers interconnected through a communication network ... ◆

- ▷ 'Systems' aspects of computer science
- ▷ Hardware, OS, networks and software architecture

[Umar 1995]:... Technically, the computers do not share main memory so that the information cannot be transferred through global variables. The information (knowledge) between the computers is exchanged only through messages over a network. ◆

- Algorithms in a '*no global information paradigm*'
- Parallel programming using *interaction mechanisms*

Internet vs. Distributed Systems

- ▷ Technical infrastructure development is a pre-requisite
 - 43 Million in 1999 / > 1 Billion DNS hosts in 2019
 - IoT Prediction: 36 (75) Billion connected devices in 2021 (2025)

[Coulouris et al. 1995]: A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software which enables computers to coordinate activities and to share the resources of the system.



- ▶ There is more to a distributed system than infrastructure:
 - software layers that allow for controlled information exchange and **coordination** among applications on distributed nodes
 - mechanisms to get work towards a common goal done
 - awareness of different *local vs. remote views* on the same DS

Why and how to build and use DS?

► Why? *Lots of motivations and reasons . . .*

I.2

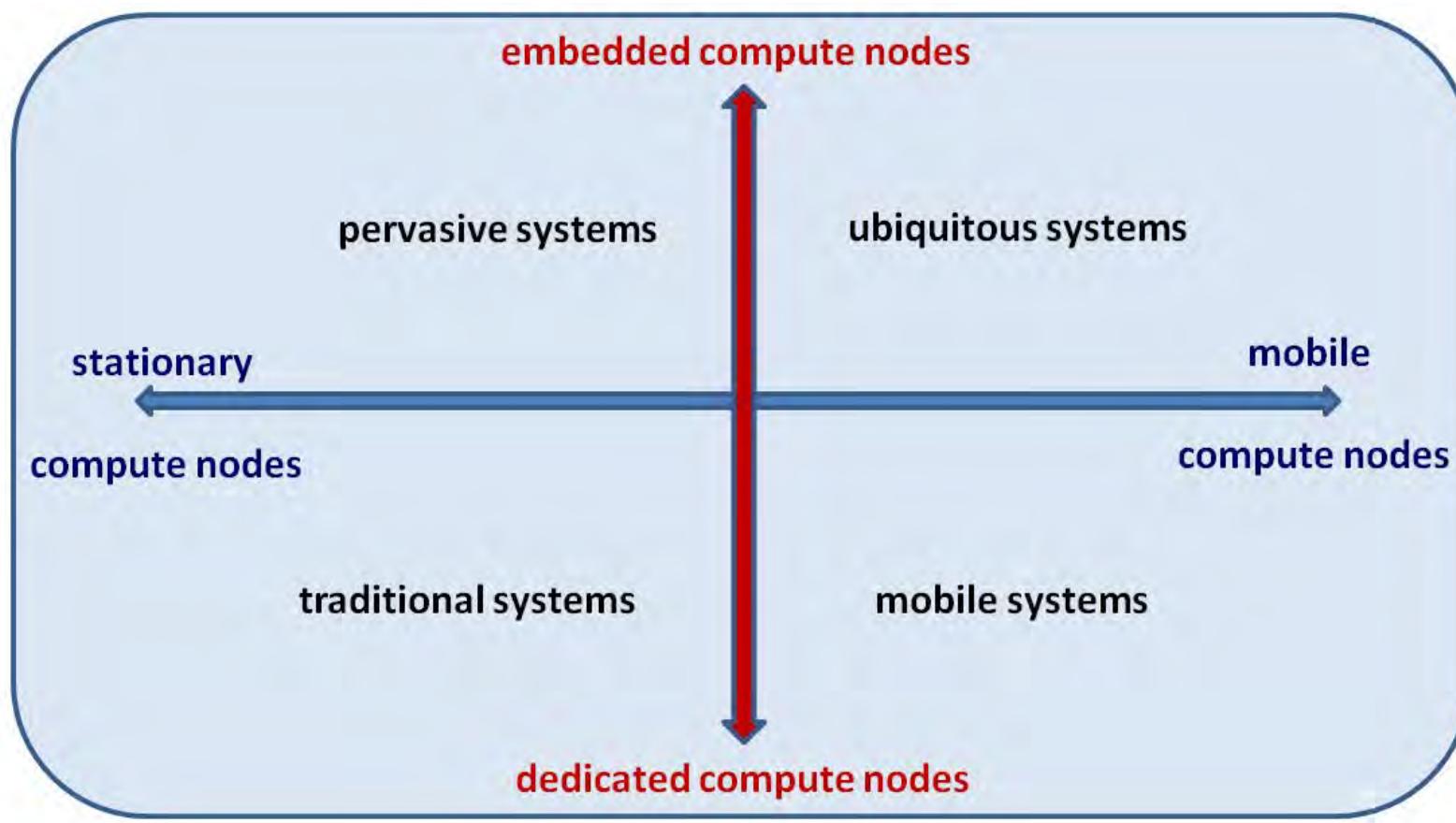
- ▷ Technical development allows for low-cost connectivity
- ▷ 'Real mobile' computing is feasible and affordable
 - ⇒ Connected Smart Environments on the rise
- ▶ Business demands for some decades
- ▶ Everyday life demands in the consumer sector more recent

◀ How? *That part is a bit tricky . . .*

I.3

- ◀ All problems of single computers are present in DS, too
- ◀ DS do not work without infrastructure: networks, services, . . .
 - ⇒ without connectivity, everything remote is not available
- ◀ Additional problems due to the *very nature of distribution*
- ◀ Theoretically, some problems cannot be solved completely at all

I.1 Different flavors of distributed systems

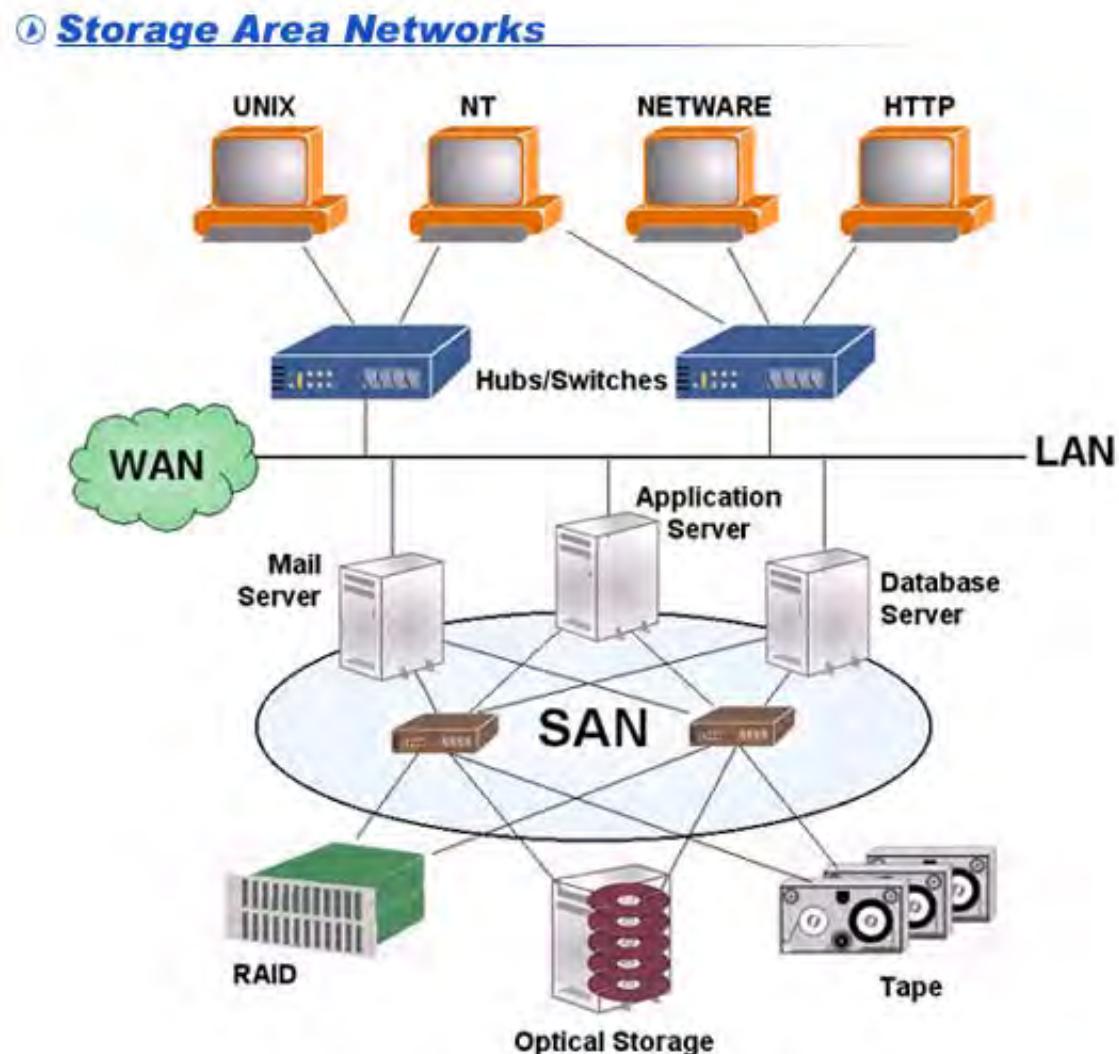


- computers connected by networks are 'traditional' today
- mobile systems (smartphones, tablets, gadgets) are common place
- *ubiquitous and pervasive computing* gain ever more importance

Example: Storage Area Networks as low-level DS

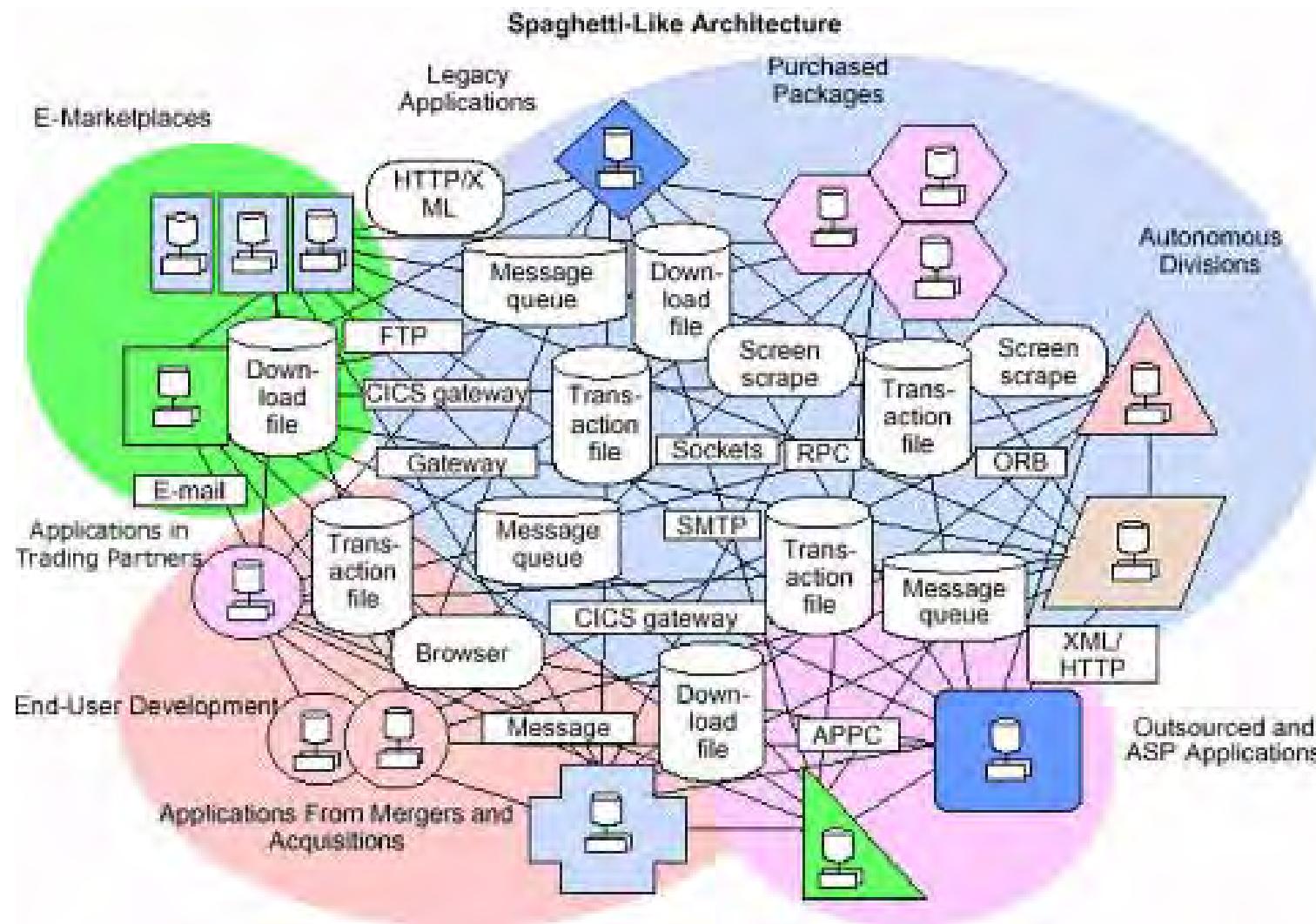
SANs abstract from:

- network details
 - heterogenous hardware
 - heterogenous OS
 - server/storage correlation
- and allow for common
- access rules
 - security policies
 - backup strategies



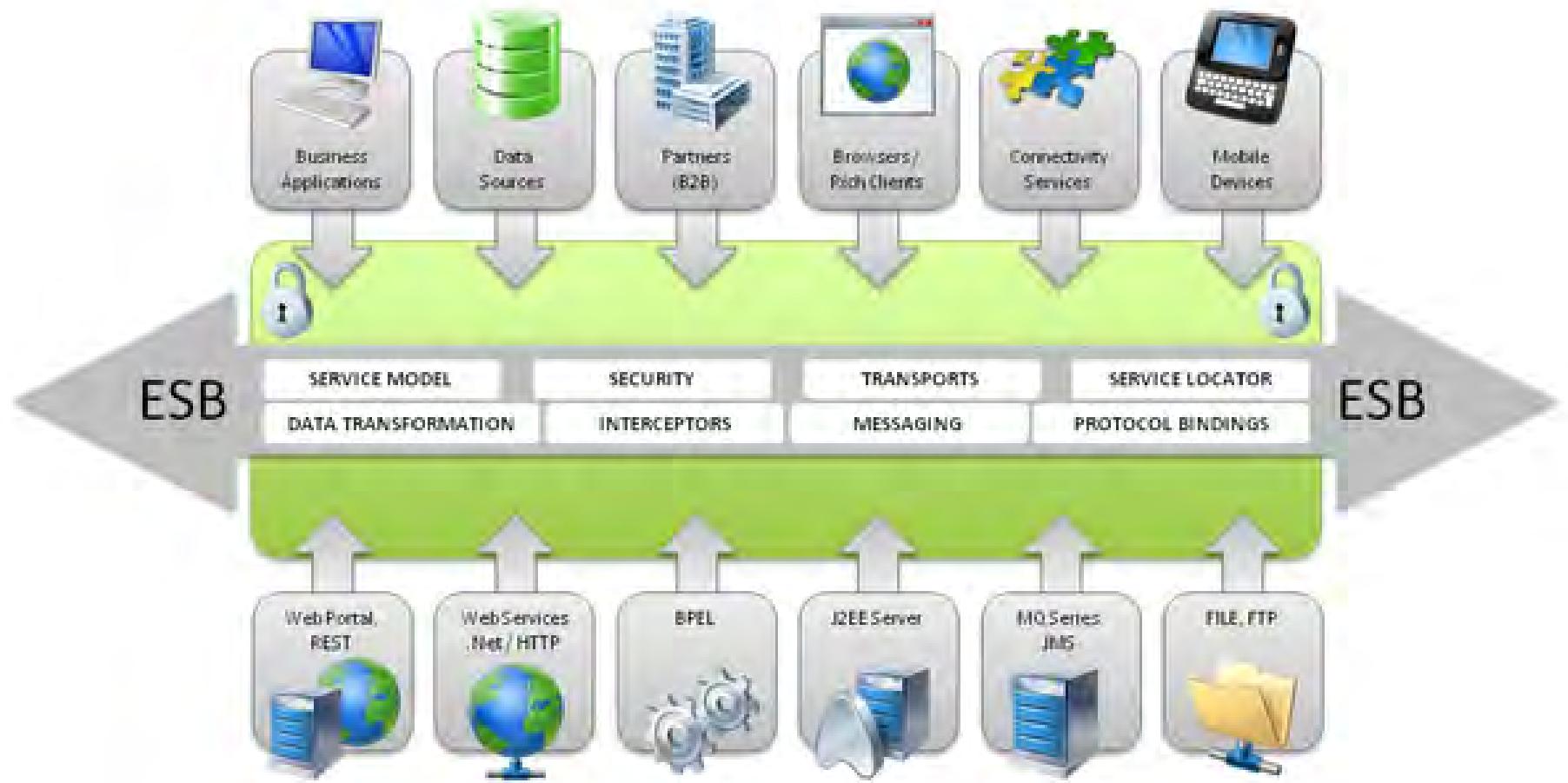
Source: allSAN Report 2001 Copyright © 2000 allSAN.com Inc

Example: Chaotic Enterprise Application Landscape



- How to manage such a system? Software Architecture matters!

Flavour 1: Structured EAI – Enterprise Service Bus



- Unified channel for linking heterogeneous systems/technologies
- Single actor has global control \implies '**centralized**' distributed
- Well-defined interface to define and implement general policies

Flavour 2: DS Potential for B2B Integration



- Dependencies between lots of businesses – maybe even competitors
- No one has global control \implies '**decentralized**' distributed
- Uses standardized protocols, needs trusted parties, . . .

Flavour 3: Peer-2-Peer Networks for Transactions

Decentralized Ledger



- Distributes information, capabilities and 'trust'
- No global control \implies '**really decentralized**' distributed
- Uses standardized protocols, needs **no** trusted parties

Distributed vs. Decentralized Systems

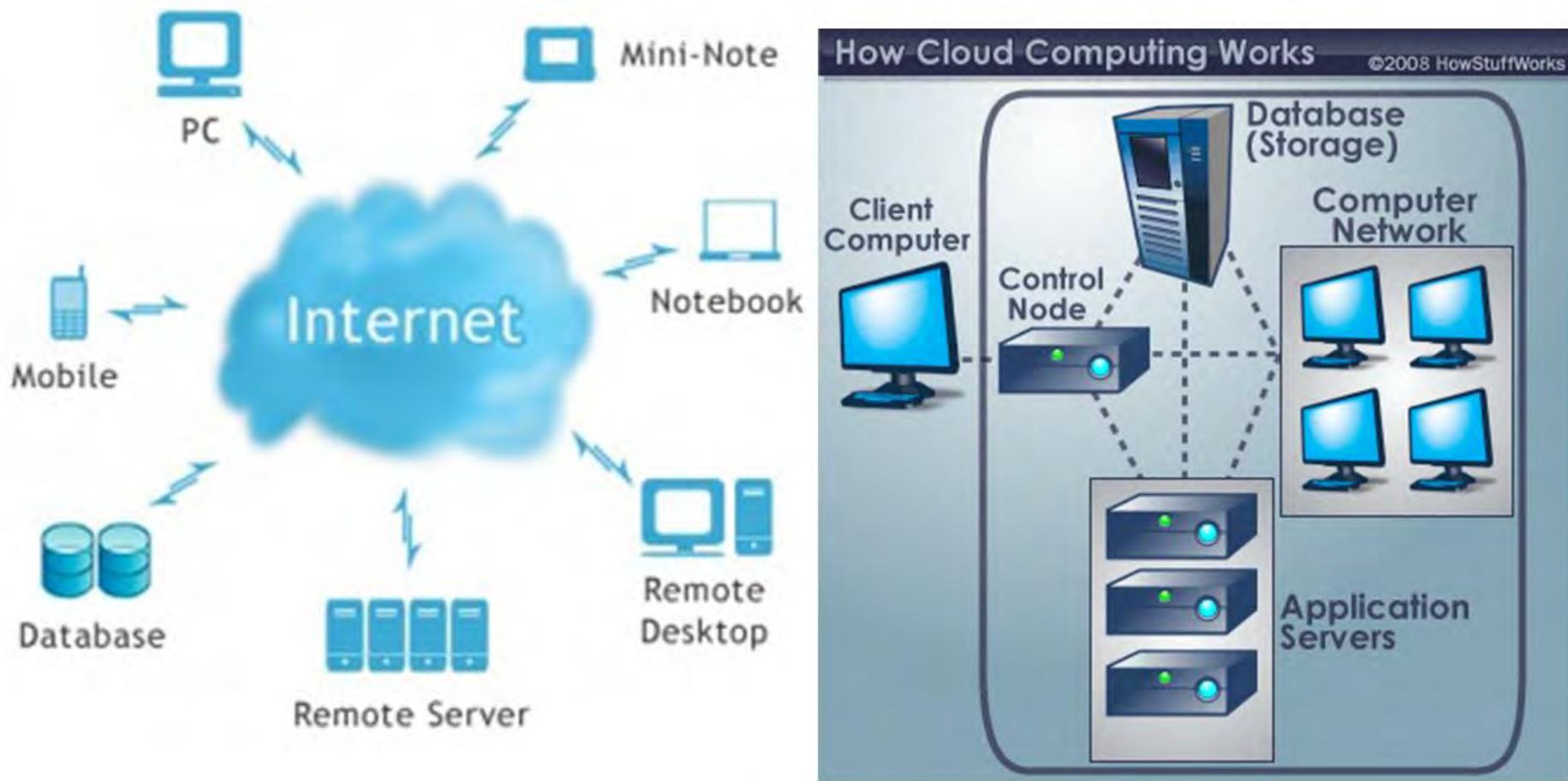
- * distributed: matter of location, technology and architecture
- * (de-)centralized: matter of control and organization

[Wikipedia]: A decentralised system in systems theory is a system in which lower level components operate on local information to accomplish global goals. The global pattern of behaviour is an emergent property of dynamical mechanisms that act upon local components, such as indirect communication, rather than the result of a central ordering influence of a centralised system. ♦

Examples:

- World wide company network including clouds etc. \implies centralized
- Distributed Ledger like, e.g. Blockchain network \implies decentralized
- P2P network for file sharing \implies decentralized

Clouds – A Modern Prototype for DS Abstraction

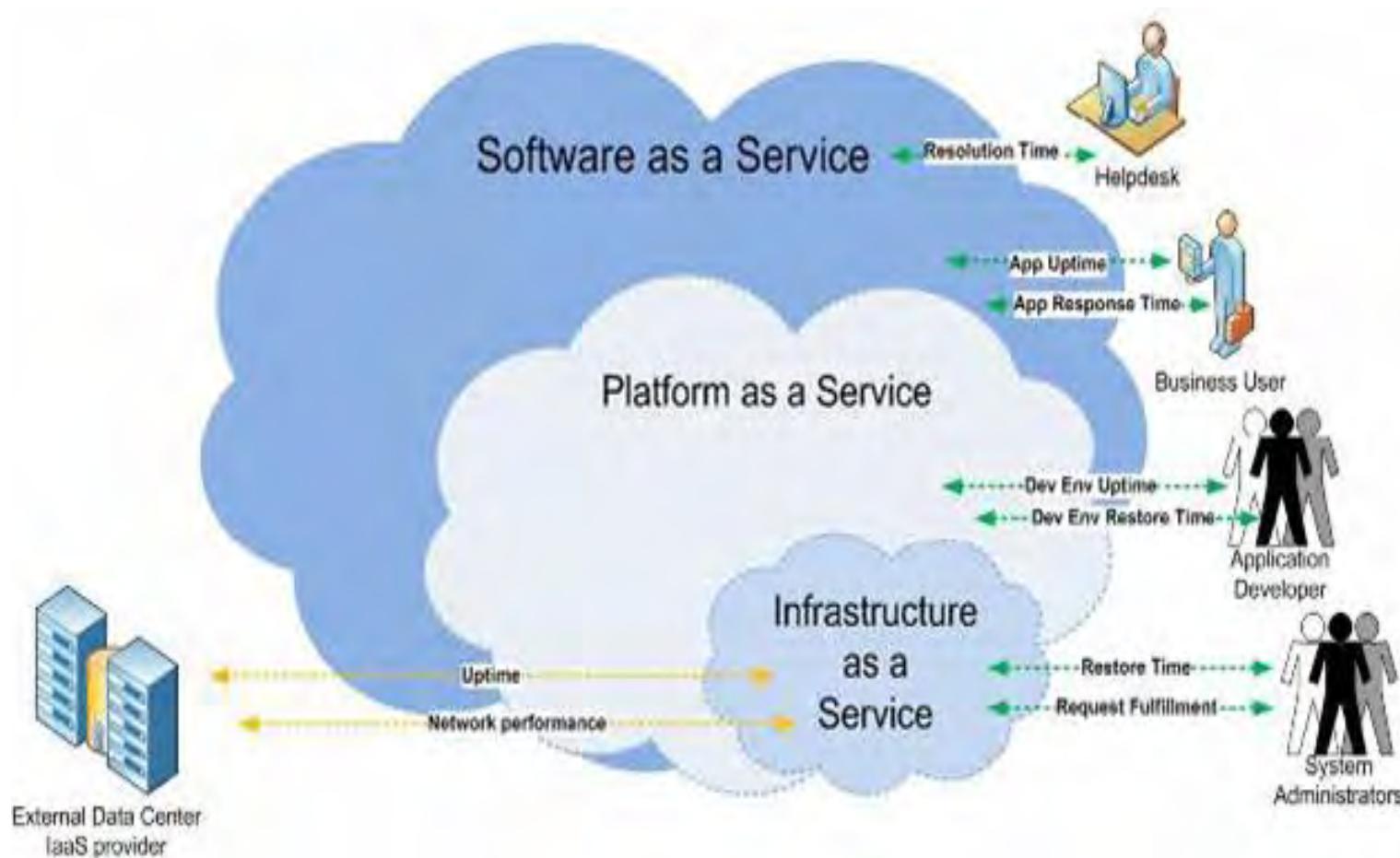


- **Cloud Usage** allows for a unified working environment despite the heterogeneity of the underlying infrastructure.
- **Cloud Implementation** details require state-of-the-art techniques w.r.t. reliability, scalability, access rights etc.

left

right

Cloud Computing – Different Abstraction Layers



Hardware **Host** → **Build** → **Consume** Software

- The lower the level, the higher the amount of own work
- The higher the level, the lower the grades of flexibility

Ubiquitous Computing - The Future?

► Goals and demands:

- mediated interaction between humans and reality using a 'computer' should be as 'normal' as the direct interaction.
- bridging the gap between the real and the virtual world(s)

⇒

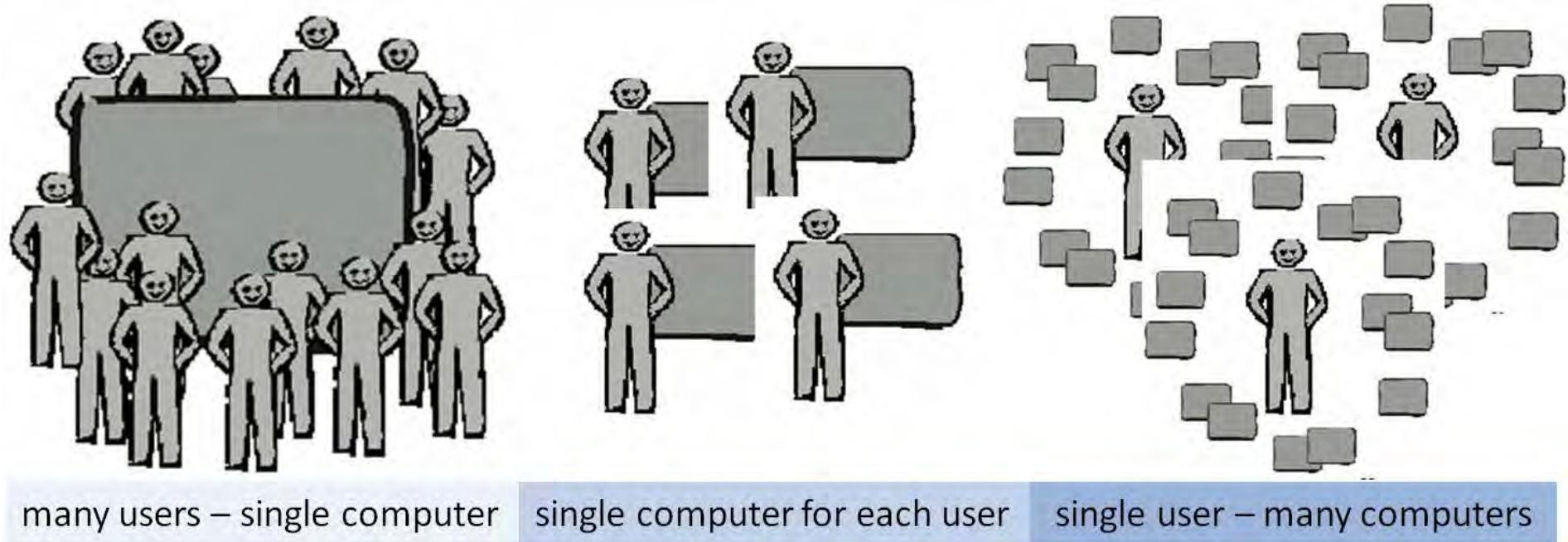
- * computers and computer support (more or less) everywhere
- * hard problem: find suitable, new modes of interaction

► Computers vanish from the conscious perception

The Concept of computers as things that you walk up to, sit in front of and turn on will go away. In fact, our goal is to make the computer disappear. [...] . The Internet is the big event of the decade [...]. We'll spend the next 10 years making the Net work as it should, making it ubiquitous. [F. Casanova, Apple, 2003]

Claim: Three stages of computer utilization

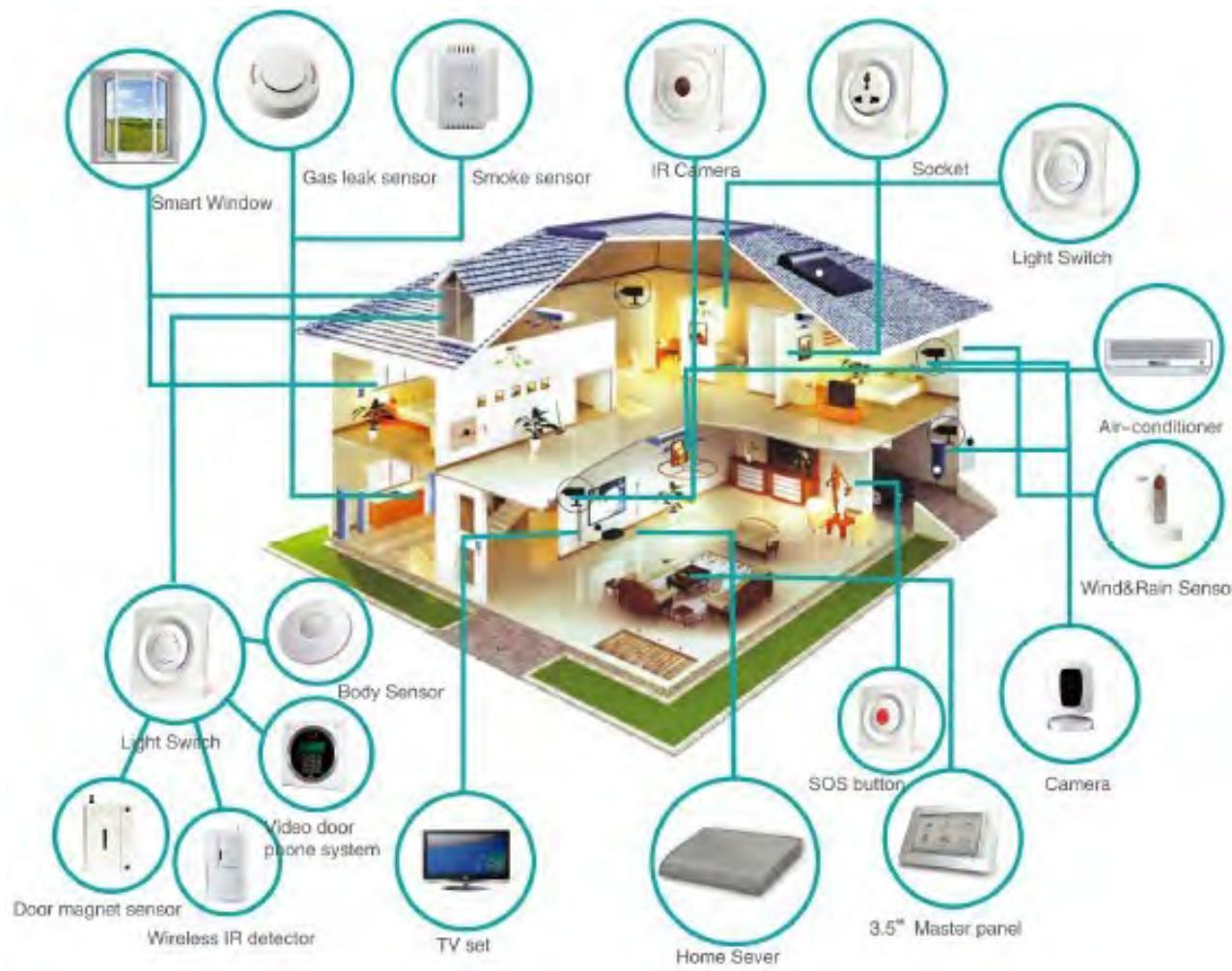
Mark
Weiser
1952-
1999



► Vision from > 30 years ago: [M. Weiser: Scientific American 1991]

Computers will act like books, windows, walks around the block, phone calls to relatives. They won't replace these, but augment them, make them easier, more fun. Dwelling with computers, they become part of the informing environment, like weather, like street sounds.

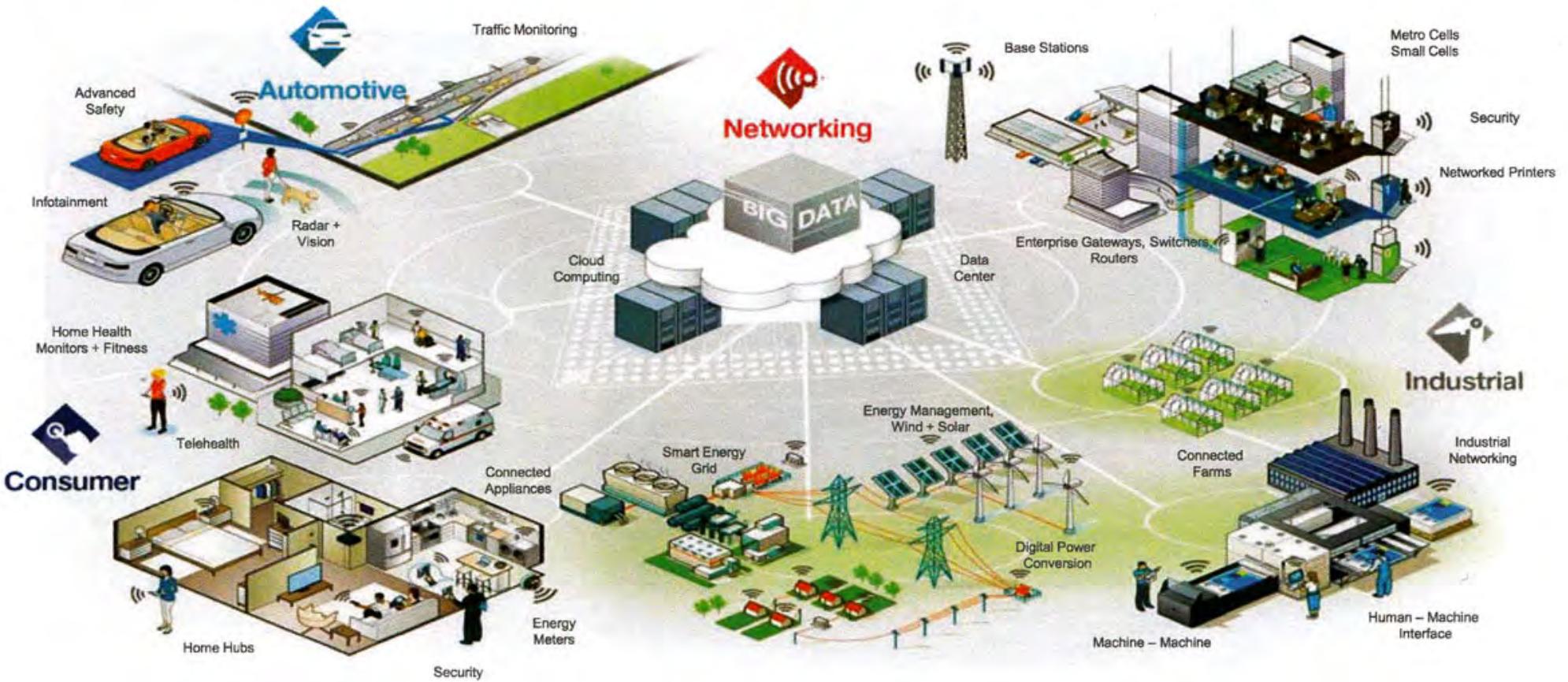
Example: Pervasive Computing at work



Applications: Energy management, security, . . . , more comfort
...assisted living, esp. for the elderly/handicapped

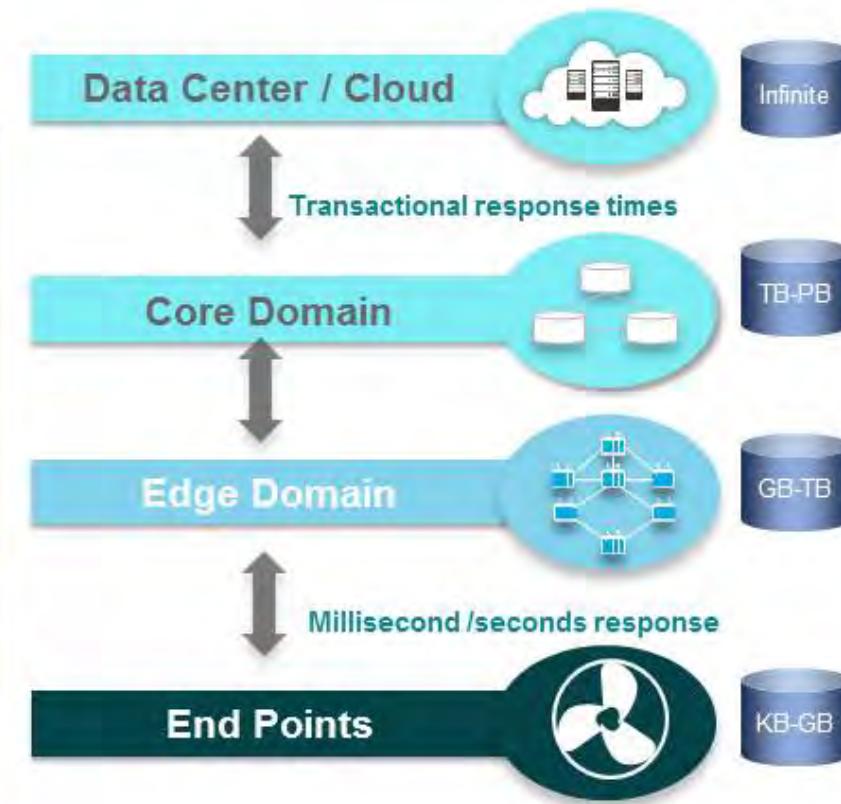
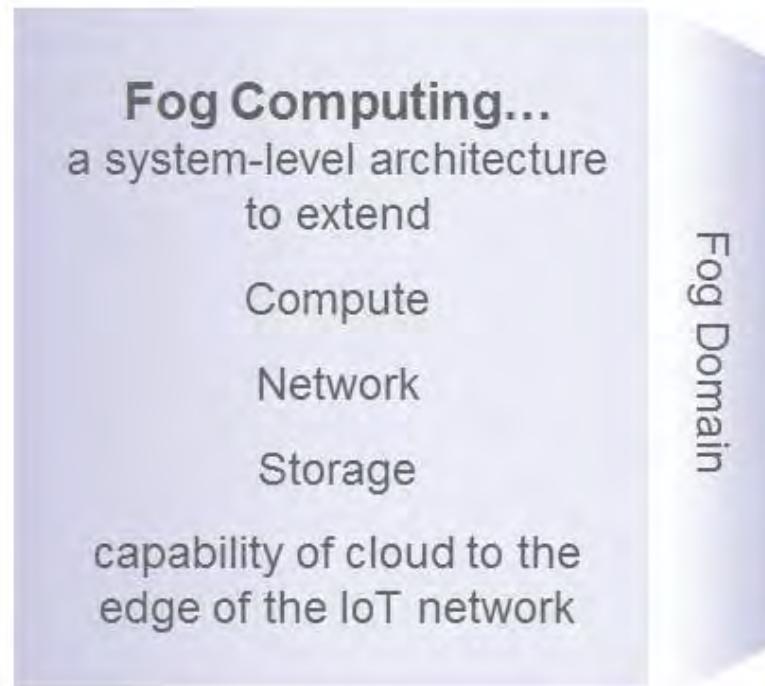
The 'final step': Smart Internet of Things (IoT)?

The Internet of Things



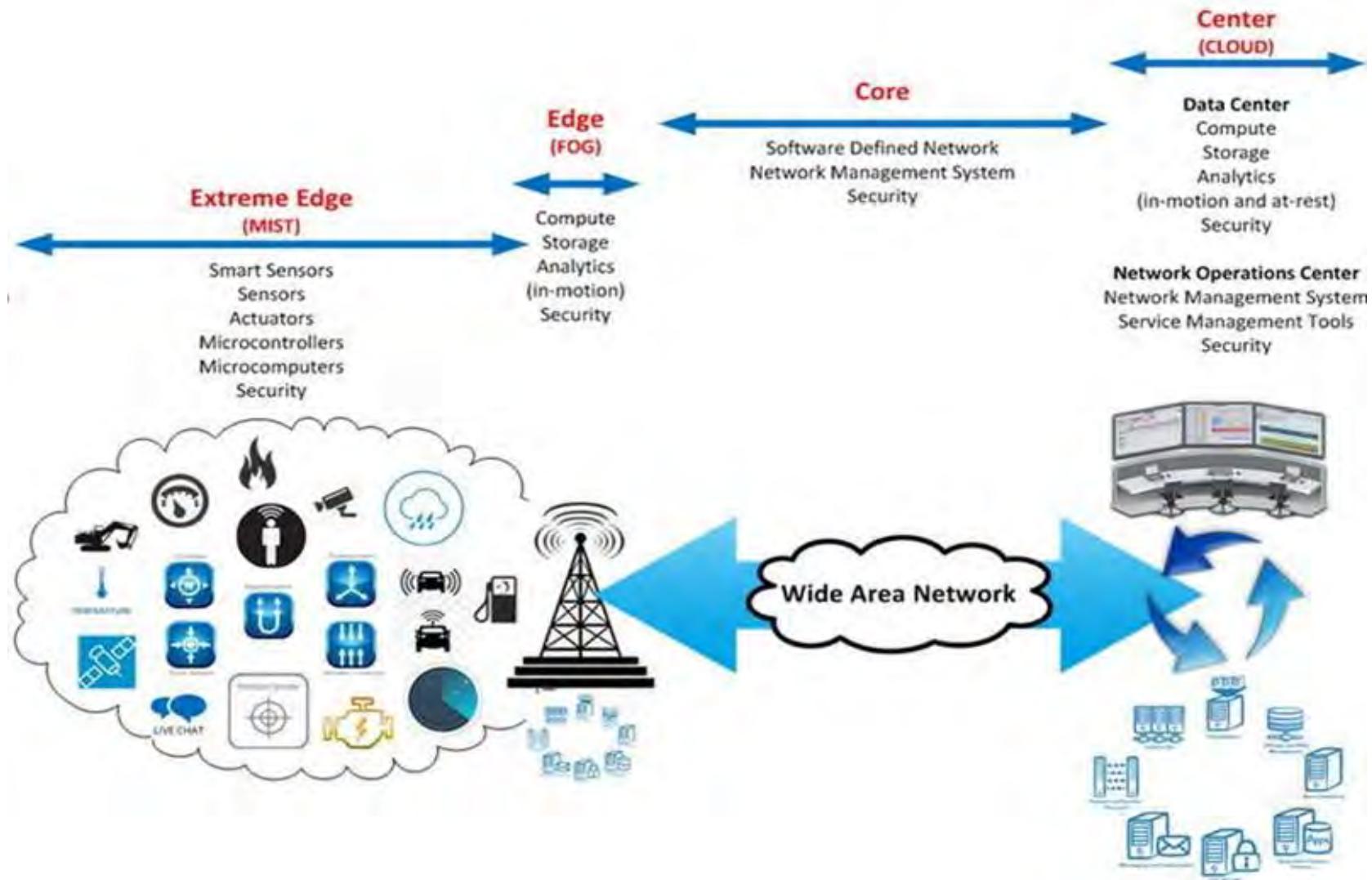
- Connectivity: Sensors – Ad-hoc Networks – LANs – WANs
- One single, global system? How to ensure reliable infrastructure?

Fog-/Edge-Computing = Cloud + IoT?



- **Edge Computing:** Use compute power of devices like Routers close to the network edges
- Reduce network traffic through data preprocessing and filtering

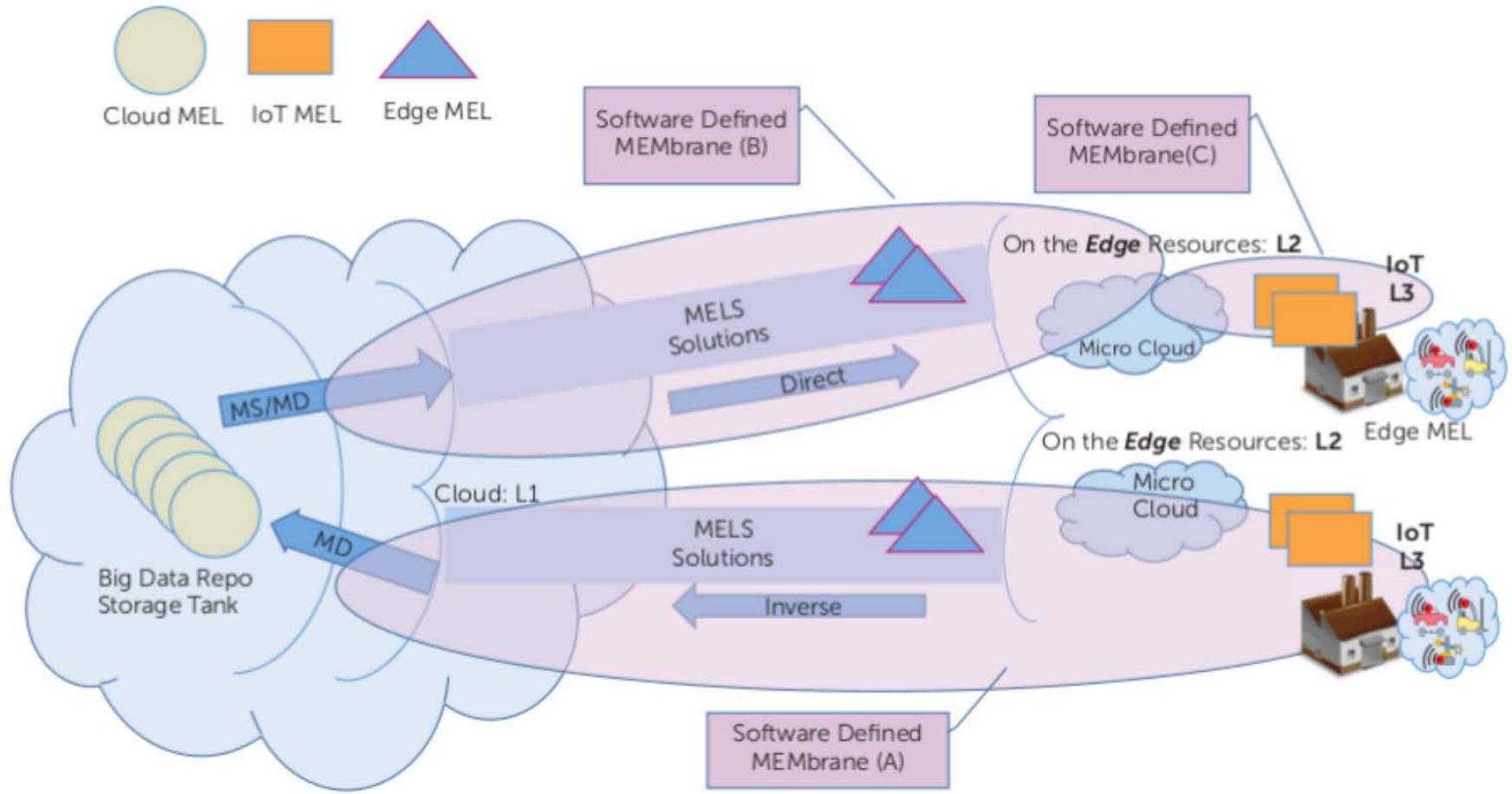
Where to Place the Compute-Load?



Trade-Off:

- Off-loading to the edge or even to the cloud vs.
- Perform work on local device, router, ...

Osmotic Computing as the connecting paradigm



Modeling metaphor:

- Software-defined 'Membranes' for shifting load(s)
- Virtualisation of resources, scheduling, access, ...

I.2 Incentives for using Distributed Systems

- *Most of our civilization's infrastructure does not work without lots of distributed systems any more, e.g.,*
 - * Energy: production and delivery including peak-load handling
 - * Traffic: traffic lights, train and airplane supervision
 - *Worldwide information exchange does not work without DS, e.g.,*
 - * telecommunication, news, internet usage, media streaming, ...
 - *Global business in its current form has been enabled by DS, e.g.,*
 - * Industrial production and logistics inside a single as well as among different companies
 - * Financial transactions, stock markets, commodity exchanges
- The question whether to use distributed systems or not is long gone!**

What kind of benefits have led to this situation?

- ▶ *Real life needs* made distribution the natural way to go as soon as it became possible due to
 - **technical** reasons: emergence of high-speed communication networks and compute power
 - **economic** reasons: off-the-shelf nodes and cheaper networks
- ▶ *Data sharing* eases the organization of work among divisions and allows for the implementation of network-supported workflows.
- ▶ *Resource sharing* like sharing special hardware, backup technology, software licenses etc. saves costs.
- ▶ Change management (should be) 'much easier' to handle:
 - * planned growth, integration of new branches and outsourcing

Technical reasons for Distributed Systems

1. Many aspects of our world are inherently distributed

- ▷ modern industrial production, financial markets, ...
 - ▶ New applications are made possible through distributed systems:
E-Commerce and *M-Commerce*

see
I-23 ff.

2. Many important systems require distribution even if the application is not distributed in order to become reliable:

- * Secure storage for important data even in the case of disasters requires geographically distributed data and compute centers \implies **Replication**
 - * Basic services like Email, internet name services etc. can only be made reliable via distributed replication utilizing different computing as well as network facilities \implies **Redundancy**

New Applications: B2B and B2C E-services

If you can imagine a way of electronically delivering something of value to a customer that will solve some problem or provide some usefulness to make their life easier, you have a viable example of an e-service. [Th. F. Stafford; CACM 6/03, pg. 27]

- ▶ **E-Services:** available on an electronic platform, e.g.,
 - Logistics: commodity markets, constant supply, ... (B2B)
 - Online Shopping: browsing, buying, payment (B2C)
 - Online Banking: transactions (B2B/B2C)
 - E-Government: support for filing an application, forms download
⇒ *Well-known, but now: always everywhere available*
- ▶ **Mobile Services:** available on smart phones and tablets
⇒ *Extend E-services based on mobility aspects*

see
I-24

Two flavors of Mobile Services

► **Location-independent Services:** E-services everywhere

- always and everywhere online
- access to information, documents and compute facilities

Examples: Email, insurance field crew, traveling salesman

⇒ *Comfortable, wider applicable mobile version of services.*

► **Location-based Services:** Completely new E-services

Usage of specific information based on current location and time

- * car traffic: traffic jams, parking situation, ...
 - * interchange facilities and timetables when leaving a train
 - * information about 'interesting' events, e.g., when coming into an airport or a trade-fair premise.
 - * mobile gaming and social networking: 'Who's around?'
- ⇒ *New application domains for E-Services.*

I.3 Problems concerning Distributed Systems

Having so much benefits, there also have to be some downsides with distributed systems?

- ▶ Working in a well-functioning distributed system is fun.
- ◀ To be at the mercy of a malfunctioning distributed system you rely on for an important task is a nightmare!
- ◀ Building a distributed system that works under good to optimal conditions is (more or less) easy.
- ◀ Designing and building a distributed system that works also under worst-case assumptions and arbitrary failures is impossible.

Designing and building a distributed system that delivers at least basic services and avoids information loss even in the presence of a moderate number of failures is a challenge.

That is the reason why we teach 'Distributed Systems' !

Characteristics that require specific consideration

- ◀ DS rely on dynamic communication networks
 - unpredictable stability, message loss etc.
 - varying network load may cause a wide variety of response times
 - connections are world-wide which implies legal uncertainty
- ◀ DS consist of *heterogenous* nodes and interconnects
 - mix of different hardware, operating systems and software
 - interaction has to be based on *standardized protocols*, i.e., *rules* for data representations (= *formats*) and interaction steps required to exchange data in well-defined language using a clearly understood common syntax and semantics, e.g.,
 - * XML/JSON/... documents with self-describing formats
 - * handshake including requests, replies and acknowledgements
 - data may have to be transformed before/after transport

A General characteristic that impedes DS design

[Lamport]: A distributed system is 'one on which I cannot get any work done because some machine I have never heard of has crashed.' ♦

- ◀ DS depend on lots of infrastructure
 - ⇒ a system may be as vulnerable as the weakest link of a chain
- ◀ DS consist of many nodes
 - ⇒ there is a high probability of failures in single nodes
- ◀ DS do not fail as a whole, but are prone to **partial failures**
 - ⇒ The more essential components, the more possibilities for failure
 - ⇒ High risk for *Inconsistencies* on a global scale

General Rule: DO and Don't for DS Design

[Mullender]: A distributed (operating) system must not have any single point of failure – no single part failing should bring the whole system down. ♦

- ▶ **OR-Distribution** using redundancy on hardware and software components on all levels \implies robust system
- ◀ **AND-Distribution** (much cheaper) \implies highly vulnerable system

$$\text{Overall working system} = \bigwedge_{\text{Nodes}} \text{Working Nodes}$$

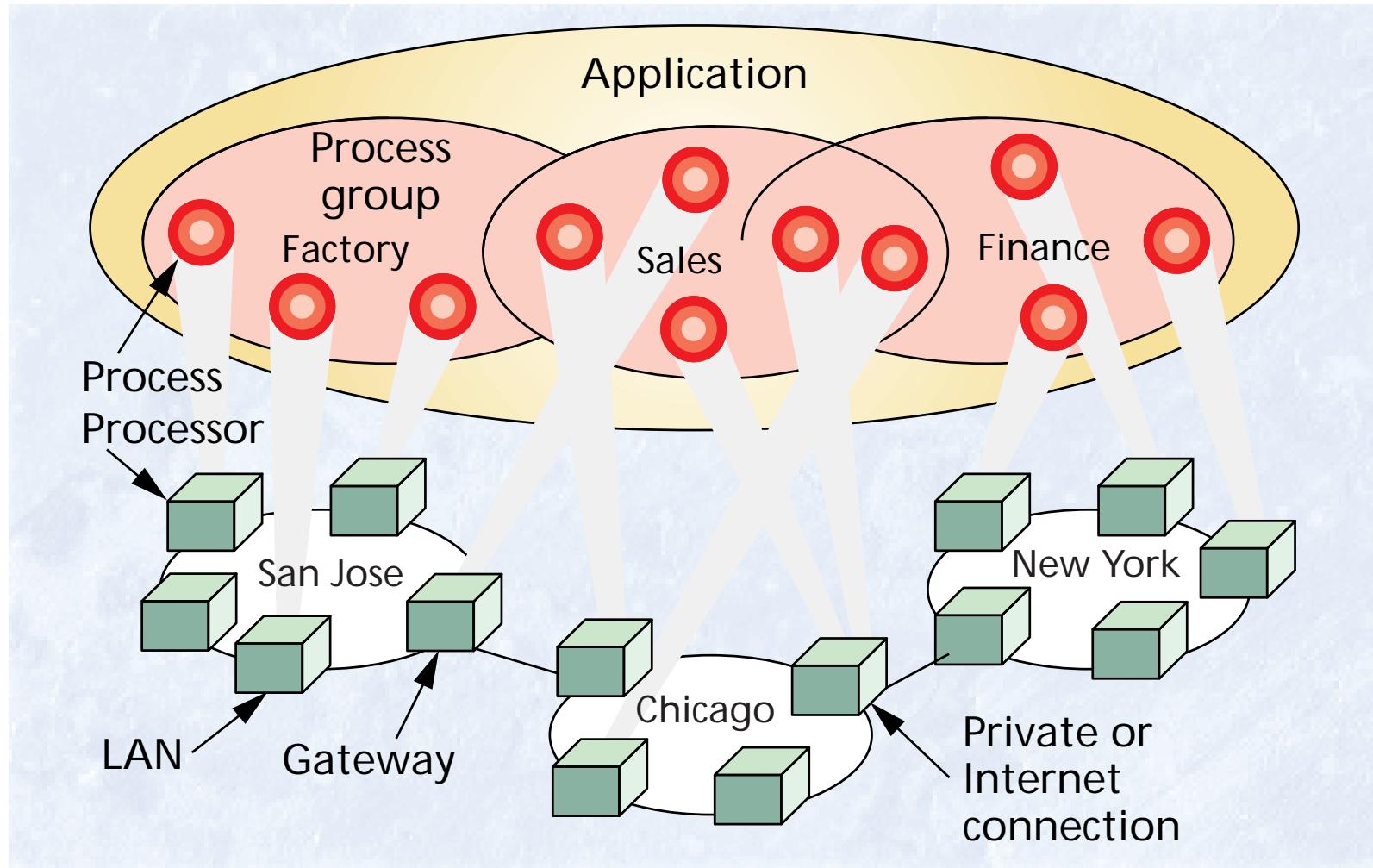
General characteristics that make building DS hard

[Shingal et al. 1994]: A distributed system consists of autonomous computers without any shared memory and without a global clock. Computers communicate using message-passing on a communication network with arbitrary delays. ◆

- ◀ programming a distributed system with information exchange requires the usage of message-passing primitives at least at the lower abstraction levels
- ◀ DS comprise a *local vs. remote view*: sharing
 - data may act as the basis for *data leaks* and data loss
 - resources may lead to *resource abuse* or *unauthorized usage*
- ◀ **No global information paradigm:**
 - ◀ *Consensus* among nodes of a DS is not always achievable
 - ◀ *physical clocks* on local nodes may be useless as time reference

⇒ Approximation requires distributed algorithms

Example: A simple consistency problem – 1

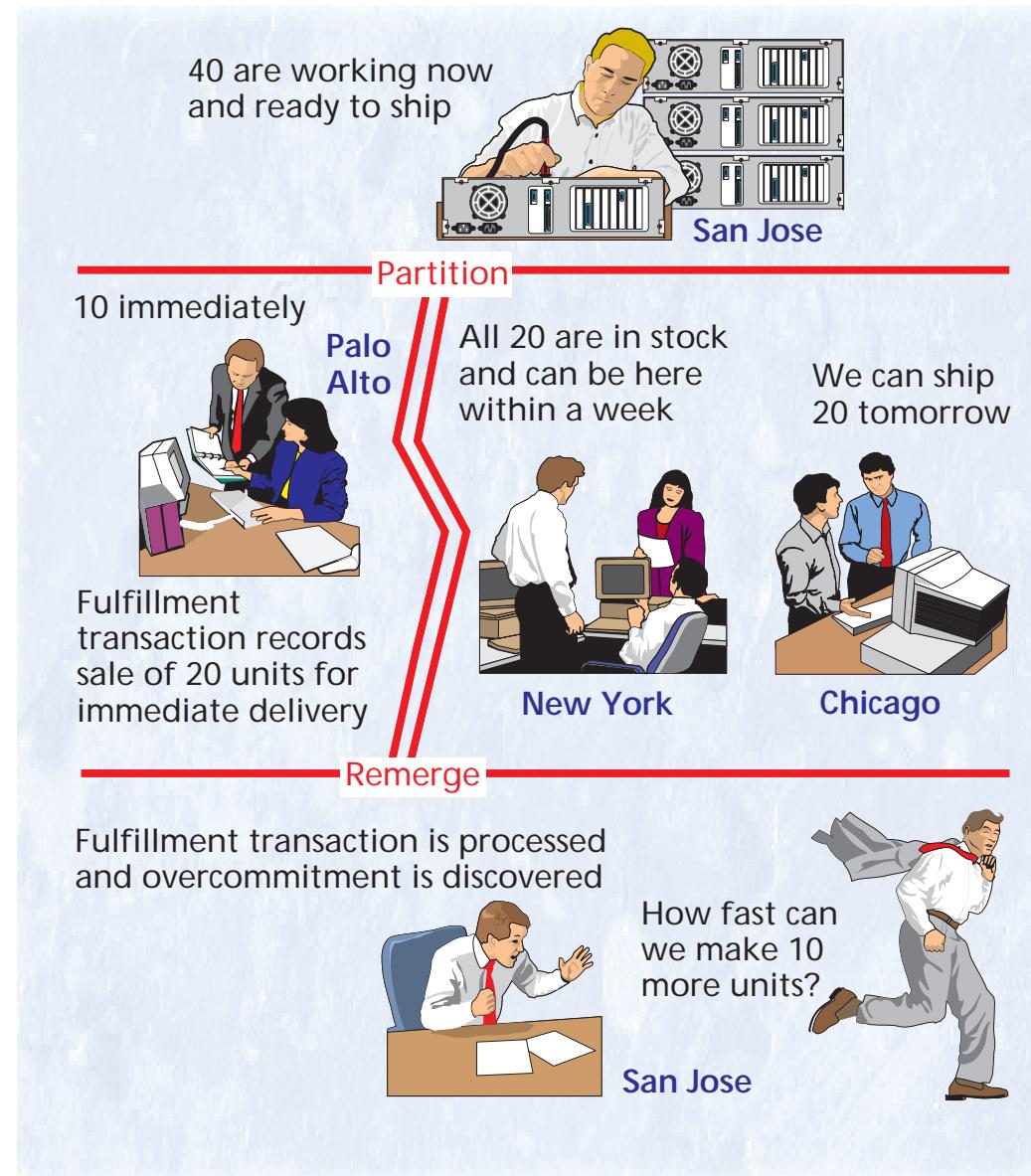


- Distributed application among 3 branches of a company
- Simple situation when network is working

Example: A simple consistency problem – 2

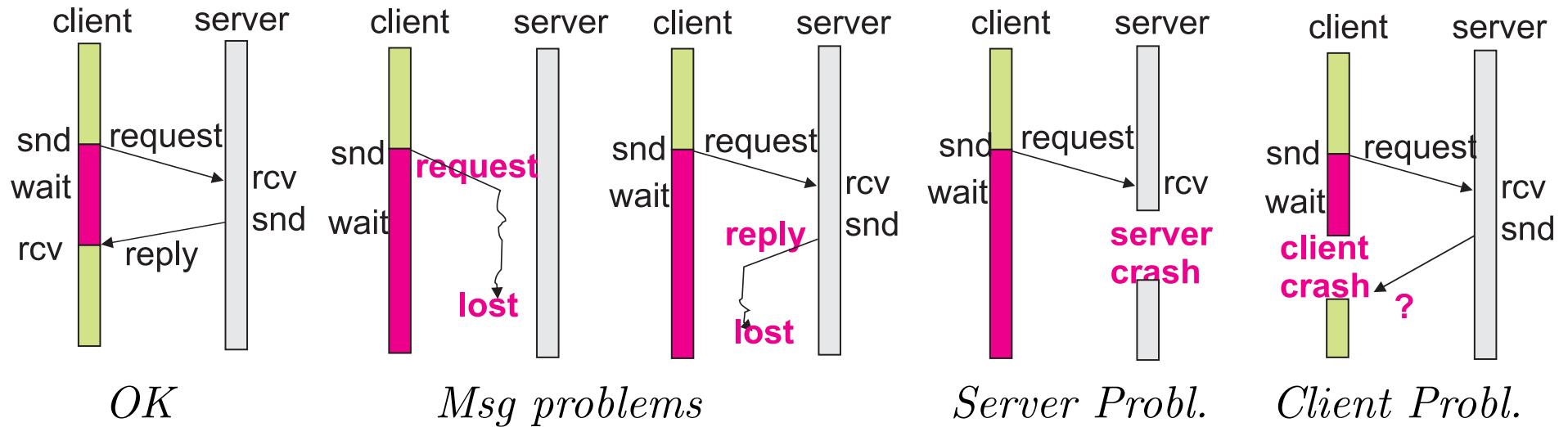
Network Partitioning:

- ▷ 40 units produced
- ◀ network problem
- $10 + (20+20)$ sold
- all local views ok
- global view wrong



Example: How to detect an Error in a DS setting?

Scenario: Client sends *request* to Server and waits for *reply*



- ◀ Message transport failures: **request** or **reply** is lost
 - ◀ *Server* is not able to process **request** due to **server crash**
 - ◀ *Server* uses unexpectedly long time to process **request** (**overload**)
 - ▶ *Client* is **crashed** and not able to receive any reply \implies possible effect on server?
- \implies *Time-Outs* are needed but: *For how long should a node wait?*

Synchronous vs. Asynchronous Systems

1. A distributed system is called **synchronous** : \iff

- (a) There exists a **system-wide known** upper limit of the time required for transferring data between different nodes of the system.
- (b) There exists a **system-wide known** upper limit for the **divergence of local clocks** between different nodes of the system.
- (c) There exists a **system-wide known** upper limit for the time, a specific node requires to compute a certain request.

2. A distributed system is called **asynchronous** : \iff

one or more of the conditions for a synchronous system are not met.

Problem: *In an asynchronous system in general there is no chance to distinguish failures from time-outs without waiting an infinitely long period of time.* (Decidability Problem)

\implies **Synchrony is a fundamental prerequisite for building DS!**

Limits of DS even in synchronous settings

CAP-Theorem: If Network Partitionings are taken into account
⇒ Trade-Off among these system properties:

1. Consistency of states among all nodes in a strong sense
2. Availability of data and services
3. Partition tolerance of underlying communication network

Dependent on the system's circumstances you have to favor different properties!

- In the *worst-case* with frequent partitions, one has to decide: compromise availability or allow for (slight) global divergence w.r.t. data updates in the system.
- In a strict data base setting using the ACID properties, availability has to be dropped to ensure consistency and isolation.
- '*Availability*' is an inaccurate term: How long is a client willing to wait before interpreting the timeout (= *latency*) as *not available*?

Brewer
2000
2012
2017
Lynch
Gilbert
2002

DS suffer from many species of failure

- ◀ **Connections:** lost, duplicated, unreadable or forged messages
- ◀ **Nodes:** different levels of harm
 - ◀ **Failstop:** node is **identifiable** as permanent **down**
 ⇒ does not respond and does not send in future
 - ◀ **Crash fault:** node is **unknowable** permanent **down**
 - ◀ **Commission fault:** incorrect processing, write wrong data etc.
 - ◀ **Send-Omission fault:** node omits messages/omits receivers
e.g. in a **broadcast** setting or a **global request**
 - ◀ **Receive-Omission fault:** drops some of the received messages
 - ◀ **Byzantine fault:** unpredictable behavior of a node
 - includes all other kinds of faults
 - time periods without reaction or even working correctly
 - forging messages, generation of spam messages etc.

It should always be clear which faults are tolerated in a system.

worst
case

I.4 Challenges when building DS

- ▷ Make the drawbacks of distributed systems manageable.
- ▶ *Try to hide as much internal details of a distributed system from the user as possible, but don't try to hide more.*

[Tanenbaum]: A distributed system is a collection of independent computers that appear to its users as a single coherent system. ◆

⇒ Single-System Image

[Schlichter]: A collection of (probably heterogeneous) automata whose distribution is **transparent** to the user so that the system **appears as one local machine**.

This is in contrast to a network, where the user is aware that there are several machines, and their location, storage replication, load balancing and functionality is not transparent. ◆

⇒ Distribution Transparency and Replication

Various Aspects to Distribution Transparency

- **Access:** Hide heterogeneity in data representations and invocation mechanisms among nodes of a DS.
- Hide where objects (data, services) reside in a distributed system
 - * **Location:** Hides detailed location of an object (in general)
 - * **Relocation:** Objects are unaware of their location changes
 - * **Migration:** Users are unaware that objects used change location
- **Replication:** Hides the fact that objects are copied and hold in different locations without knowing which copy is actually used.
- **Concurrency:** Isolation of different users and hiding of internal coordination mechanisms, e.g., in order to achieve consistency.
- **Failure:** hides and compensates failures in single components or the network of a distributed system.

Two main reasons for Transparency

1. **Reliability:** *Failures and malfunctioning components of a DS are not visible to the user but will be compensated internally.*

- **Replication** is at the very basis of any robust system.
- **Abstraction** from detailed hardware combined with migration and relocation transparency allows for exchange of components.

2. **Performance:** *Distribution does not impede the perceived performance for users. A distributed system scales for high loads as well as for load variations.*

- **Migration** to spots of heavy use increases performance
- **Replication** is also the key technique for performance:
 - * Data replication and *Caching* for fast access
 - * Service replication based on current system load
- **Concurrency** when performing work for a single user utilizes resources much better and reduces response times.

How to build 'good' distributed systems - 1

- ▶ *Know your internal system characteristics.*
- ▶ *Make well-documented assumptions and decisions.*
- ▶ *Use Abstraction:* Do not rely on specific servers, locations or names because this impedes transparency.
- ◀ Avoid *single points of failure* because there is a very high probability of overall system failure as the result.
Be careful: *multiple single points of failures are even worse.*
- ◀ Thinking in a '*global, consistent knowledge*' paradigm is error-prone because it does not hold in distributed systems.
- ◀ Avoid the typical pitfalls caused by a '*classical system view*'.

see
next
page

Common Pitfalls: too optimistic assumptions

◀ *Forget most of the assumptions you use in monolithic systems:*

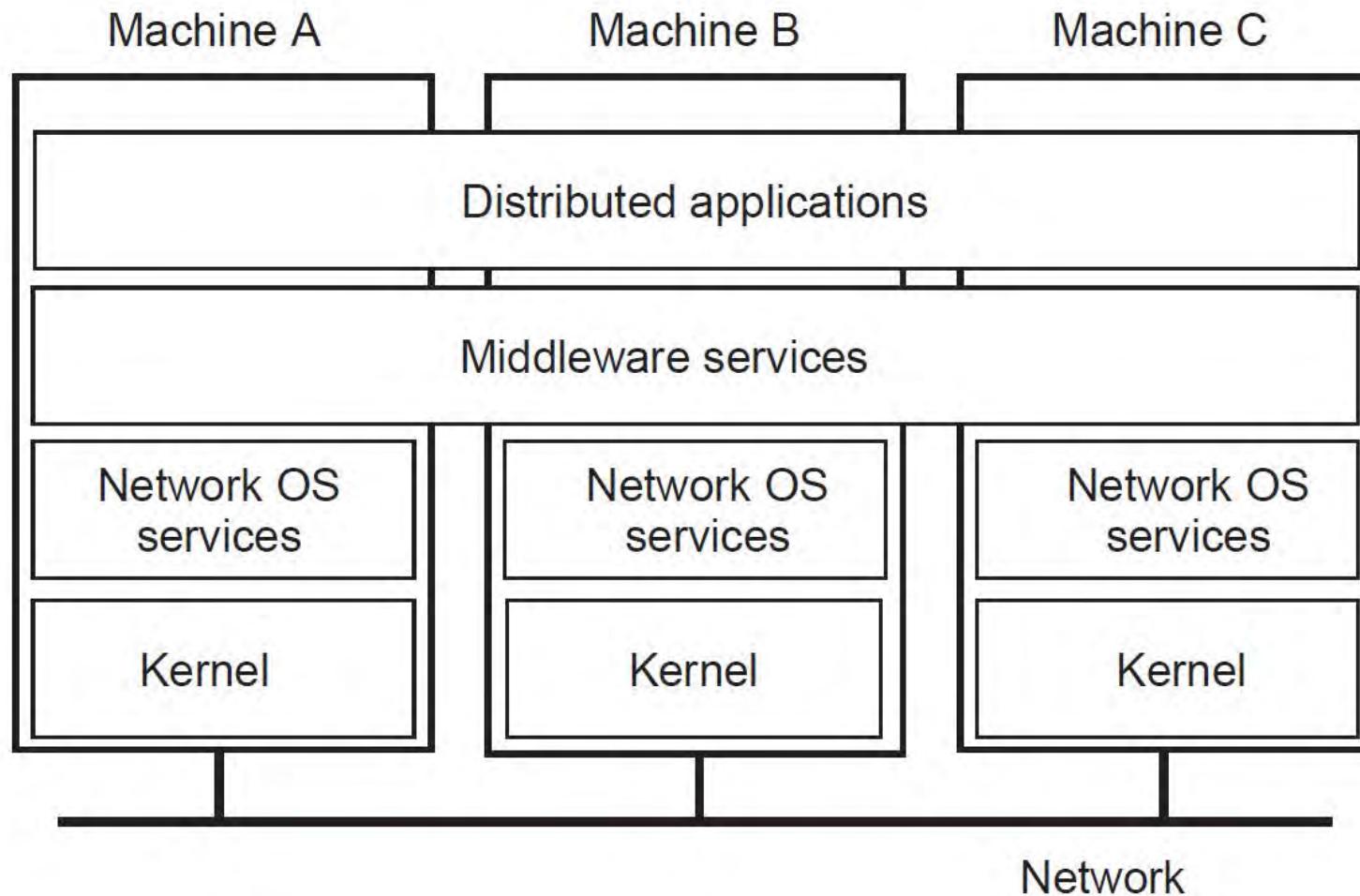
- Everything is homogeneous.
- The topology of the system does not change.
- One can rely on the location of specific data and services.
- The network is reliable and secure.
- Latency is zero.
- Bandwidth is infinite and transport cost is zero
- Messages are delivered in the same order they have been sent.
- There is one global administrator who can access everything.

To build a distributed system means to overcome all these false assumptions when you design and program the system but to provide as many of these assumptions to the user of your system.

How to build 'good' distributed systems - 2

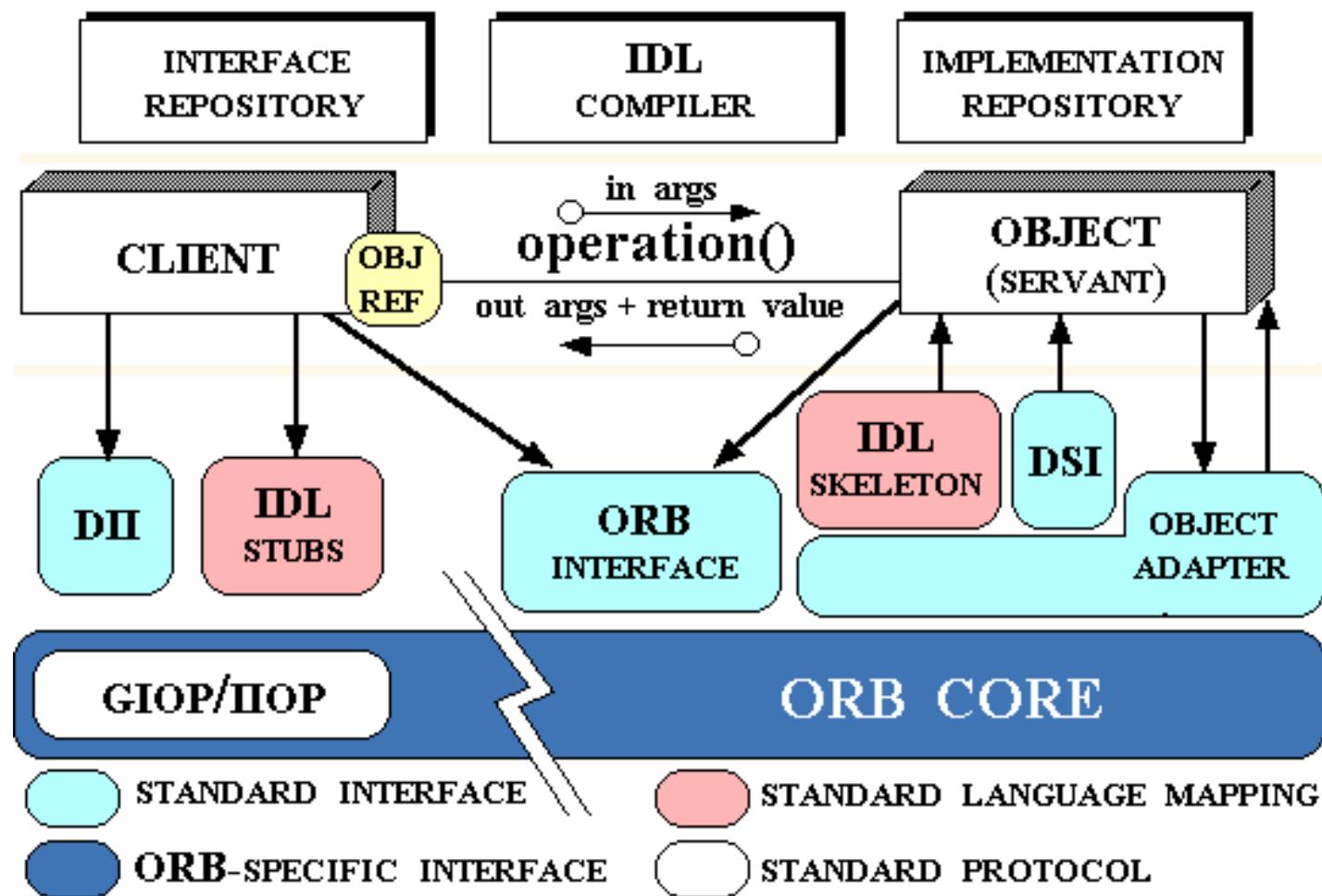
- ▶ Distributed systems are complex software systems
 - ⇒ **Good software architecture is the key to success!**
- ▶ Try to apply *separation of concerns* as much as possible.
- ▶ Break the system into interacting *layers of functionality* and don't try to solve all issues in a single layer, e.g.,
 - ◀ communication issues should be separated from application logic
 - ◀ database access should be abstracted from application code
- ▶ Use standards and standard tools whenever available.
- ▶ Use available *middleware systems* iff they fit your needs instead of programming everything from scratch:
 - ◀ complex and requires time and efforts to get familiar with
 - ▷ in the end you get much more work done ...

Typical Middleware Layer Architecture for DS



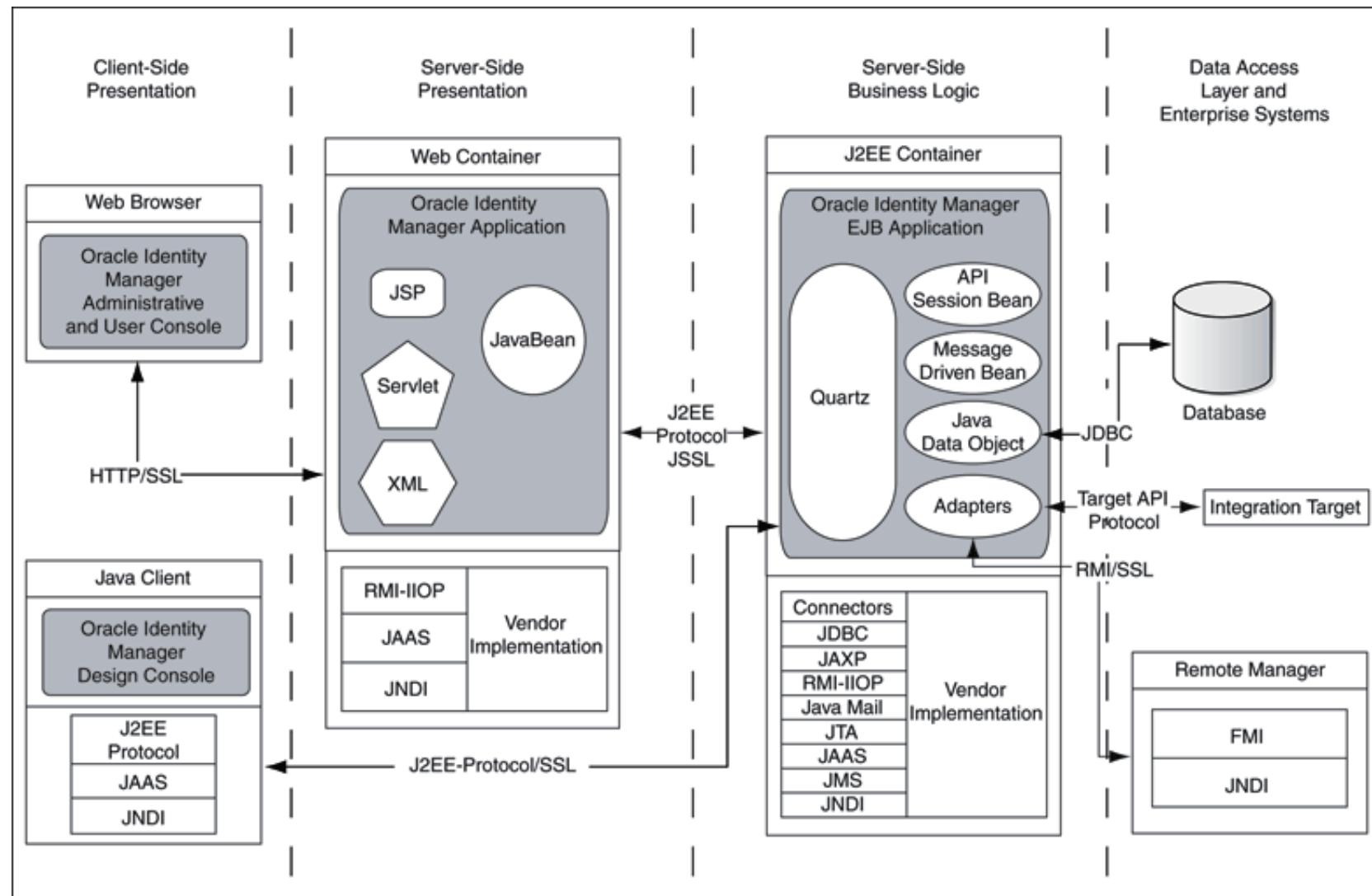
- abstracts from network and heterogeneity
- additional services, e.g., global naming, messaging, DB interface
- neutral w.r.t. applications: basis for many different applications

Example 1: CORBA as an ESB-based DS (1991)

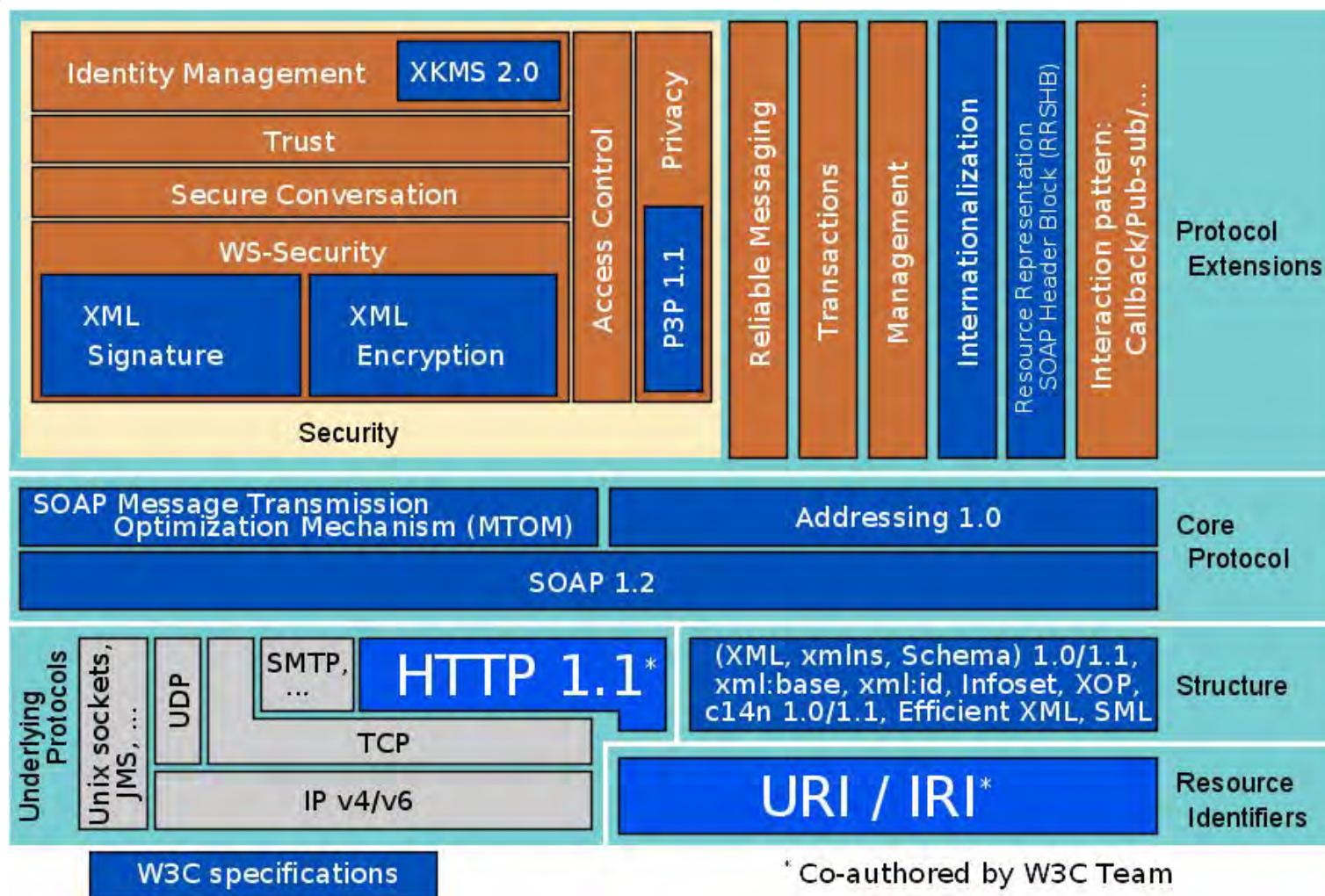


- Common Interface Definition Language for multi-language RPCs
- standardized messaging, event handling, service directories, ...

Example 2: J2EE multi-tier architecture (1999)

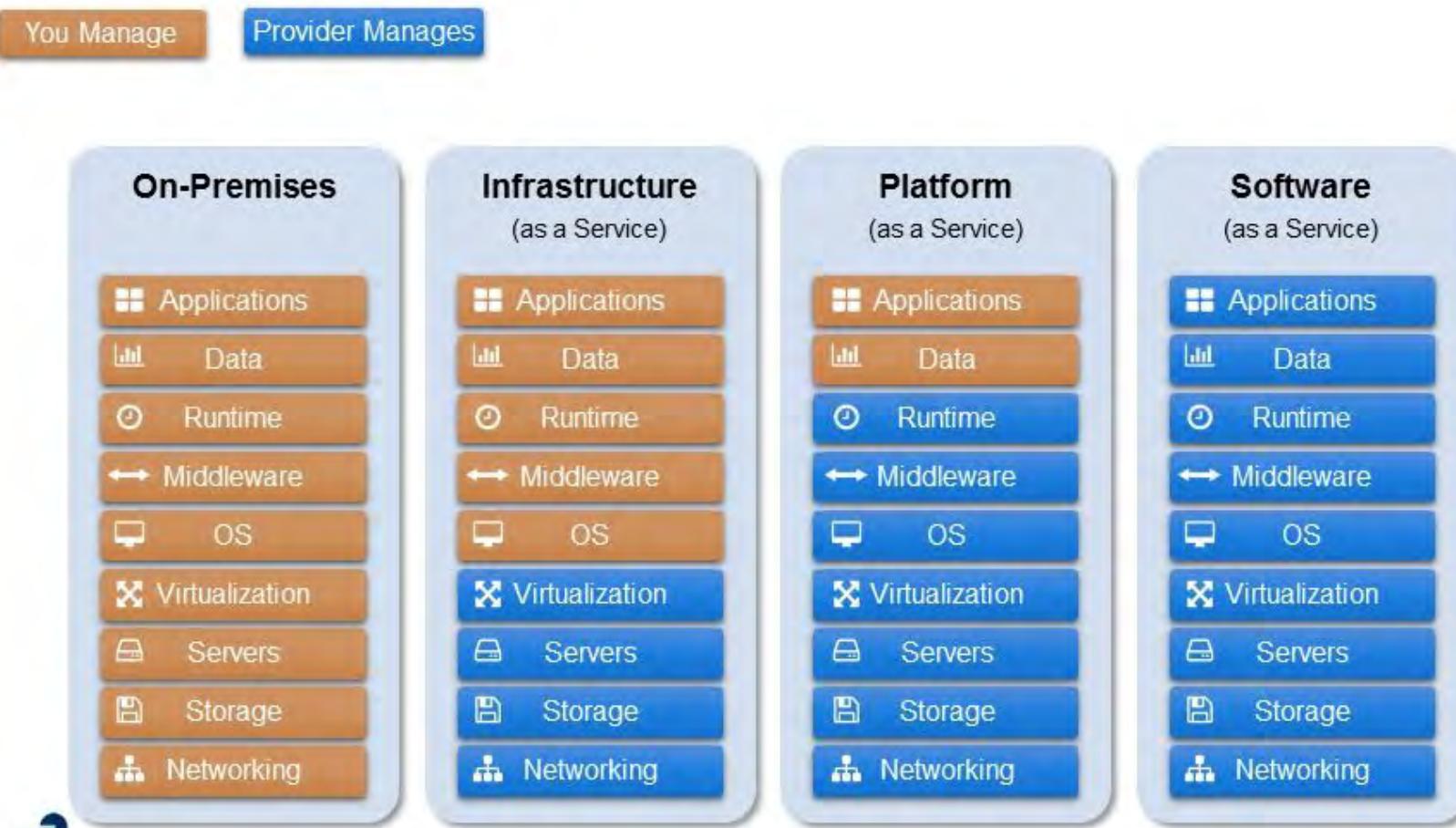


Example 3: Service Oriented Architecture Stack



- 4+ layer structure: networking ... *Quality of Service* issues
- hierarchical as well as horizontal organization
- standard as the basis for different vendor implementations

Example 4: The Cloud View of the Service Stack



- Nowadays Enterprise Software Systems are inherently complex
- Different Users → Abstract as much as possible from details
- Provide different levels of Abstraction and Control

Course Goals and Overview

You should know and understand

- ▷ the most important problems coming with DS
- ▷ techniques to make DS work despite of all these problems
- ▶ how to apply your knowledge to build real DS yourself

Course Syllabus: (besides basics)

- Basic Interaction Mechanisms and Paradigms:
 - * Shared Variables and Synchronization vs. Message Passing
 - * Message Passing (Sockets) vs. Messaging Services (AMQP)
- Programming in a Client/Server Paradigm
 - * Remote Procedure Calls (RPC)
 - * WSDL-, gRPC and REST-based Web Services
- Fundamental distributed algorithms
- Backbones for usable DS: Transparency and Replication
- Outlook: different flavors of middleware technologies

II. Technical Basis of Distributed Systems

Distributed Systems run on top of **heterogenous**

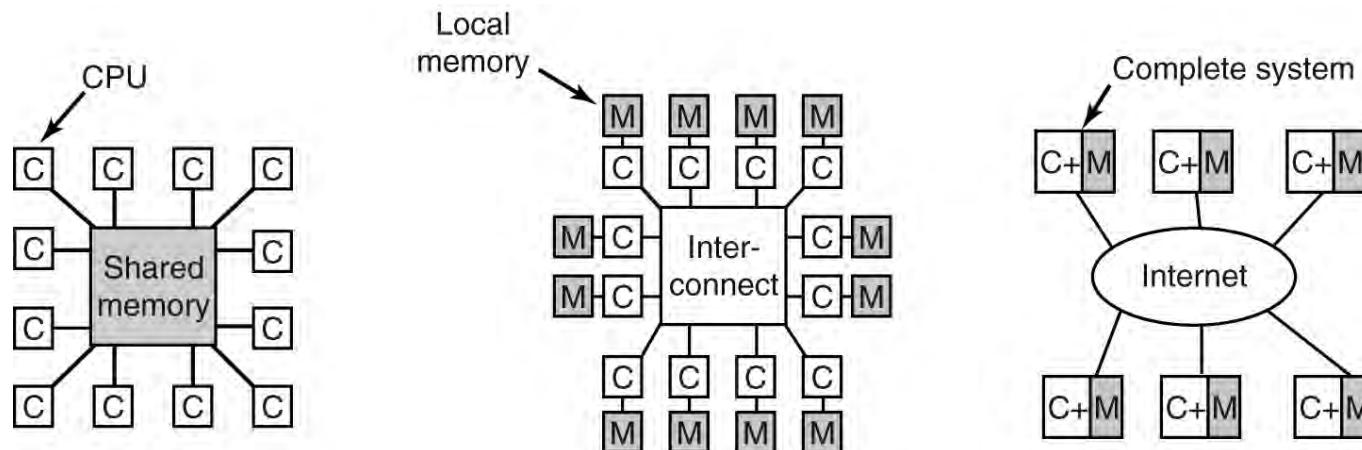
- **Hardware:**
 - single computers: mobile devices . . . high-end servers
 - wire-bound and wire-less interconnection networks
- **Operating systems:** heterogeneous data formats, environments

Distributed Application: 'Sets' of many 'programs at work', i.e.,
processes running on different, heterogeneous nodes across a wide-area communication network \implies

- \triangleleft different local operating system rules apply
- \triangleleft processes change their state independently from the distributed application due to local *scheduling*, availability of *resources* etc.

Some of the details of the underlying basis cannot be hidden completely when programming a distributed system.

II.1 Hardware and Programming Models for DS



'History': development of computer architecture and systems

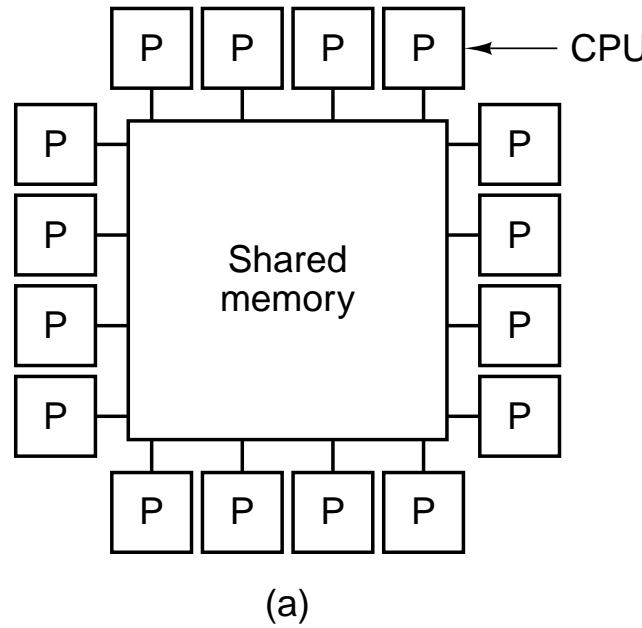
1. All sorts of internal parallelism inside a 'von-Neumann' computer nowadays, e.g., DMA, pipelining, ... multi-core architectures
2. Different levels of interconnection architectures and types:
 - ▷ **Multi-Processors:** High-performance 'on premise' connect
 - lots of CPUs and a single, global, shared MEM (SMS)
 - lots of CPUs with dedicated local MEM on their own (DMS)
 - ▷ **Multi-Computers:** Wide-area, moderately performing connection of autonomous, full-fledged computers (DMS)

Programming Models for different architectures?

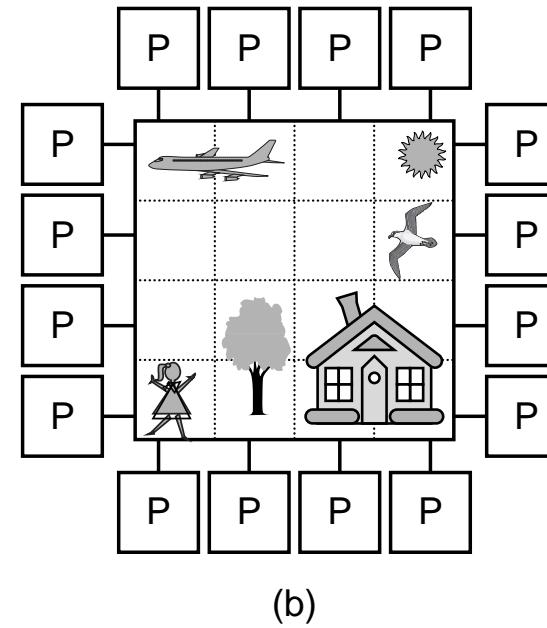
- **Shared Memory (SMS) models:**
 - △ parallel processes (threads) accessing global variables
 - ▶ **synchronized** objects using locks, semaphores, monitors
 - ▷ *Black-board* models of shared interaction spaces
- **Distributed Memory (DMS) models:**
 - ▶ *message-passing* with various abstractions for
 - * *addressing* via process names, ports, groups, properties
 - * strong vs. loose *coupling* w.r.t. synchronization and time
 - ▷ autonomous *actor* and *agent society* models

Even today, interaction paradigms on programming language level reflect different hardware architectures in order to allow for dedicated, well-performing programming models.

Example: Shared Memory Problem Solving



(a)



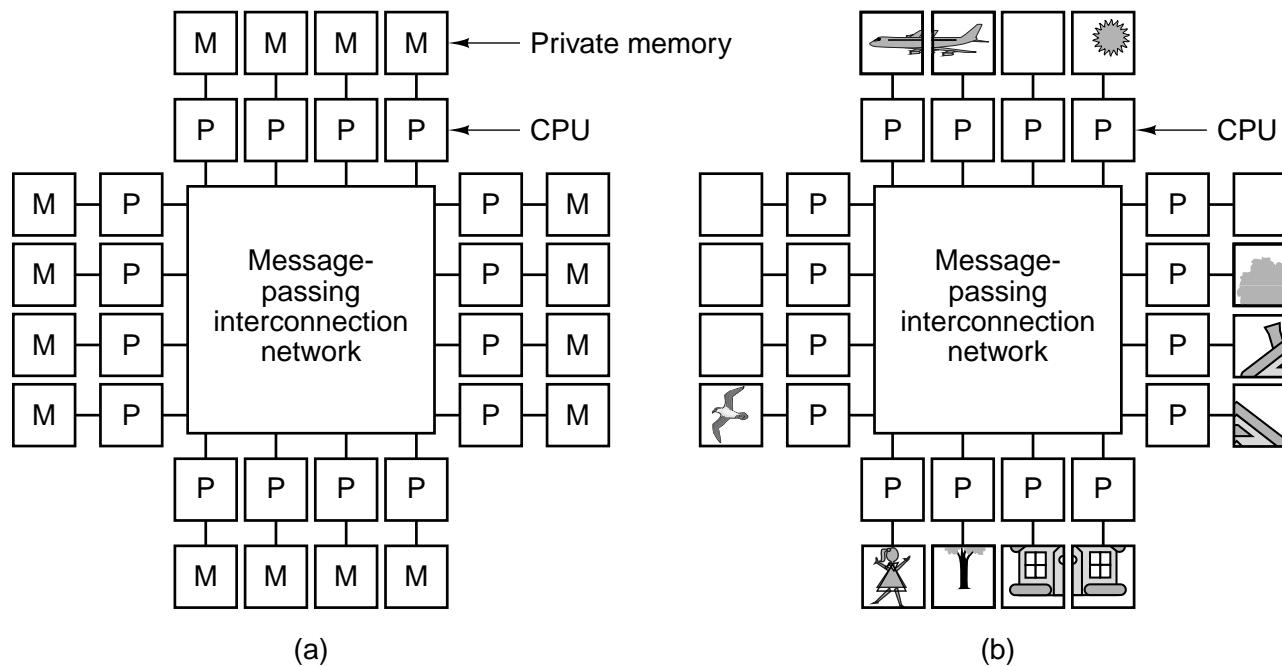
(b)

Example: Image analysis

- All data accessible for all processors in global MEM
 - *Data distribution* among processors is used to organize work
 - Exchange: process reads data written by other processes in MEM
- ◀ **Problem:** *Race-conditions* iff not properly synchronized
Good data distribution is essential for efficiency and load balancing,
but no costly explicit transport needed.

Tanenbaum Computer-Architectur 1999⁴
Fig. 8.1

Example: Distributed Memory Problem Solving

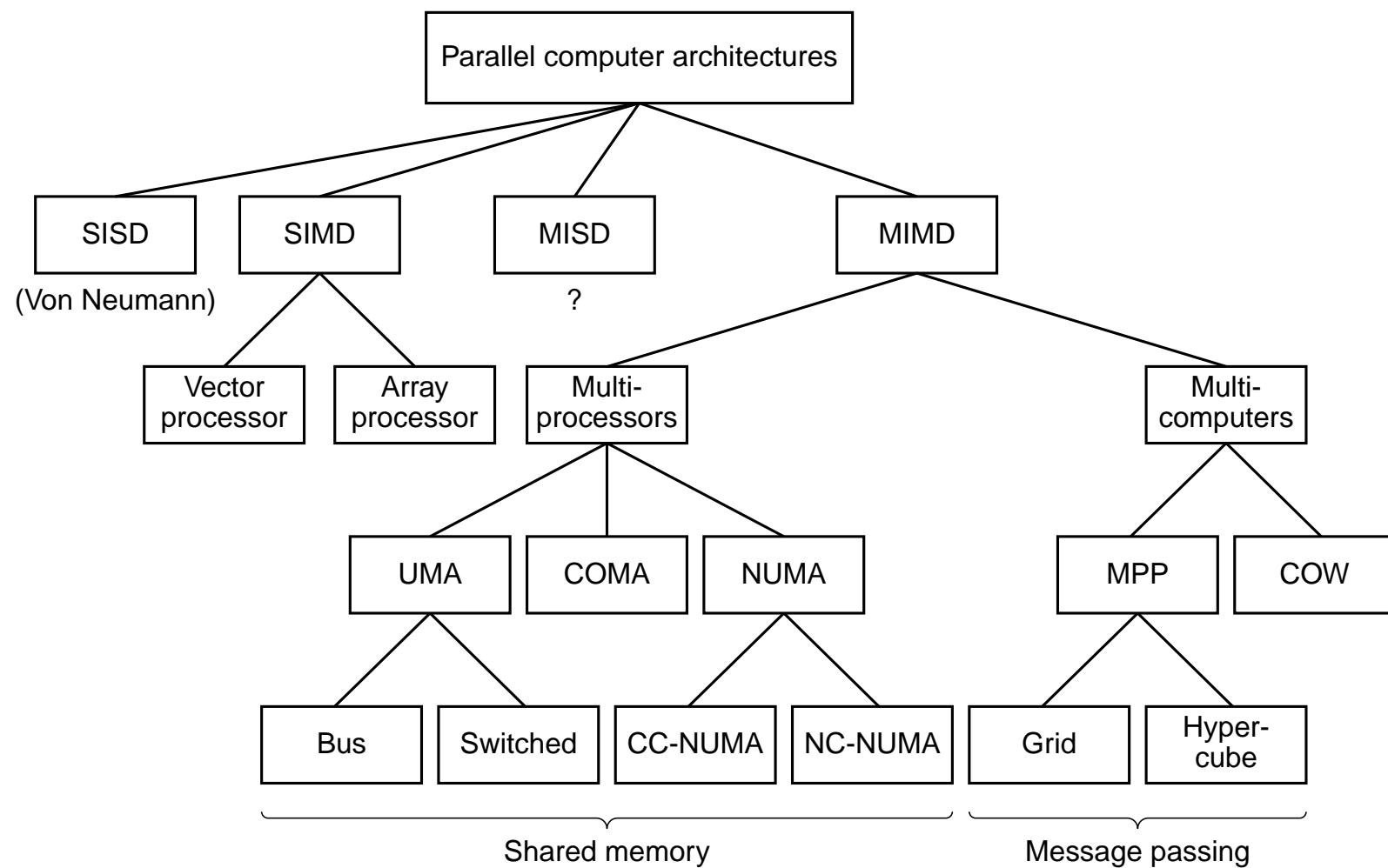


Example: Image analysis

- Data are distributed among local MEM of all processors
 - *Data distribution* among processors is used to organize work
 - Exchange: processes send/receive data to/from other processes
- ◀ **Problems:** Wrong or changing data distribution is costly;
Networks with bad performance or reliability result in waiting processes; lost messages lead to blocking processes etc.

Tanenba
Compute
Architekt
1999⁴
Fig.
8.2

Putting DS into Computer Architecture Context

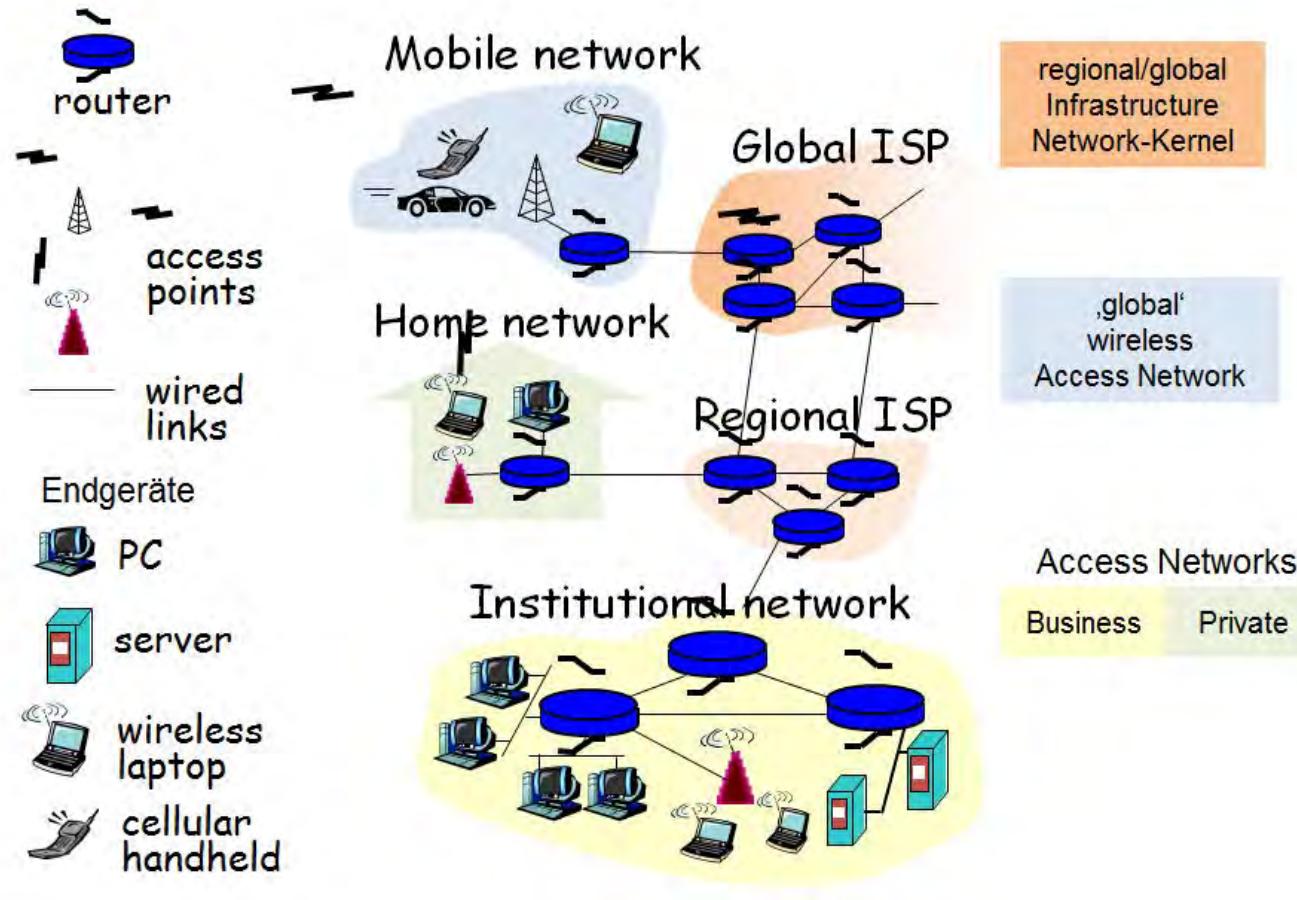


- 'Complete DS' fit only in a small part of this classification system
Technically: **M**ultiple **I**nstruction **M**ultiple **D**ata **M**ulticomputer
- Single nodes of the same DS may well be of any other class

Tanenbaum
Computer-Architektur
1999⁴
Fig.
8.14

Flynn
Tanenbaum

II.2 Underlying Network Technologies

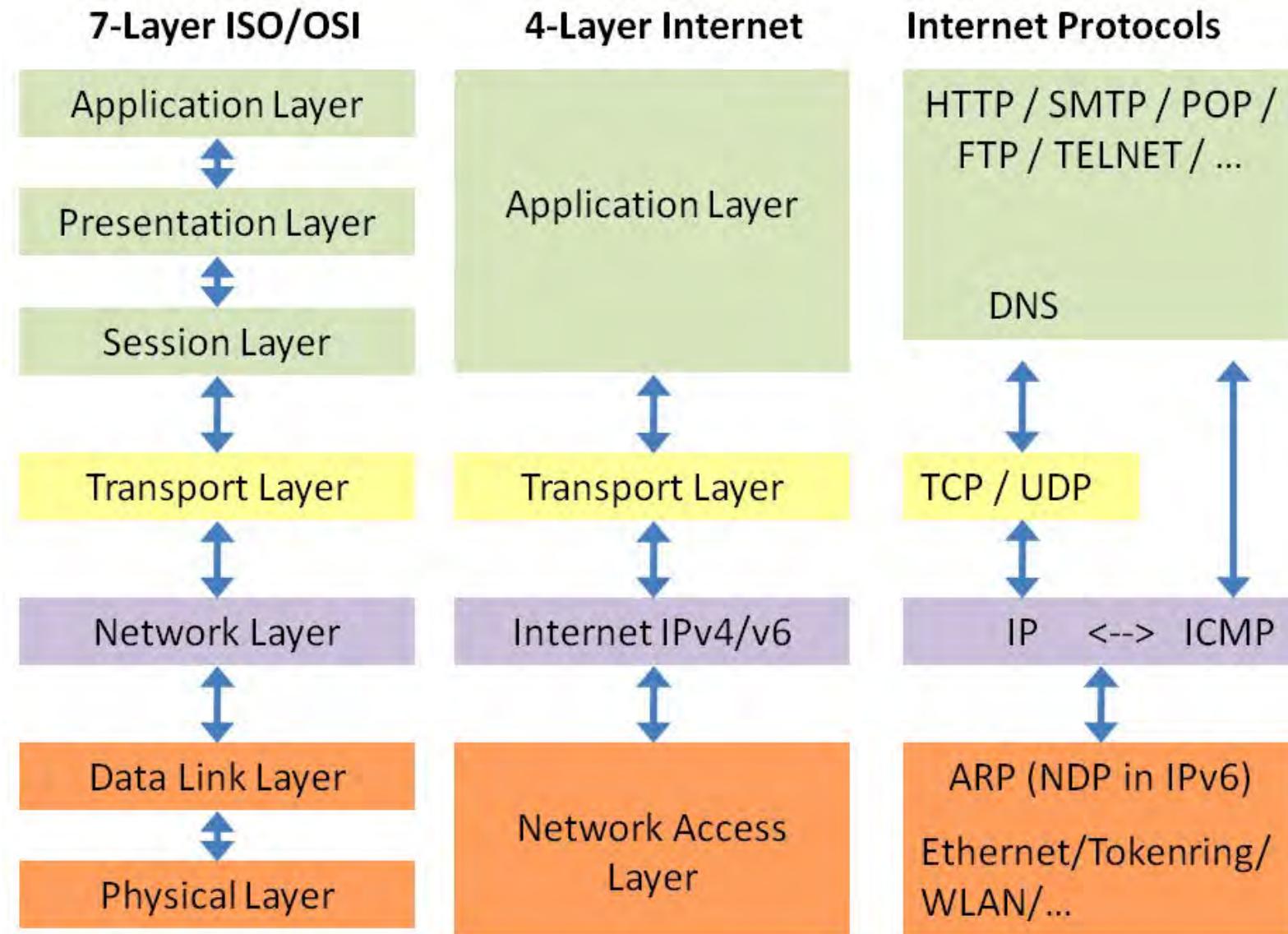


- Interconnection of a variety of (Heterogeneity)
 - * Technologies: coax/fiber cable, wireless, telecommunication, ...
 - * Algorithms for routing, fault-tolerance, load-balancing, ...
- always evolving in terms of connectivity, technology, standards.

Heterogeneity ruled by world-wide standards ...

OSI (Open Source Interconnection) 7 Layer Model				
Layer	Application/Example		Central Device/Protocols	DOD4 Model
Application (7) Serves as the window for users and application processes to access the network services.	End User layer Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management	User Applications SMTP	GATEWAY Can be used on all layers	Process
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	Syntax layer encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • Character Set Translation	JPEG/ASCII EBDIC/TIFF/GIF PICT		
Session (5) Allows session establishment between processes running on different stations.	Synch & send to ports (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.	Logical Ports RPC/SQL/NFS NetBIOS names		
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	TCP Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	F I L T E R P A C K E T Routers TCP/SPX/UDP		Host to Host
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	Packets ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting	IP/IPX/ICMP		Internet
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	Frames ("envelopes", contains MAC address) [NIC card — Switch — NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control	Switch Bridge WAP PPP/SLIP		Network
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	Physical structure Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts	Hub Land Based Layers		

... but: the real world is a bit simpler

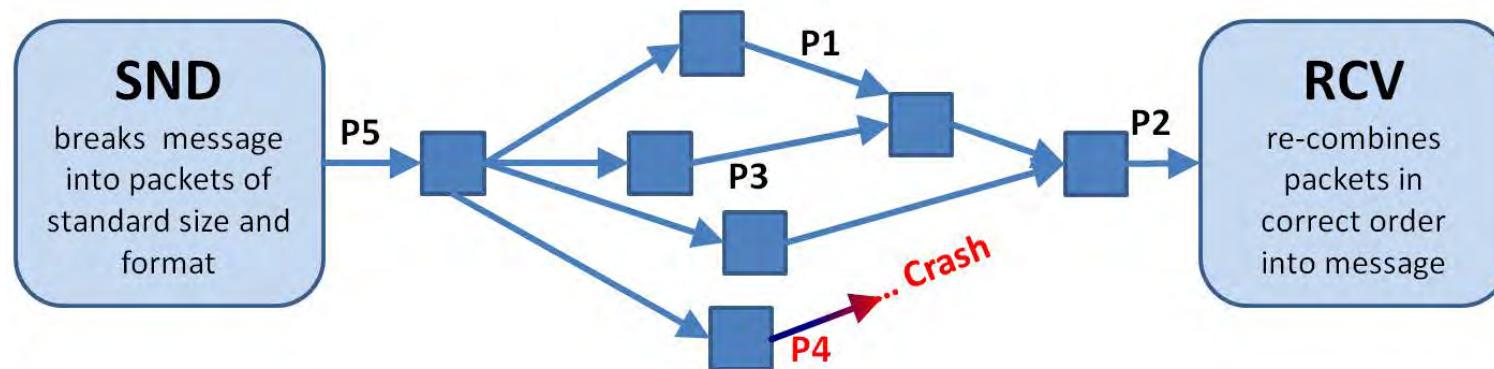


Network-related Problems for building DS – 1

◀ Sending a message from node A to B causes a lot of **Overhead**

1. in **A**: *User-Process* \longleftrightarrow Operating System \longleftrightarrow Network Interface
2. **Network**: \longleftrightarrow Network Transmission, Protocols, Error-Handling \longleftrightarrow
3. in **B**: Network Interface \longleftrightarrow Operating System \longleftrightarrow *User-Process*

◀ **Message-Ordering** is not granted in all network models



◀ failing components: re-routing due to fault-tolerance techniques

◀ changing load: different routes due to load-balancing

◀ **Lost Messages** have to be taken care of in some environments

Network-related Problems for building DS – 2

High-level **programming languages** rely on 'linked data structures',
e.g., objects, structs, records, ...

- ◀ Local references, pointers or direct addresses (`int`) are meaningless on a different remote machine.
- ◀ Networks usually transmit data as simple fixed-length, contiguous, encoded byte blocks.
⇒ complex data structures cannot directly transferred 'as is' in a DS

Missing transparency requires

- when sending: **Marshalling**
 - * *Externalization/Serialization* of data structures
 - * *Splitting* data into smaller segments fitting into network packets
- when receiving: **De-Marshalling**
 - * *Assembling* packets back to meaningful data structures
 - * *De-Serialization*: storing data and building new link structures

Heterogeneity-related Problems in Networks

- ◀ **Internal representations of data diverge** among different machines, operating systems and language runtime systems, e.g.,
 - Big vs. Little Endian byte order (HW)
 - ASCII vs. EBCDIC vs. Unicode (OS)
 - 32 vs. 64 Bit integers and addresses (HW+OS)
 - Byte vs. Word alignment of data in local memory (OS+PL)
 - padding and order of components for structured objects (PL)
- ▶ **Low-level Middleware layers** should allow for Standards-based
 - * representations of data types: `char`, `int`, `floating-point`
 - * rules for *encoding/decoding* between different data formats
 - * descriptions and services for serialization/de-serialization rules
 - * descriptions for complex data like agreed-upon or self-describing standardized formats, e.g. XML- based schemata.

Example: Alignment and Padding in MEM-Layout

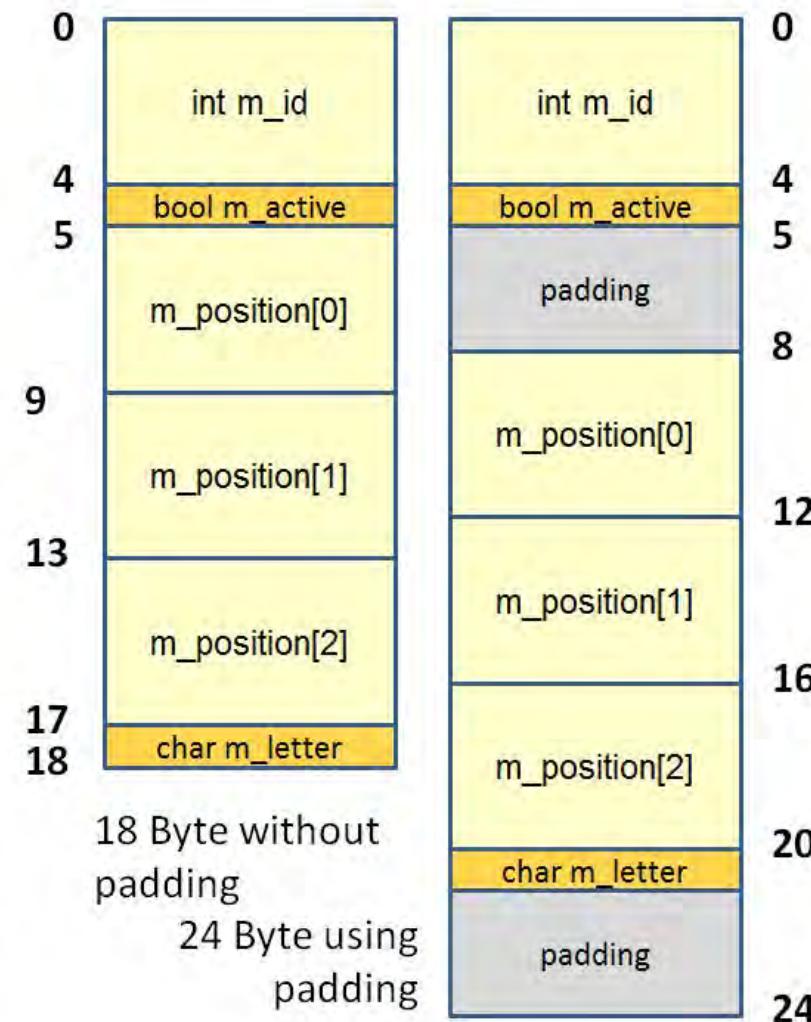
```
struct WaypointInfo
{
    int m_id;
    bool m_active;
    float m_position[3];
    char m_letter;
};
```

Expl.: C struct

- 1 Integer m_id
- 1 Bool m_active
- 3 Floats (Array)
- 1 Char m_letter

MEM Layout with Byte-Address

- left: unaligned, optimal w.r.t space
- right: 4-Byte word aligned



- * **Alignment:** structures start always at a multitude of word-size
- * **Padding:** un-used Bytes filled by '00000000' due to alignment

II.3 Operating and Run-time System Issues

A Distributed Application consists of

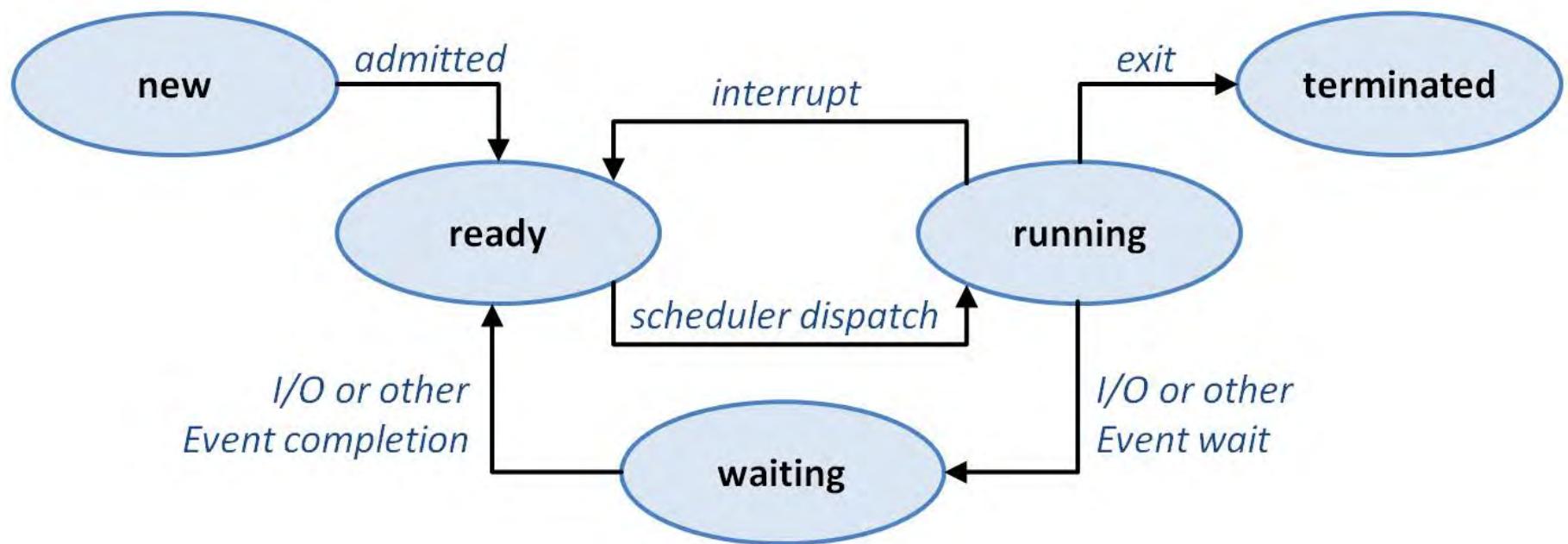
- 'program parts' of the application program
- supported by more or less levels of Middleware
- running on the participating nodes of a distributed system.

Simple System Model: A 'bunch' of **running programs** that – besides interacting over a communication network – are ruled by executing as **local processes**:

- * processes are embedded into the local *run-time system environment* of the programming language used
- * that depends on the local *operating system environment*
- * which uses the *hardware resources* of the local machine.

The handling of local processes in each participating node influences the overall execution of the DS.

Process Handling in an Operating System



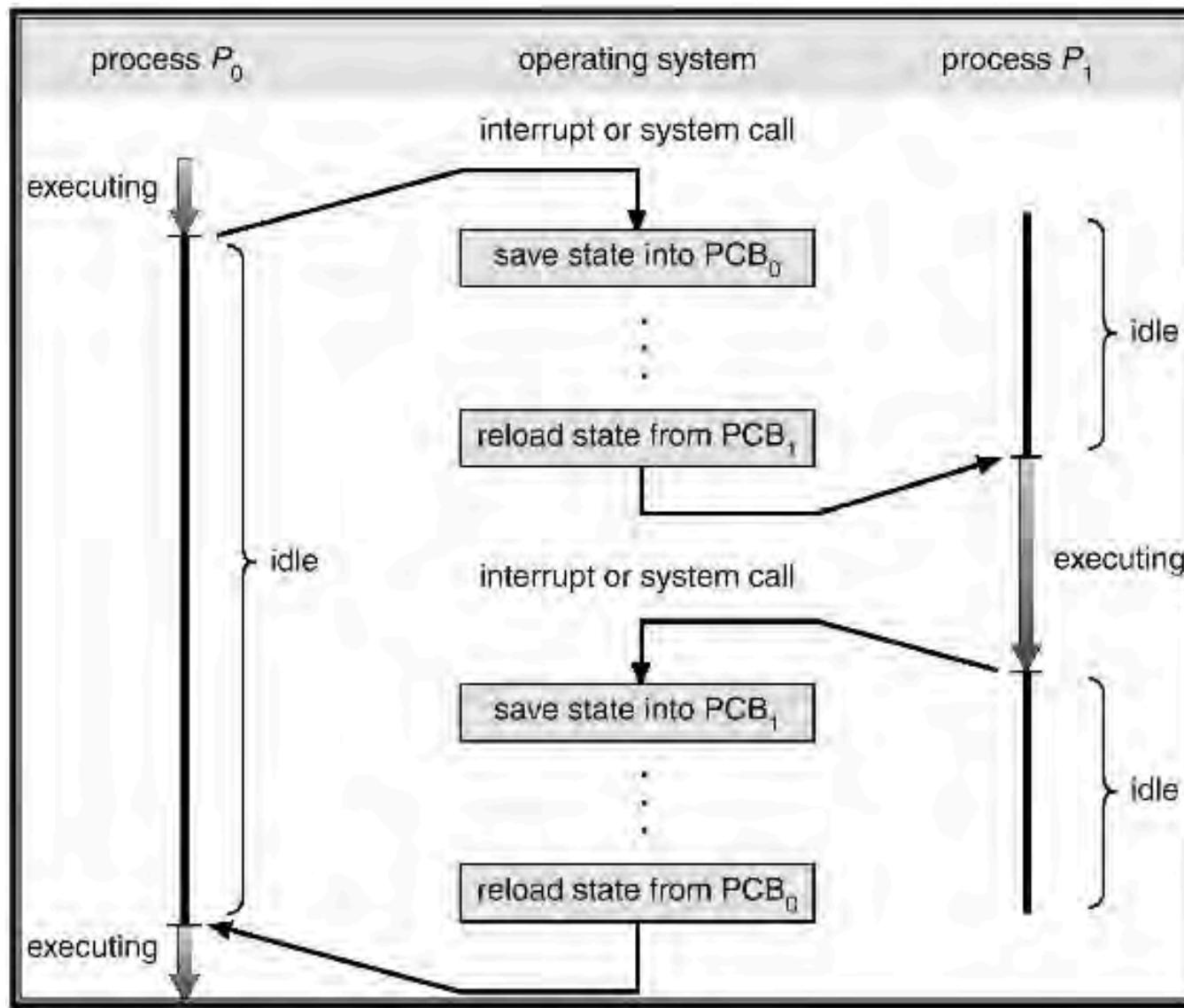
READY-Processes typically do not run to completion due to:

- OS scheduling decisions may prioritize 'other' processes
- Processes may require **resources** that are not yet available
Expl.: Interaction requires the network interface resources
- Processes may get stopped when trying to violate access rules

Processes are prevented from progress for long time periods.

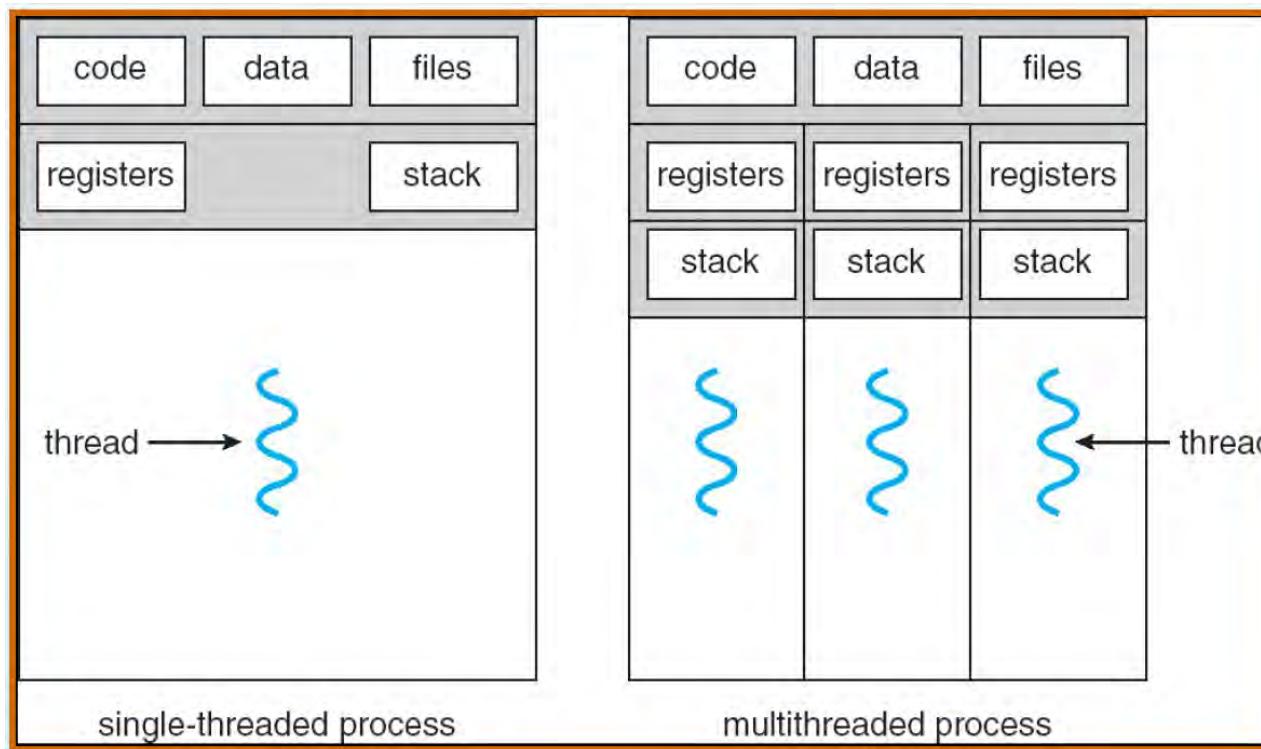
Process Switch between processes P_0 and P_1

aus:
Silber
schatz
Galvin
Gagne:
Opera
ting
Systems
Concepts
Fig.4.3



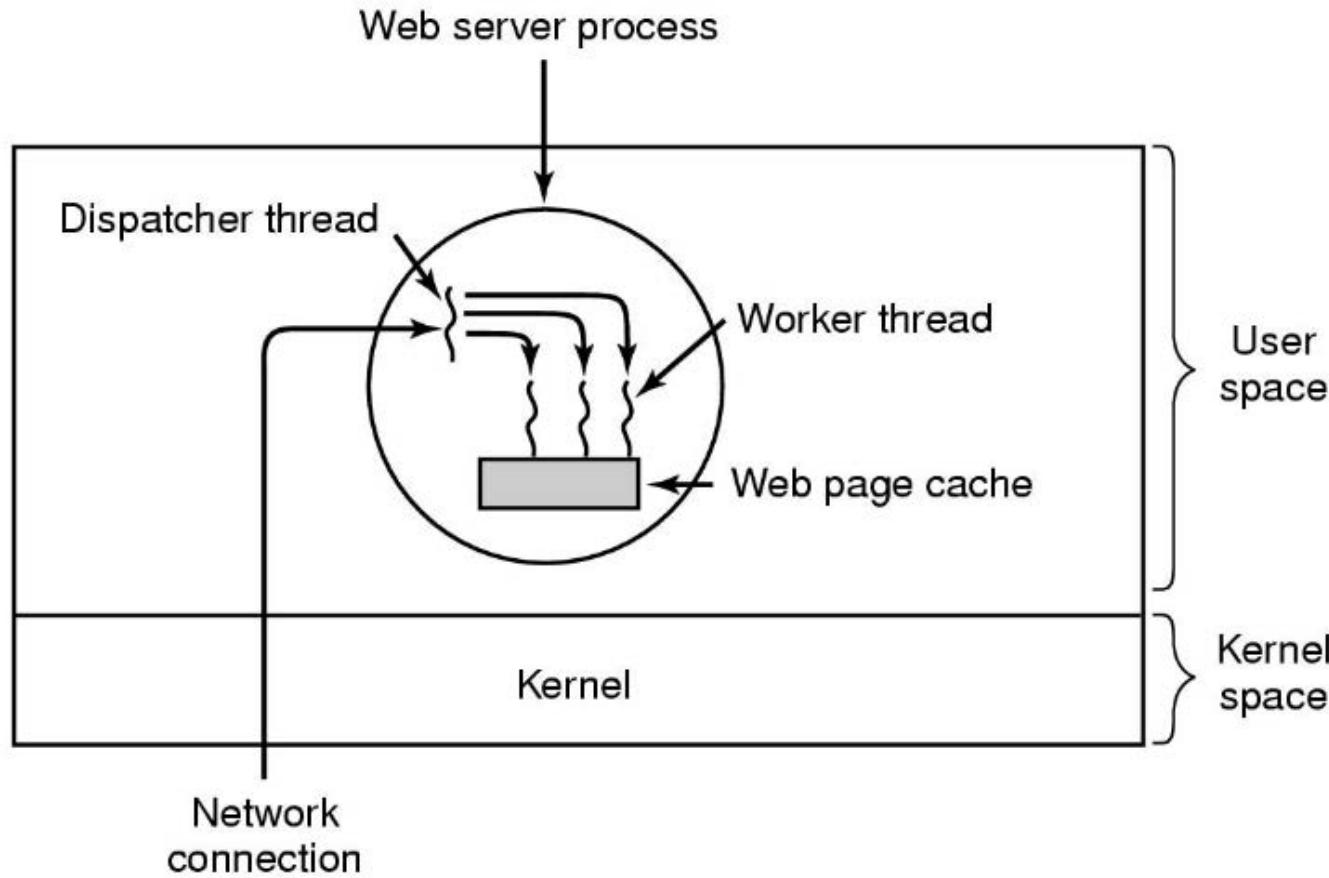
◀ Overhead caused by swapping processes in and out, esp. OS-time

Operating System Processes and Threads



- ◀ Process switches write/read **Process Control Block (PCB)**, e.g.,
 - ▶ Execution state: program counter, registers, stack content
 - ◀ Process environment: directory, open files, file pointer, memory
 - ◀ OS management: User/P-ID, priorities, accounting information
- ▶ Distinguishing among **heavy-weighted processes** and **light-weighted threads** allows for considerable optimization.

Example: Usage of Threads in a DS context



Architectural Pattern: *Keep your server always available !*

- *Dispatcher* thread listens for incoming requests and delegates work to *Worker* threads that work independently from communication
- Worker threads may have to be *synchronized* w.r.t. common data

II.4 Implications for programming DS

see
also
I-40

I-42

1. *Never assume that your data structures can be sent over the network without encoding/decoding or marshalling/demarshalling.*
2. *Always describe data and message formats in detail, either as a well-documented 'common agreement' among all nodes of a DS or by using standardized self-describing message formats.*
3. *Always be sure about the properties of your communication network concerning message ordering and possible message loss.*
4. *Never assume that your processes are running to completion without any interruption.* This does not hold for:
 - combinations of entire processes or threads
 - single method calls in a thread
 - single operations or statements in a thread
 - evaluation of simple expressions

⇒ *Fixed orders have to be implemented iff needed.*

Implications for programming DS – 2

5. '**Atomicity**' does **not** even hold on programming language level.

- ▷ reading **or** writing a variable like an `int` may be atomic
- ▶ reading **and** writing a variable normally is not!

Atomicity has to be implemented iff needed.

6. Never assume that you have full control over the *execution order* or isolation of your processes without extra programming efforts.

- OS scheduling leads to arbitrary *non-deterministic interleaving*, i.e., re-ordering of statements among different processes.
- Parallel multi-core hardware leads even on a 'single node computer' to *true parallelism*, i.e., different statements of different processes are executed at the same time.
- On different compute nodes, *true parallelism* is the given normalcy.

III. Basic Interaction and Cooperation Mechanisms

Basis:

(c.f. Definition of Distributed Systems)

- *Hardware*: Distributed, heterogenous compute nodes connected by various types of networks.
- *Software*: Application and operating system processes (or threads) in a mix of *sequential*, *nondeterministic interleaving* or *truly parallel* execution on the same or different nodes.

Target Distributed Systems Model:

- System(s) of **active processes** get some *common* work done
 ⇒ *Information exchange through interaction mechanisms*
- **Interaction** defines two principal **Roles**: '*provider*' vs. '*recipient*'
 - Shared Memory: '*writer*' vs. '*reader*'
 - Distributed Memory: '*sender*' vs. '*receiver*'
 ⇒ *Information exchange introduces a **causal dependency** where the recipient depends on the provider.*

Preview: Dependency degrees and coupling

1. **No direct or in-direct interaction** \implies *no coupling*
asynchronous processes with isolated local states only
2. **Signal: existence** (pure) of information representation
'pure' synchronization \implies *very tight coupling*
Example: P_r waits for signal from P_p ; check for non-null ref.
3. **Data: existence plus content** \implies *tight coupling*
Example: P_r reads content of data or message provided by P_p
4. **Task: existence plus content plus interpretation**
of information representation \implies *moderate coupling*
Expl.: Server P_r gets method call and parameters from client P_p
5. **Self-contained: existence plus content plus procedure**
to deal with content \implies *loose coupling*
Expl.: P_r gets method call, parameters and executable from P_p

III.1 Process System Model

1. Static Structure prescribes the scope for behavior

- a finite set of **States** Q with a specific initial state $q_0 \in Q$ and a (possibly empty) set of final states Q_F
- a finite set of **Rules** R describing all system steps permitted, i.e., all allowed **state changes** $\rightarrow \in Q \times Q$

2. Dynamic Behavior results from possibly infinite sequences of state changes, so-called **processes** that adhere to all rules of R .

Definition III.1: (Process)

Process is a **sequence of steps** $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \dots$ where

- each step is performed by an **action** a_{i+1} that results in a state change $q_i \rightarrow q_{i+1}$ ($i \in [0 : \infty]$)
- all resulting states q_i of a process are permissible, i.e., $q_i \in Q$
- the action for each step is permitted by some rule in R
- each single action is assumed to be **atomic**, i.e., indivisible. ◆

Processes as Execution Sequences

Two dual aspects of a process are equivalent:

- *passive view* as a sequence (trace) of states q_0, q_1, \dots
- *active view* as a sequence (trace) of actions a_1, a_2, \dots

Note: '*Trace*' is often used for single execution sequences

Sequential vs. Parallel Execution

- ▶ A **single process** as a sequence of steps is always **sequential**
 - * the steps of the same process are executed in a *linear total order*
 - * for any two actions a and b in a process holds: $(a \sqsubset b)$ or $(b \sqsubset a)$
- ▶ **Parallelism** is made possible by potential and execution means:
 - * Execution of several processes in different components that are (more or less) *independent* from each other (*partial order*).
 - * Combining more than one subsystem to a new system that allows for more than one execution step at the same time.

Definition III.2: (Process System Model)

1. A **process system** PS consists of a set $\{P_1, \dots, P_n\}$ of sequential processes P_1, \dots, P_n .
2. The **Action Execution Order** \sqsubset_{PS} of PS describes the admissible overall execution orders of all processes of PS :
 - (a) For each $P \in PS$ all actions are in a linear, sequential order.
 - (b) Actions from different processes $P_a \neq P_b$ are not ordered and said to be independent and **concurrent**.
 - (c) \sqsubset_{PS} is an irreflexive, asymmetric and transitive **partial order** relation among the actions of all processes of PS

◆

- The starting point of any PS is an almost unordered system where each process adheres to its own linear order but all actions from different processes are completely independent from each other.
- Introducing interaction among processes defines additional dependencies and restricts the overall execution space.

Definition III.3: (Properties of Processes and Systems)

1. **P terminates:** P is in a state $q \in Q$ where the rule R permits no more actions $\implies q \in Q_F$ is a final state.
2. PS is **deterministic** : \iff
 $\forall q \in Q \exists$ exactly one process P starting with q .
3. PS is **determinate** : $\iff \forall q \in Q$ holds: All terminating processes starting with $q \in Q$ also end in the same finite state q' .

◆

- A non-terminating process has no well-defined final state, hence, Def.III.3 (3) does not apply.
- **Deterministic vs. Determinate Systems:**
 - * a deterministic system is always also determinate.
 - * a determinate system is not required to be deterministic.

Determinacy is more relaxed but ensures the correct result.

Two Aspects of Non-Determinism

◀ Don't Know-Non-Determinism as a *modeling mechanism*

- complex systems with high spectrum of possible behavior
- not every detail can be captured in a model of reasonable size
- worst-case-assumptions are needed to avoid system malfunctions

Examples: securing a user interface

integration of a black-box device in a software system

► Don't Care-Non-Determinism as an *abstraction mechanism*

- problem allows for different approaches to find 'a' solution
- system allows for tolerance in execution orders due to independent 'concurrent' system parts
- different execution orders end up with the same results

Examples: non-deterministic automata models

evaluation orders for expressions, functional languages

Specification Level vs. Execution Level

Programs specify the rules R that restrict the **processes** of a system.

1. A **sequential** program specifies exactly one process P .
2. A **non-deterministic** program specifies an arbitrary process P_i of a possibly infinite set $\{ P_1, P_2, \dots \}$ of processes.
3. A **concurrent** program specifies a set $\{ P_1, P_2, \dots \}$ of more or less independent processes with partially ordered actions.

⇒ '*Concurrency*' is a specification level property.

Program execution uses the potential of relaxed strictness tailored to the underlying hardware:

- Availability of more than one processor allows for *true parallelism*.
- Otherwise, *non-deterministic interleaving* (pseudo-parallel execution) by choosing an overall total execution order \sqsubset_{ex} that respects the partial order \sqsubset_{PS} ($\sqsubset_{PS} \subseteq \sqsubset_{ex}$) becomes possible.

⇒ '*Parallelism*' is an execution level property.

Example: Concurrency, Parallelism and Atomicity

- Concurrent program does not imply an order on a_1 and a_2

$i := 5; \text{parbegin} \{ \underbrace{i++}_{a_1} \} \parallel \{ \underbrace{i++}_{a_2} \} \text{parend}; y := i; \text{end};$

- 'Three' admissible execution orders:

- ▷ quasi-parallel execution orders $a_1; a_2$ or $a_2; a_1 \implies y \approx 7$
- ▷ truly parallel execution $a_1 \parallel a_2 \implies y \approx 6$
actions read simultaneously 5, increment by 1 and write 6 back
 $\implies \text{Pseudo-Parallelism does not simulate true parallelism.}$

- ◀ On execution level there are not only 2, but 'at least' 6 actions:

$i++ \approx i=i+1 \approx \text{LD R1}, i; \text{INC R1}; \text{ST R1}, i$

- * Load $\approx \text{MEM} \rightarrow \text{Register} / \text{STore} \approx \text{Register} \rightarrow \text{MEM}$
- * OS-scheduling may interrupt processes at all times
- * Example: LD_a1; INC_a1; LD_a2; ST_a1; INC_a2; ST_a2;

- ◀ more than three admissible execution orders on machine level

$\implies \text{Atomicity must not be assumed at program level.}$

III.2 Coordination, Interaction and Causality

- ▶ **Coordination** is needed to guide the work of different processes running in a distributed system towards a common goal, e.g.,
 - * organizing a parallel search in a bunch of documents
 - * reaching a common agreement concerning a trusted entity
- ▶ **Interaction** is required to implement coordination efforts.
 - a process **publishes** (part of) its state, e.g.,
 - * handing a file descriptor to another process, e.g. OS
 - * forwarding a search result for filtering
 - a process **re-acts** based on the state of another process, e.g.,
 - * Printing a file when signalled as 'completed'
 - * Continue with a calculation when all partial results are available

Interaction is implemented based on the underlying paradigm:

- SMS: Writing and reading the same data
- DMS: Sending and receiving messages

Dependencies, Causality and Execution Orders

- ▶ Interaction defines two principal **Roles**: '*provider*' vs. '*recipient*'
- ◀ Interaction introduces **dependencies** between actors of these roles:
 - * SMS: 'Reader' depends on 'Writer' to provide current data
 - * DMS: 'Receiver' depends on 'Sender' for sending message

Principle of Causality: '*Cause and Effect*'

- Information is only accessible *after* it has been provided.
- The *execution order* between different parts of an interaction is essential for the overall result, e.g.,
 - ◀ A Reader may read outdated (current) data when the read access is executed before (after) the corresponding write actions, which may violate determinacy.
 - ◀ A Receiver may be blocked from further execution while waiting for a message that does not arrive which results in a non-terminating 'blocked' system.

c.f.
pg.
III-1

Example: One-way Interaction and Dependency

(i) Shared-Memory-Model

```
g=0; parbegin {g=F(3); a=2*g} // {x=3*g} parend
```

- * P2 reads variable written by P1

⇒ **Execution order is important for P2, but not specified**

SMS requires additional restriction on execution order \sqsubset_{PS}

(ii) Message-Passing-Model: two alternative specifications (A/B)

(A) parbegin {g=0; g=F(3); snd(g,P2); a=2*g}
 // {g=0; rcv(g,P1); x=3*g} parend

(B) parbegin {g=0; snd(g,P2); g=F(3); a=2*g}
 // {g=0; rcv(g,P1); x=3*g} parend

- * P2 receives and uses ('old' vs. 'new') value from sender P1

⇒ **Execution order is important and specified via snd/recv**

DMS: Access and order restriction in a single statement

Example: Mutual Interaction and Dependency

(i) Shared-Memory-Model

```
g = 0; h=0; parbegin {g=h+1} // {h=g+2} parend;
```

- Processes P1 and P2 read and write the same variable:

P1 before P2: $g \leftarrow 1$ and $h \leftarrow 3$

P2 before P1: $g \leftarrow 3$ and $h \leftarrow 2$

⇒ **Execution order relevant for both processes.**

Additional Problem: true parallelism and atomicity

P1 and P2 in parallel: $g \leftarrow 1$ and $h \leftarrow 2$

⇒ **Execution sequences without interrupts are needed**, i.e., guaranteeing atomicity for sequences consisting of many steps.

(ii) Message-Passing-Model: no simple one-to-one correspondence, but sending and receiving of data among more than two processes causes similar effects and problems.

c.f.
pg.
III-9

A Dependency Model for SMS and DMS – 1

Abstract Model independent of underlying programming model

- Information exchange is implemented by data exchange.
- Supplying data by a *provider role* is done by **Write**-Actions.
- Using data by a *recipient role* is done by **Read**-Actions.

Definition III.4: (Basic Interaction Primitives)

Let PS be a process system, P a process and a an action of P .

1. The effect of an action a on the state of a process P is characterized by the following sets of data used:

$Read(a)$ denotes all **data read in** a

$Write(a)$ denotes all **data written in** a

$Data(a) := Read(a) \cup Write(a)$ denotes all data **used** by a .

2. The overall effect of P is defined by its overall data usage

$Read(P) := \bigcup_{a \in P} Read(a); \quad Write(P), Data(P) \text{ resp. } \blacklozenge$

A Dependency Model for SMS and DMS – 2

- The **basic inter-action step** between two processes P_1 and P_2 is executed by actions $a_1 \in P_1$ and $a_2 \in P_2$ such that $Data(a_1) \cap Data(a_2) \neq \emptyset$.
- The detailed structure of $Data(a_1) \cap Data(a_2)$ defines the type of dependency between the actions and, hence, the processes.

Definition III.5: (Different Types of Dependencies)

Let PS be a process system and $a_1 \in P_1; a_2 \in P_2$ actions.

1. One-way Read-Write-Dependency:

$$a_1 \xrightarrow{rw} a_2 : \iff Write(a_2) \cap Read(a_1) \neq \emptyset$$

2. Mutual Read-Write-Dependency:

$$a_1 \xleftrightarrow{mu} a_2 : \iff (a_1 \xrightarrow{rw} a_2) \wedge (a_2 \xrightarrow{rw} a_1)$$

3. Mutual Write-Write Conflict:

$$a_1 \xleftrightarrow{ww} a_2 : \iff Write(a_1) \cap Write(a_2) \neq \emptyset$$
◆

Example: Write-Write Dependencies

(\xleftrightarrow{ww}) : $x := 1; z := x; a := 3;$
 parbegin $y := \underbrace{x+z}_{a_1} \parallel \underbrace{y := 2*a}_{a_2}$; parend;
 $a := x+y;$

$a_1 \xleftrightarrow{ww} a_2$, because $Write(a_1) \cap Write(a_2) = \{y\} \neq \emptyset$

- ▷ a_1 executed before $a_2 \implies a \approx 7$
- ▷ a_2 executed before $a_1 \implies a \approx 3$
- ▷ $a_1 \parallel a_2$ executed truly parallel $\implies Value(y) \in \{2, 6\}$ undefined

Note: Dependency relation is not always transitive!

... parbegin ... $x := y+1$... $\parallel \dots y := 2*z$... $\parallel \dots z := a+b$... parend; ...

- $a_1 \xrightarrow{rw} a_2$ caused by y and $a_2 \xrightarrow{rw} a_3$ caused by z
- a_1 and a_3 with disjoint Read/Write-sets are 'independent'.

Problem: if a_1 is executed after $a_2 \implies$ the order between a_2 and a_3 matters for a_1 *(debugging nightmare)*

Dependencies, Concurrency and Non-Determinism

- ▶ *Imperative, sequential programming* is built upon MEM accesses, i.e., '*assignments*', and dependencies but can rely on a fixed total linear execution order.
- ◀ *Imperative, concurrent programming* utilizes the potential of non-determinism and partial execution orders:
 - Different executions of the same system in different orders may lead to different results, so-called **Race-Conditions**.
 - The causal order implied by dependencies has to be respected.
⇒ otherwise, the result is an **in-determinate process system**.
- ▶ *Imperative, concurrent programming* makes use of *synchronization* to restrict the permissible execution orders such that
 - * dependencies are respected as much as needed in order to guarantee determinacy, and the
 - * potential of non-determinism is preserved as much as possible.

Definition III.6: (Synchronization)

Let Prog be a concurrent program that describes the process system PS and its action execution order \sqsubset_{PS} . The extension of \sqsubset_{PS} by means of additional **order restrictions** for ensuring determinacy of PS is called **synchronization**.

Statements used in the modified program Prog' in order to specify synchronization are called **synchronization mechanisms**. ◆

- ▷ *Shared Memory* uses special mechanisms that guard parallel access to data or block processes until current data are available.
- ▷ *Distributed Memory* uses `snd` and `rcv` which are inherently causally ordered because a message can only be received if it has been sent before, i.e., $\text{snd} \sqsubset_{PS} \text{rcv}$.

Caution: Synchronization should be used with great care!

- ▶ actions waiting for predecessors that are not executed are **blocked**.
- ▶ cycles of order restrictions among processes may lead to **deadlocks**.

Example: Synchronizing One-Way Dependencies

1. **One-way dependency** $a_1 \xrightarrow{rw} a_2 \implies \sqsubset_{PS} := \sqsubset_{PS} \cup a_2 \sqsubset a_1$
is respected using one-way synchronization

- ▶ a priori known order \implies fixed **statically** in program code
 Example: P2 requires result from P1
- ▶ **Simple Mechanism:** $signal(k)/wait(k)$ where $k \in \mathbb{N}$
 Semantics: $\forall k \in \mathbb{N}$ holds $signal(k) \sqsubset wait(k)$

Example:

(left side without – right side using synchronization)

```
g=0; parbegin
  {g=F(3); a=2*g}
  //
  {x=3*g}
parend
```

```
g=0; parbegin
  {g=F(3); signal(S); a=2*g}
  //
  {
    wait(S);
    x=3*g
  }
parend
```

c.f.
pg.
III-12

- * Undesirable order no longer admissible: system is determinate
- * Message Passing uses `snd/rcv` with 'implicit' `signal/wait`

right
side

Example: Synchronizing Mutual Dependencies

2. **Mutual Dependency** $a_1 \xleftrightarrow{mu,ww} a_2 \implies (a_2 \sqsubset a_1) \text{ or } (a_1 \sqsubset a_2)$
has to be respected using multi-way synchronization

◀ **Problem:** often no real preference for a specific order
 \implies static decision for one-way synchronization makes no sense.

► **Typical Scenario:** lots of processes access common data

- * actual execution order is insignificant, but
- * specific Read and Writes should be executed without interrupt
i.e., *Atomicity for a sequence of actions* is required

 \implies **dynamic mutual exclusion synchronization mechanism**

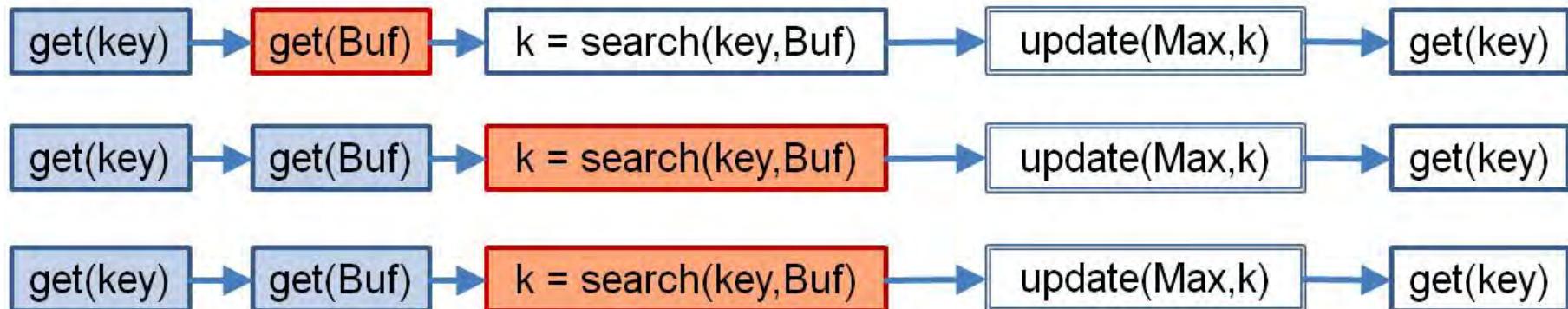
Example:

- calculation of the frequency of key occurrences in text buffers Buf
- single processes search *locally* for a single key in a single buffer
- *global* updates add local frequencies up to a global result

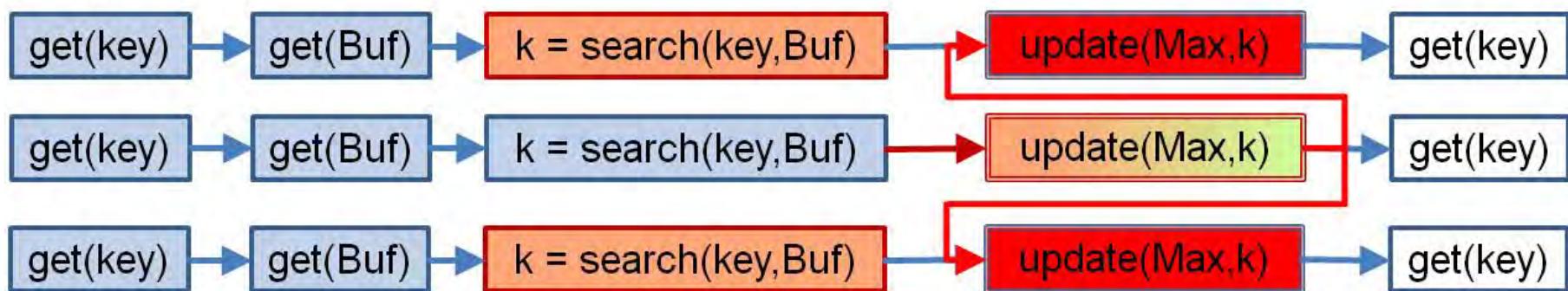
c.f.
pg.
III-21

Example: 3 Processes with local orders plus update

State 1 of PS



State 2 of PS



Execution state of process actions

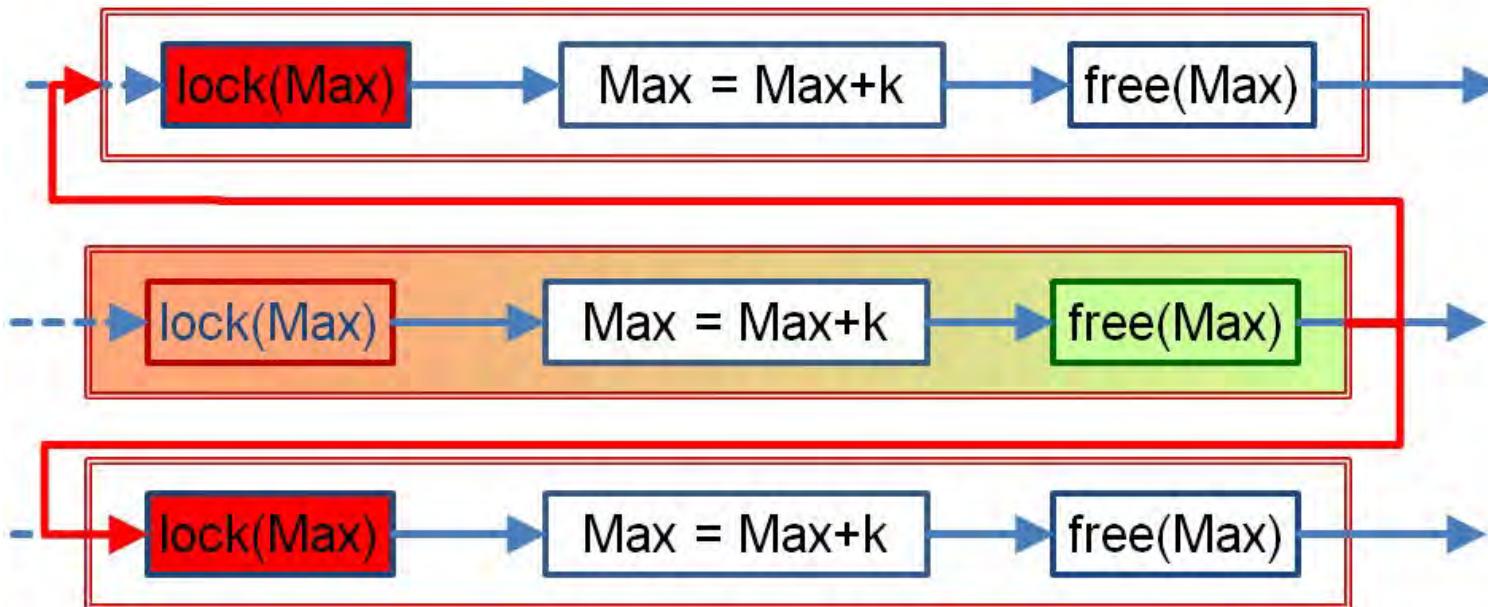
Original action execution order

Additional order pairs due to synchronization



- Max used by all processes in update \implies all processes are mutual dependent w.r.t. Max, i.e., \xleftarrow{mu} and \xleftarrow{ww}

Example: Internal Mechanism for Synchronization



Programming language construct: lock/free for variables

- Detailed execution order does not matter for overall result.
- 'First' successful `lock(Max)` introduces new \sqsubset_{PS} pairs between `free(max)` at the end of the active update and *all other* `lock(Max)` actions yet to be executed.
- Executing corresponding `free` supersedes new pairs from \sqsubset_{PS}
- Reading and writing Max is done without any other process accessing Max during this sequence of steps.

Definition III.7: (CriticalSection and Mutual Exclusion)

Let $K \subseteq PS$ be a set of processes from PS that are mutual dependent and let M be the data this dependency is based upon, i.e., M is part of the intersection of data sets from all processes.

1. A sequence of actions $cs = a_l a_{l+1} \dots a_f$ of $P \in K$ that causes the dependency is called **critical section** w.r.t. M : \iff
If cs is **active**, the execution order \sqsubset_{PS} of PS ensures that no other process from PS executes an action a which uses data from M , i.e., $Data(a) \cap M \neq \emptyset$.
2. The class $CS(M)$ of all cs from K is **strictly mutual exclusive** w.r.t. data M . ♦

Remark: A finite sequence of actions $a_l a_{l+1} \dots a_f$ is **active** in a state of PS if at least a_l has been executed and a_f has not yet finished its execution.

A weaker Notion of Critical Sections

- ▷ A **critical section** of **Order** k ($k > 0$) allows for a **maximum** of k active critical sections cs in parallel (at the same time).
- ▷ Strict mutual exclusion is the special case of a cs of order $k = 1$

Example: Reader-Writer Problems

- k Reader processes R_1, R_2, \dots, R_k use data M
- m Writer processes W_1, \dots, W_m write data M
- Restrictions on parallel access:
 1. Only one Writer is allowed at a specific time ($\xleftarrow{\text{ww}}$)
 2. Writers and Readers are mutual exclusive ($\xrightarrow{\text{rw}}$)
 3. A maximum of $k = 3$ Readers are allowed for parallel access, e.g., caused by insufficient resources

c.f.
pg.
III-43

Definition III.8: (Blocking, Deadlocks, Starvation)

Let PS be a process system with processes $\{P_1, \dots, P_n\}$.

1. PS is **safe** w.r.t. *critical sections* : \iff Definition III.7 holds for all critical sections $CS(M)$ of PS .
2. A process $P \in PS$ is **permanently blocked** if the rule R does prevent P permanently from the execution of its next action because P has to wait for actions of other processes from PS .
3. A subset $D \subseteq PS$ of processes is called **Deadlock** if all processes of D are permanently blocked due to a cyclic wait for actions from other processes from D caused by \sqsubset_{PS} .
4. A process $P \in PS$ is treated **unfair**, i.e., is subject to **Starvation** if the process P cannot proceed for an infinite amount of steps, although the next action of P is allowed by the rule R . ◆

- *Blocking* and *Deadlock* are caused by the specification level \sqsubset_{PS} .
- *Starvation* is caused by the execution environment, typically by keeping a process from resource access for an infinite long 'time'.

Cooperation, Coordination and Interaction

- ▷ In order to get common work done, processes have to *cooperate*.
- ▷ Cooperation is made possible by *coordination* using *interaction*.
- ▷ *Interaction* implies interaction roles and causes *dependencies*.
- ▷ Combining dependencies with non-deterministic specifications of execution orders may lead to *race conditions* and, hence, indeterminate systems.
- ▷ In order to *ensure determinacy*, additional restrictions on execution orders and, thus, reduced potential for parallel execution may be required.
- ▷ *Synchronization* is used to implement execution order restrictions.
- ▷ *Synchronization mechanisms* are dependent from the underlying interaction model.

Cooperation: *Coordinated Interaction ruled by means of synchronization mechanisms to ensure determinate systems in order to get some common work done.*

SMS vs. DMS Synchronization Specification

Shared-Memory paradigm:

- Interaction \approx Read/Write of common variables, data blocks or files
- Order restrictions among processes:
 - ◀ Imperative programming languages use dedicated statements for synchronization, e.g., signal/wait or locks.
 \implies **Synchronization and interaction in separate constructs**
 - ▶ Object-oriented languages combine synchronization with attributes or methods (code blocks) of classes.
 \implies **Synchronization and interaction object-based.**

c.f.

pg.

III-10

Message-Passing paradigm:

- Interaction by means of sending/receiving messages
- Order restrictions are built into Message-Passing constructs
 \implies **Synchronization and Interaction in the same construct.**

III.3 Implementing Synchronization for SMS

Important Facts:

- Atomicity on single nodes is an indispensable prerequisite for synchronization among different nodes.
 - On a single node, hardware-based atomic actions are required.
 - ◀ Synchronizing $n > 2$ processes is much harder than synchronizing only 2 processes, esp. due to *fairness* considerations.

Different Levels and Efficiency Considerations:

Example: No synchronization without atomic basis

Objective: Data M should be used **mutual exclusive** by P_1 and P_2

Attempt: M 'guarded' by int d : M free $\iff d==0$; d initial 0

- processes check actively whether ($d == 0$) before entering cs
- processes assign $d = 1$ when entering and $d = 0$ before leaving cs

Entry Prologue for P_1 $\dots \overbrace{\text{while } (d \neq 0) \{ \}}; d=1 \dots cs_1 \dots \overbrace{d=0};$	$\overbrace{\text{Epilogue}}$ $\dots \overbrace{\text{while } (d \neq 0) \{ \}}; d=1 \dots cs_2 \dots \overbrace{d=0};$
Entry Prologue for P_2	$\overbrace{\text{Epilogue}}$

Problem: Reading a value and writing afterwards is **not atomic**

- ◀ P_1 and P_2 read d in state 0; both assign $d = 1$; both enter cs
 \implies process system is *not safe* w.r.t. M .
- ◀ Scheduling may allow P_1 access $n > 1$ times, even if P_2 is also trying to get access to M \implies process system is *not fair*.

Hardware Basis for Atomic Actions

Atomicity basis depends on underlying hardware model:

- ▷ based on **Processors**:
 - **Single cores/processors**: masking interrupts for some steps
 - **Multi-cores/processors**: locking access to bus/interconnect
- ▷ based on **Memory** access
 - ▷ locking memory regions for single process access
 - requires special memory HW support, e.g., SMS supercomputers
 - ▷ special hardware support for **atomic read-and-write**
 - ▷ easy to implement efficient operating system level primitives
 - ◁ not always available and not portable across architectures
 - ▷ serialized MEM support for **atomic reads or writes**
 - ◁ algorithms a bit more 'tricky' to ensure atomicity
 - ▷ work on all machines due to lower demands
- Note:** restricted to *basic data types* like boolean or int

Atomic read-and-write for synchronizing 2 processes

Assembler construct: `test_and_set(x,R)`

- Parameters: *local* register R; boolean variable x from MEM
- Semantics: **uninterrupted** execution of a 3 statement sequence

```
begin R = x; if (R == 0) then x=1 fi; end
```

Implementing mutual exclusion: initial state (`bolt == 0`)

$x_1=1; \dots$ <div style="text-align: center; margin-left: 100px;"> <i>Entry Prologue for P_1</i> </div> <div style="margin-left: 100px;"> $\overbrace{\text{while } (x_1==1) \text{ test_and_set}(bolt,x_1)} \dots cs_1 \dots$ </div> <div style="text-align: right; margin-right: 100px;"> <i>Epilogue</i> </div> <div style="margin-right: 100px;"> $\overbrace{bolt=0;}$ </div>	$x_2=1; \dots$ <div style="text-align: center; margin-left: 100px;"> <i>Entry Prologue for P_2</i> </div> <div style="margin-left: 100px;"> $\overbrace{\text{while } (x_2==1) \text{ test_and_set}(bolt,x_2)} \dots cs_2 \dots$ </div> <div style="text-align: right; margin-right: 100px;"> <i>Epilogue</i> </div> <div style="margin-right: 100px;"> $\overbrace{bolt=0;}$ </div>
---	---

- **Interpretation:** cs is free, i.e., bolt is open $\iff (\text{bolt} == 0)$
- Epilogue is not critical as it is only a single write operation
- **Additional assumptions:** cs_1/cs_2 last only finite time
bolt is only used in prologue/epilogue

Atomic read-and-write for $n > 2$ processes

Basic Concept: *Epilogue hands cs over to 'next' waiting process*

- boolean `Wait[0:n-1]` initial `[0, ..., 0]` registers waiting processes
- $\text{Prologue}_i \approx$ register in `Wait[i]` and set flag X_i (1)
- Condition for waiting: P_i is registered **and** flag X_i is set (2)
- End of Prologue_i : unregister via (`Wait[i] = 0`) (3)
- Epilogue_i : iff no process is waiting $\implies cs$ is freed: `bolt=0`; (4-6)
otherwise free next process P_j by (`Wait[j] = 0`) (7)

▷ Prologue_i :

- (1) `Wait[i]=1; Xi=1;`
- (2) `while (Wait[i]==1) & (Xi==1) test_and_set(bolt,Xi);`
- (3) `Wait[i]=0;`

▷ Epilogue_i :

- (4) `j = (i+1) mod n; /* next process in array */`
- (5) `while (j <> i) and (Wait[j] == 0) do j = (j+1) mod n; od;`
- (6) `if (j == i) then bolt=0;`
- (7) `else Wait[j]=0;`

A two-process solution using atomic read-or-write

Characteristics:

- **Weak assumption:** only a single Read **or** Write is atomic
- Processes synchronize based on active waits, i.e., spin locks

Techniques to ensure safeness under these conditions:

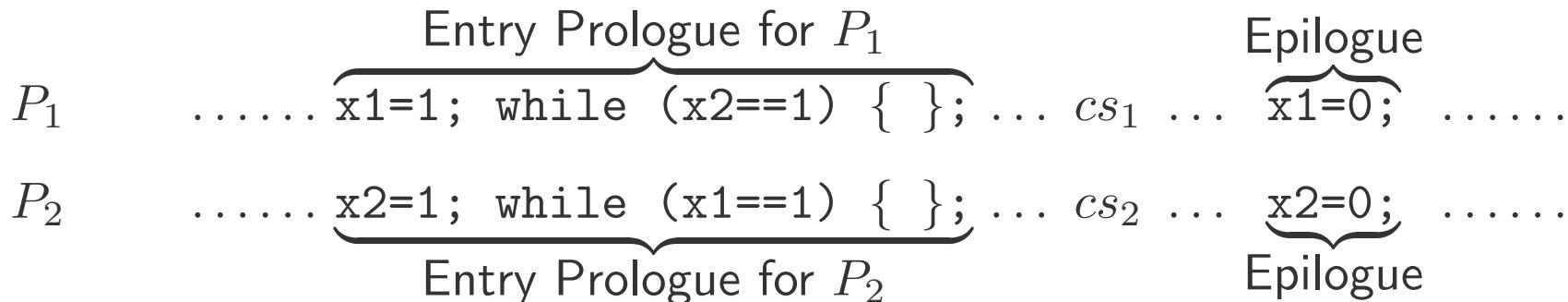
1. Always register first before trying to enter a critical section in order to ensure safeness.
Counterexample: Naive Algorithm with parallel access to bolt
2. Avoid cyclic waits and deadlocks by introducing additional decision criteria that respect fairness in case of conflict.
3. Avoid starvation by means of introducing a round-robin mechanism among all waiting processes.
4. Fairness considerations are only meaningful if there is more than one waiting process.

c.f.
pg.
III-29

Unsuccessful Attempts using atomic read-or-write

1. separate global variables; register first, then test

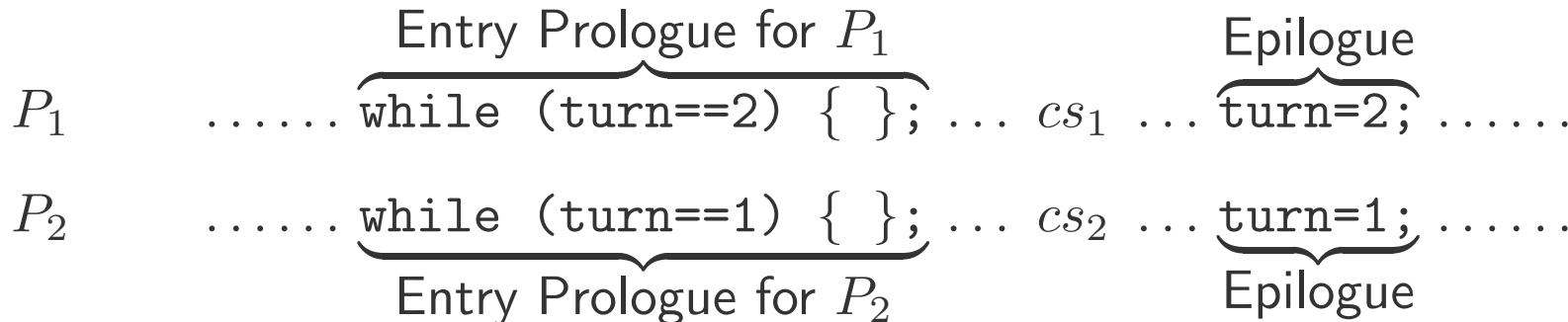
$x1=0; x2=0;$



⇒ safe, but system may end up in **Deadlock**

2. same variable but processes use distinct values (round-robin)

$turn \in \{ 1, 2 \} ;$



⇒ safe, no deadlock, fair, but prone to **permanent blocking**

Peterson-Algorithm synchronizing 2 processes

(1981)

x1=0;

 $\{0, 1\} \approx P_1$ {not registered, registered}

x2=0;

 $\{0, 1\} \approx P_2$ {not registered, registered}

turn=1;

 $\{1, 2\} \approx$ decides in case of conflict

Prologue₁ x1=1; turn=1;
Process P₁
 while (x2==1) and (turn==1) {};

critical section cs₁

Epilogue₁ x1=0;

Prologue₂ x2=1; turn=2;
Process P₂
 while (x1==1) and (turn==2) {};

critical section cs₂

Epilogue₂ x2=0;

Case-based Analysis of Peterson-Algorithm – 1

- ▶ Processes register using private variable **before** $cs \implies \text{Safeness}$
- ▶ 'turn' ensures round-robin access $1-2-1-2-1-\dots \implies \text{Fairness}$
- ▶ 'turn' influences execution in case of conflict only (and)
 $\implies \text{no permanent blocking}$

1. **Safeness:** P_i in $cs_i \implies$

- (a) $(x_{3-i} == 0) \vee (turn \neq i)$ due to while-Condition **and**
- (b) $(x_i == 1)$, due to registration; reset at the end of cs_i only.

Case Analysis for P_i in cs_i :

- I: $(x_{3-i} == 0) \implies P_{3-i}$ not in cs_{3-i} due to (b)
- II: $(x_{3-i} == 1) \implies (turn \neq i)$ due to (a) $\implies (turn == 3 - i)$
 $\implies P_{3-i}$ waits in Prologue $_{3-i}$ $\implies P_{3-i}$ not in cs_{3-i}

Result: always at most one of the processes is in **critical section**

Case-based Analysis of Peterson-Algorithm – 2

2. **P_i permanently blocked?** Prologue $_i$ is only critical sequence

$\implies P_i$ permanently in Prologue $_i$

\implies Condition $(x_{3-i} == 1) \wedge (turn == i)$ holds permanently

$\implies P_{3-i}$ registered & $(turn = 3 - i)$ **before** P_i

$\implies P_{3-i}$ is allowed to enter critical section cs_{3-i}

As cs_{3-i} is finite $\implies P_{3-i}$ executes Epilogue $_{3-i}$, esp., $x_{3-i} = 0$

\implies Condition for P_i does not hold anymore (Contradiction)

3. **Starvation for P_i ?** Reason: P_{3-i} '**passes**' P_i $n > 1$ times

P_i waits in Prologue $_i$; P_{3-i} in cs_{3-i}

As cs_{3-i} is finite $\implies P_{3-i}$ executes $x_{3-i} = 0$ in Epilogue $_{3-i}$

But: P_{3-i} re-enters Prologue $_{3-i}$ **before** P_i executes its test anew

P_{3-i} executes $(x_{3-i} = 1) \implies P_i$ still has to wait

P_{3-i} executes $(turn = 3 - i) \implies P_{3-i}$ blocks itself in Prologue $_{3-i}$

$\implies P_i$ evaluates $(turn \neq i)$ and leaves Prologue $_i$ (Contradiction)

Alternative Analysis – State-Space Exploration

Graphical Model: Algorithm is ruled by three variables

- ▷ States $Q \approx < x_1, x_2, turn > \in \{0, 1\} \times \{0, 1\} \times \{-, 1, 2\}$
- ▷ Initial state $q_0 < 0, 0, - >$ (initial value of turn is irrelevant)
- ▷ **Actions:** Prologue-Steps, cs_i ; cs_{3-i} , Epilogue-Steps
- ▷ only finite set of states \implies finite diagram by using loops

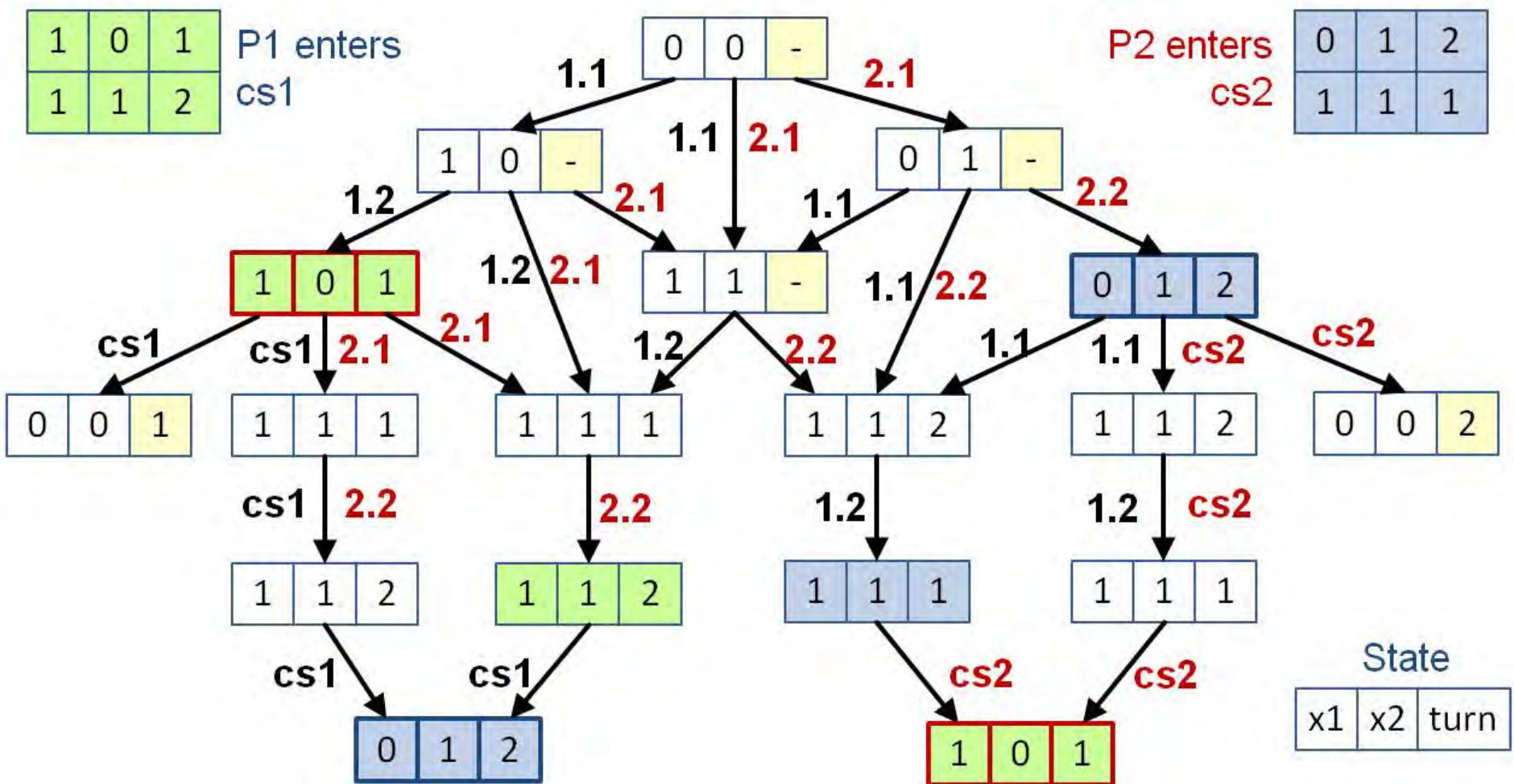
Properties of Algorithm defined in terms of diagram states:

1. If there is no state s.t. cs_1 **and** cs_2 can be entered \implies **safe**
2. If there is no state s.t. **only** Prologue-Loops are executed in both processes \implies **no permanent blocking**
3. If in each cyclic, i.e., possibly infinite run cs_1 **and** cs_2 are allowed in a round-robin fashion iff both are registered \implies **no starvation**

Pros and Cons for these kinds of models:

- ▷ Case Analysis is prone to the same errors as the algorithm.
- ◁ State-Space tends to get huge for complicated systems.

State Space Analysis - Peterson Algorithm



```
x1=1; turn=1; while (x2==1) and (turn==1) { }; ... cs1 ... x1=0;
x2=1; turn=2; while (x1==1) and (turn==2) { }; ... cs2 ... x2=0;
```

Assessment of low-level Synchronization Solutions

- *Indispensable basis for any kind of synchronization*
 - Assembler level is not the ultimate basis for a solution.
 - Similar problems in electronics due to varying signal propagation times \implies *glitches* caused by *hazards*, i.e., race conditions
- ◀ *Missing abstraction*: Low level results in ill-structured code.
- ◀ *Wasted CPU-time*: Basic level has to rest on possibly long-time *active waiting* \implies costly in multi-process environments.

Solution: Multi-layered SW Architecture w.r.t. synchronization

- HW-supported *spin locks* for 'short' critical sections on OS level
- Higher-level lock-constructs based on *sleep locks* due to OS scheduling and WAIT-Queues in programming languages.
- Well-structured *object-oriented* and *pattern-based* techniques embedded in languages and libraries for parallel programming.

c.f.
pg.
III-28

III.4 Higher level SMS Synchronization

- ▷ Generalized high-level concepts
 - ▷ **Semaphores** as an abstraction from active wait
 - ▷ **Monitors** as the basis for object-oriented synchronization
- ▷ Standard data structures with synchronization properties

Definition III.9: (Semaphore – User View)

A *Semaphore* is a *shared integer variable* that is only manipulated by the following three operations:

- **init(sem, n):** initializes a semaphore **sem** where $n \in \mathbb{N}_0$.
- **P(sem):** decrements **sem** by 1
- **V(sem):** increments **sem** by 1

A semaphore **sem** respects always the following two rules:

1. All operations on **sem** are **mutually exclusive** actions.
2. A process trying a P() operation that would render the value of **sem** **negative** is blocked.

Dijkstra

Using simple Semaphores for Synchronization

1. One-sided synchronization: (a_2 before a_1)

Example: P_1 needs result in a_1 that is provided by P_2 in a_2

c.f.
pg.
III-19

Initialization: Semaphore s; init(s,0);

P_1 : P(s); a_1 ;

P_2 : a_2 ; V(s);

⇒ Semaphore implements a signal/wait mechanism.

2. Mutual exclusion: A specific semaphore for **each** data structure

Example: global g is updated in $n > 1$ processes by $g = g + N$

c.f.
pg.
III-13

Initialization: Semaphore s; init(s,1);

P_i : P(s); cs_i ; V(s);

⇒ Semaphore implements a lock/release mechanism.

Alternative Variants for Semaphores

- ▷ **boolean** Semaphore uses only $\{0, 1\}$ as its value domain
- ▷ **integer** Semaphore uses value domain \mathbb{N}_0 (**counting** Semaphore)
- ▶ **additive** Semaphore: Integer-Sem. with comfortable operations
 - $P(sem, k)$: decrements sem by $k \in \mathbb{N}$
 - $V(sem, k)$: increments sem by $k \in \mathbb{N}$

Note: $P(sem, k)$ blocks if $sem < k$

Application for an additive semaphore:

3. Reader/Writer system allowing 3 Readers maximum in parallel

Initialization: `Semaphore rw; init(rw,3)`

Reader: `... P(rw,1); READ; V(rw,1); ...`

Writer: `... P(rw,3); WRITE; V(rw,3); ...`

Writer is blocked by trying $P(rw,3)$ if a Reader is currently active.

c.f.
pg.
III-24

Definition III.10: (Semaphore – Implementation View)

A **Semaphore** is a data structure built from an integer **counter variable** `sem` and a **waiting room** for processes (`PCBs`) `susp`. The semaphore implements the following **mutual exclusive** operations:

1. `init(s,n) :: sem = n; susp = PCBQueue(); where n ∈ N0`
2. `P(s) :: sem = sem-1;`
 `if (sem < 0)`
 `{ susp.Enqueue(myself.PCB()); myself.suspend(); }`
3. `V(s) :: sem = sem+1;`
 `if (sem ≤ 0)`
 `{ proc = susp.DeQueue(); proc.resume(); }` ◆

Note:

- `|s.sem|` yields the number of blocked processes for integer `sem`.
- **Fairness** for a *single* semaphore is warranted if the `PCBQueue` is a **FIFO-Queue**.

Monitors for Object-Based Synchronization

Brinch-Hansen
1973
Hoare
1974

- ▶ Monitors explicitly connect the shared data to be guarded from parallel access and the operations used to do so.
- ▶ Monitor concept incorporates an explicit concept of **interface**.

Definition III.11: (Monitor (naive))

A **Monitor** is a **data structure** D consisting of

1. a set of **shared variables** and
2. a set of allowed **access operations** $OP = \{ op_1, \dots, op_n \}$.

OP constitutes a class of critical sections $CS(D)$ of order 1. ◆

- ◀ Naive monitor concept is much too restrictive:
 - No parallel actions on D permitted at all
 - Lots of meaningful interaction techniques are not implementable
- ▶ Relaxed monitor concept with additional functionality needed.

Example: Bounded-Buffer using Event-Variables

Motivation: $op_i \in OP$ only meaningful after $op_{j \neq i}$ changes data
 e.g., 'get' ('put') makes no sense if buffer is empty (full) \implies
 process has to release monitor in order to proceed eventually.

```

TYPE buffer = MONITOR                                (* BOUNDED BUFFER EXAMPLE *)
VAR count: INTEGER;
VAR not_full, not_empty: EVENT;
PROCEDURE put_buf(r: DATA)
  BEGIN IF is_full? THEN wait(not_full); ... signal(not_empty) END;
PROCEDURE get_buf(VAR r: DATA)
  BEGIN IF is_empty? THEN wait(not_empty); ... signal(not_full) END;
FUNCTION is_full? BEGIN is_full? := (count = 100) END;
FUNCTION is_empty? BEGIN is_empty? := (count = 0) END;
BEGIN count := 0 END;                               (* Initialization: Number of Elements *)
  
```

Language Construct: Monitor enhanced by *Event Variable* e

- `wait(e)` blocks calling process until `signal(e)` is issued
- `signal` has no effect if waiting queue for e is empty (\neq Semaphore)
- `signal(e)` re-activates one of the waiting processes: *Which one?*

Problem: Strategies for Passing Monitor Access

Process finishes monitor op : How to hand over monitor access?

- 'global' waiting queue for monitor may hold $n > 1$ processes
- 'local' queues for several events may also hold $n > 1$ processes

Application semantics is crucial: *No 'generic best solution'!*

1. P is granted mutex by op_i and blocks immediately due to $\text{wait}(e)$

P did not alter state \implies

processes P' waiting internally are not re-activatable \implies

release for processes waiting in 'global' queue outside monitor.

2. P holds mutex via op_i and alters state of data in monitor

\implies processes from all wait-queues may be allowed to proceed

\implies **Competition** among local and global wait-Queues

Note: Fairness considerations are in favor of processes waiting internally w.r.t. events.

Case Study: Synchronization in Java – 1

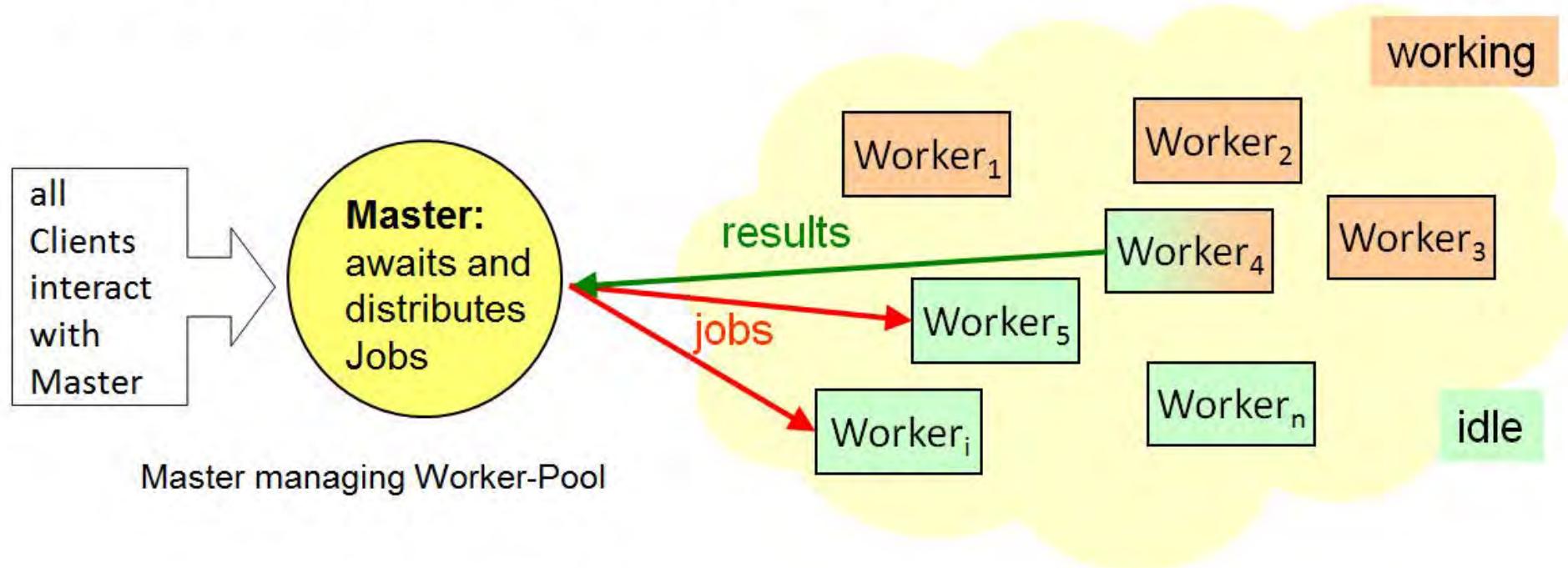
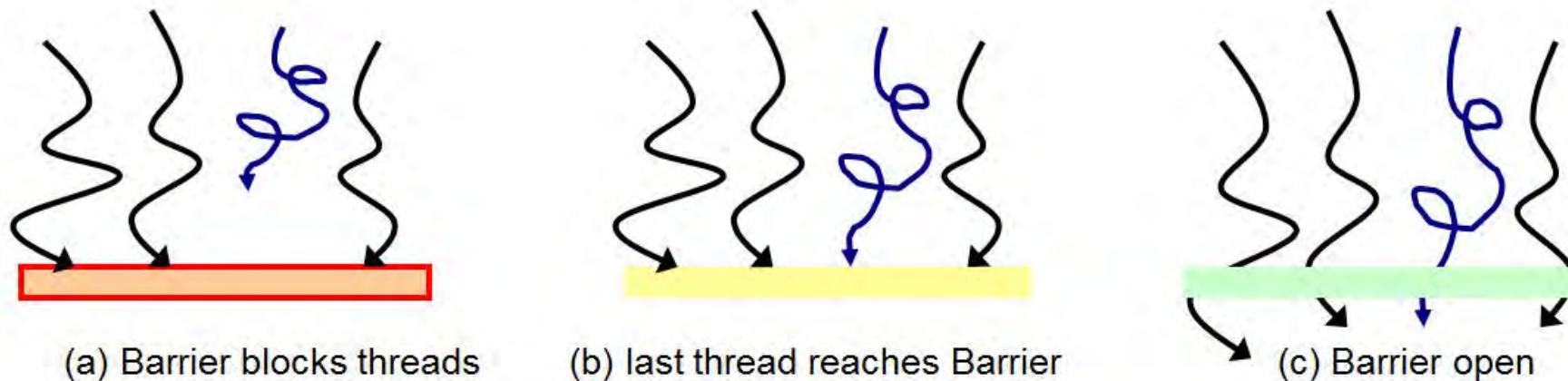
- ▷ Access to basic 32-Bit data is *atomic*, but
 - ◀ 64/128-Bit require $n > 1$ accesses and may be interrupted.
 - ◀ Internal optimization like *caching* prevents timely propagation of shared values between different threads on the same JVM.
⇒ Use `volatile` in order to eliminate both problems.
 - ▶ All objects are *monitors* and support *locking* by use of
 - `synchronized`: entire objects, methods or critical code blocks
 - Set of waiting threads (*WaitSet*) that is manipulated by
 - * Threads trying to acquire a 'used' lock end up in *WaitSet*
 - * Threads explicitly waiting (using timeouts) to release lock
 - * Thread interrupts and end of timeouts to quit waiting
 - * Object `notify` to re-activate a *random* single thread
 - * Object `notifyAll` to re-activate *all* waiting threads
- Problem:** *Fairness is not ensured, esp. with `notify`!*

Case Study: Synchronization in Java – 2

- ◀ Controlling threads and critical accesses explicitly using interrupts, wait-conditions or scheduling constructs, e.g., sleep, in code.
- ▷ `java.util.concurrent.Semaphore`: acquire/release
 - * constructed as boolean/additive as well as fair/unfair variant
 - * acquire interruptible, non-interruptible or non-blocking ('try')
- ▷ Atomic *Objects* for basic data structures supporting atomic `compareAndSet`, `getAndAdd` ... more efficiently than using locks.
- ▷ Queues, linked lists, hash maps etc. as high-level data structures supporting synchronized access efficiently.
- ▷ High-level thread control: `CountDownLatch`s or `Barriers` facilitating work distribution and master-worker architectures.
- ▷ High-level execution control: `Executors` and `Futures` in order to implement 'asynchronous systems' using postponed or *delayed executions* based on application logic.

c.f.
pg.
III-50

Example: Barrier and Master-Worker Paradigm



III.5 Message Passing Interaction

Constructs: $P_1: \text{SND}(\text{exp}, \dots) \rightarrow \dots$

$P_2: \dots \rightarrow \text{RCV}(\text{var}, \dots)$

Effect ($P_1 \mapsto P_2$): ' $\text{var}_{P_2} := \text{VAL}(\text{exp})_{P_1}$ '

Causality: Message transport requires time, i.e., $\text{SND} \sqsubset_{PS} \text{RCV}$

Characteristics of different Message-Passing Paradigms:

1. Message payload specification:

- Result of expression evaluation: int, reference to a linked DS
- Externalization and marshalling of *message* content

c.f.
II-11/
—
II-13

2. Destination address(es) specification

Note: this is not needed in a SMS paradigm

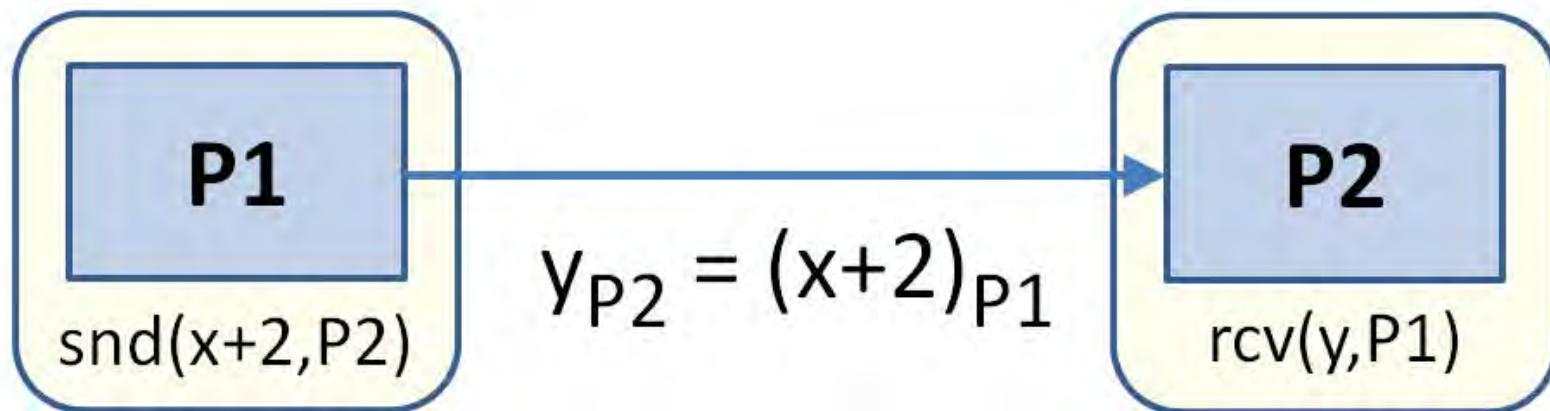
3. Different levels of synchronization and coupling between sending and receiving processes beyond $\text{SND} \sqsubset_{PS} \text{RCV}$.

4. Transient vs. Persistent Communication ranging from no buffering, short-term buffering of msgs, ..., message queueing

Identification of Message Destination - 1

1. Direct explicit Naming: Usage of 'process names'

- ◀ fixed identifier in code \implies barely usable
- ▷ Flexible, internal logical identifier management
 - * standard in parallel programming environments, e.g., MPI
 - * process identifiers as return values from process start managed using logical abstractions, e.g., 'neighborhood'

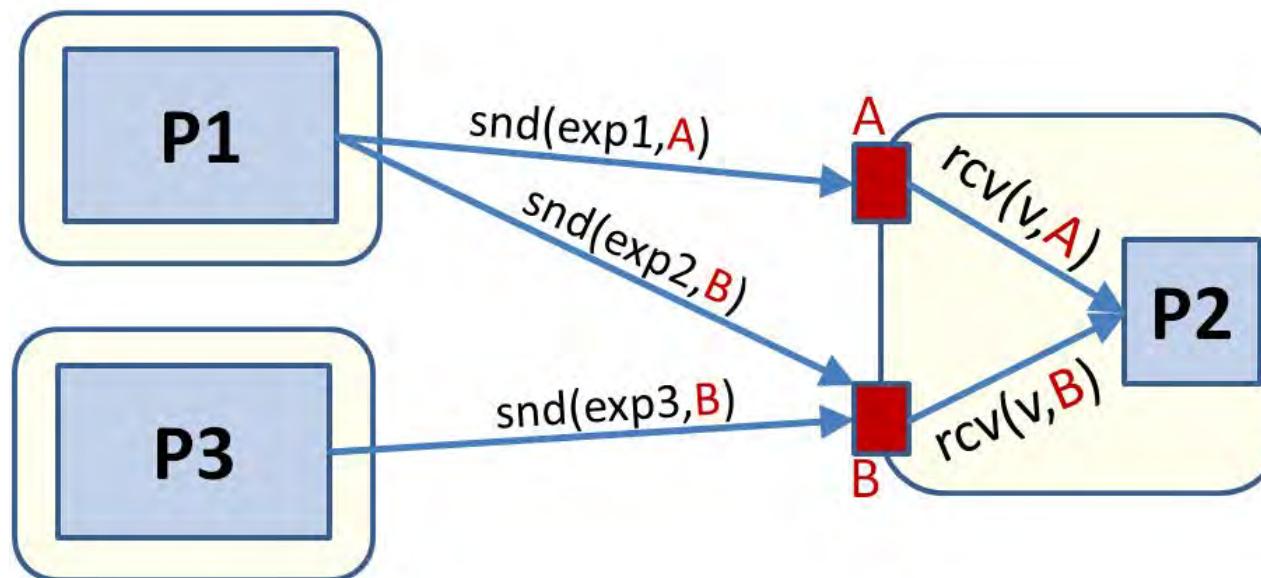


- ◀ Modularization and Encapsulation only sufficient when using well-documented, fixed naming conventions.

Identification of Message Destination - 2

2. Indirect Naming using Ports yields useful abstraction

- Sender and receiver process remain anonymous
- A single process may use different Ports
- Different processes may use the same port
- typically no (or only restricted) buffering at ports



permits n–1 and 1–n interaction
Ports not part of processes but at OS or NW layers
e.g. UDP sockets

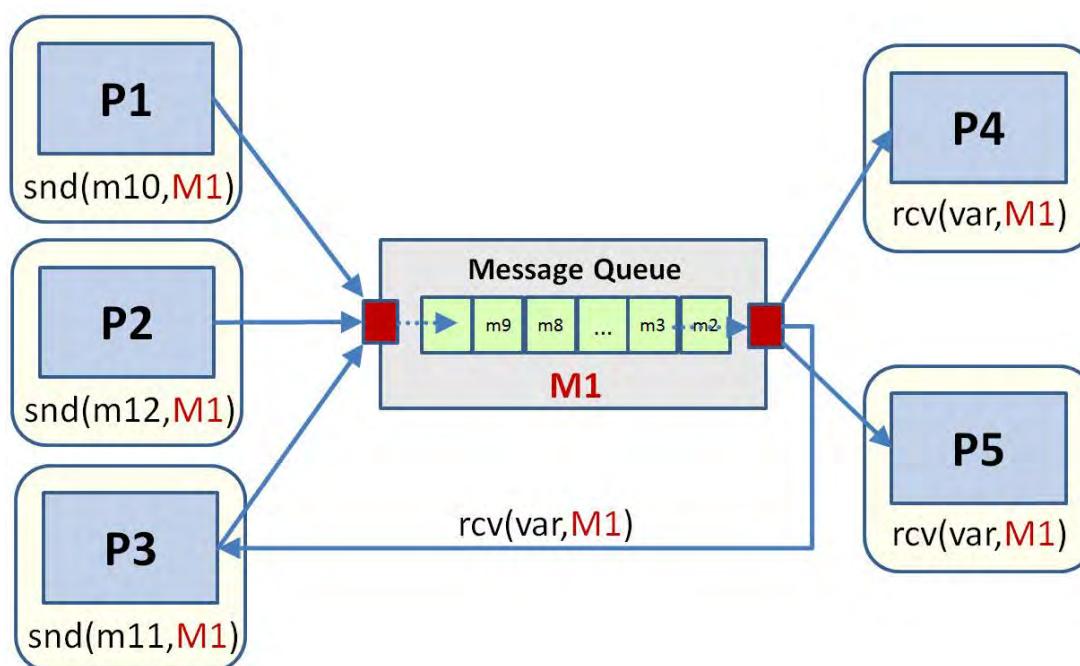
- ▷ Easy to re-use by parameterizing applications with logical port Identifiers, e.g., in program libraries.

Identification of Message Destination - 3

3. Indirect Naming using Channels

- Connect logical ports by means of **Channels**
- Definition/set-up by infrastructure specification or during process creation, e.g., TCP sockets

4. Mailbox: connections act as buffered channels, e.g., JMS



- ▷ high level of de-coupling w.r.t.
- Identification and Time**
- ◁ copying needed
- ▷ m-n-communication
- ▷ **broadcast/multicast** via non-destructive message receive

Synchronization among Communicating Processes

$$P^1 : q_0^1 \xrightarrow{a_1^1} q_1^1 \xrightarrow{a_2^1} \dots \dots \dots q_k^1 \xrightarrow{\text{snd}(P_2)} q_{k+1}^1 \xrightarrow{a_{k+2}^1} \dots$$

$$P^2 : q_0^2 \xrightarrow{a_1^2} \dots q_m^2 \xrightarrow{\text{rcv}(P_1)} q_{m+1}^2 \xrightarrow{a_{m+2}^2} \dots$$

c.f.
pg.
III-56

- **Asynchronous Communication:** inevitable effect $\text{snd} \sqsubset_{PS} \text{rcv}$

- q_{m+1}^2 in P_2 only reachable if P_1 in q_{k+1}^1
- $\xrightarrow{a_{k+2}^1}$ executable in P_1 even if P_2 not in q_m^2

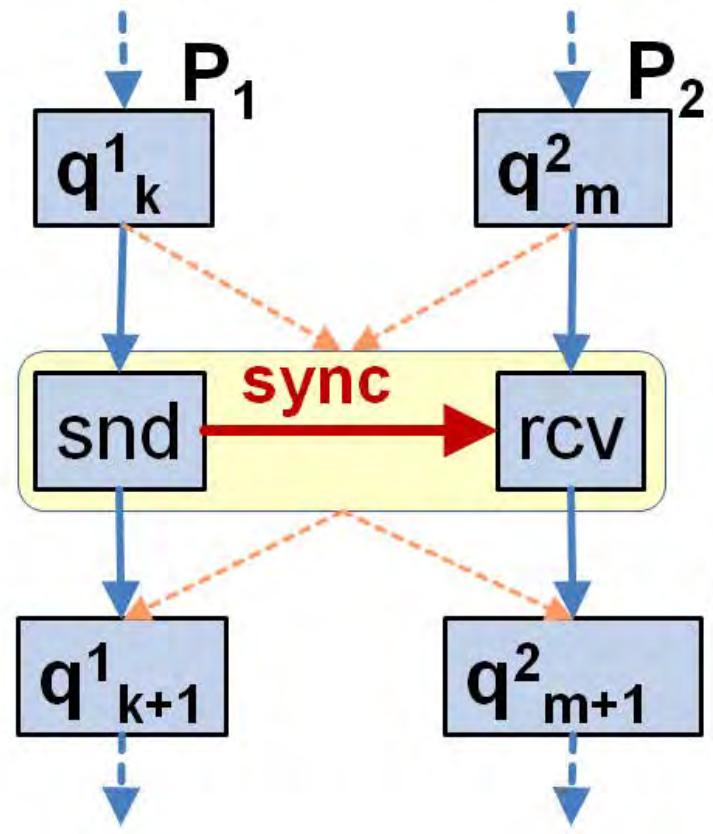
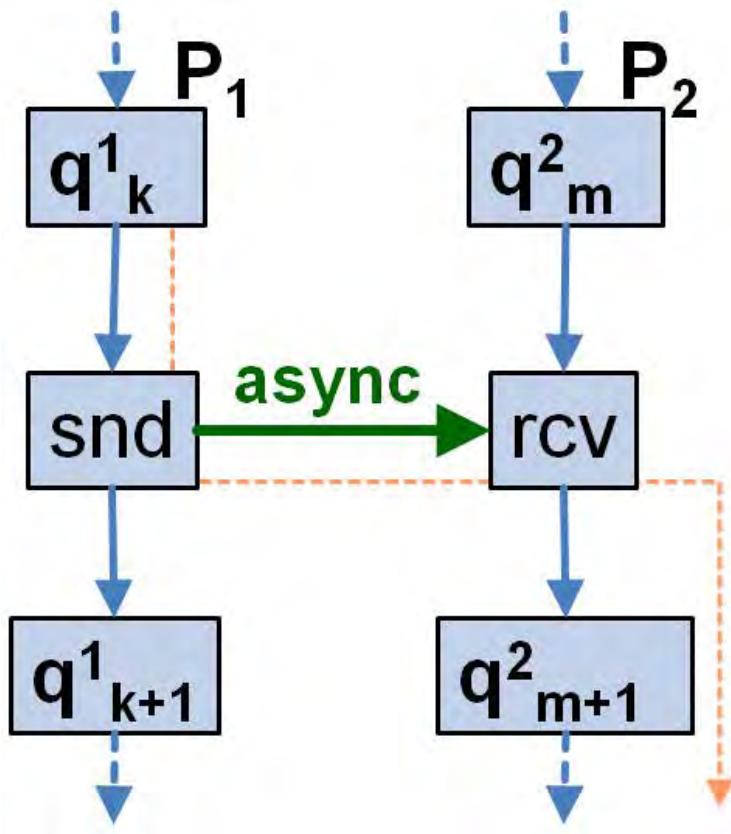
\implies Only P_1 affects execution of P_2 (**one-sided effect**)

- **Synchronous Communication:** maximum effect in both procs.
 rcv **after** $\text{snd} \wedge \text{snd}$ **after** $\text{rcv} \implies \text{coincident as a single action!}$

- q_{m+1}^2 in P_2 only reachable if P_1 in q_{k+1}^1
- q_{k+1}^1 in P_1 only reachable if P_2 in q_{m+1}^2
- $\xrightarrow{\text{snd}(P_2)}$ in P_1 ends if P_2 is in q_{m+1}^2

\implies **Symmetrical effect** for executing both, P_1 and P_2

Asynchronous vs. Synchronous Communication



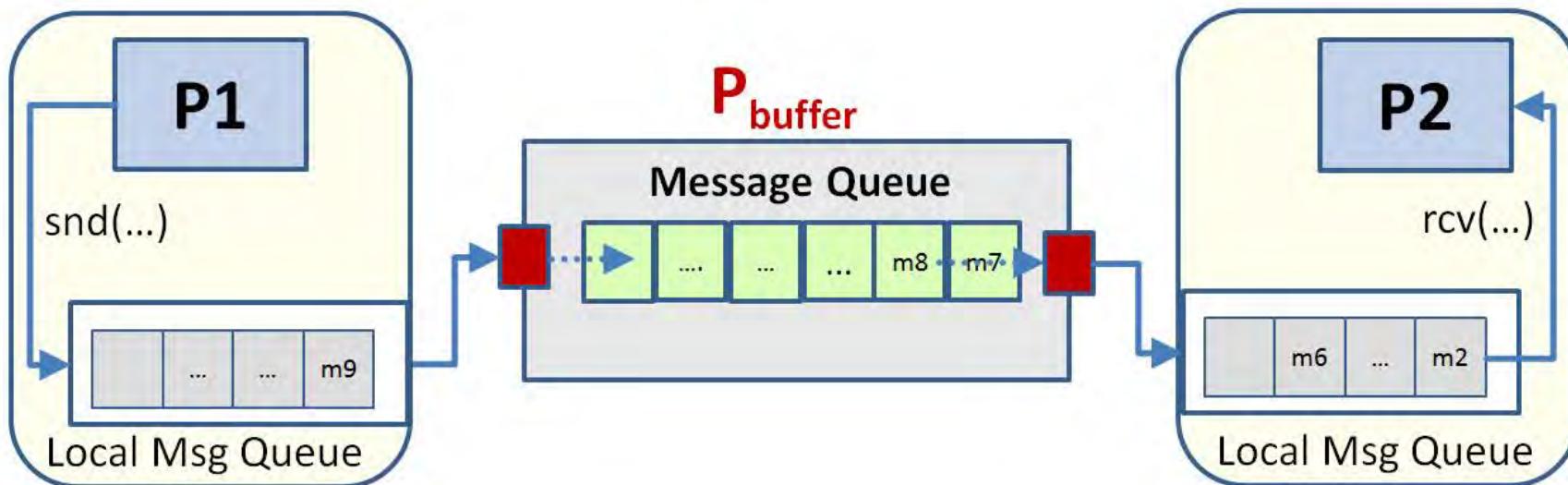
- ▷ easier in programming
- ▷ simulates sync (*handshake*)
 - $\text{snd}(P_2); \text{ rcv}(P_2)$
 - || $\text{rcv}(P_1); \text{ snd}(P_1)$
- ◁ but: limits to asynchrony
- ▷ easier to implement
- ◁ 'easier' to *Deadlock*, e.g.,
 - $\text{snd}(P_2); \text{ rcv}(P_2)$
 - || $\text{snd}(P_1); \text{ rcv}(P_1)$
- ◁ prone to programming errors

Limits in Implementing Asynchrony

Problem: P_1 sends *unbounded* many message to P_2

P_2 fails to execute `rcv(P_1)` frequently

⇒ How to build a buffer for infinite many messages?



Different Locations for Buffering:

- ▷ local buffer for sending process (OS-I/O-Queue)
- ▷ special buffer process P_{buffer} like, e.g., in a mailbox system
- ▷ local buffer for receiving process (OS-I/O-Queue)

Theory: Unfeasible as any buffer has only finite capacity

Approximating Asynchronous Semantics

Concept: Sending process acts like asynchronous communication is possible as long as there is sufficient buffer space; otherwise one of several possible programming models is used: *If buffer is full*

◀ **Buffer-blocking** sender semantics

- ⇒ block sender as in synchronous communication
- ⇒ \exists maximum **synchronous distance** caused by buffer size
- ⇒ Communication becomes synchronous if buffer is full

◀ **Lost messages** semantics

- ⇒ sender is not blocked but keeps sending
- ⇒ drop oldest messages ⇒ Non-blocking bounded queue.

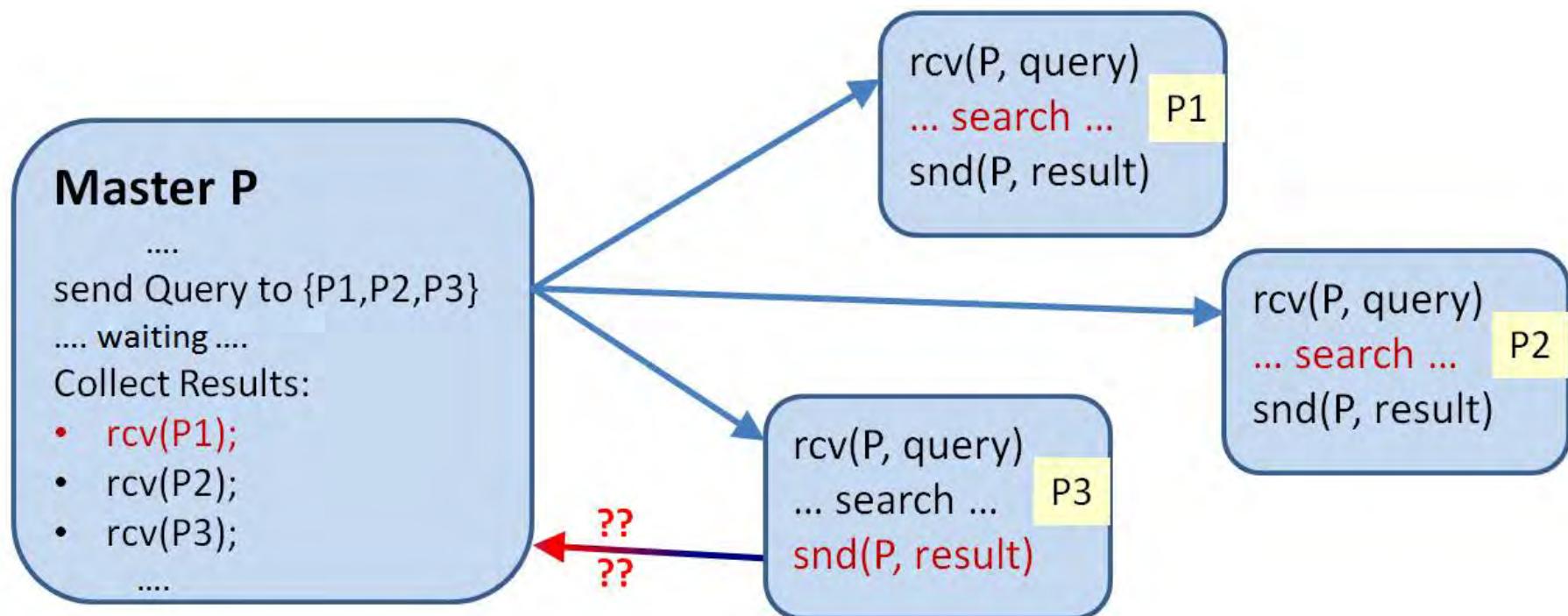
Problem: *Semantics is dependent from actual data.*

► **Raise an exception**, i.e., the programmer is handed the opportunity to wait, to drop messages (FIFO, LIFO, attributes) or to block the sender until some buffer space becomes available.

De-Coupling using Selective Receives – 1

Setting: $P \longleftrightarrow \{P_1, P_2, P_3\}$ interacts with $n > 1$ processes.

- ◀ Behavior of computation process is not predictable in detail
- ◀ no a priori known order of exchanges among processes known
- ⇒ programming a fixed order of `rcvs` is likely to block processes



⇒ P is blocked by `recv(P_1)` although P_3 is able to send a result.

De-Coupling using Selective Receives – 2

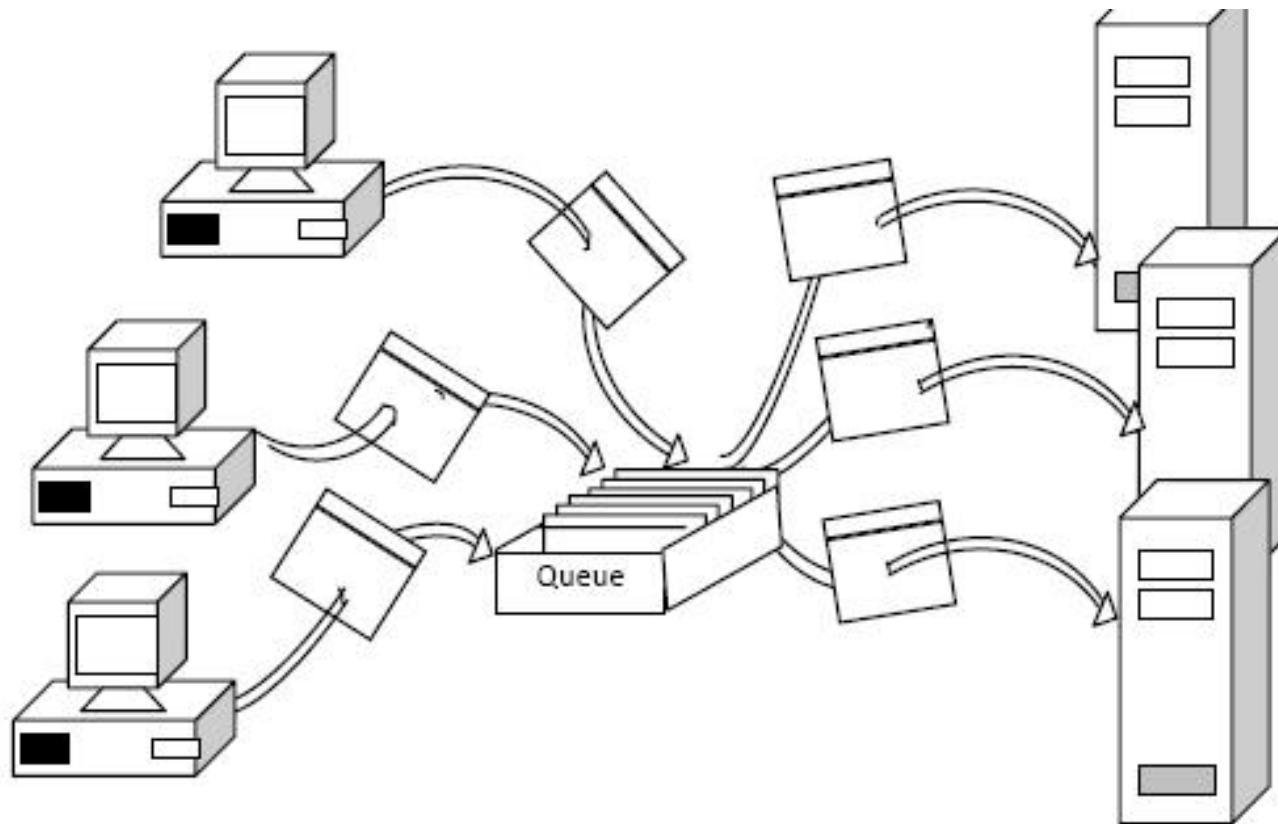
Design Space for Programming Language Solutions:

1. **Thread-per-port**: Each sender is served by a **thread** of its own
 - ⇒ check for messages in a round-robin fashion
 - ⇒ computation process is not blocked.
2. **rcv-from-any**: Use the same **port** for all sending processes
 - ⇒ blocks only if no message is found at the port at all
 - ⇒ blocking is no problem as there is no work to be done.
3. **Time-outs**: use receive constructs that allow for timeout settings
Problem: program tends to react differently due to variable loads and response times.
4. **probe** as a non-blocking test whether there are messages or not.
5. **Snd/Rcv as guarded commands**, e.g., in CSP or select in ADA
 - DO guard₁ → stmt₁ | ... | guard_n → stmt_n | ELSE stmt; OD;

Distributed termination iff no more communication is possible.

c.f.
II-18Dijkstra,
Hoare

Preview: Decoupling using Message Queueing

c.f.
IV.

- ▶ *De-couples systems w.r.t. time and space*
- ▶ bridges heterogeneity through standardized message formats
- ▶ standard part of (almost) all middleware systems, e.g., Java JMS
- ◀ administration and server overhead only pays off in large systems

Overview: Java Message Passing Using Sockets

Two principal types of sockets are of interest here:

1. **TCP:** Stream sockets: are used in a *Client/Server* fashion
 - * ServerSockets wait for accepting connection requests
 - * Client Sockets request connections
 - * successful connections work as two-way *streams*
2. **UDP:** Datagram sockets: similar to traditional message passing
 - * DatagramPackets: payload plus sender/receiver addresses
 - * DatagramSockets are bound to ports for packet send/receive

Addressing uses (IP-Address, Port-Address) based on `java.net`

Java message passing using UDP sockets will be introduced in the exercises for the first assignment. Message passing using TCP sockets is – for example – one of the topics of DSG-PKS-B 'Programmierung komplexer Systeme'.

III.6 Message Passing Synchronization

1. **One-way synchronization:** implement signal/wait by `snd/recv`
asynchronous `snd`; blocking `recv`c.f.
pg.
III-19
2. **Mutual Exclusion** for 'Shared data' may be implemented by
 - (a) **Centralized Server:** hosts data and is known to all processes
 - (b) **Migrating Server:** migrates data to exactly one requesting process at a time
 - (c) **Distributed Agreement:** allows for many copies of data spread among processes

Safeness: It has to be guaranteed that at any 'point in time' only one single process has write access to 'the data'

Additional Problems:

- ◀ unreliable message transport \implies message loss and re-ordering
- ◀ no global time to 'order' concurrent requests for ensuring fairness

How to Overcome Unreliable Transport – 1

- ◀ Ordering messages or arbitrary events for $n > 2$ processes is done using so-called '*logical time*' maintained by algorithms like Lamport-Time or Vector-Time (see chapter VI)
- ▶ Ordering messages among *pairs of processes* (P_s, P_r) is easy.

1. Use counters in order to detect lost messages:

- \forall process pairs (P_s, P_r) *counters* $S_{s,r}$ and $R_{s,r}$ are introduced.
 - * each sender P_s uses $S_{s,r}$ to count messages sent to P_r
 - * each receiver P_r uses $R_{s,r}$ to count messages received from P_s
 - * all counters in all processes are initialized by 0
 - * for each `snd` in P_s , the local counter $S_{s,r}^{P_s}$ is incremented
 - * for each `rcv` in P_r , the local counter $R_{s,r}^{P_r}$ is incremented
 - * up-to-date counters are part of each message: *piggy backing*
- When receiving a message, P_r checks whether $(S_{s,r}^{P_s} \neq R_{s,r}^{P_r} + 1)$ holds in order to expose a missing or untimely message from P_s .

How to Overcome Unreliable Transport – 2

2. Storing messages and Ackn/Resend control messages based on 1. implement reliable messaging in case of 'finite' errors.

- P_s stores all messages sent to other processes including the corresponding counter as long as there is no Ackn from P_r .
- P_r sends a receipt Ackn iff everything is ok, otherwise a Resend message is sent from P_r to P_s .
- P_s re-sends missing messages as long as no receipt has been received; afterwards the messages delivered are dropped.

Timeouts may be used to overcome permanently crashed processes.

3. Message ordering between exactly two processes:

- If P_r notices that ($S_{s,r}^{P_s} == R_{s,r}^{P_r} + k$ where $k > 1$), the message is *buffered* at P_r until the missing messages have been received.
- After receiving a timely message, P_r checks its local buffer for messages that may be ready to be consumed now.

Centralized Server for Shared Data

- **Global Server:** specific $P \in PS$ for all critical sections $cs(D)$

```

FORALL D DO csd.state = 1; csd.Queue.create(); OD;
WAIT FOR                                /* Server-Loop as a Guarded Command */
    recv(Pi,csd,GET)  =>
        IF (csd.state==1) THEN csd.state = 0; snd(Pi,csd,OK);
        ELSE csd.Queue.enqueue(Pi); snd(Pi,csd,WAIT);      FI;
    recv(Pi,csd,FREE)  =>
        IF (csd.Queue.isempty()) THEN csd.state = 1;
        ELSE P' = csd.Queue.frontdequeue(); snd(P',csd,OK); FI;
END WAIT;

```

- Arbitrary **Client:** process $P_i \in PS \setminus P$

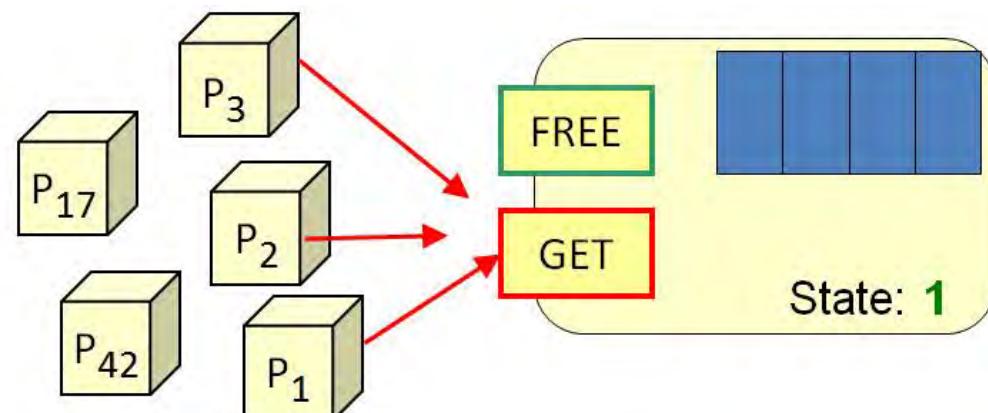
```

snd(P,csd, GET); recv(P,csd, X);      /* Prologue with blocking recv */
IF (X==WAIT) THEN recv(P,csd, Y) FI;    /* any more waiting? */
csdPi;
snd(P,csd, FREE);                      /* Epilogue */

```

Example: Typical Run with Centralized Server - 1

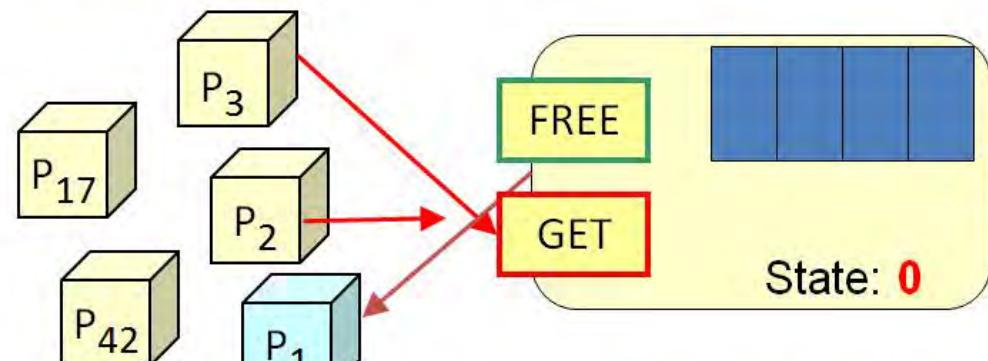
(1) P1, P2 and P3 starting requests for cs;
P1 request arrives at server



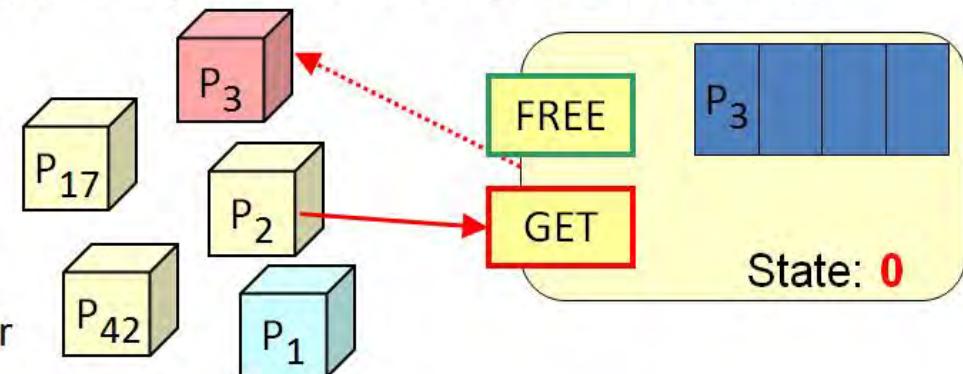
Legend:

- Client States:
 - no request or no answer (yellow)
 - WAIT from server received (red)
 - active in critical section (cyan)
- Different messages:
 - GET (red arrow)
 - OK (red arrow)
 - WAIT (dotted red arrow)
 - FREE (green arrow)

(3) P3 receives 'WAIT' and is stored in queue
P2 request arrives at server

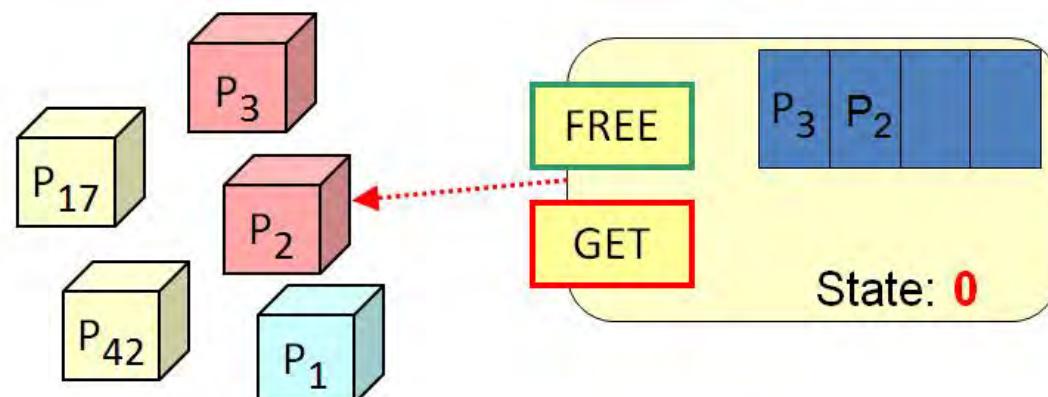


(2) P1 is granted cs; P3 request arrives at server

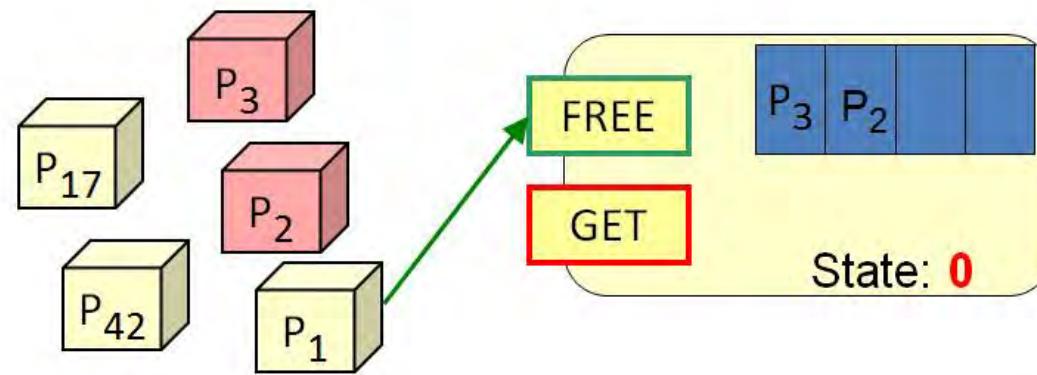


Example: Typical Run with Centralized Server - 2

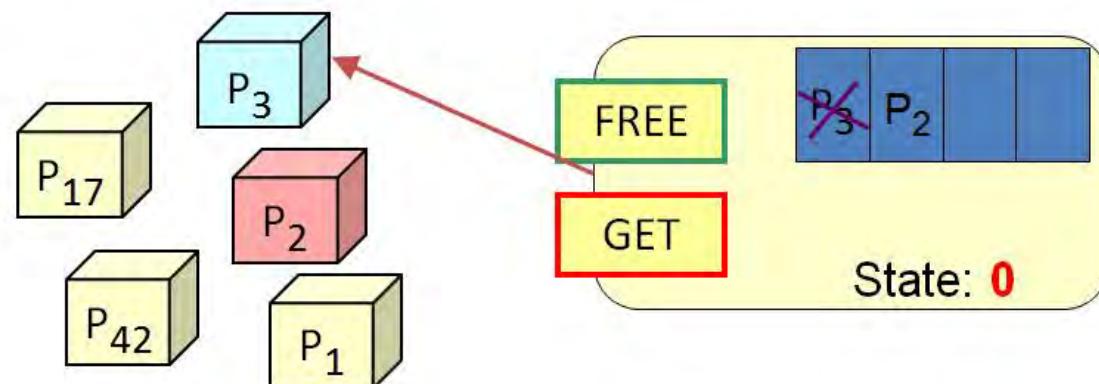
- (4) P2 receives 'WAIT'
two processes wait in
server Queue now



- (5) P1 leaves cs and 'FREE' message from P1 arrives at server



- (6) P3 is removed from Queue
and gets 'OK' message to
enter cs;
P2 wait still in Queue



... etc.

Assessment of Centralized Server Solution – 1

- ▶ P grants at most a single OK at a time \implies *System is safe.*
- ▷ Local FIFO Queue inside P is fair, but what about messaging?
 - * Server is only able to treat *known requests* fair.
 - * If messages from a process or subsystem are arbitrarily delayed or dropped, the system cannot be fair.
 \implies **Starvation** depends on communication system properties.
- ◀ **Overhead:** maximum of 3 messages for a single access to csd.
- ◀ **Errors** render entire system useless:
 - ◀ **Lost GET/WAIT/OK messages** block Client process indefinitely
 - ◀ **Lost FREE messages** from Client block server side and csd
 - ◀ **Crash** of server process P or process P_i currently holding csd block entire system.
 \implies *Server process as a single point of failure combined with Client processes as multiple single points of failure.*

Assessment of Centralized Server Solution – 2

Pragmatism: Centralized Algorithm works fine iff

1. Message transport is reliable and to some extent fair.
 2. Server runs on reliable hardware, e.g., virtual servers with backup.
 3. Additional Measures:
 - Use distinct server processes for managing different data sets D
 - Replicate server for the same data set D on different hardware and interact among replicated servers in order to ensure consistent queues and preserve safeness.
 - Use **time-outs** if waiting for control messages lasts too long:
 - * Clients should take actions if servers are down.
 - * Server should take actions if FREEs from clients are overdue.
- ⇒ *Centralized Server works only in a reliable environment.*

Note: *Migrating servers work similarly and exhibit almost the same properties.*

Preview: Truly distributed synchronization?

Two principal classes of really distributed algorithms:

1. Negotiation based on local conditions in client processes

- P_i sends requests to P_j s asking for permission
- Many algorithms using different *request* and *inform* sets
 - * a process has to ask all processes from its *request* set
 - * a process has to answer all requests from its *inform* set
- Sets have to be constructed such that always a **majority** of all processes is asked for permission in order to ensure safeness.

2. Access based on exclusive ownership of a control token

- Methods vary w.r.t. how token is acquired from other processes
- Simple case: in a *ring* of processes, token is passed around.

Problems are similar to those detected for the centralized version

- ◀ *single point-of-failure* → *multiple single points-of-failure*
- ◀ algorithms do not work with unreliable communication

Conclusion: Basic Interaction Mechanisms

- ▶ Sufficient abstractions from hardware and network in order to implement portable distributed systems.
- ◀ Insufficient abstraction from underlying system to program on a really comfortable level (apart from msg queueing).
- ◀ Only (very) tightly coupled systems implementable by directly interacting based on *Read/Write sets* of data, no matter whether interaction is based on `read/write` or simple `snd/rcv`.

c.f.
pg.
III-2

Higher levels of abstraction and de-coupling:

- ▶ *Message Queueing* is a first step from basics to better de-coupling via more advanced middleware technology.
- ▶ *Client-Server* is a more abstract paradigm based on remote procedures, objects and services suitable for typical settings in an internet-based environment.

End
of
III

IV. Message Queuing Systems

Application Domain: Bridging heterogenous applications

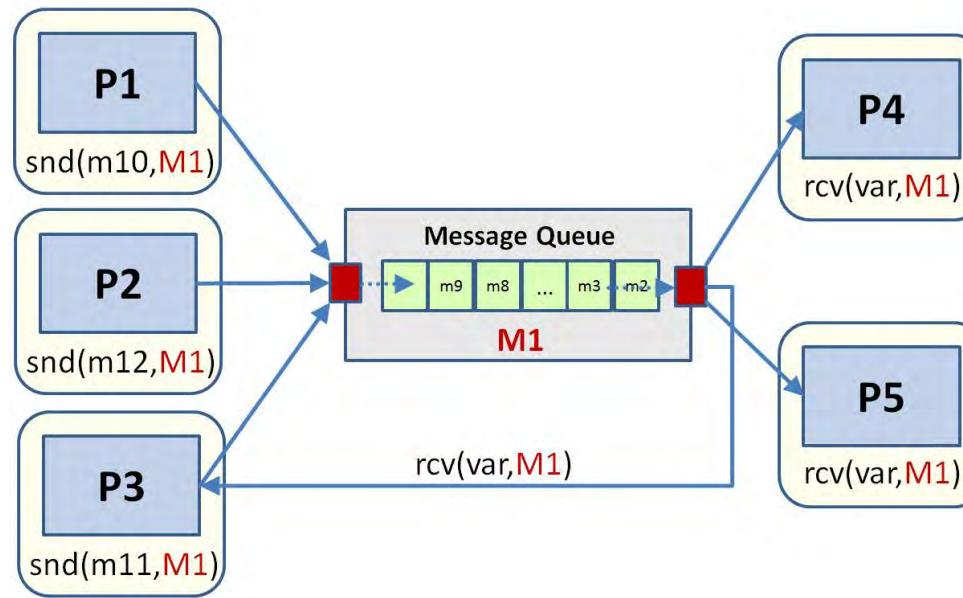
- ⇒ suitable for EAI as well as B2Bi
- ⇒ low-level use for the Internet of Things

- ◁ Applications run in different geographic locations
- ◁ Application environments evolved over decades
- ◁ Global standardization (often) not feasible
 - ⇒ Bottom-Up **Integration** of isolated applications towards interacting IT 'landscapes'
 - ⇒ Evolving and always changing IoT environments

Requirements: Explicit message-passing interaction

- ◁ Arbitrary complex messages: size and structure for 'Clients'
- ◁ Interaction among separately developed applications
 - ⇒ **Wrapper** for message alignment needed
- ◁ Tight synchronization tends to be too error-prone

Messaging = Message Passing plus Queuing



Basic Principles: Support for Asynchronous Msgs and Queues

- * interfaces to drop/pick-up messages
- * logical naming schemes for **multi-cast** interaction
- * reliable message transfer based on **persistent Queues**

Decoupling Effects \implies convenient and secure message passing

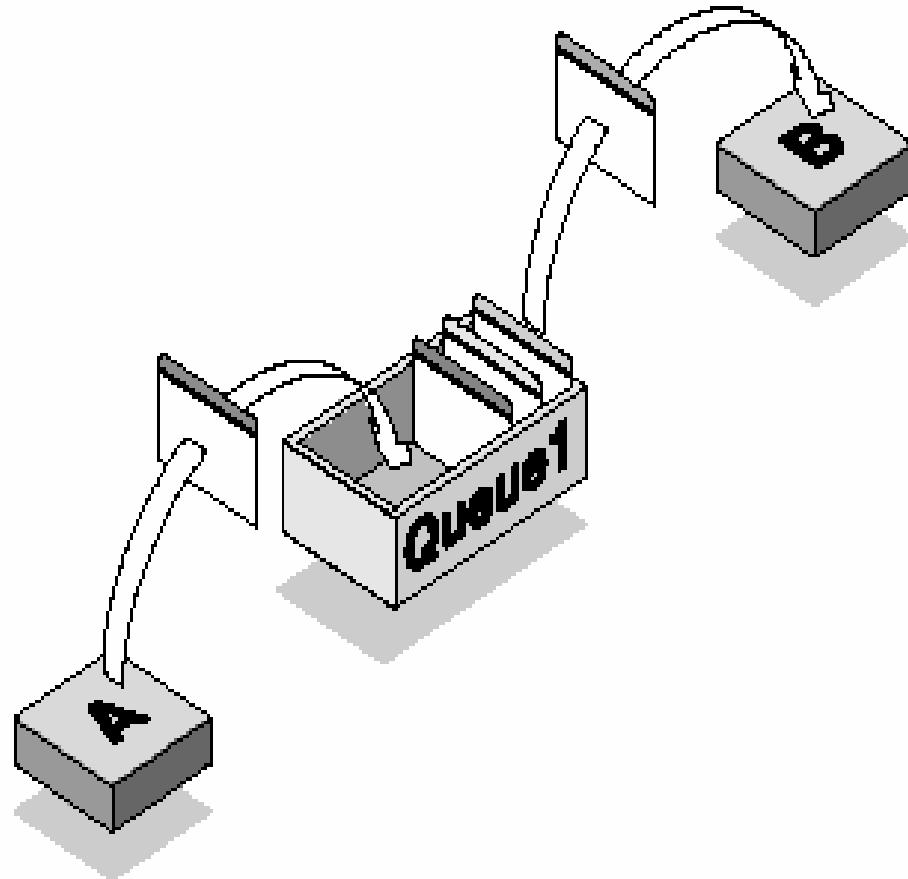
- **Asynchronous:** decoupling w.r.t. time
- **Multi-Cast:** decoupling w.r.t. concrete 'sender' or 'receiver'

IV.1 Basic Characteristics of the Messaging Model

- **Queue Addresses** instead of naming communication partners
 - ⇒ Abstraction decouples one-way interactionIV-??
- ◀ Queues are typically **unidirectional** channels
 - ⇒ **Two-way interaction** requires two distinct QueuesIV-??
- A wide range of unidirectional interaction 'styles' is easily supported:
 - Multi-destination queues: '*get consumes*'IV-??
 - Selective Routing: disjoint vs. shared Queue-BindingsIV-??
 - Multi-Source–Multi-Destination interactionIV-??
 - ⇒ **unidirectional $m-n$ channel**
 - Publish/Subscribe–Model: variants depend on *read* vs. *consume*
'logic of 'get' operation.IV-??
 - ⇒ **unidirectional $m-n$ -Multicast**

Uni-directional One-to-One using a single Queue

Fig.:
IBM TR
GC33-
0805
1995

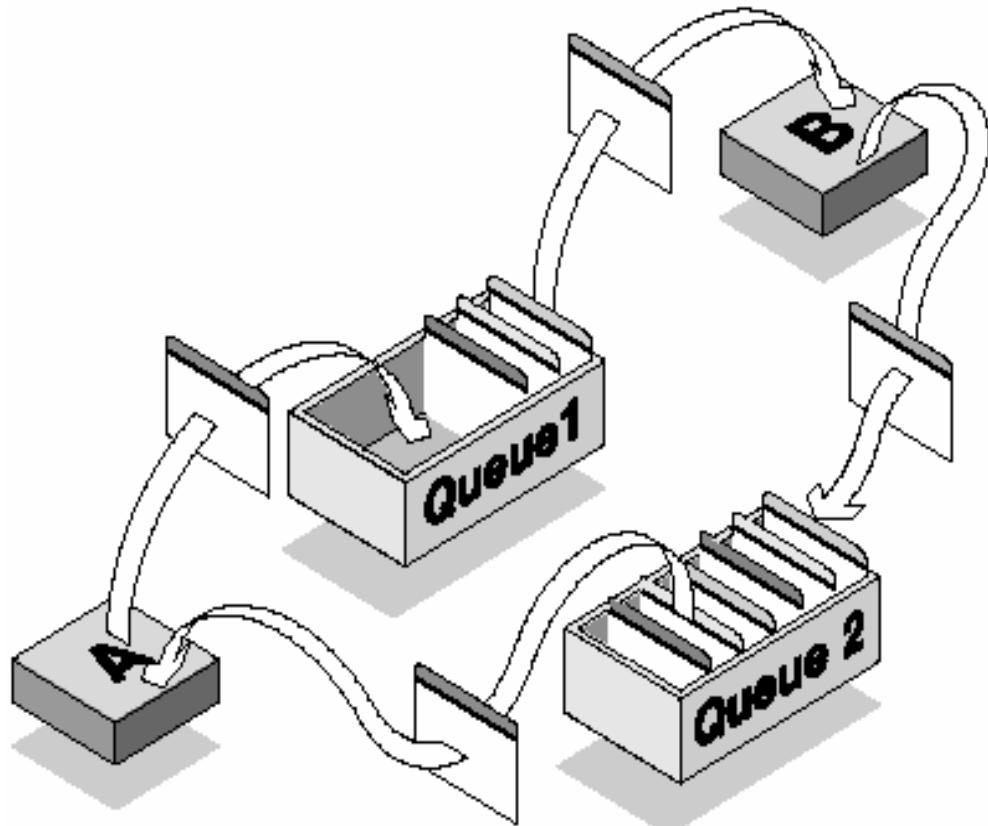


Abstraction:

1. *A* hands Msg to local Queue Interface Queue1 is used as the 'Address'
2. Queue stores and transfers Msg
3. *B* extracts Msg from local Queue Interface

Bi-directional Point-to-Point requires two Queues

Fig.:
IBM TR
GC33-
0805
1995



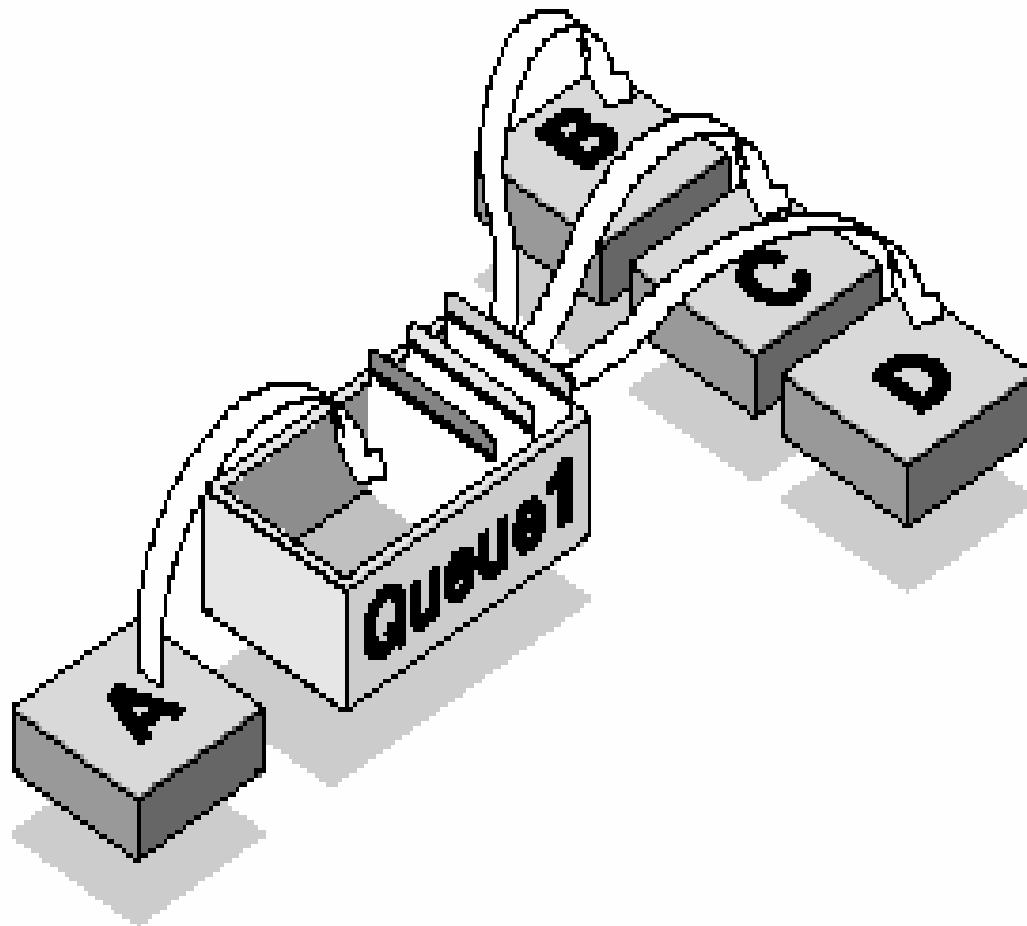
A sends *Msg1* via *Queue1*;
B replies with *Msg2* using
Queue2;
Decoupling through
asynchronous reaction

Applications:

- Inquiry/Result
- Order with confirmation
- RPC: Call and Reply

Uni-directional One-to-Many using a single Queue

Fig.:
IBM TR
GC33-
0805
1995



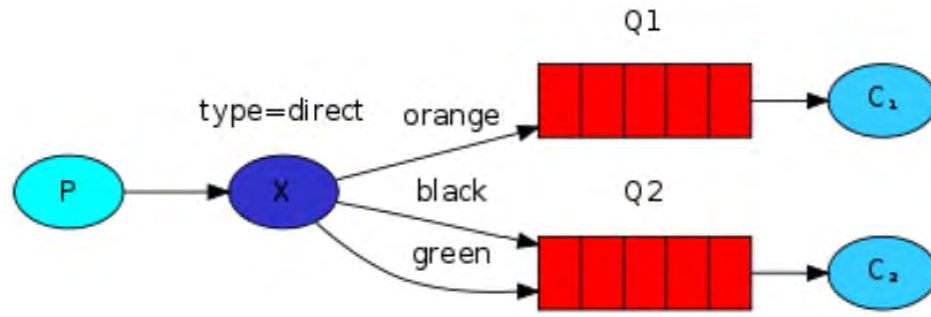
Abstraction:

1. *A* hands Msgs to local Queue Interface
2. Queue stores and transfers Msgs
3. *B, C, D* **consume** Msgs from local Queue Interface
4. Each Msg is processed by **one** Receiver

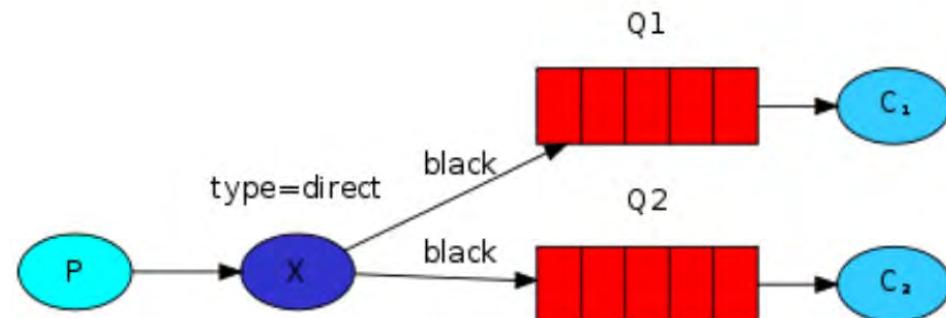
Work Queues \implies Competing Consumer Pattern

Selective Routing with Filtering Attributes

Attribute(s)/Bindings: between 'Exchange' and 'Queues'



disjoint \implies
Msgs go to specific queues

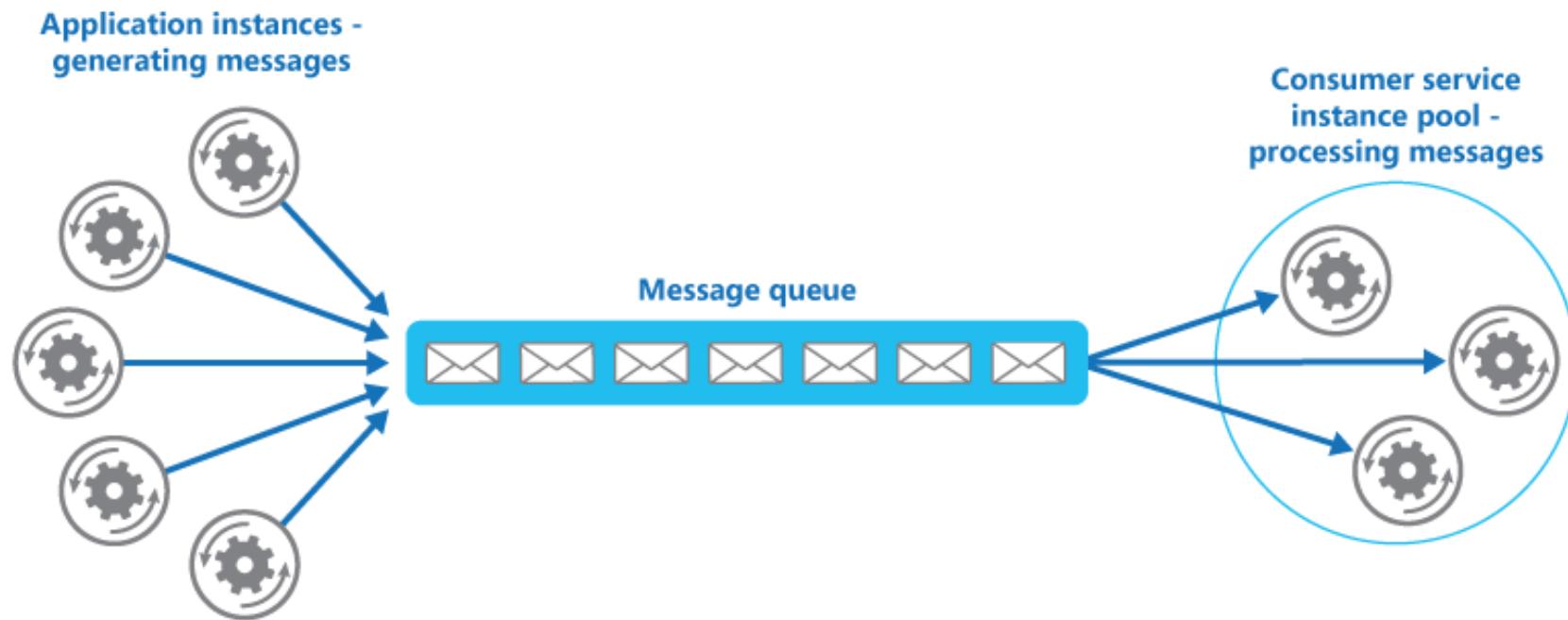


shared \implies
Msgs go to multiple queues

Remark: More on RabbitMQ in section IV.4

Multiple One-to-One using a single Queue

Fig.:
docs.
micro
soft.
com/
en-us/
azure/
archi
tecture/
patterns/

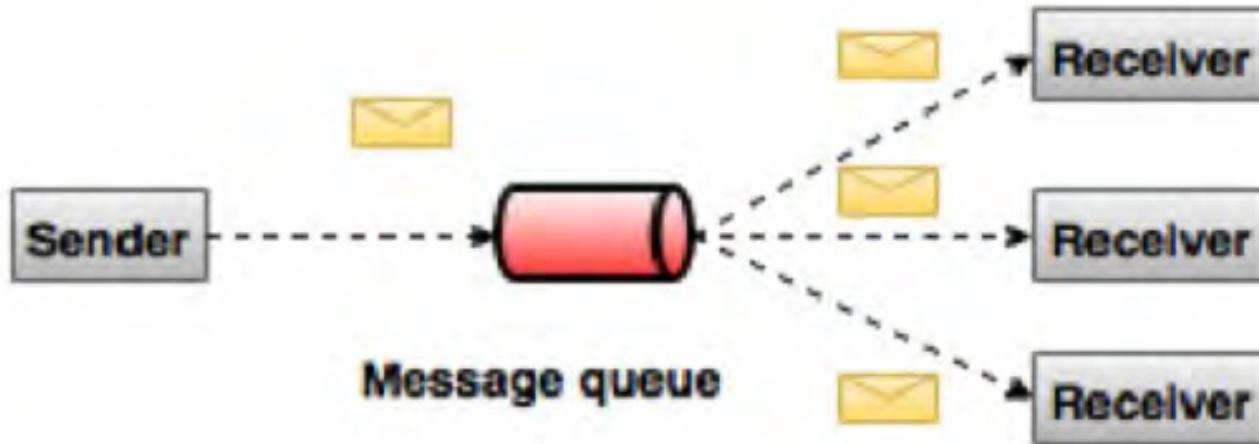


- Msgs are extended to hold specific receiver addresses or **logical attributes** allowing inquiries by potential receivers
- All Sender/Receiver use common queue endpoint
- Lots of processes use the same queue \implies reduced overhead
- Each Msg goes to a single Receiver

Application: load balancing, work-list processing (Workflow)

Publish-Subscribe Organization using 'Topic's

Fig.:
www.
tutorials
point.
com/
apache_
kafka/
apache_
kafka
_intro
duction
.htm



- Abstraction: tagging msgs with/subscribing to specific **topics**
- Enhanced Decoupling w.r.t.
 - * **Logic:** no direct Sender/Receiver-Roles required
 - * **Time:** *durable subscription* msgs stored for offline subscribers
expiration date limits costs for storing msgs indefinitely

Application: Order goes to processing, logging and accounting

IV.2 Message Queuing 'Products'

► **Messaging – been around for decades and is still alive:**

 ⇒ Starting Point IBM MQSeries in 1993

- IBM MQ V.9.1 cloud-based product suite today
- Microsoft MSMQ Series: almost the same . . .

► Support from all **Integration** and **Cloud Providers**:

- Important part of ESB-, SOA- or Cloud-Suites
 - * MS Azure Service Bus or Storage Queues
 - * Amazon AWS Simple Queue Service (AWS SQS)
 - * Open Source Products: Apache ActiveMQ, RabbitMQ, . . .
- Used to connect all kinds of modern infrastructure
 - * Microservice architectures, Serverless (AWS Lambda), . . .
 - * Streaming applications (e.g. Apache Kafka); . . .

► **New application areas:** Embedded systems and the IoT

www.
ibm.
com/
support/
know
ledge
center/
en/
SSFKSJ...
com.ibm.
mq.pro.
doc/
q001010.
htm

Modern Offerings and Standards

- ▶ Java: **Java Message Service JMS 2.0a Spec.** (03/2015)
 - **Interface Architecture:** Java \longleftrightarrow Messaging started in 1998;
Jakarta Messaging 3.0 in Jakarta EE 9
 - Open Source implementations and commercial version(s)
 - * Amazon SQS, JBoss messaging, ...
 - * Apache ActiveMQ, Rabbit MQ Client/Plugin, ...
- ▶ **Emerging 'Standards':**
 - Advanced Message Queuing Protocol (**AMQP**)
alternative to http \implies general interaction (Vers.1.0, 2014)
 - Message Queuing Telemetry Transport (**MQTT**)
based on TCP/IP \implies light-weighted for IoT (Vers.5.0, 2019)
 - Streaming Text Oriented Messaging Protocol (**STOMP**)
based on http \implies Text-based interaction (Vers.1.2; 2012)

jakarta.
eeOASIS
ISO/IEC
19464

OASIS

IV.3 Java Message Service API – Overview

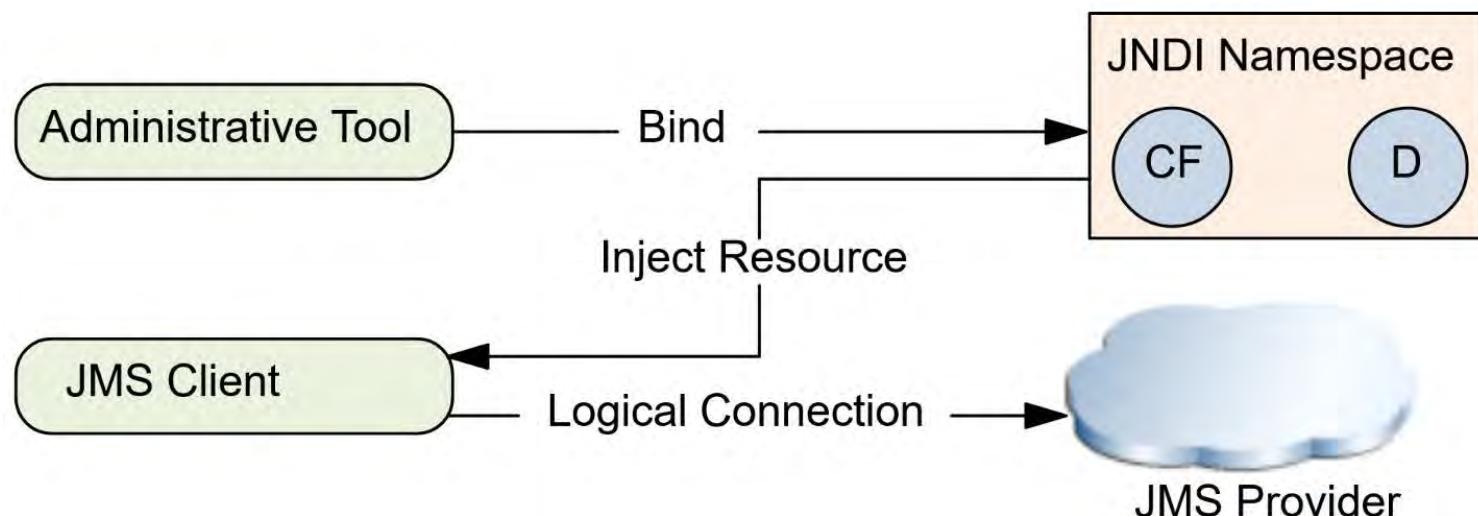


Figure 48-2 Jakarta Messaging Architecture

1. **JMS Provider:** Platform, Control and Administration
2. **JMS Clients:** Use platform to produce/consume messages
alternative: native clients that adhere to the API rules
3. **Messages** using predefined Java types and formats
4. **Administered Obj.:** destination(D)/connection factories(CF)
Interfaces for creating and managing connections

from:
Jakarta
EE
Tutorial,
chap.
48
eclipse
-ee4j.
github.
io/
jakar
taee-
tutorial/
2021

Fig.48-2

lookup
vs.
injection
DSG-
DSAM

IV-??

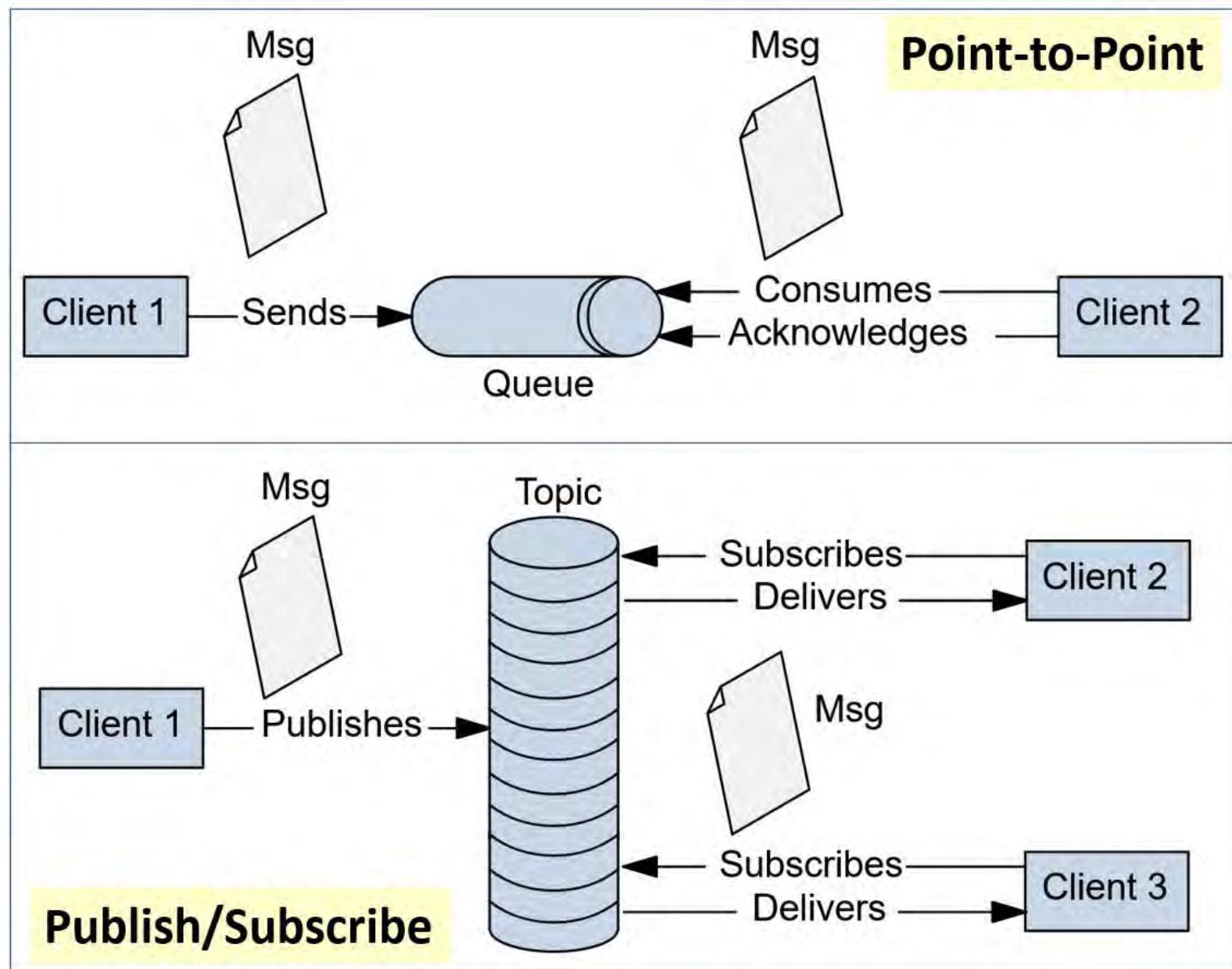
IV-??

Java Message Service API: Functionality

- **Styles/Domains:** specific paradigms of messaging usageIV-??
 - Point-to-Point interaction
 - Publish/Subscribe paradigm
 - * durable subscription supports **de-coupling w.r.t. time**IV-??
 - * unshared/shared subscriptions: single/multi consumers
- **Interaction:** **synchronous** via receive with/without **timeouts**
asynchronous via message listener: `onMessage`method
- **Robustness:** series of messaging ops in transaction context
- **Messages:** header with meta information
additional: properties used as criteria for queue selection
 - ⇒ application-specific queue handling implementable
 - ⇒ IDs from foreign MQ systems may be integrated
- **Content:** 6 different types of Java messagesIV-??

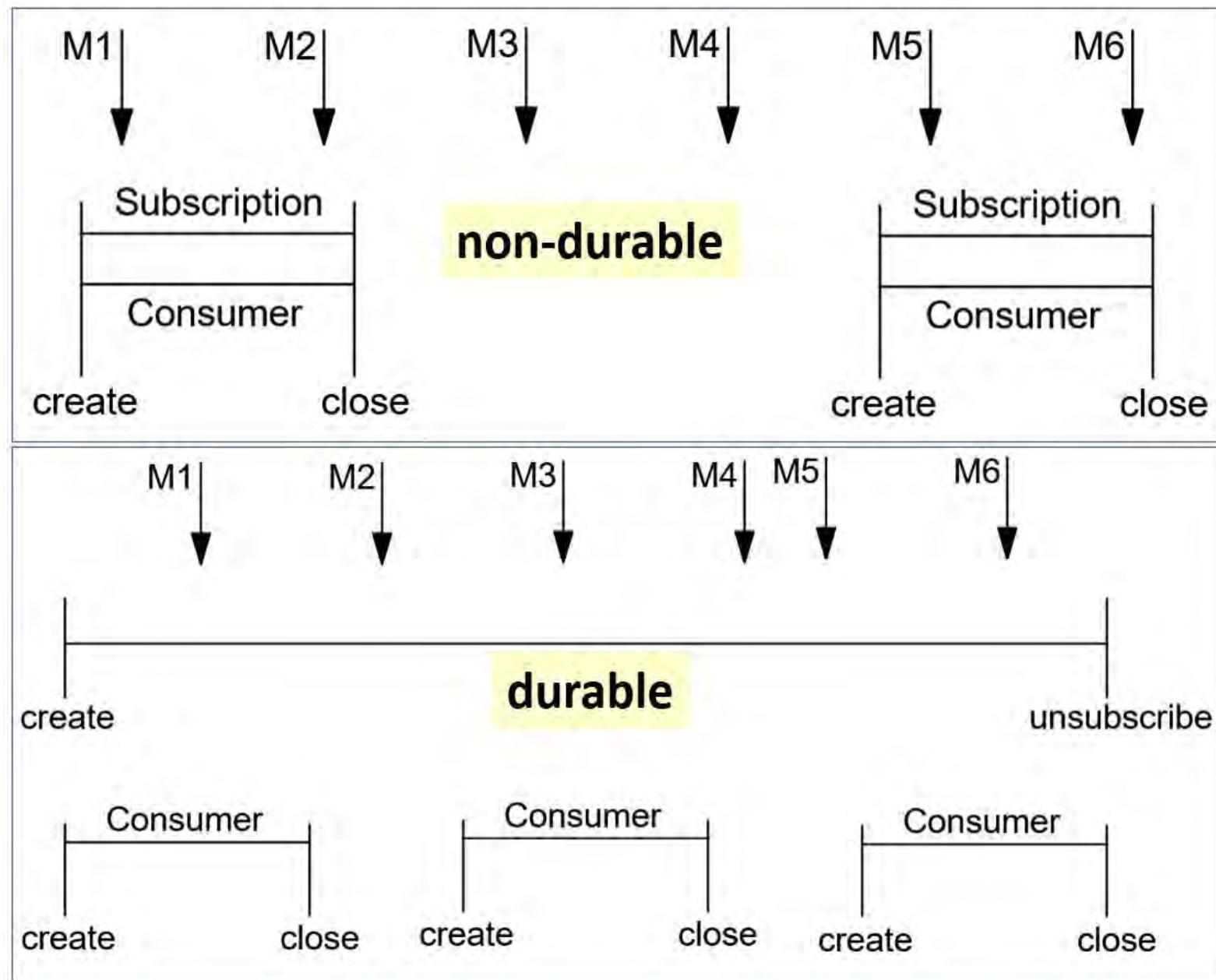
Java Message Service API: Messaging Styles

from:
Jakarta
EE Tut.
Fig.48-
3/4



Java Message Service API – 4: Subscription Types

Jakarta
EE
Tutorial,
chap.
48
Fig.
48-6
and
48-7



Java Message Service API: Message Types

Jakarta
EE
Tutorial,
chap.
48
Table
48-2

struct

Message Type	Body Contains
TextMessage	A <code>java.lang.String</code> object (for example, the contents of an Extensible Markup Language file).
MapMessage	A set of name/value pairs, with names as <code>String</code> objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
StreamMessage	A stream of primitive values in the Java programming language, filled and read sequentially.
ObjectMessage	A <code>Serializable</code> object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

JMS API: Software Architecture

1. Configuration: administered objects via asadmin (pre-runtime)

- Create and parameterize factories using JNDI Namespace
- Specify msg source and destination ≈ Destination

res-
source
anno-
tations

2. Usage: Create and assemble messages

(a) Creating a JMSContext object provides

- * a connection to a JMS provider and
- * a session as a single-threaded context

(b) Create JMSContext communication objects

- message producer: used to send messages
- message consumer: synchronous via receive(timeout)
asynchronous via MessageListener

Additional support for message selectors to filter messages and QueueBrowser objects to inspect queues.

Remark: If interested in more details, please refer to Jakarta EE tutorial.

JMS API: Software Architecture

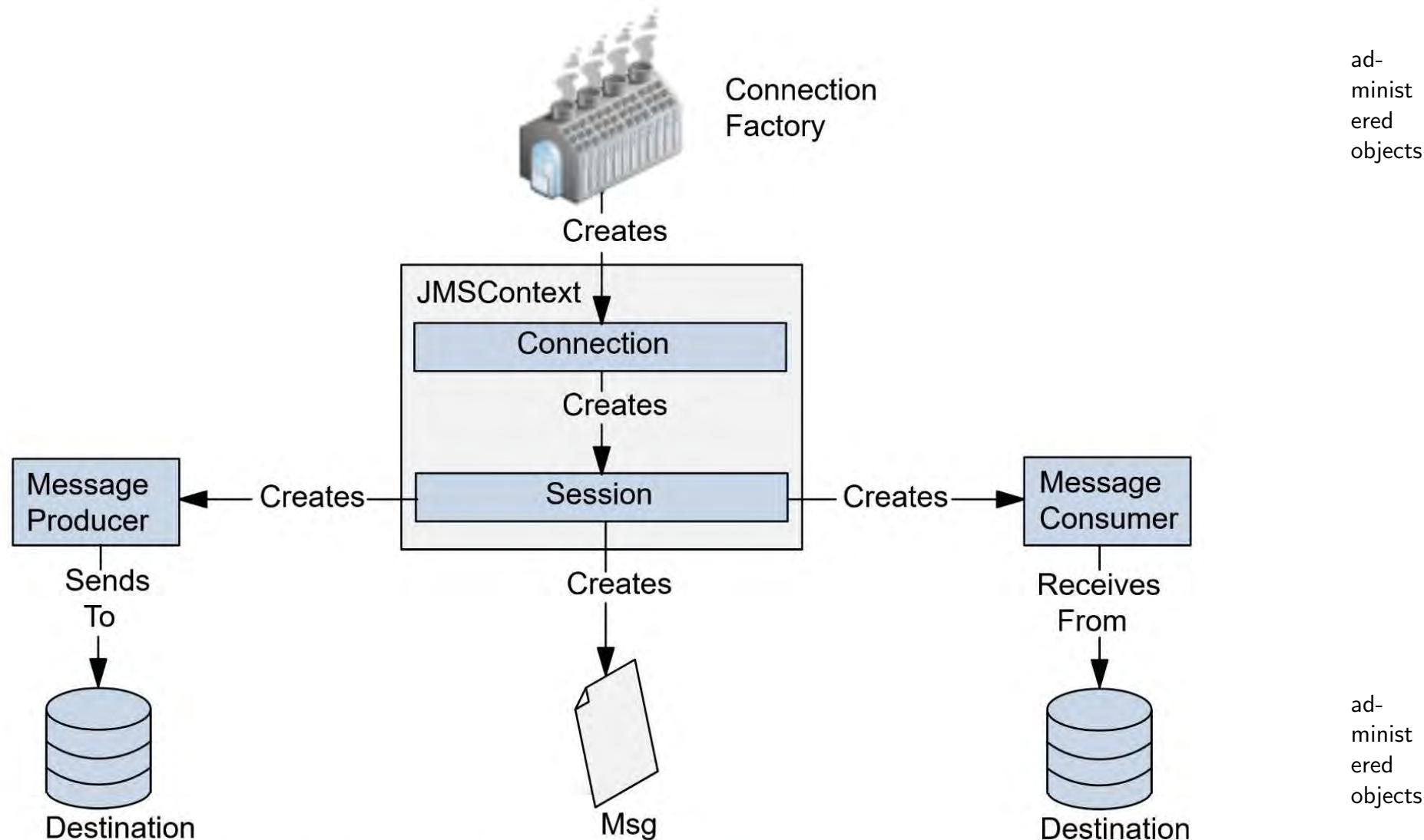


Figure 48-5 Jakarta Messaging Programming Model

IV.4 Advanced Message Queuing Protocol

Fig.:
www.
amqp.
org
/prod
uct
/over
view



Advanced Message Queuing Protocol (AMQP)

OASIS

- ▶ ISO/IEC19464 standard since 2014
- ▶ general alternative protocol to http
- ▶ based on TCP; SSL/TLS and SASL usable
- ▶ used for almost all distributed interaction scenarios from cloud infrastructures to mobile clients
- ▶ support from (almost) all important vendors over the last years

AMQP

Basic Model

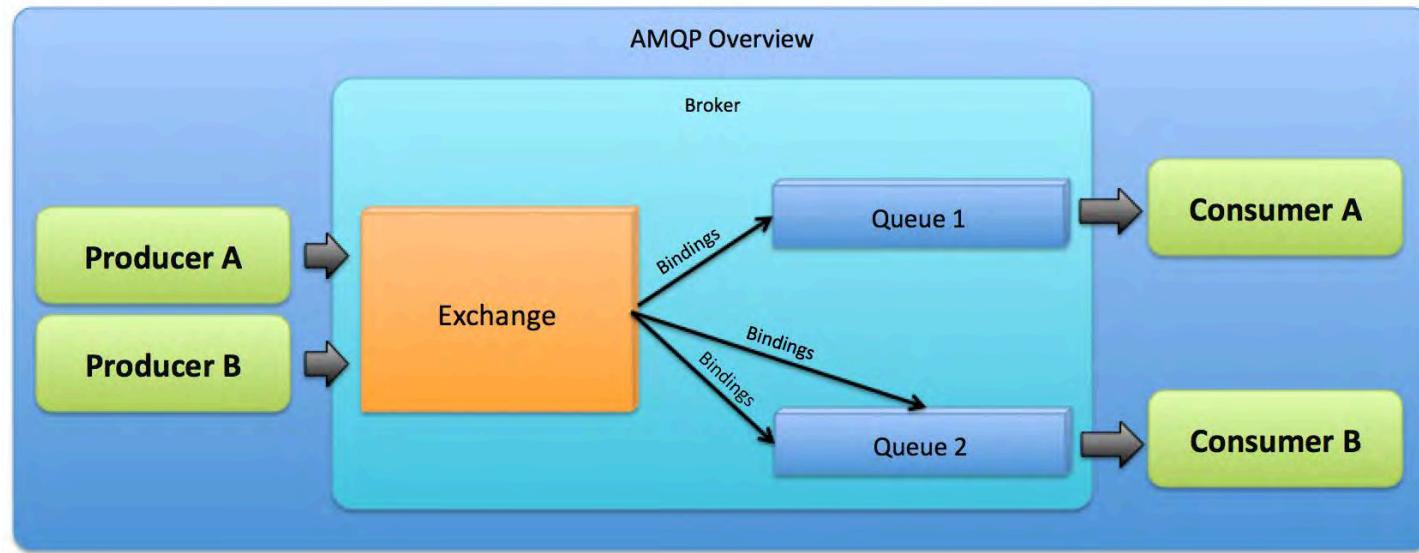


Fig.:
alex
volov
.com/
2016/06/
amqp/

- Message Broker as the central instance to implement Queues
- Producer (P) and Consumer (C) act as 'Clients' of the Broker
- Exchange: connection between Producer and Queues
 - * Name: used for discovering and to setup connections
 - * Type: interaction 'style': direct, fanout, header, topic
 - * Bindings: Keys as basis für routing messages 'to' queues
- Queues store the messages
 - * have to be attached to an Exchange after creation
 - * can be used as a **pull** or a **push** medium

c.f. pg.
IV-21

AMQP Interaction Styles and Implementations

Realizing different interaction styles:

- ▷ organized by Exchange component in Broker
- ▷ based on Strings as routing keys in messages
- ▷ based on Strings as binding keys when attaching queues

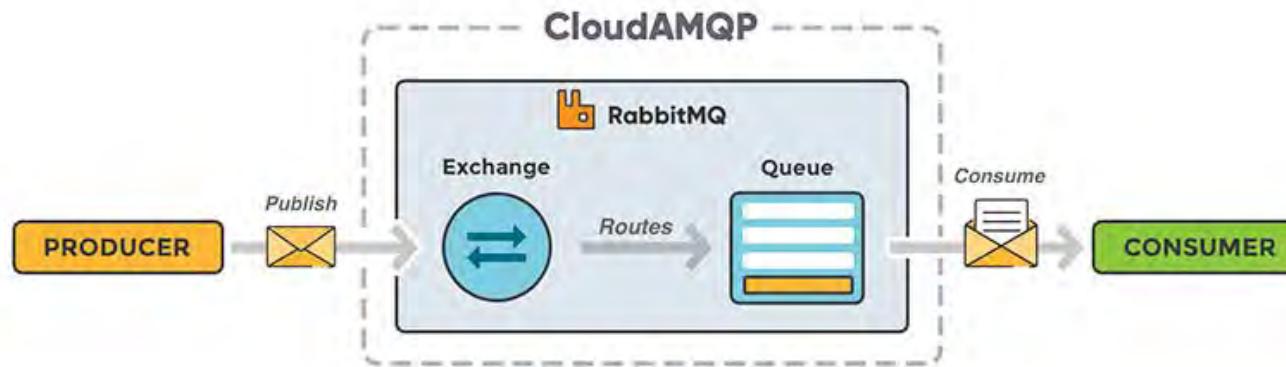
Four different methods to route messages:

- direct: matches msg routing keys with queue binding keys
- fanout: msg goes to all attached queues ignoring routing keys
- topic: implements publish/subscribe pattern
- header: uses msg attributes (data types) instead of routing keys
- ▶ Many implementations on offer today:
 - * Apache Qpid/ActiveMQ/Artemis
 - * MS Azure Service Bus, SwiftMQ, . . . , RabbitMQ
- ▶ Additionally: Bridges to other protocols, e.g. MQTT

IV.5 AMQP Implementation: RabbitMQ

- Open Source Message Broker Software (based on Erlang)
- Pivotal Software (VMWare, SpringSource, General Electric)
- Multiple **Standards** supported:
 - * AMQP implementation (widely used)
 - * MQTT implementation (IoT)
 - * STOMP integration
 - * JMS client plugins etc.
 - * HTTP, WebSockets
- **Multi-platform/language support:**
 - * Spring, .NET, JVMs, ...
 - * Java, Java Script/Node Ruby, Python, PHP ... Haskell
- **Light-weighted** and not too hard to host
- CloudAMQP: Cloud hosting at AWS Marketplace and heroku

RabbitMQ - a short Overview



Remark: Tutorial material online: www.cloudamqp.com/docs/index.html

Interface to Programming: P and C connect to the Broker

simplified

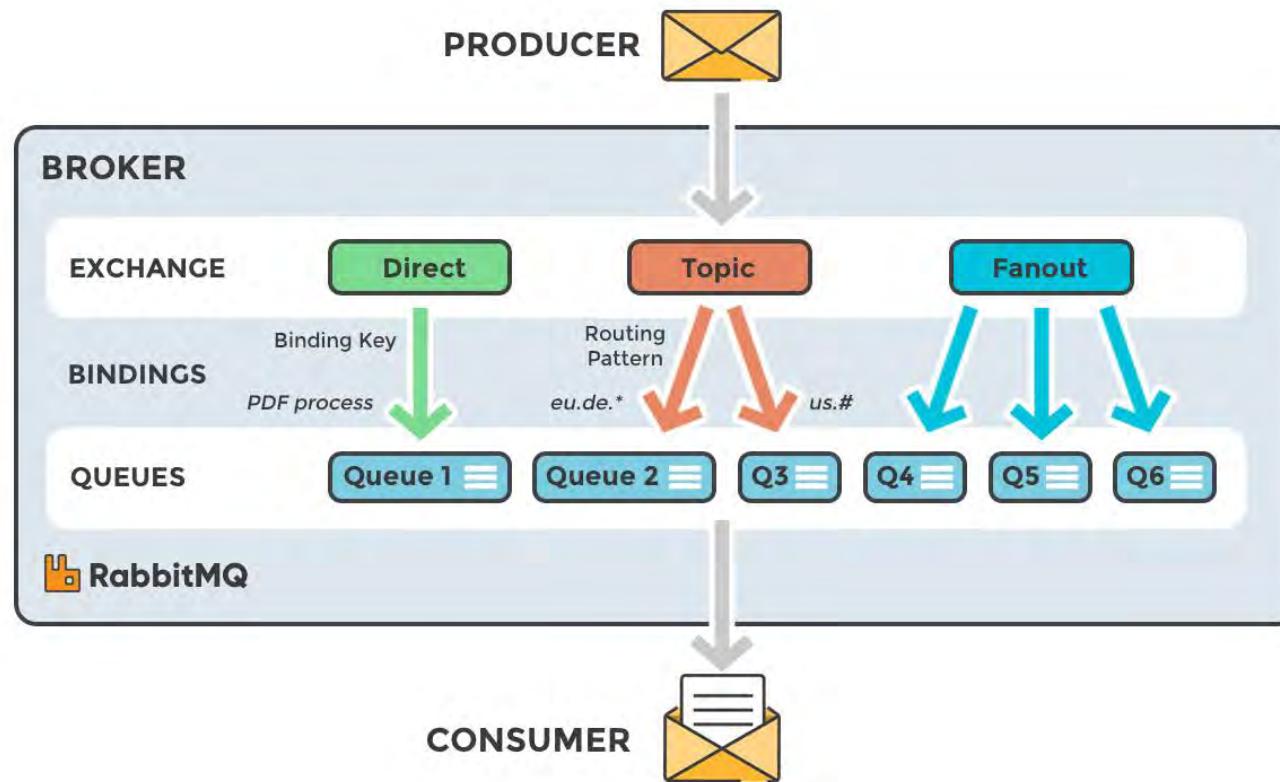
- Client **Producer**:

1. setting up a connection which provides a channel that allows to
2. declare (an exchange that binds to) a queue.
3. publishing uses an exchange, a routing key and the payload
4. close the connection at the end

- Client **Consumer**:

1. setting up a connection which provides a channel that allows to
2. declare (an exchange that binds to) a queue (idempotent)
3. receiving requires defining a callback function to handle the message
4. consume declares a queue name and the on_message_callback
5. consuming is done in an 'endless' waiting loop start-consuming

RabbitMQ - Different Interaction Styles



Supported Interaction Paradigms:

- * direct 1-1 or 1-n with named queue and anonymous exchange
- * Competing Consumer Pattern
- * publish/subscribe with single or multiple topics
- * routing via Strings as routing/binding keys
- * simulating remote procedure calls using Request/Reply queues

Summary: Importance of Message Queuing

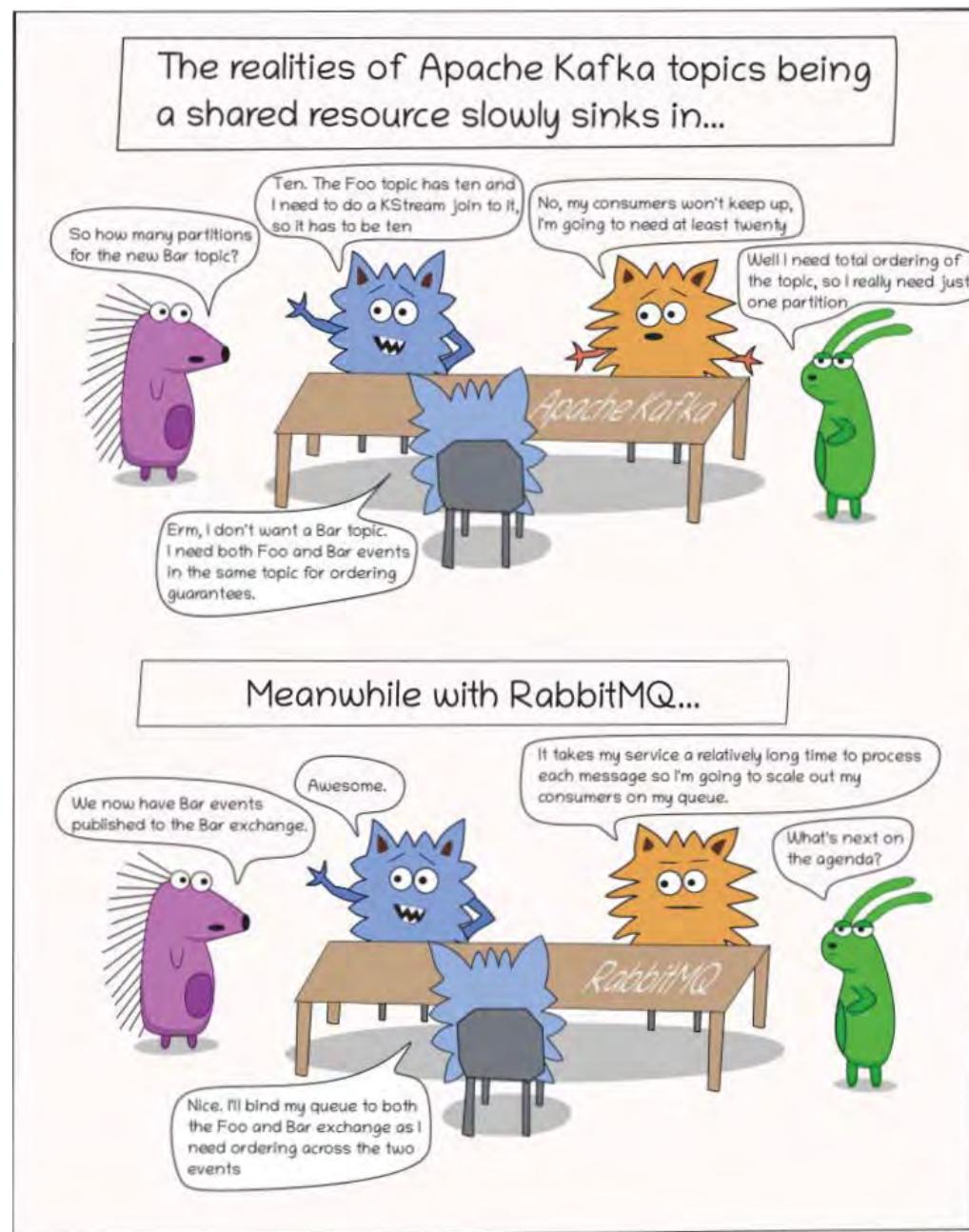
- ▶ **Integration** of already existing, heterogenous applications, esp.
 - Widespread solution for **inter middleware integration**
 - Integration of **Server-side components**, e.g., EJBs
 - Combination of **Online and Batch** processing
 - Implementation of reliability and load-balancing for msg passing
- ▶ Suitable for all kinds of **loosely-coupled systems**
 - (highly) asynchronous
 - often not at the same time active

Applications in big corporations or between different enterprises
- ◀ Basic paradigm that is (almost) universally applicable, **but:**
 - low-level **message** view not always adequate
 - configuration/administration of messaging systems rather tricky

⇒ **too cumbersome for tightly-coupled systems**

c.f.
DSG-
DSAM
-M

from:
jack-
van
lightly.
com/
sketches



End of
chapter
IV

V. Client/Server-Interaction

Message passing too closely coupled for modular systems \implies
higher level of de-coupling needed: **Remote Procedure Calls**

Nelson
1981

► **Basic Model:** simple role-based architecture

- *Server* offers services
- *Client* requests and uses services

Examples: Email, file server, time server, name server, . . .

► **Well-known programming paradigm:** Procedure Calls

\implies simple interaction protocol

at
the
time

► **Moderate coupling**

Server \approx no global knowledge besides own state

Client \approx service signatures of Services (**Service-Broker**)

Substitution Test: new *Clients* may use running *Server*
new *Server* may offer the same/new *services*

Remark: really old concept www.ietf.org/rfc/rfc707.txt 1976

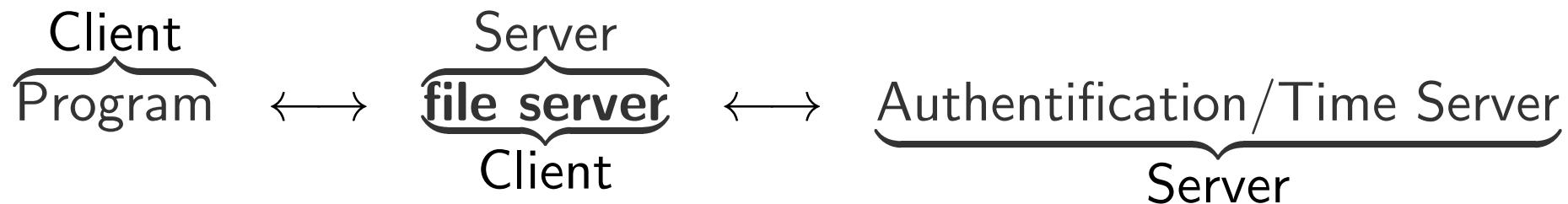
Client-Server-Interaction – Roles

Roles allow for a suitable level of **abstraction**:

- ▶ Independent from specific **kind of Request**
 - compute function: Compute server replicated
 - read/write data: File system server replicated

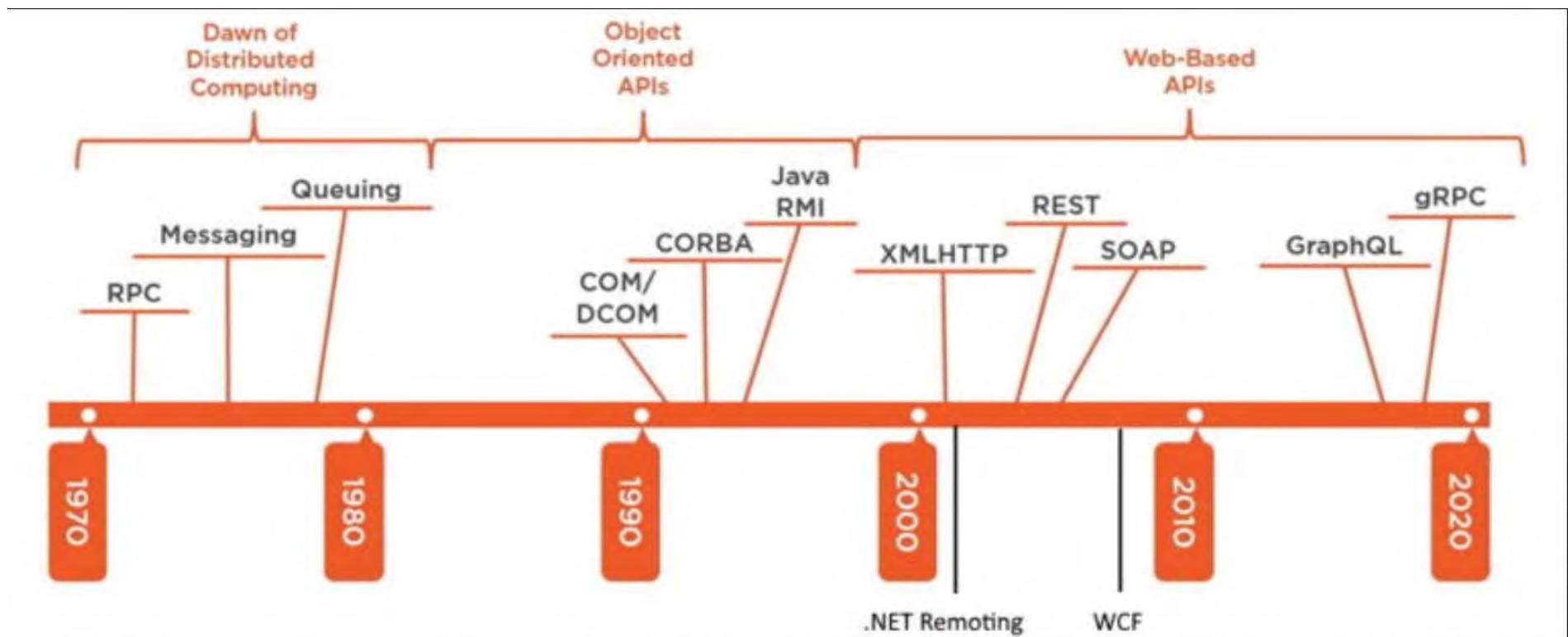
⇒ **Transparency w.r.t. active and passive components**
- ▶ Role may **change due to point of view**
 - ⇒ context-sensitive roles
 - ⇒ same object may play different roles

Example: User-Programm reads data from file



Remark: C/S model is even more general than RPC interaction

C/S-Paradigm Development over 45+ Years



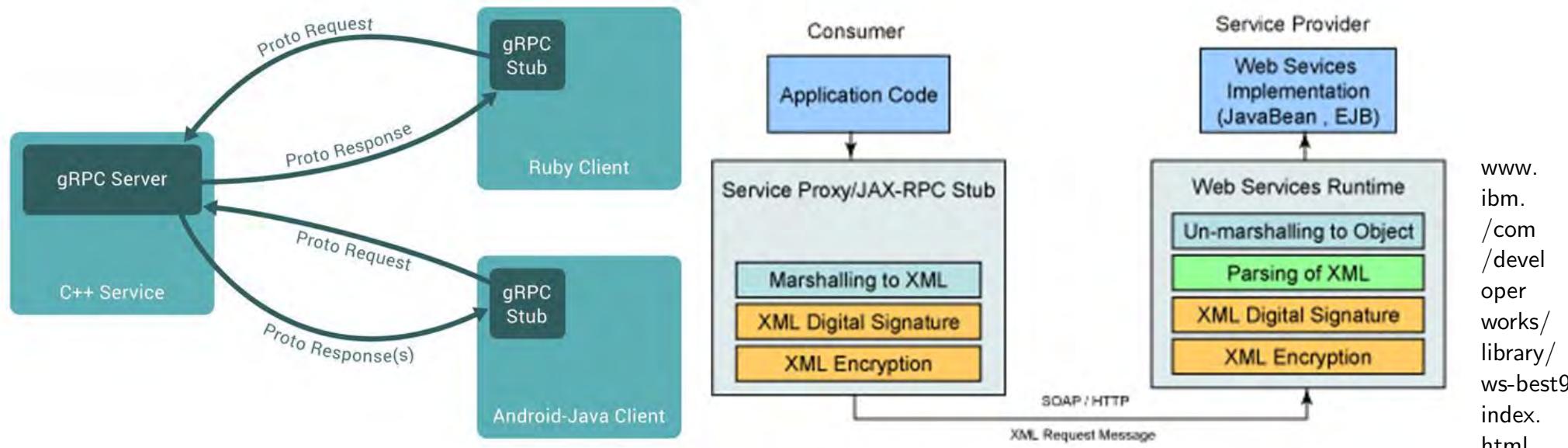
- OS/Network: Stream I/O message exchange using TCP-Sockets
 ⇒ C/S role determines which kind of socket used in program
 ... ⇒ ...
- All kinds of distributed models for **Remote Procedure Call**
 Description, Addressing, Protocols, Data Formats, Sync, ...

Different Approaches for the C/S-Paradigm

- ▷ **'Classical' Remote Procedure Call Envs** \implies **Characteristics**
 - * Distributed Computing Environment (DCE)
 - * Microsoft MSRPC; COM; COM+; DCOM;NET Remoting
 - * Common Object Request Broker Architecture, e.g., in C based on C-like Interface Definition Language (IDL)
- ▷ **Object-Oriented C/S Implementations**
 - * Java Remote Method Invocation (RMI)
 - * Using CORBA and IDL in a non-Java OO Environment, e.g., C++
- ▷ **Client/Server on the Internet – Services**
 - * Google gRPC re-inventing IDLs in JSON (grpc.io, 2015) v.3
 - * Heavy-weighted XML/SOAP-based Web Services (SOA) (v.5)
 - * Light-weighted http-based APIs using a REST Architecture (REpresentational State Transfer; Roy Fielding 1994/2000) v.4

Preview – 1: Google RPC vs. Webservice

grpc.
io.docs.
guides



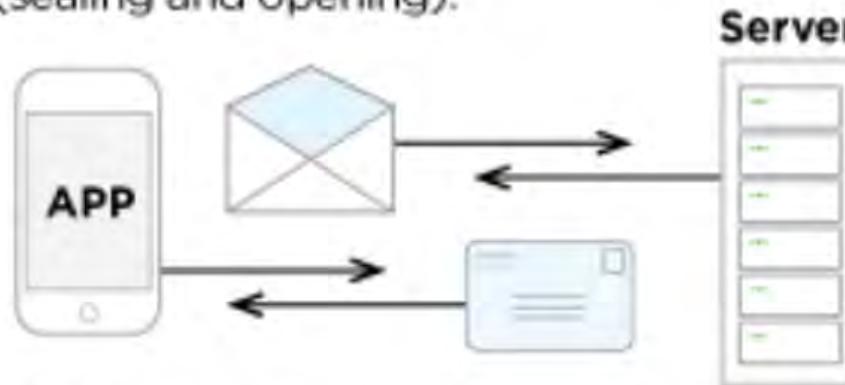
- Left: gRPC setting for multi-language clients
protocol buffers as Interface Definition Language
- Right: IBM Webservice using 'RPC' technology
XML Ecosystem as common Interface Definition Language

IDL

Preview – 2: **SOAP vs. REST APIs**

SOAP is like using an envelope

Extra overhead, more bandwidth required, more work on both ends (sealing and opening).



REST is like a postcard

Lighterweight, can be cached, easier to update.

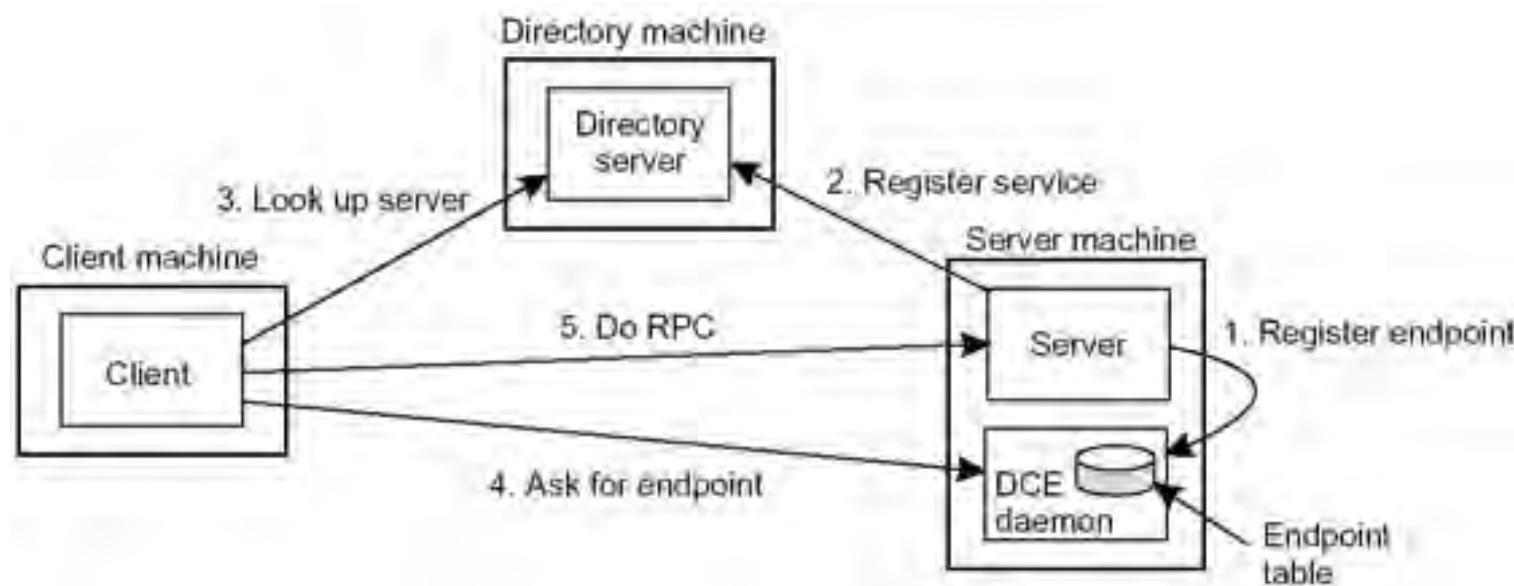
Fig.:
www.
upwork.
com/
hiring/
develop
ment/
soap-vs-
rest-
compa
ring
-two-
apis

Question of Architectural Philosophy vs. 'plain' Efficiency:

- Interface-based service environments requiring heavy messages
- Efficiency-based API 'style' using light-weighted http 'only'

V.1 Remote Procedure Call Characteristics

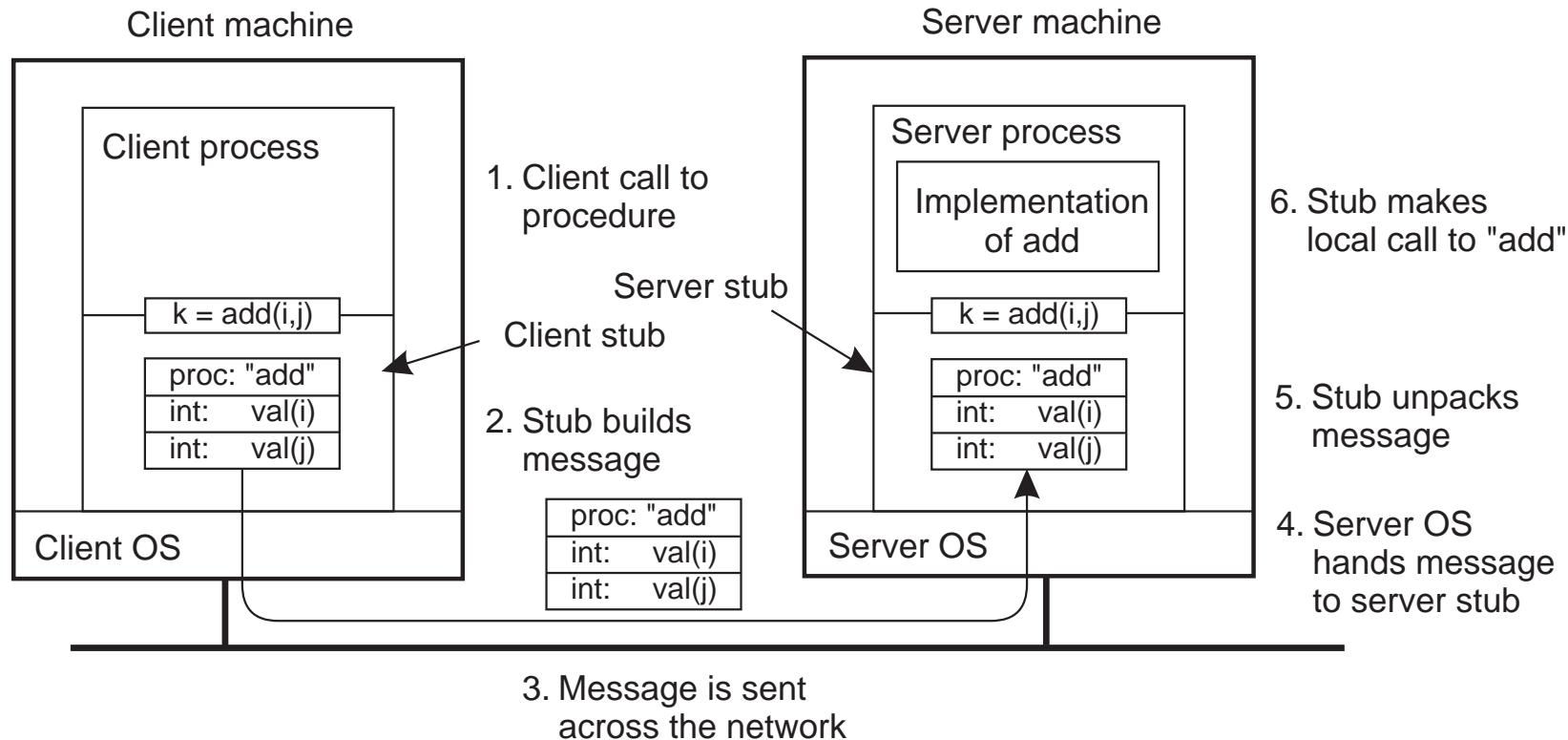
- Client program needs procedure/function signature for call
- Client process needs address of a server 'serving' the procedure



- Server needs a **registry**/directory to publish procedures infrastructure to accept remote calls as local calls
- Client needs an initial address for a **registry** or directory **lookup** functionality to find published procedures
- Client and Server need transport protocol(s) to interact at all

Tanen
baum
Distri-
buted
Systems
Fig.2.15
DCE

Internals of an RP Call with already known Server



- Steps 1. and 6. constitute a 'classical' local procedure call
- Client calls a *client stub* that mimics exactly a local call
internal steps 2./3. handle remote aspects and interaction
- Server side remote aspects are hidden by a *server stub*
 - * 4./5. transform message back into call and call local procedure
 - * takes result and sends it back to client stub (not shown)

RPC – Levels to 'guarantee' Transparency

Procedure call using in/out-Parameters **implemented** as:

- ▷ Procedure locally available \implies standard local *Library-Call*
- ▷ Procedure only remotely available \implies **Stubs** (Proxy procedures)

1. Caller (client): call	(User)	
Server lookup; marshal call; WAIT ...	(client stub)	c.f.
Usage of directory or meta-servers (<i>Trader/Matchmaker</i>)		pg. V-8
SendRequest with call;	(Client Network IF)	c.f.
2. Callee (server): ... wait using a GetRequest		pg. V-7
receive Request;	(Server Network IF)	
un-marshall request;	(server stub)	
local procedure-call ... WORK ... return result;		
marshal result	(server stub)	
SendReply with result; ...	(Server Network IF)	
3. Caller (client): WAIT;		
receive Reply with result;	(Client Network IF)	
un-marshall result;	(client stub)	
return result	(User)	

Problems w.r.t. RPC–Transparency

- **Performance:** Remote Calls imply lots of Overhead
 - ◀ transfer from (to) programming language MEM via serializing/packing (deserializing/unpacking) to (from) network
 - ◀ Operating System(s): process handling on both sides
 - ◀ Network: routing, varying load, errors, re-transmit, ...
⇒ Local Call should be much faster.
- **Infrastructure:** *Server* has to be identified and contacted
 - * **naming:** symbolic interface names, e.g., ports
 - * **locating:** fixed network address vs. inquiry via *broadcast*
more flexible: Meta-*Server* manages *Server* registrations
 - * **binding:** decide on concrete *Server* for runtime or a single call
⇒ Wide Spectrum of Supporting Environments:
 - fixed known URI, registry, ..., Service Eco Systems
 - basic functionality ... advanced Service-Level Agreements

SLAs

Problems w.r.t. RPC–Transparency – cont'd.

- **Parameter Handling:** has to respect distributed memory

- ▷ *call-by-value* easy via copy and marshalling
- ▷ *simple value results* easy via copy and marshalling

- ◀ **Problems:** 'Remote Addresses' in Distributed Memory?

- * *call-by-reference*: reference is useless on server side
- * Global variables or implicitly used local references in code
- * Additional resources, e.g., open file pointers

- ▷ **'Solutions':**

- ▷ Objects (references) may be marshalled and migrated to Callee
- ▷ *Call-by-visit/move*: temporary vs. 'permanent' migration

Remark: Object-oriented concepts reduce these problems!

- ◀ Files do not fit into the 'Remote' Computing Model

⇒ Store data in **Database Servers**

Problems w.r.t. RPC–Transparency – cont'd.

- **Synchronization** (Caller)

- **Standard:** Caller blocks until return-value is available exactly as in a local Procedure Call \implies Transparency

- Trade-Off:** simple control vs. idle time in Client-Processor

- **Optimizations** possible but forfeit transparency advantage

- **Asynchronous Calls without waiting for results** \implies Request-Protocol without Reply, e.g., SunRPC (timeout 0)

- **Asynchronous Calls with delayed waiting for results**

- Client control structures for 'postponed wait' required

- Concept:** promises \approx proxy for expected result; Client waits via claim-Operation which is blocked until result arrives

- Implementation:** Callable \approx Runnable with return value Future \rightarrow send to Executor; compute asynchronously, etc.

- Example.: `java.lang.*/java.util.concurrent` since Java 1.5

Liskov
1988DSG-
PKS-B

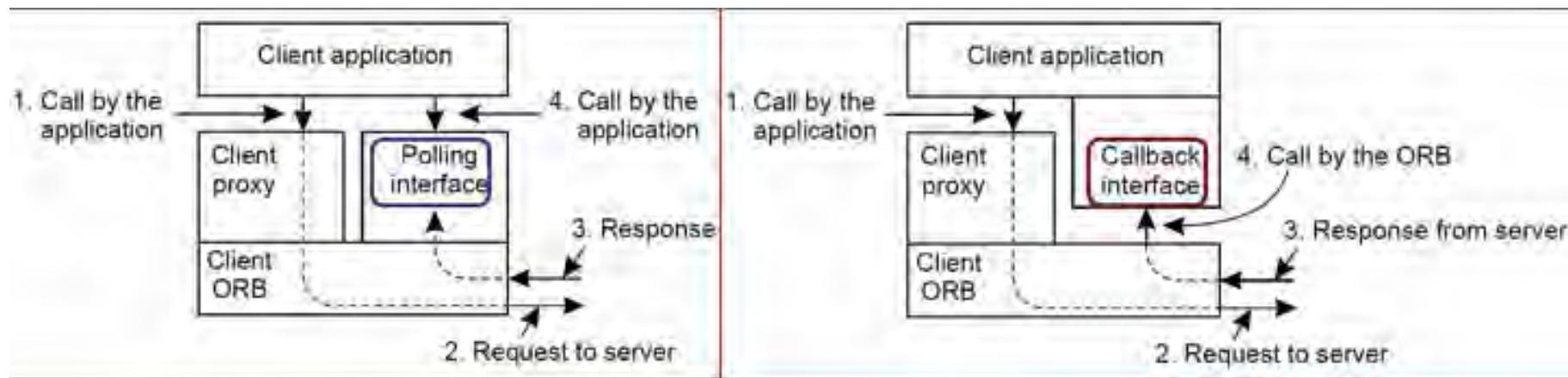
Relaxed RPC Model: Callback vs. Polling

Example: CORBA

Common Object Request Broker Architecture

- Client issues RPC (via stub) and resumes local work asynchronously
- Server gets requests and computes procedure
- Asynchronous model less transparent but much more flexible

Tanenbaum
Distributed
Systems
Fig.
9.7/8



New Issue: *Who triggers delivery of result? Client vs. Server*

- **Polling:** *Client* 'asks' explicitly for result when needed
- **Callback:** *Server* calls callback interface provided at client side
responsibility reverses Client/Server role for result

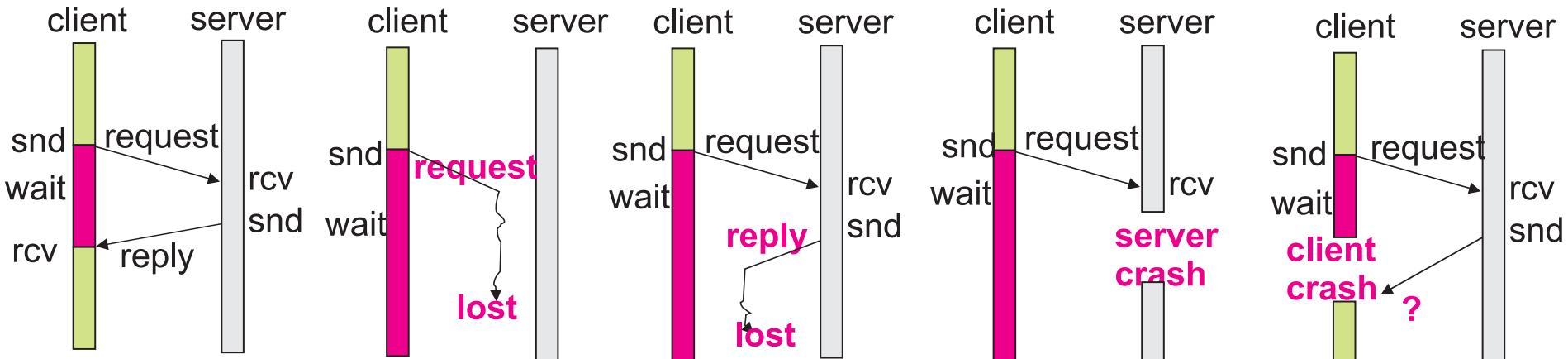
left

right

V.2 Client-Server-Interaction: Handling of Failures

Reasons: communication channels and/or compute nodes

- ◀ Messages: **request** or **reply** lost
 - ◀ Server unable to accept or process **request**, e.g., due to a **crash**
 - ◀ Server takes unexpectedly long to process **request** due to high load
 - ◀ Client is **crashed** and not able to accept result
- ⇒ Errors not easy to distinguish on client side



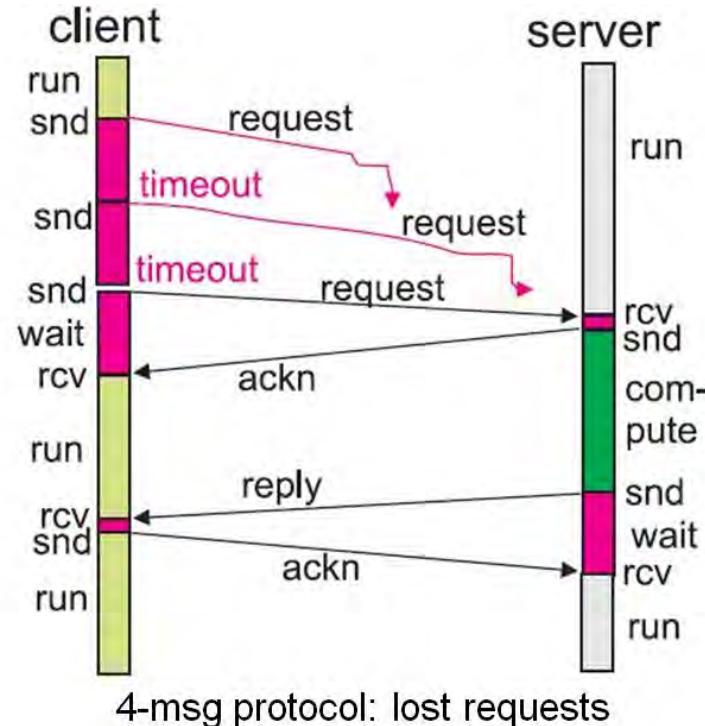
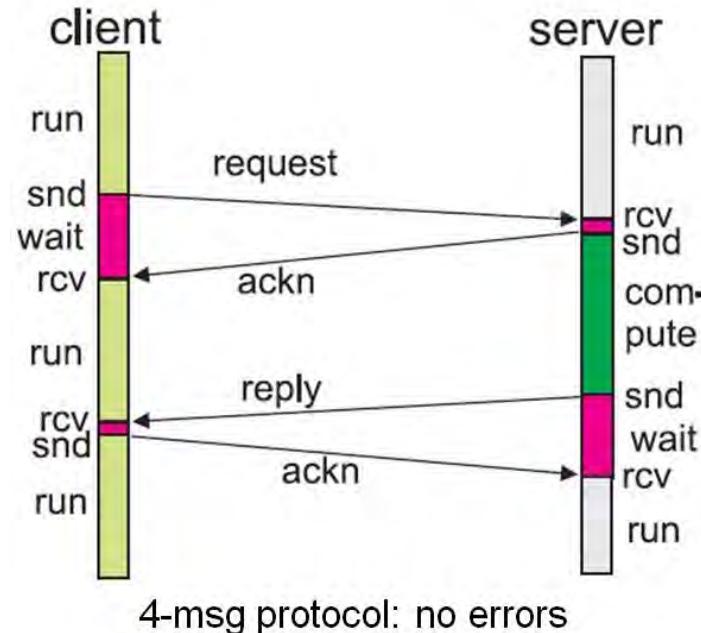
- ▶ Time-Outs and re-sends are critical to ensure successful interaction.
- ▶ Different levels of robustness can be achieved by different communication protocols for implementing RPCs.

C/S-Interaction Protocols: 4-MESSAGE-Protocol

rcv uses *Time-out* > transfer time + expected msg handling time

Time-out: Transfer time Client → Server + Server → Client

C.snd(req); C.rcv(ackn); ... C.rcv(reply); C.snd(ackn);



Advantages:

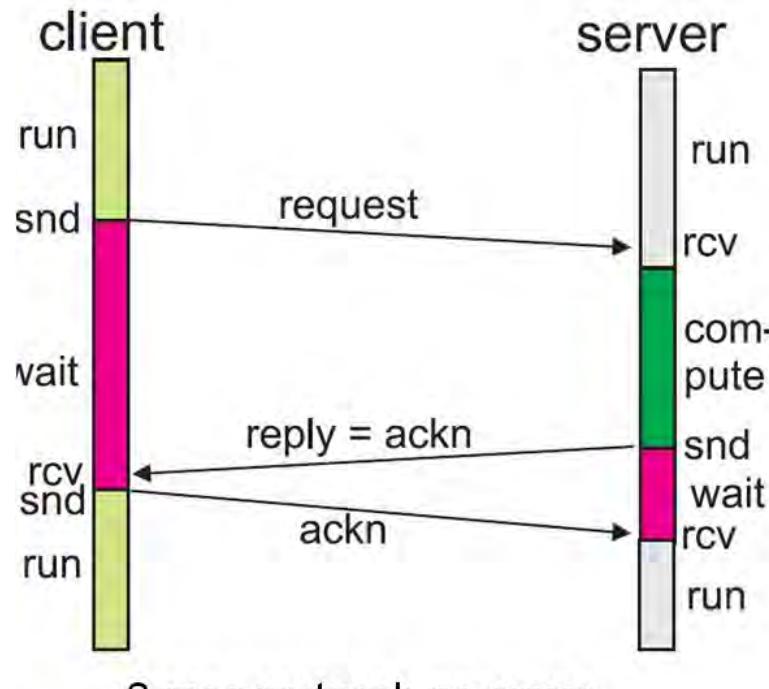
- ▷ Client is not blocked during entire Request processing
- ▷ Time-out is independent from 'job' dependent compute time

C/S-Interaction Protocols: 3-MESSAGE-Protocol

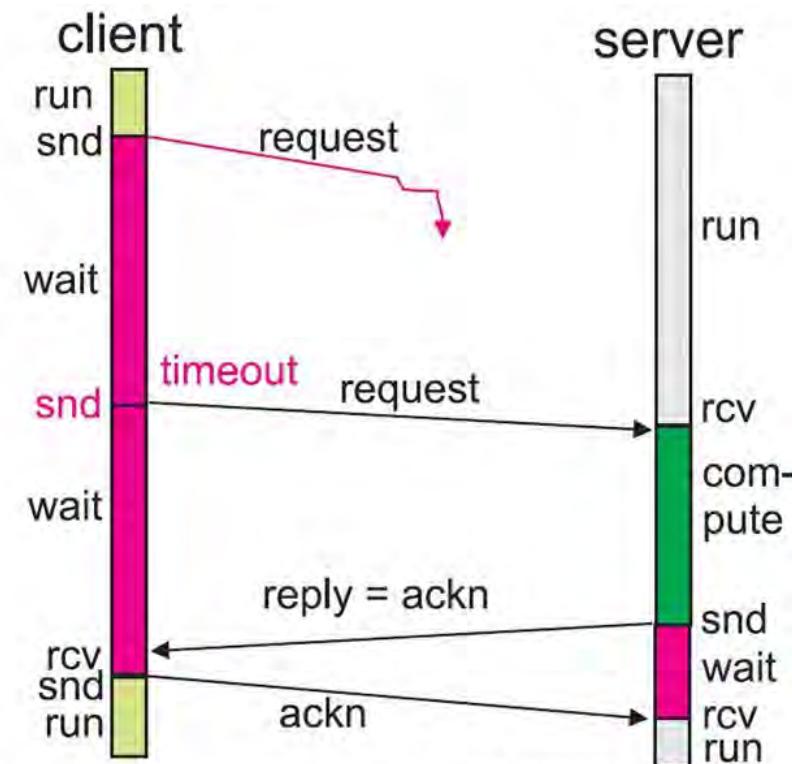
Request is not explicitly acknowledged;

Reply is used as implicit acknowledgement

`C.snd(req); C.rcv(reply); C.snd(ackn); ...`



3-msg protocol: no errors

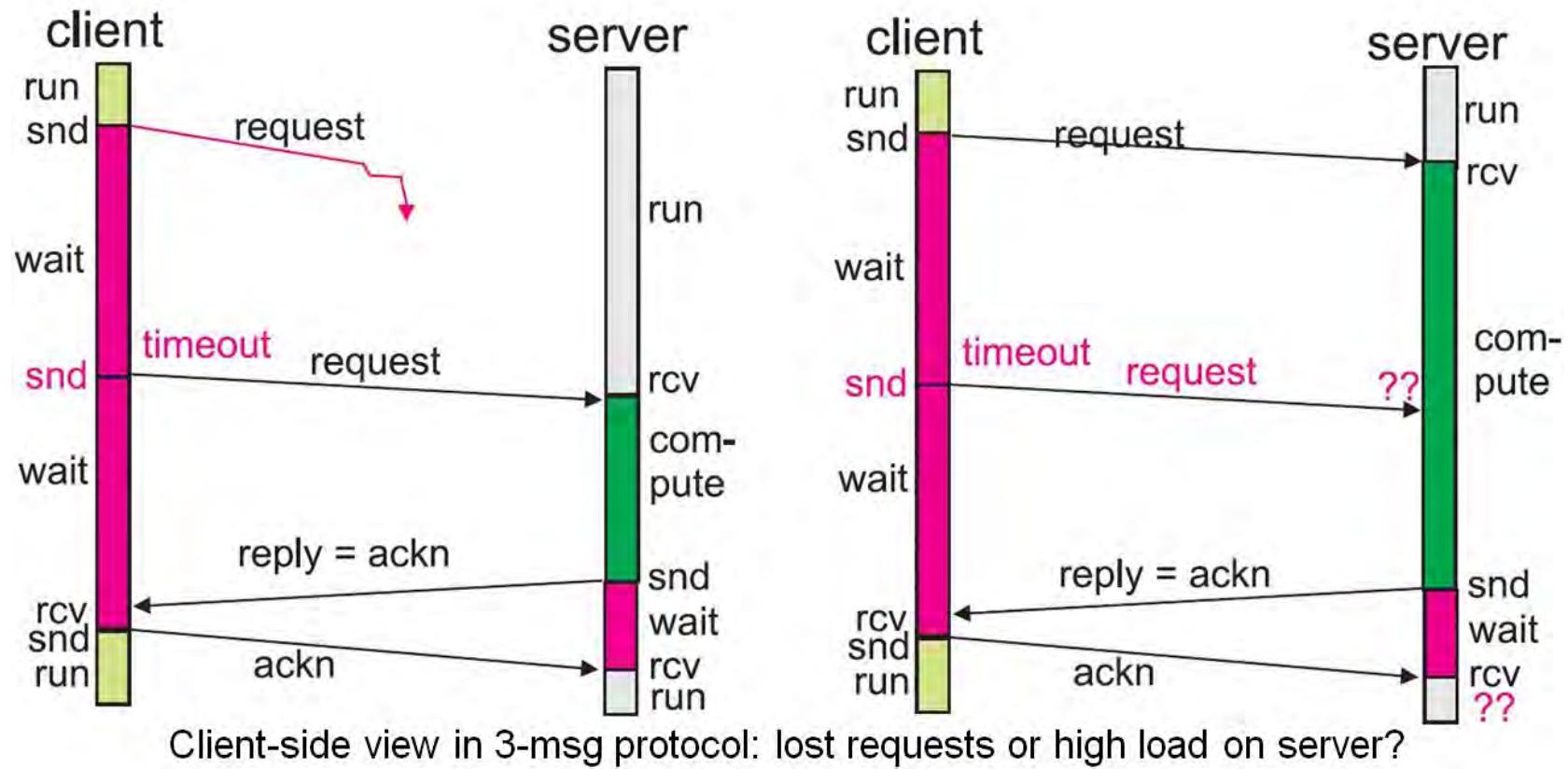


3-msg protocol: lost requests

Drawbacks: Client blocks for entire time of Req processing
 'long' timeouts have to respect compute time

Problem: Setting adequate Time-outs is critical

- ◀ too short: long compute times lead to repeated re-sends
- ◀ too long: errors are detected after long blocking times only



- ▷ Compromise for Time-Out on client side
- ▷ Server should always use short timeouts for ackn: avoids blocking.

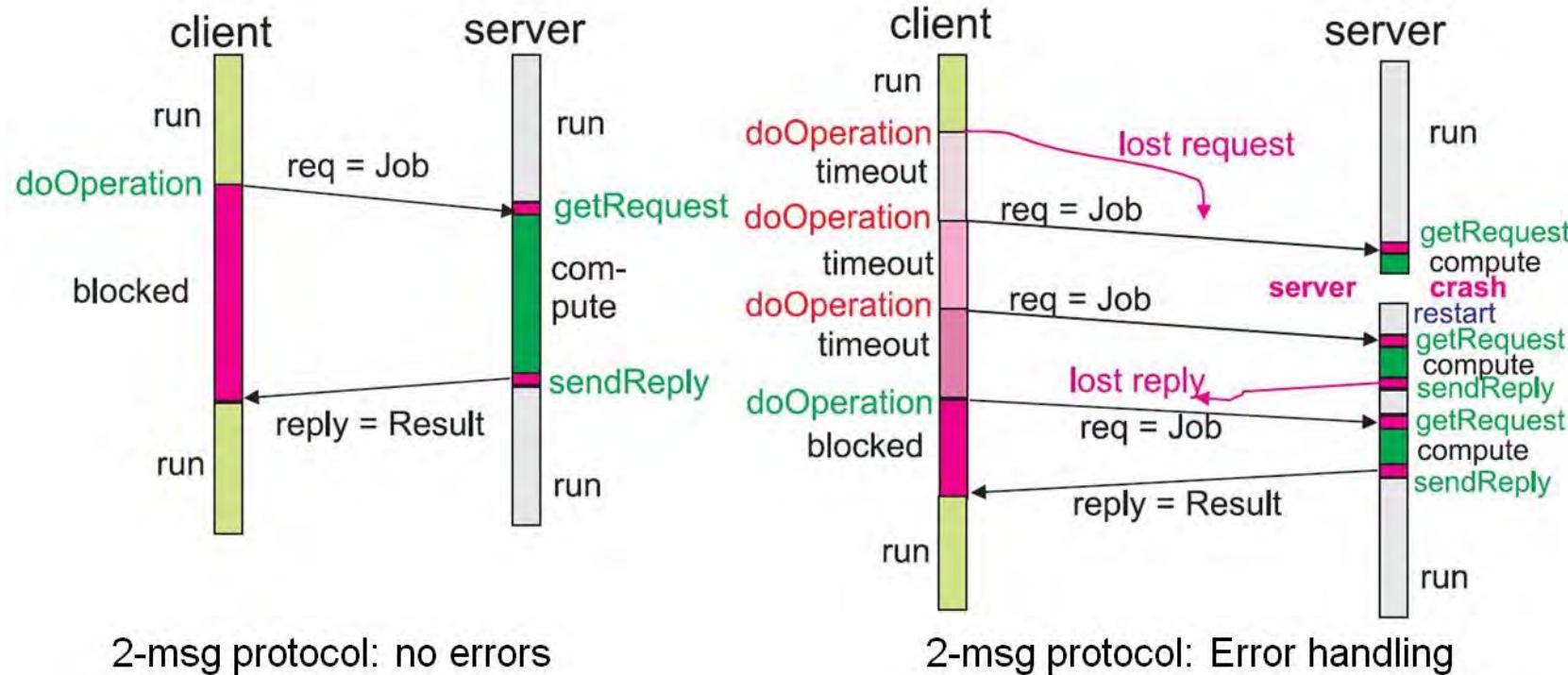
C/S-Interaction Protocols: 2-Message-Protocol

Concept: Do not use any explicit acknowledgments: optimal w.r.t
 $C.\text{snd}(\text{req}); C.\text{rcv}(\text{reply})$ msg-count

If reply is lost, simply re-send the request as 'new'

⇒ Protocol consists of 3 operations only:

$C.\text{doOperation}(\dots); S.\text{getRequest}(\dots); S.\text{sendReply}(\dots)$



Transparency: Client waits *exactly as in a local Procedure Call*

Semantic Models w.r.t. Failure Handling

Problem: *How to handle partial failures?*

- local calls: Caller/Callee on the same machine \implies both **crash**
- remote calls: typically only one side **crashes**
lost messages as an additional source of failures

Critical: **non-idempotent** operations causing side-effects

Example: `read(i)` vs. `increment(i)`; repeated money transfer

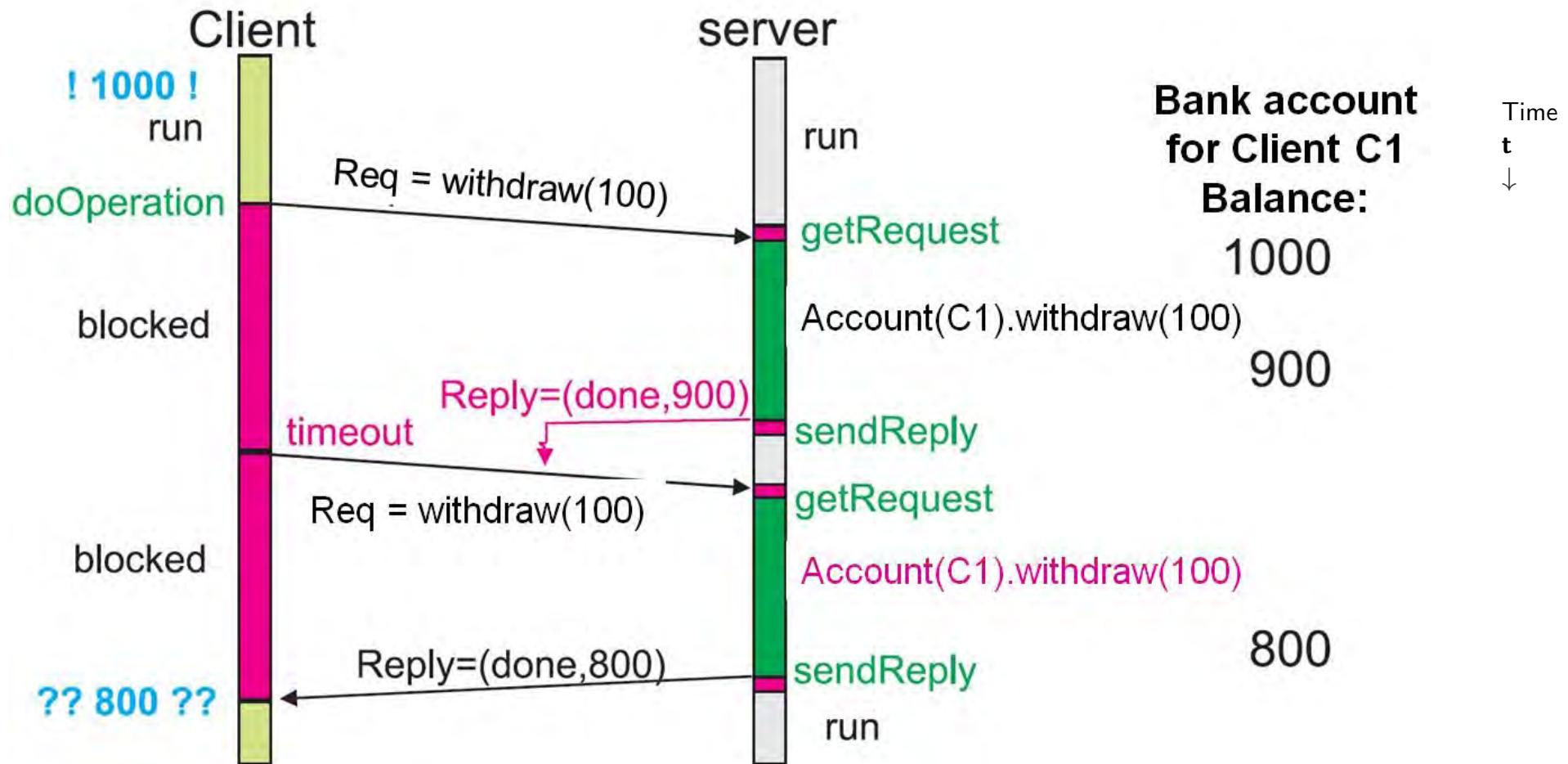
c.f.
pg.
V-20

Hierarchy of Models:

- (a) **Maybe/best effort:** No guarantees \implies hard to use
simple and efficient; Call is executed either once or not at all
- (b) **At-least-once:** System ensures execution of call
 - **Client:** repeats requests (`doOperation`) until executed
 - **Server:** doesn't check for *duplicates* \implies repeated executions?
 \implies high load levels combined with short **Time-outs** are critical
 \implies **multiple side-effects lead to unwanted system state**

c.f.
pg.
V-18

Problem: Lost Msg and non-idempotent Operation



- ◀ Lost replies as well as high loads may lead to multiple re-sends
- ◀ Server state gets corrupted due to unexpected side effects

Semantic Models w.r.t. Failure Handling – cont'd

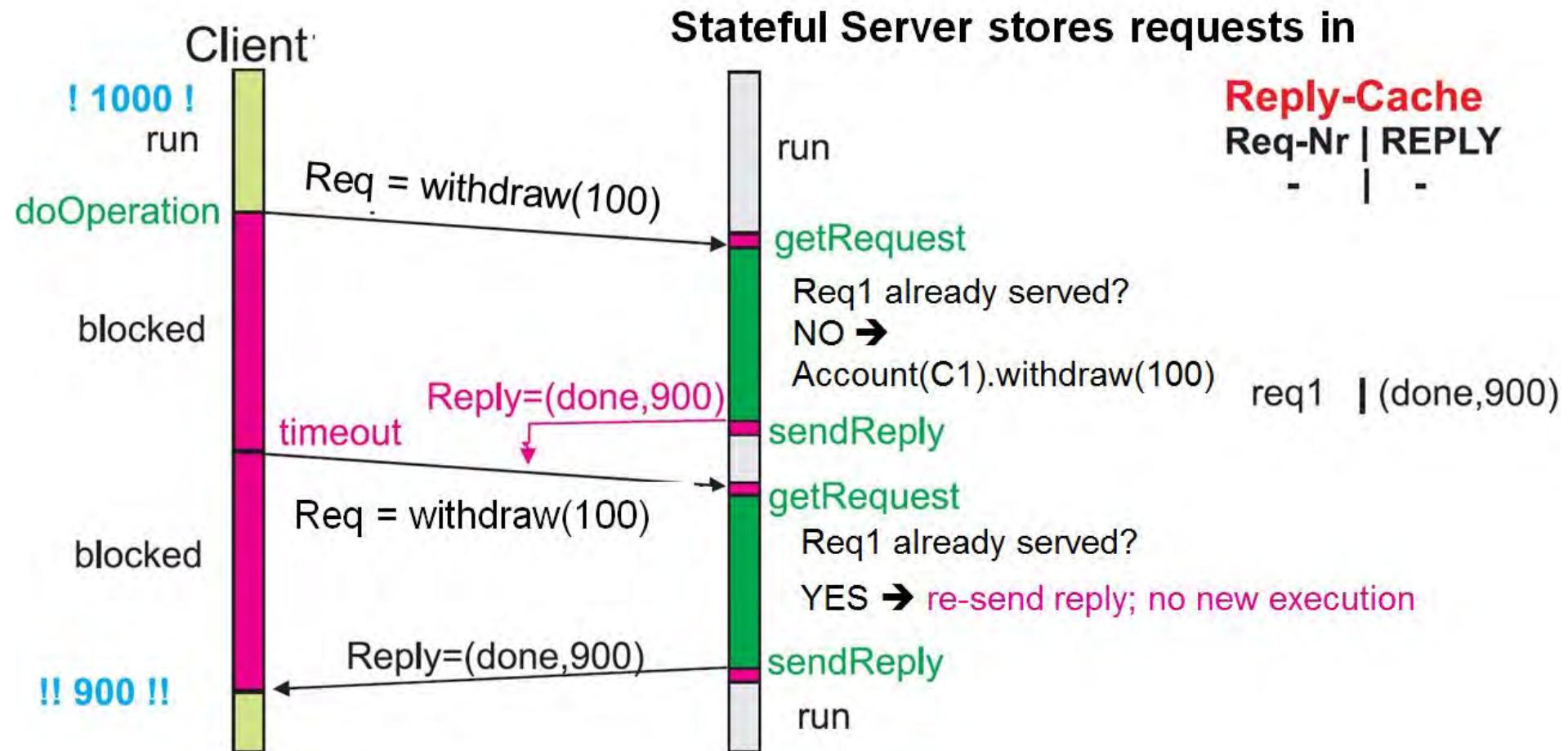
Hierarchy of Models (cont'd):

- (c) **Last-of-many**: Result from last Call is accepted only, but nested RPCs may lead to so-called **Orphans** for intermediate results
 - (d) **Last-one**: last-of many plus **Orphan-Handling** for nested calls
Technique: version numbers for calls; detects/discards **Orphans**
 - (e) **At-most-once**: Server detects and handles duplicate Requests
Technique: Server stores Request and Replies (for re-send)
⇒ **works well for non-idempotent calls**
 - (f) **Exactly-once**: Ensures execution also in the presence of **crashes**
Implementation: **at-most-once** plus *persistence* techniques
Req/Reply **Caching** detects duplicates
Problem: high overhead for global snapshot/rollback algorithms
- Trade-Off:** *High Quality-of-Service vs. Implementation Overhead*

c.f.
pg.
V-22

c.f.
chapter.
VI

Example: Server Recognizes Duplicate Calls



- Request cache and request numbers needed for all clients
- Request can be flushed from cache when acknowledgement arrives

Communication Protocols to implement RPCs

RPCs are a general mechanism to implement DS:

- ▶ wide range of requirements based on actual application
 - ▶ different Quality-of-Service models with varying overhead costs
- ⇒ **Support for different failure models helps:**

1. **R-Protocol (Request):** asynchronous RPC without return value
Optimal efficiency in case of lots of repeated calls
Examples: Screen **updates**; time signals of a 'master clock'
c.f.
pg.
V-12
2. **RR-Protocol (Request,Reply):** optimistic standard
 Reply_i acknowledges Request_i ; Request_{i+1} acknowledges Reply_i
Facilitates **at-least-once** implementation without much overhead
c.f.
pg.
V-18
3. **RRA-Protocol (Request,Reply, Acknowledge-Reply)**
Facilitates **exactly-once on Server**: store reply only until Ackn
Client should store order of **Requests** and acknowledgements
c.f.
pg.
V-16

Summary: Client/Server using classical RPCs

- ▶ Interaction model supporting moderate de-coupling of roles
- ▷ Synchronous version close to traditional procedure call
- ▶ Frequently used in DS: DCE; SunRPC; Corba; RMI; ...

- ◀ No transparency w.r.t. call-by-reference and resources
- ◀ No transparency w.r.t. infrastructure for servers etc.
- ◀ comfortable levels of failure transparency are costly

- ◀ Optimizations on client side lose transparency advantage
- ◀ No integrated concept of 'server state', e.g. for handling situations where calls cannot be executed due to program logic.

Modern Variant(s):

- *Light-weight RPC* models: XML-RPC; JSON-RPC; google gRPC
- *Services* based on Web Service Stack or REST-based APIs

V.3 Google gRPC

- 'Classical' RPC implementation based on an IDL
- Result of several steps of internal RPC implementations at google
- 'Incubating Project' of the *Cloud Native Computing Foundation* (c.f. www.cncf.io/projects/)

Characteristics:

- ▷ Designed for Interaction on the Internet via HTTP/2 transport
- ▷ Multi-Language Support (> 10 languages): protocol buffer as IDL, but other formats possible, e.g. JSON
- ▷ **Advanced Interaction Styles** between a single Client and Server:
 - * Single Message: Request-Reply
 - * Stream-based models: 1-n/n-1/n-m Message Stream Flow
 - * Synchronous and Asynchronous Interaction (Deadlines/Cancelling)
 - * Advanced Error Handling (easily mapped to HTTP Error Msgs)

Fig.
taken
from:
www.
slide
share.
net/
borisov
alex/
enabling
googley-
micro
services-
with-
http2-
and-
grpc?
next_
slide
show=1
pg.86

IDL: Protocol Buffers specify Payload and Services

- message declarations for payload structures for de/serialization
- service used to bundle one or more rpcs
- Single RPC: Keyword `rpc` plus Parameterlist and Result

`rpc <Name>(<Parmlist_Msg_Types>) returns (<Result_Msg_Type>)`

Interaction Styles specified via Parameter/Result:

- * message type names only \implies single Request and/or Reply
- * Keyword `stream` used for Parameter or Result
 \implies Server-/Client-/Bidirectional Streaming

Unary	Server streaming	Client streaming	BiDi streaming
Unary RPCs where the client sends a single request to the server and gets a single response back, just like a normal function call.	The client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages.	The client send a sequence of messages to the server using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response.	Both sides send a sequence of messages using a read-write stream. The two streams operate independently. The order of messages in each stream is preserved.

gRPC Interaction Styles and Synchronisation

- ▶ newStub \implies **asynchronous** calls from client
 - extended RPC model supports all interaction variants on both sides
 - * streams are not 'synchronized' between Client and Server
 - * interaction uses 'Observers': onNext; onComplete ...
- ▶ blockingStub: \implies **synchronous** calls from client
 - 'classical' RPC model; no streaming on client side possible
- ▶ newFutureStub \implies **synchronizes** explicitly when result retrieved
 - 'lazy evaluation' model; no streaming calls possible

Additional Synchronization:

- *Timeout*: how long waits a client for RPC result
- *Deadline*: fixed time when RPC result should be received
- *Cancelling*: always possible on client and server side

depends
on lang
uage

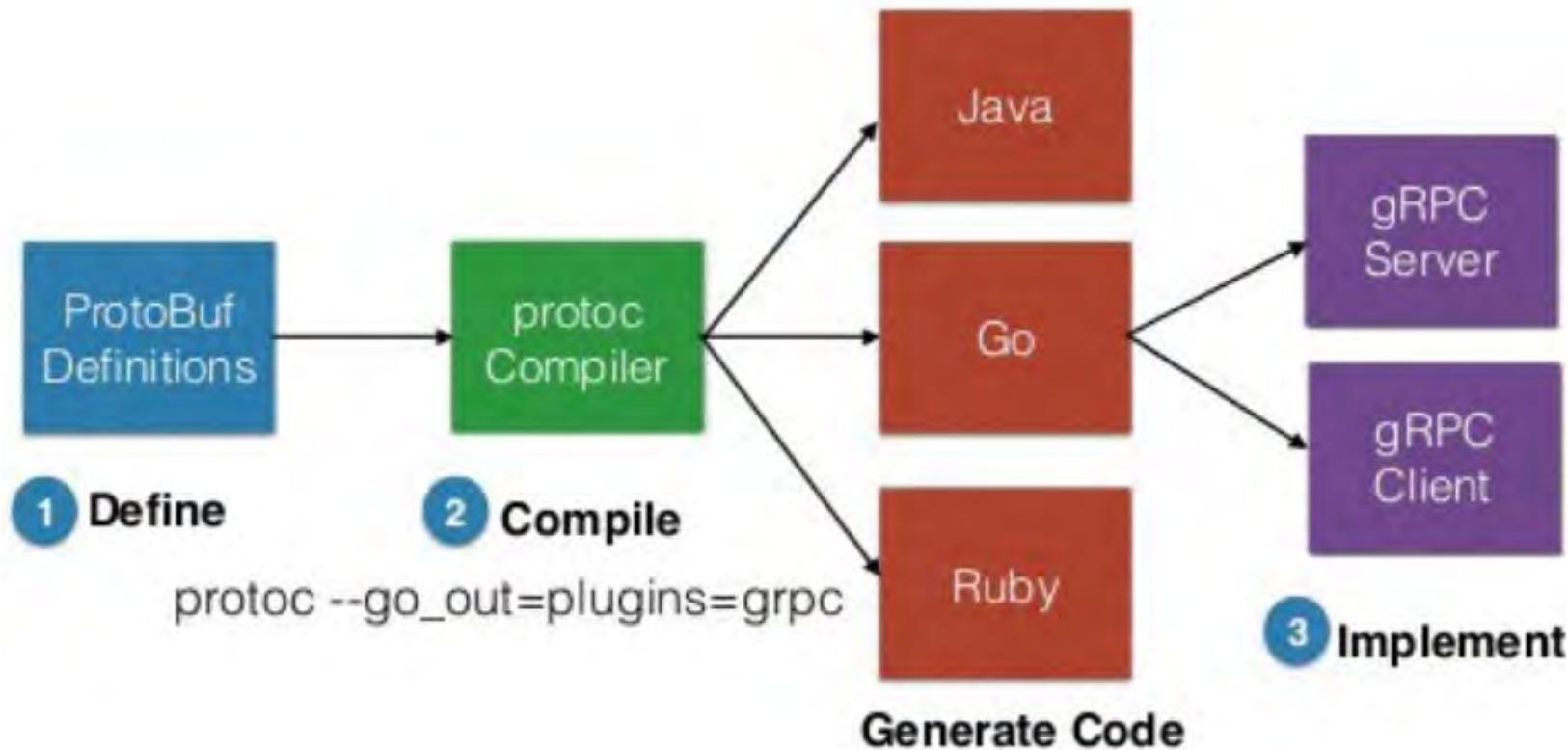
Problem: Inconsistent views w.r.t. 'Success' on client vs. server,
e.g. deadline matched on server side BUT deadline
exceeded on client caused by network transfer on reply

gRPC Error Handling Issues

- ▷ **Standard Error Model:** OK iff success, Error-Codes otherwise
 - * supported by all languages, implementations, stubs
 - * 16 general **gRPC Error Codes** supported by any model, e.g.
OK, UNKNOWN, INVALID_ARGUMENT, DEADLINE_EXCEEDED, ... PERMISSION_DENIED,
UNAUTHENTICATED, UNIMPLEMENTED, RESOURCE_EXHAUSTED , UNAVAILABLE, Data_LOSS
- ▷ **'Rich' Model:** detailed reasons and/or how to overcome an error specified in `status/error_details.proto` files:
 - * BadRequest info w.r.t. wrong parameter types or ranges
 - * RetryInfo for suitable timeout; QuotaFailure; ...
 - * Stack trace, meta-level info for request, documentation

Remark: codes can be matched to HTTP/2 error codes
(github.com/googleapis/googleapis/blob/master/google/rpc/code.proto)

Developing a gRPC Multi-Language Application



- Specify the service interface
- Implement the server code
- *Tooling:* Generate stubs needed for your language(s) for both sides
- Implement your Client

Fig.:
medium.
com/
@akshit
jain_
74512/
inter-
service-
commu-
nication-
with-
grpc-
d815a56
1e3a1

Szenario: Combine Multi-Language Services

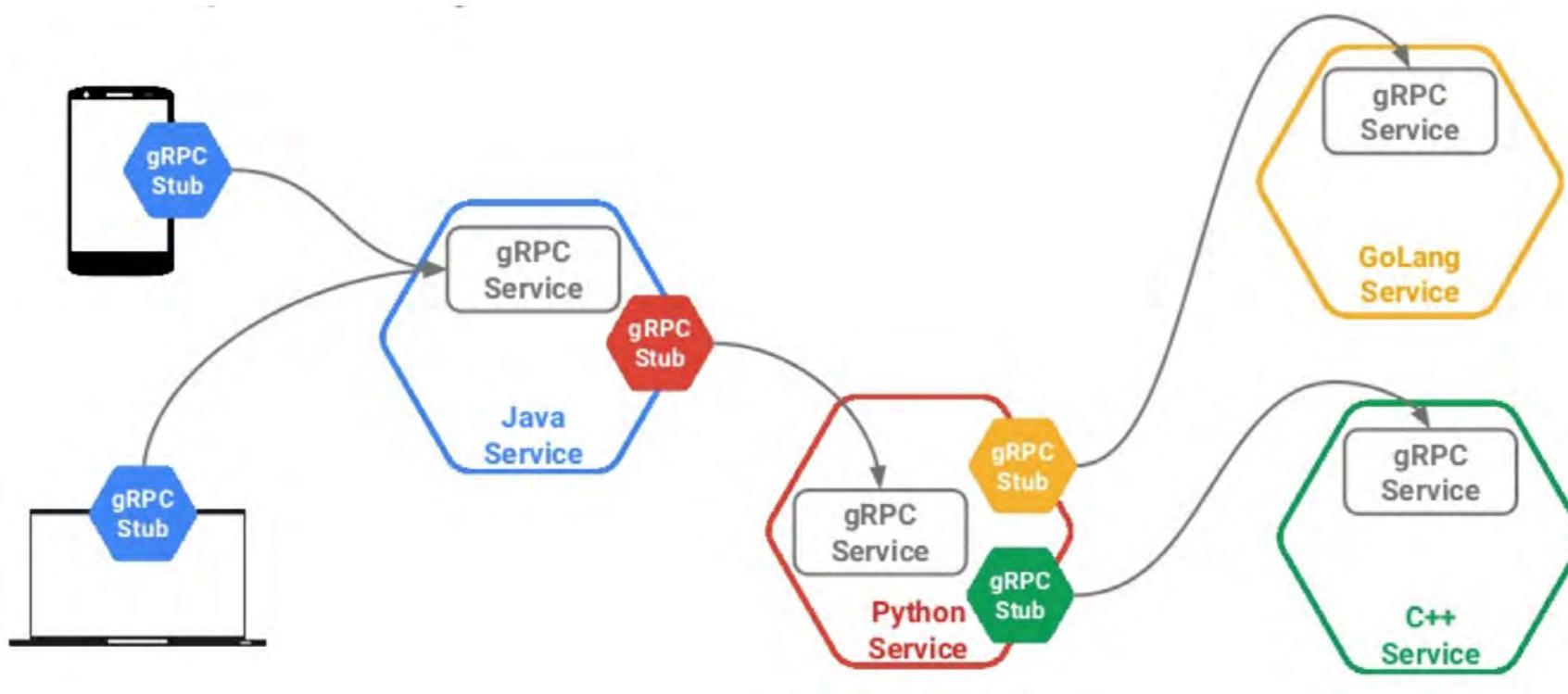
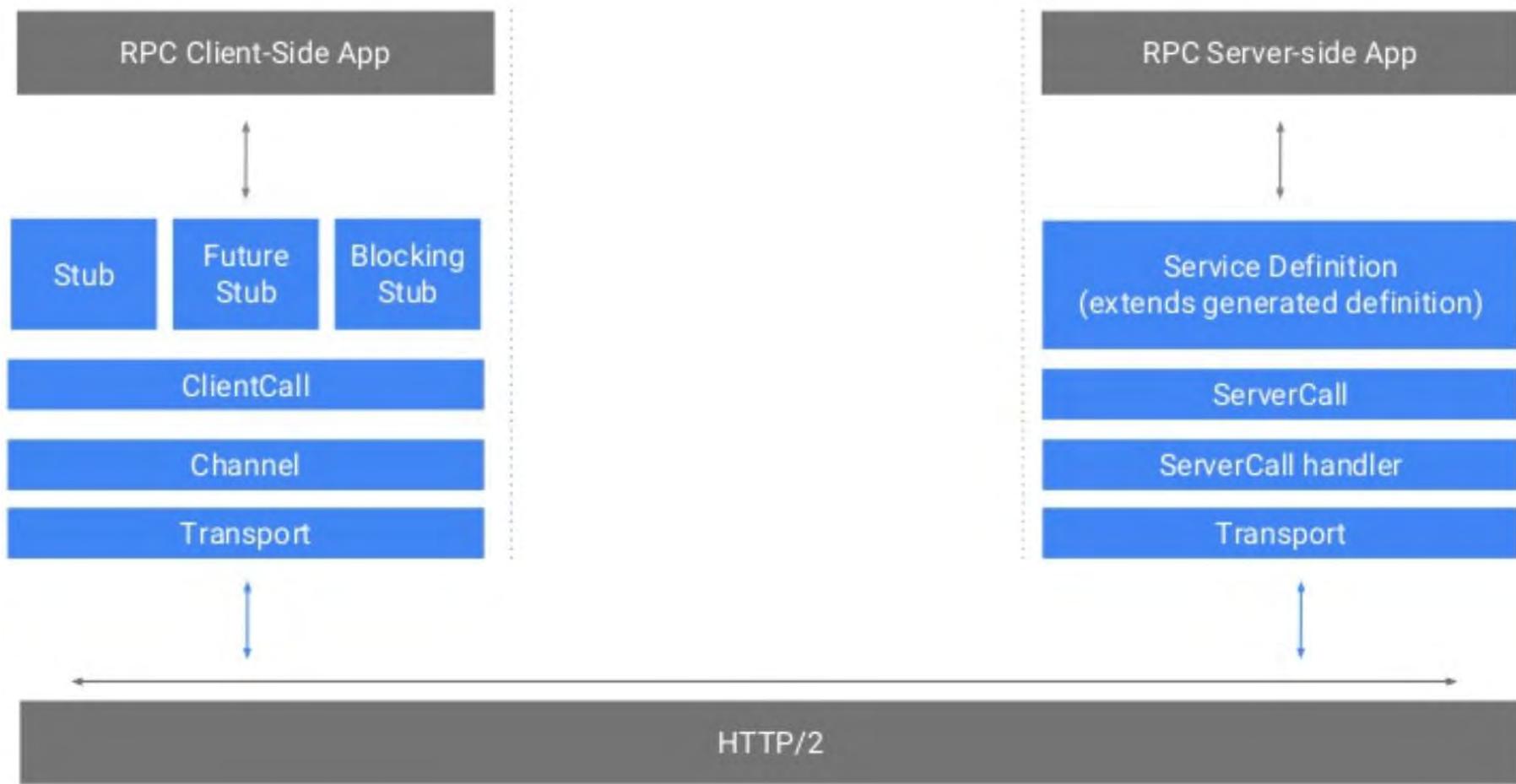


Fig.
taken
from:
www.
slide
share.
net/
borisov
alex/
enabling
googley-
micro
services-
with-
http2-
and-
grpc?
next_
slide
show=1
pg.99

- ▶ IDL as a special 'extra' language guarantees independence
 - proto files describe Msg formats \implies Transport Independence
 - proto files describe Service Interface \implies Call Independence
- ▶ IDL plus Stub-Compilers allow for easy language support

Behind the Scenes – 1: HTTP/2-based RPC



- Classical RPC implementation (see DCE)
- Stubs for handling a/synchronous calls and synchronization
- Channel: HTTP/2 connection between client stub and server

Fig.
taken
from:
www.slide-share.net/
borisov
alex/
enabling-googley-microservices-with-http2-and-grpc?
next_slide
show=1
pg.165

c.f.
pg.
V-8

Behind the Scenes – 2: Multi-Channel Support

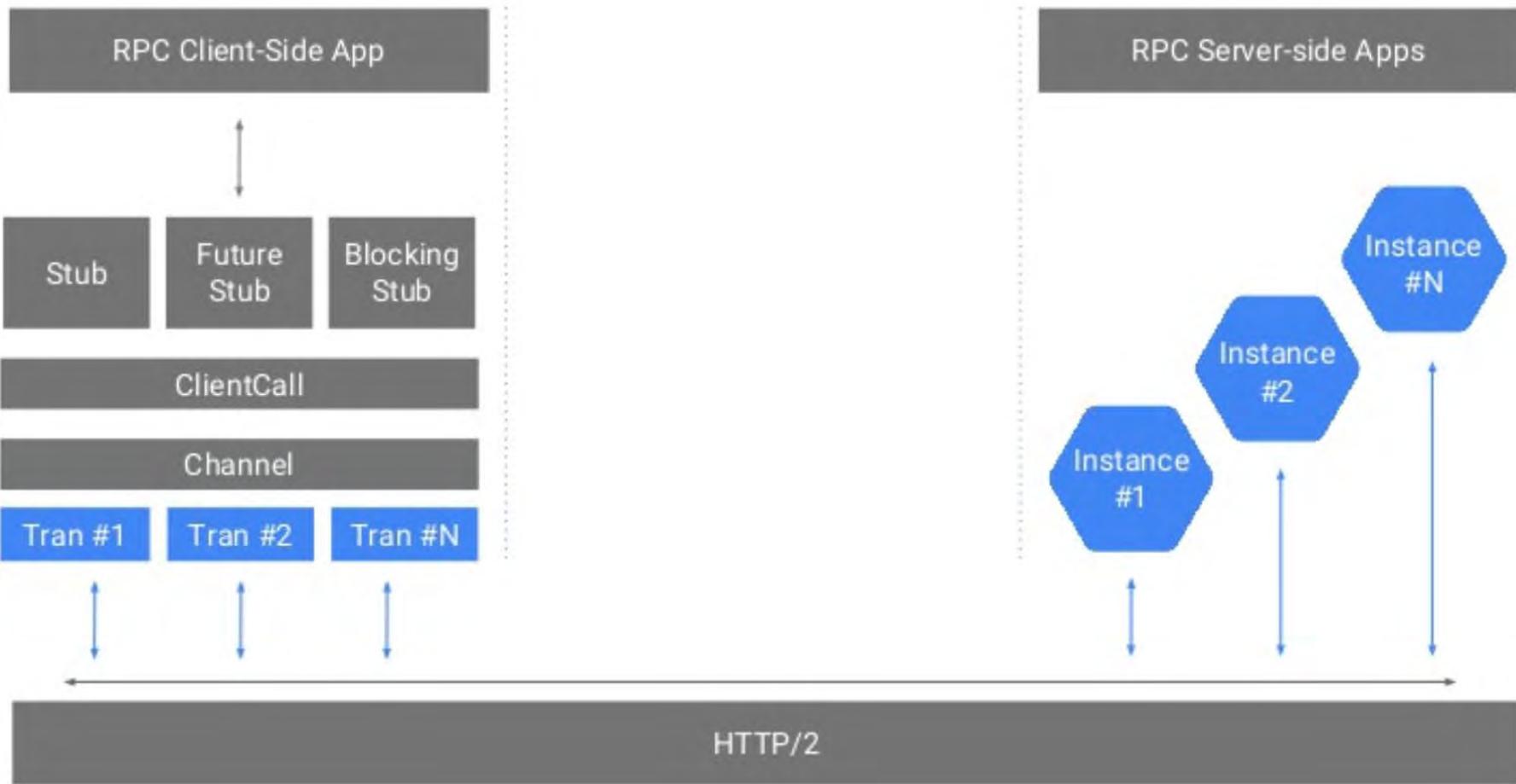
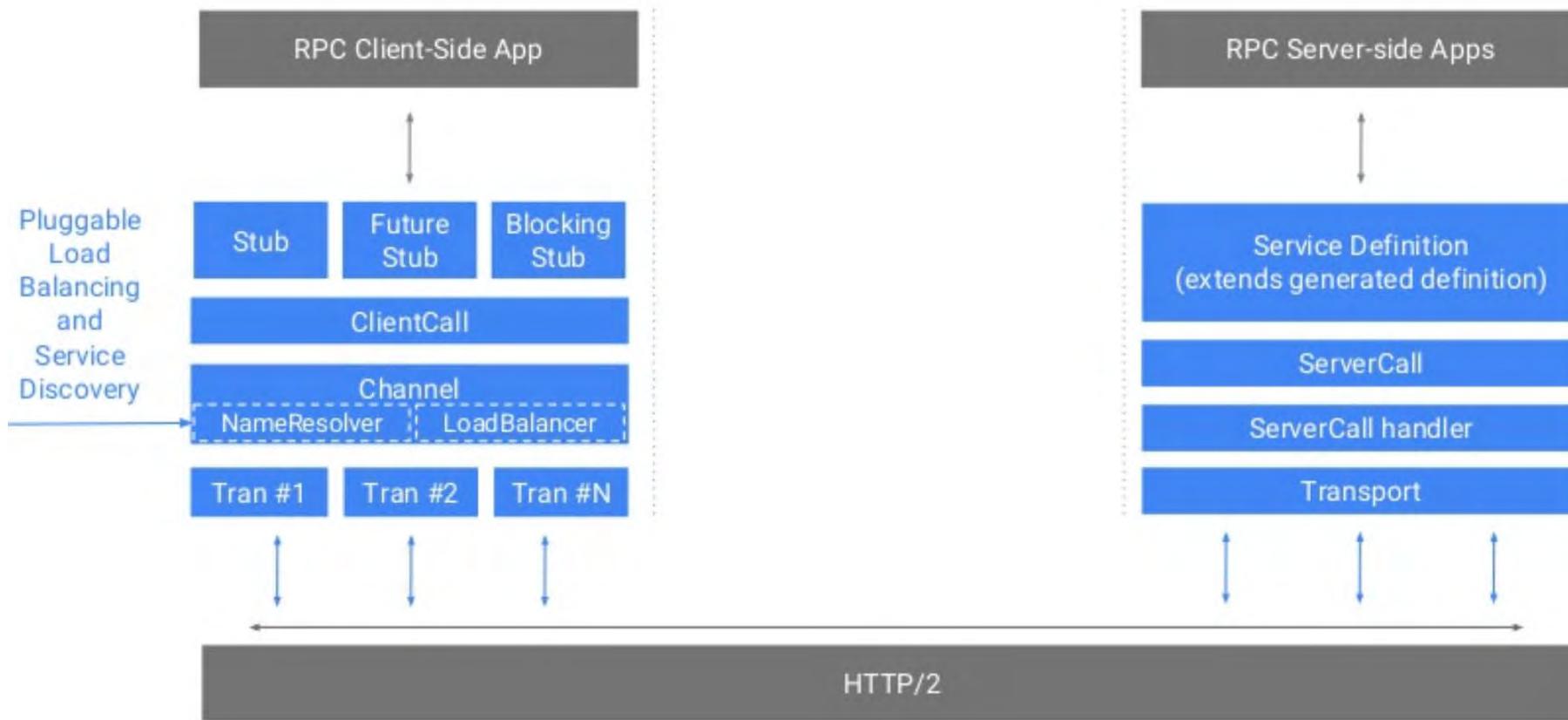


Fig.
taken
from:
www.slide-share.net/borisovalex/enabling-google-microservices-with-http2-and-grpc?next_slide.show=1
pg.167

- service may be implemented by many servers ('instances')
- Transport may be duplicated in order to gain performance

Behind the Scenes – 3: Channel Loadbalancing



- Higher-level techniques usable through 'Interceptors' as 'plugins'
- Finding suitable servers based on Discovery Service
- Include 'Load balancing' for optimizing channel usage etc.

Fig.
taken
from:
www.slide-share.net/borisovalex/enabling-google-microservices-with-http2-and-grpc?next_slide=show=1

End
of
V.3

V.4 REST-based Services

[Roy Fielding, Diss. 2000]: '*The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. [...] It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.*'

www.
ics.
uci.
edu/
~fielding/
pubs/
dissertation/
top.htm

REST: REpresentational State Transfer

- Architectural Style, not a concrete technology \implies 'very abstract'
- inspired by/dedicated to the World Wide Web
- Handling of Web resources, esp. hypermedia content of all kind
- developed 1994 – 2000 'in parallel' to URI and HTTP/1.1

Remark: We discuss only an overview of REST in general and focus on the C/S (RPC) aspect in interpreting **RESTful services** as HTTP-based RPCs. There is much more to the REST architectural style when it comes to the detailed architecture guidelines and the Web in general. A more concise treatment is offered in the master DSG-SOA-M module. ♦

REST Characteristics – Resources and Addresses

'A resource can be anything that has identity.'

'[...] abstract concepts can be resources.'

- 'Everything is a **Resource**' as the basic concept:
 - * documents, images, video/audio, (structured) objects, **services**
 - * works on single as well as multiple entities (collections)
 - * resources may be **linked** together and navigated like **Hypertext**
 - * single resource may have various different representations/formats
- **Addressing:** Uniform Resource Identifier (URI)
 - * different types of 'Identification' : **UR Name** vs. **UR Location**
 - * extensible syntax based on schemes, e.g. `https`, `mailto`, ...
`scheme://[userinfo@]host[:port] path [?query] [#fragment]`
`https:// en.wikipedia.org /wiki/Uniform_Resource_Identifier#URLs_and_URNs`

- * direct access or dynamically navigating through **links**

⇒ **general and universally applicable concept.**

tools.
ietf.
org/
html/
rfc2396
(1998)
rfc3986
(2005)

REST Characteristics cont'd – Interaction

- **Interaction:** 'Messages' based on HTTP interaction model
 - * **uniform** requests: POST, GET, PUT, DELETE, ...
 - * **uniform** responses via HTTP-defined formats and codes
 - * typical API style: **C**reate **R**ead **U**pdate **D**elete (CRUD)

c.f.
pg.
V-59

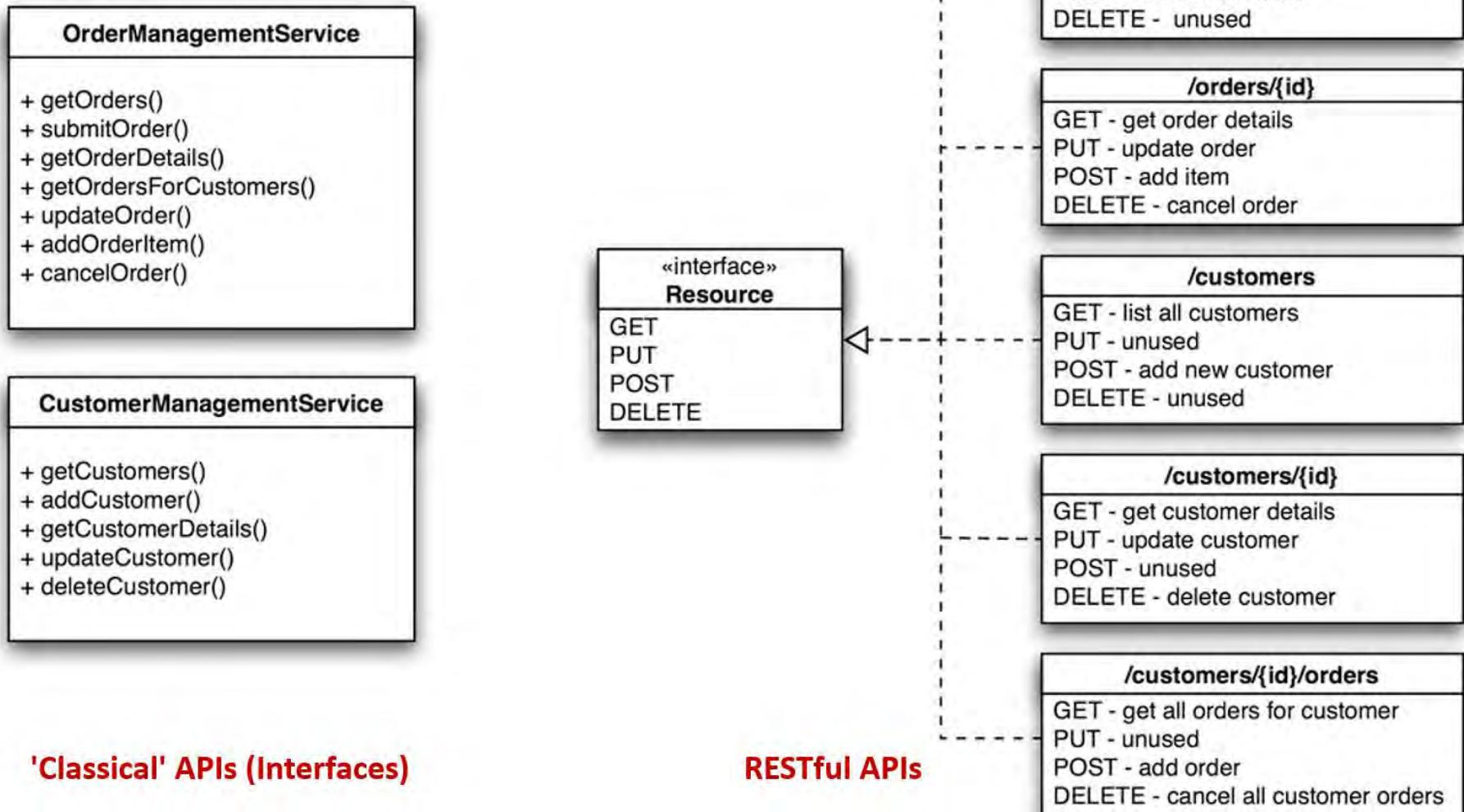
Method	REST Operation	Description
POST	CREATE (INSERT)	Create or update
GET	READ (QUERY)	Query about the resource
PUT	UPDATE (CHANGE)	Update
DELETE	DELETE (DELETE)	I want to delete what-ever-it-is
HEAD		I'm something like 'GET' [1]
OPTIONS		JAX-RS mumbles something about me.

Fig.:
de.
slide
share.
net/
imcin
stitute/
java-
web-
services-
15-
rest-
and-
jaxrs

- Service APIs are always syntactically uniform
- No fixed API (syntactical service spec) needed **before** contacting a service \implies **highly flexible and supports dynamic change!**
- Service Semantics in (more) resources and operation parameters

Example – Classical vs. RESTful API Style

Figure(s) taken from: Stefan Tilkov (INNOQ)
<https://www.infoq.com/articles/rest-introduction/>



REST Characteristics cont'd – Stateless Services

- **Stateless Services** as an architectural principle
 - ◀ C/S applications without state are mostly useless
 - ▶ state should **not** be kept 'implicitly' on the server side, but
 - ▷ store server-side aspects of the state explicitly in resource state
 - ▷ keep everything else on the client side
 - ◀ avoid client-specific server states like, e.g., sessions, cookies
- ⇒ **scaling, load balancing, replacing resources much easier**

- **Messages** (Request/Replies):

- * have to carry always all relevant information
- * transfer *Representations of Resource States*
- * may be *cacheable* on Client side for performance reasons
- * trigger effects on states by sending changed representations

⇒ **flexible and easy for distributed systems**

hence
the
Name

see:
coupling
levels
pg.
III-2

Combining gRPC with REST-based Environments

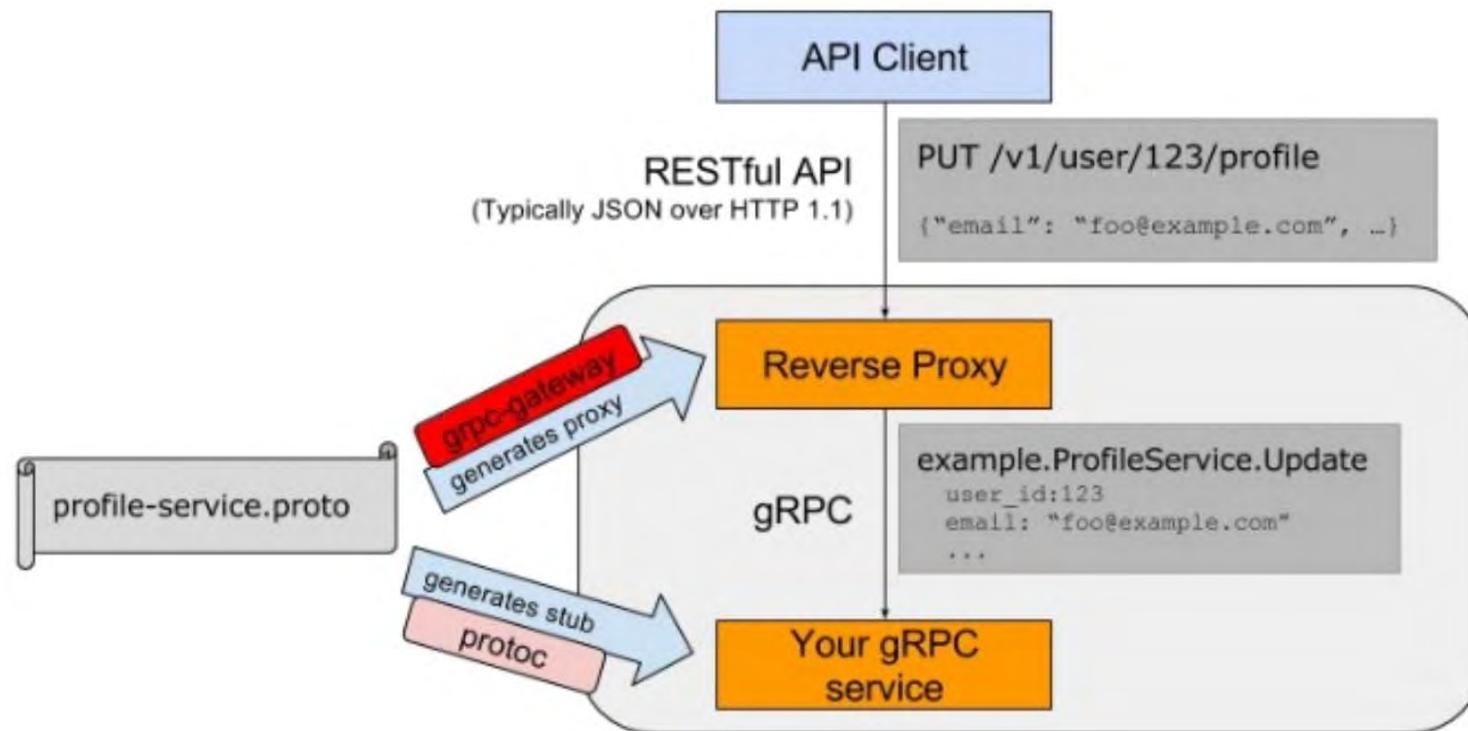


Fig.
taken
from:
www.
slide
share.
net/
borisov
alex/
enabling-
googley-
micro
services-
with-
http2-
and-
grpc?
next_
slide
show=1
pg.193

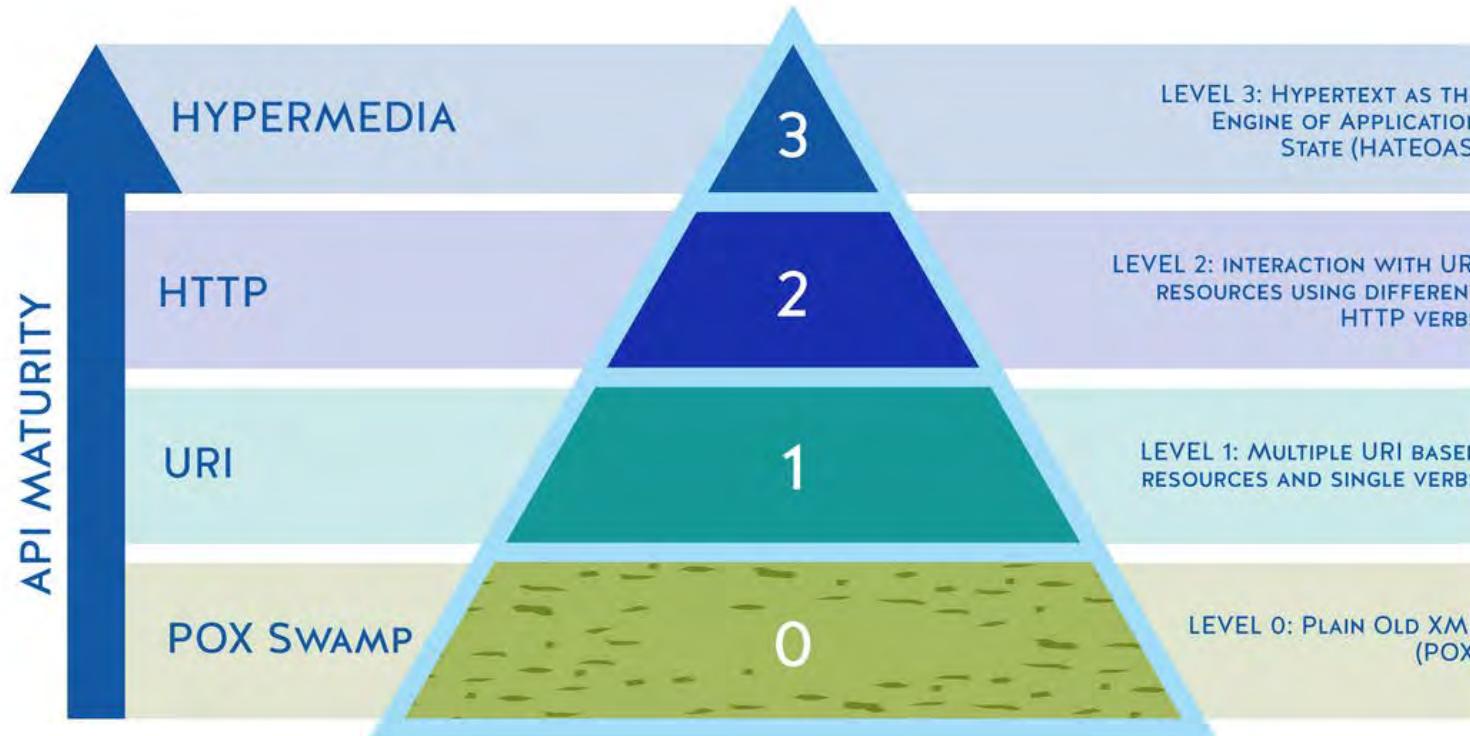
- ▷ gRPC as part of Google eco-system \implies Interoperability
- ▷ Interfacing other popular techniques, esp. REST-based APIs is done in a way that a gRPC service is reachable and usable by:
 - * 'ordinary' gRPC using the generated stub
 - * gateway to a RESTful API via generated 'Reverse Proxy'

Taking Restful Services serious – HATEOAS

Hypertext As The Engine Of Application State

Term coined by Leonard Richardson (2008)

- ▷ Enhancing C/S architectures step by step towards 'solid REST'



www.
crummy.
com/
writing/
speaking
2008-
QCon/
act3.
html

Fig.:
nordi
capiis.
com/
what-
is-
the-
richard
son-
maturity-
model/

- ◀ Level 3: only RESTful level in the sense of R. Fielding's definition

Representation Formats and Java-'Binding'

- Level 1 up: No need for 'classical' operation interface specification
- Needed: format(s)/structure definition for representations (msgs)

⇒ *Description Languages* for RESTful APIs?

- ◁ Web Application Description Language (WADL) ≈ WSDL4REST?
- ▷ RESTful API Modeling Language (RAML): YAML-based
- ▶ OpenAPI (Swagger): resource structure specifications
- ▶ needed: structure/format of resource data, e.g. JSON, XML

Java API for RESTful Webservices JAX-RS: (*Intro 2nd Assignment*)

- Code-First approach using Java-annotations: @PATH, @POST, @GET ...
- JSON is used for msg content specification
- Java-based tooling (Jersey as reference implementation)

V.5 Webservices and SOA

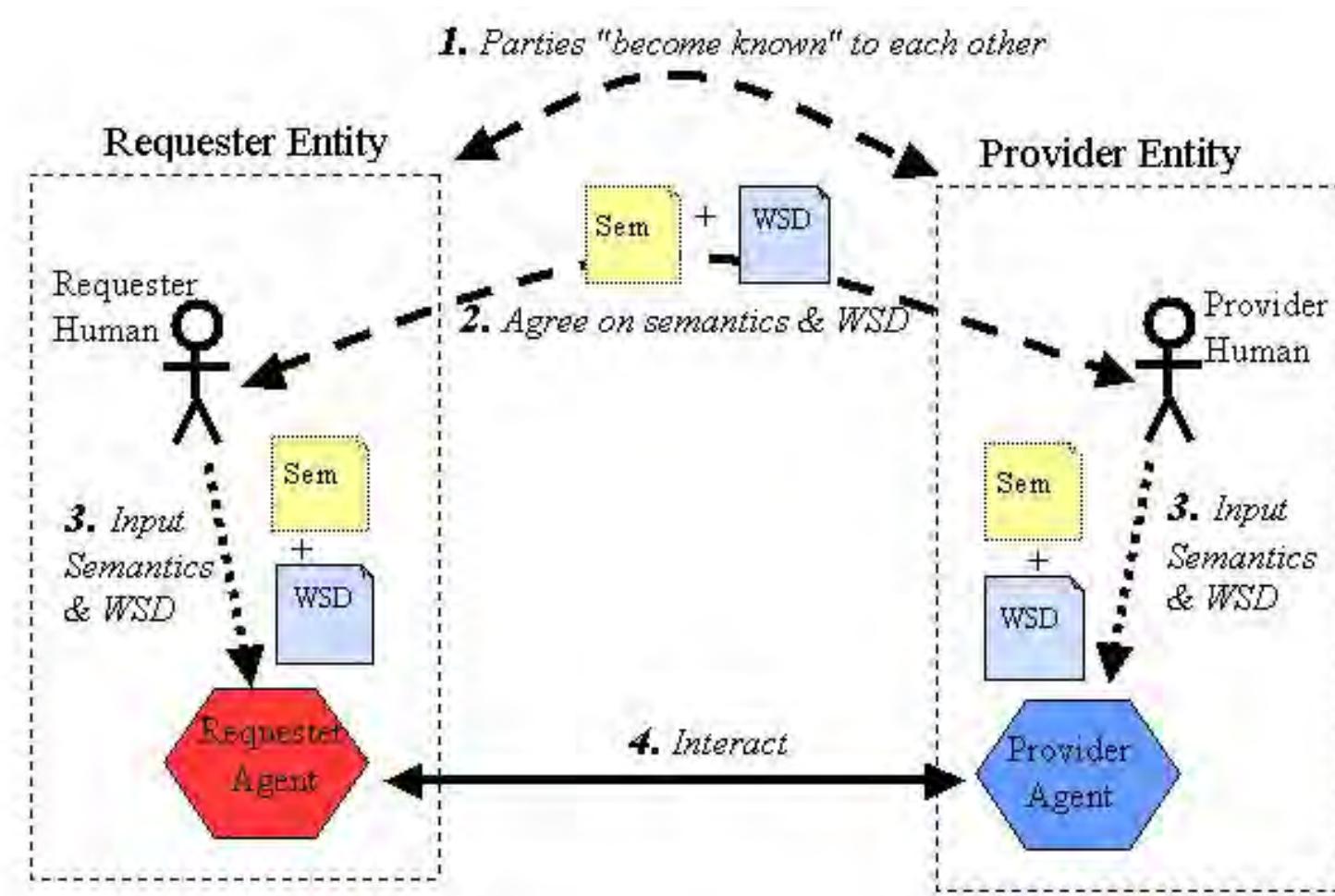
- ▷ **Client/Server-paradigm** uses '*Service*' notion implicitly
- ▶ **Service-Oriented Computing (SOC)**
 - * centers around an explicitly defined notion of '*Service*'
 - * provides '*the*' C/S implementation from approx. 2000-2015
 - * fosters a *description-oriented style* of programming
- ▶ **Driving force:** '*Business Needs*' for **EAI** and **B2Bi**

Definition V.1: (W3C WSA Group 2004)

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.



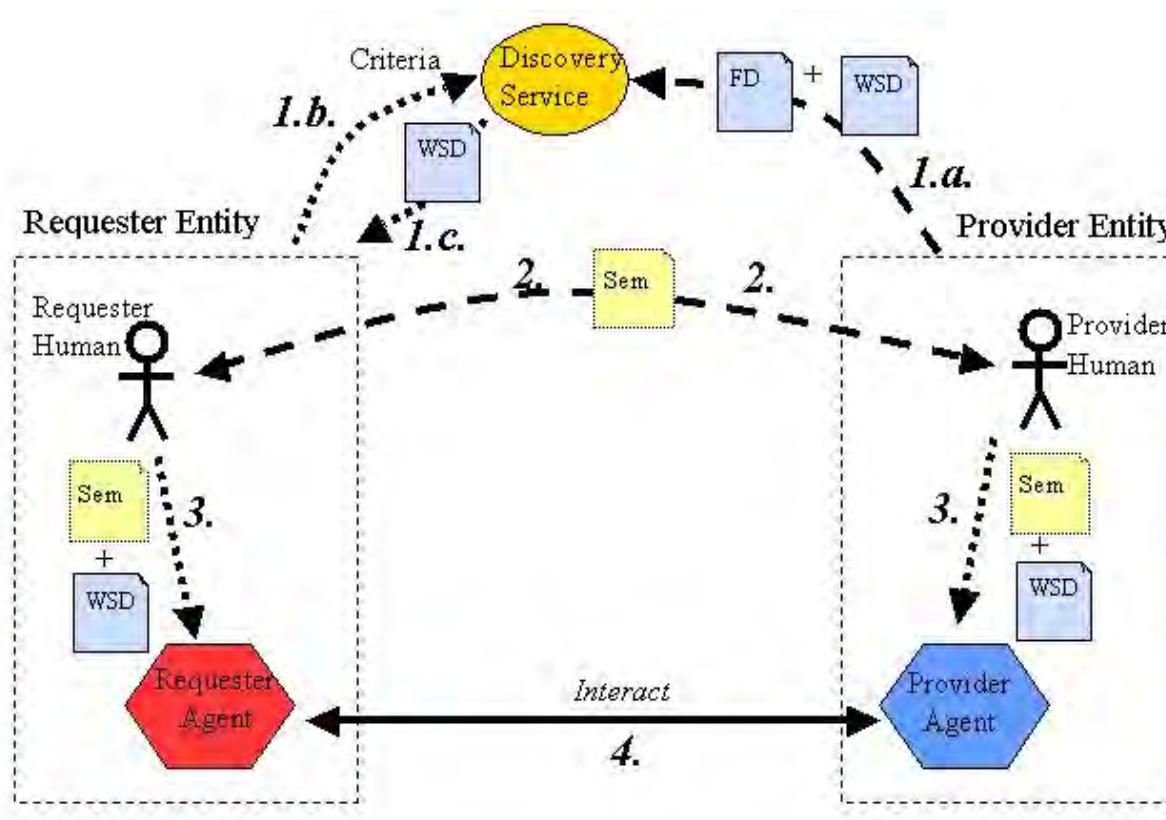
Webservice Model: Generic Usage Scenario



www.
w3.org/
TR/2004
NOTE-
ws-arch-
2004
0211
Fig 1.1

- ▷ **Roles:** Requester vs. Provider for a single interaction
- ◁ generic paradigm, esp. regarding implementation of step 1.

Practical Usage Scenario using Discovery Service

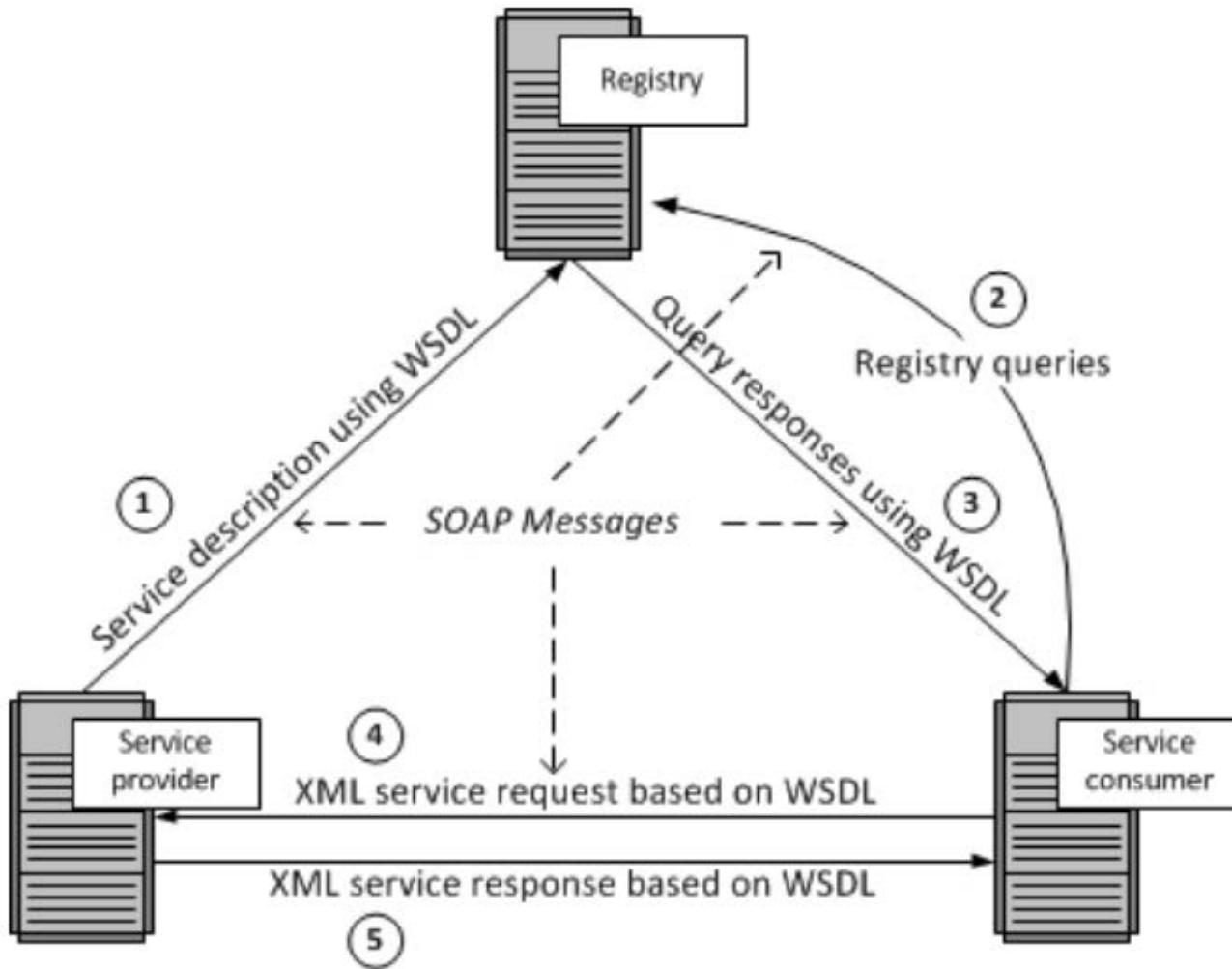


- * 1.a: Provider exports functional description *FD* for DS
- * 1.b: Requester looks up criteria in DS
- * 1.c: Requester gets description(s) from DS
- * 2: Requester and Provider negotiate w.r.t. service usage
- * 3./4.: Requester and Provider interact to execute Webservice

www.
w3.org/
TR/2004
NOTE-
ws-arch-
2004
0211
Fig.3.2

c.f.
pg.
V-43

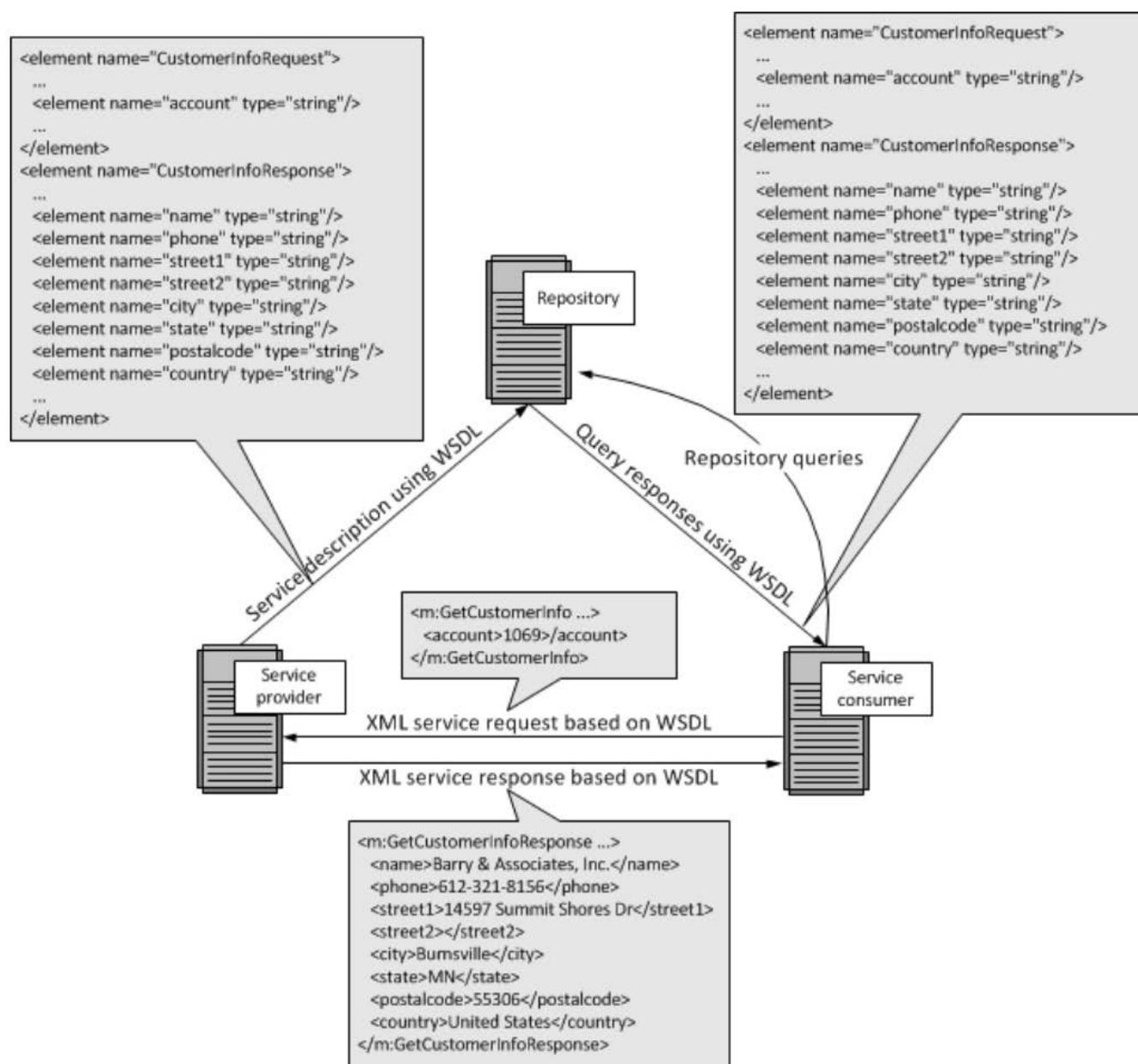
WS Reality: Participants, Protocols and Formats



www.service-architect.com/web-services/articles/web-services-explained.html

- ▷ How to find your service' address? Directory/Registry Service for service publishing and finding vs. 'known' address, e.g. (IP, port)
- ▷ C/S interaction using a Remote Procedure Call via SOAP

Example: Interaction Information in a WSDL file



Service-Oriented Architecture

Service-Oriented Architecture (SOA): more than Webservices

- Complete architectural paradigm for distributed systems
- Ranges from business processes to detailed implementation
- Extends to '*Service Landscapes*' or '*Service Eco Systems*'

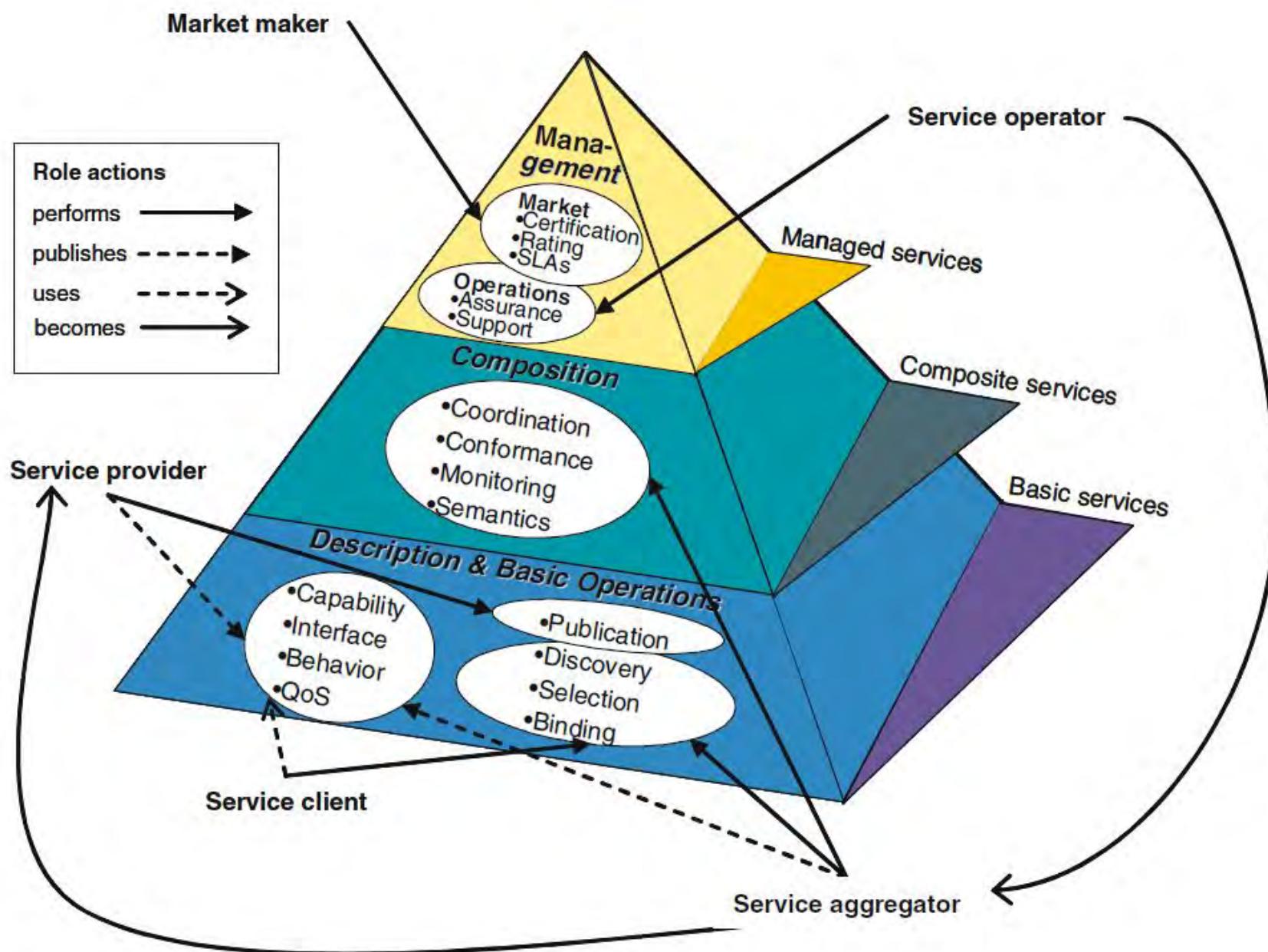
From the point of view of:	SOA is
Business executive and <u>business analyst</u>	A set of services that constitutes IT assets (capabilities) and can be used for building solutions and exposing them to customers and partners
Enterprise architect	A set of architectural principles and patterns addressing overall characteristics of solutions: modularity, encapsulation, loose coupling, separation of concerns, reuse, compositability, and so on
Project manager	A development approach supporting massive parallel development
Tester or quality assurance engineer	A way to modularize, and consequently simplify, overall system testing
Software developer	A programming model complete with standards, tools, and technologies, such as Web services

Note: Boris Lublinsky: Defining SOA as an architectural style.

www.ibm.com/developerworks/architecture/library/ar-soastyle/

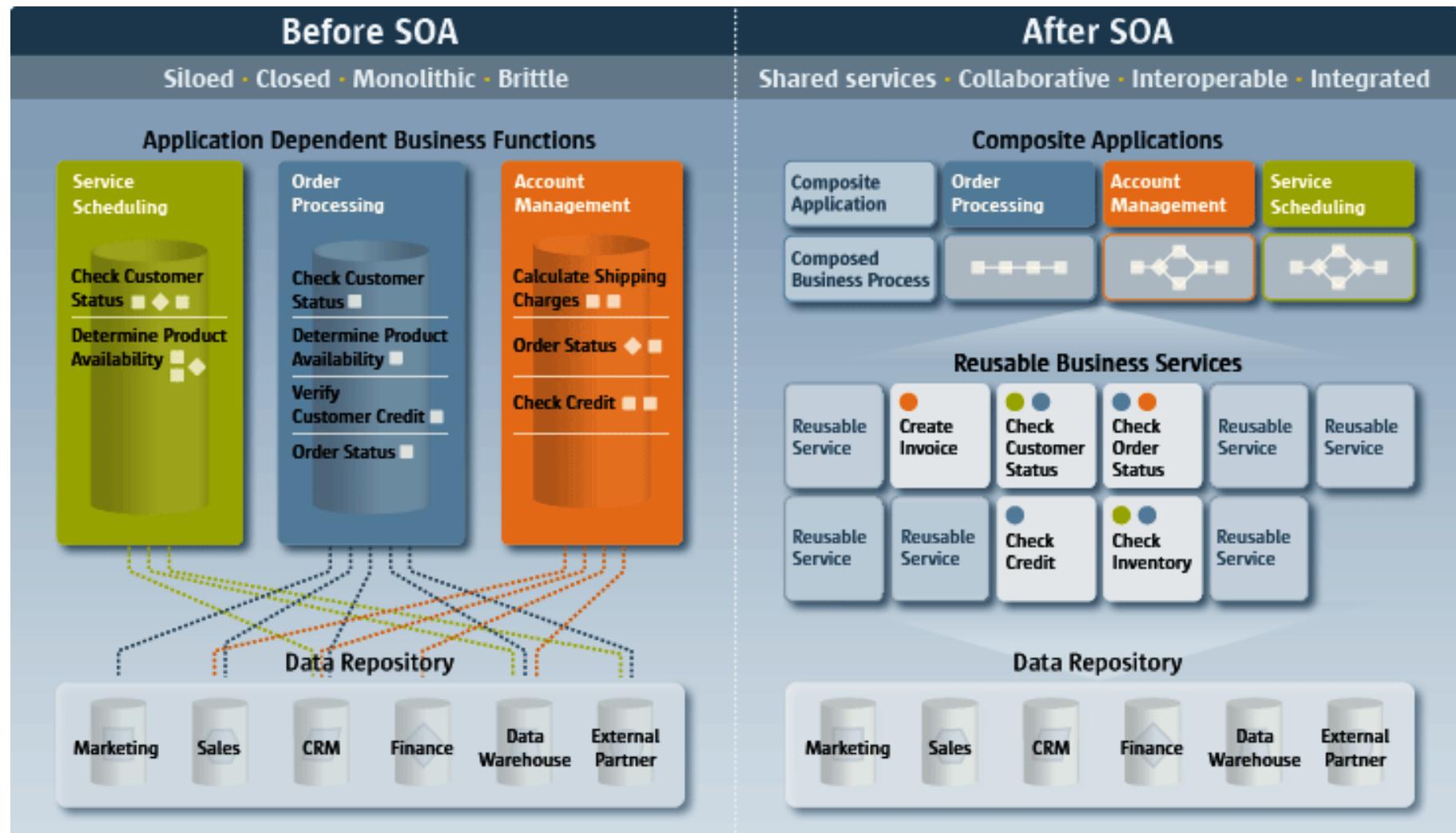
c.f.
vc
docs

A 'holistic' view on SOA



Vision – 1: Expected Impact: SOA-fication on Enterprise IT

www.
the
server
side.
de



Vision – 2: Service Landscapes and Eco Systems

'Service Eco Systems are electronic market places and emerge as a result of the shift toward service economies. The aim of service ecosystems is to trade services over the internet. The vision of service ecosystems is an evolution of service orientation and takes services from merely integration purposes to the next level by making them available as tradable products on service delivery platforms.'

c.f.
Barros,
Dumas:
The
Rise
of Web
Service
Ecosyste
in:
IT
Profes
sional,
8(5),
2006

SES infrastructure must provide support for service

1. **discovery** based on service descriptions
2. **selection** by comparing and evaluating descriptions
3. **contracting**, esp., *Service-Level-Agreements (SLAs)*
4. **consumption** by calling and running a service call
5. **monitoring** at runtime, esp., w.r.t. SLA fulfillment
6. **profiling** combines 4.-5. for further service evaluation

c.f.
Scheit
hauer,
Augustin
Wirtz:
Des
cribing
Services
for
Service
Eco
systems.
in:
ESBE
2008

Benefits of SOA vs. Status of Webservices?

- ▶ Holistic support from business level down to technological details
- ▶ Explicit integration of **contracts** into the core concepts
- ▶ Clean designed paradigm that
 - * demands explicitly defined **interfaces** and type definition
 - * demands explicitly defined **interaction protocols** up front
 - * distinguishes between specification and implementation
 - * de-couples program logic from transport issues

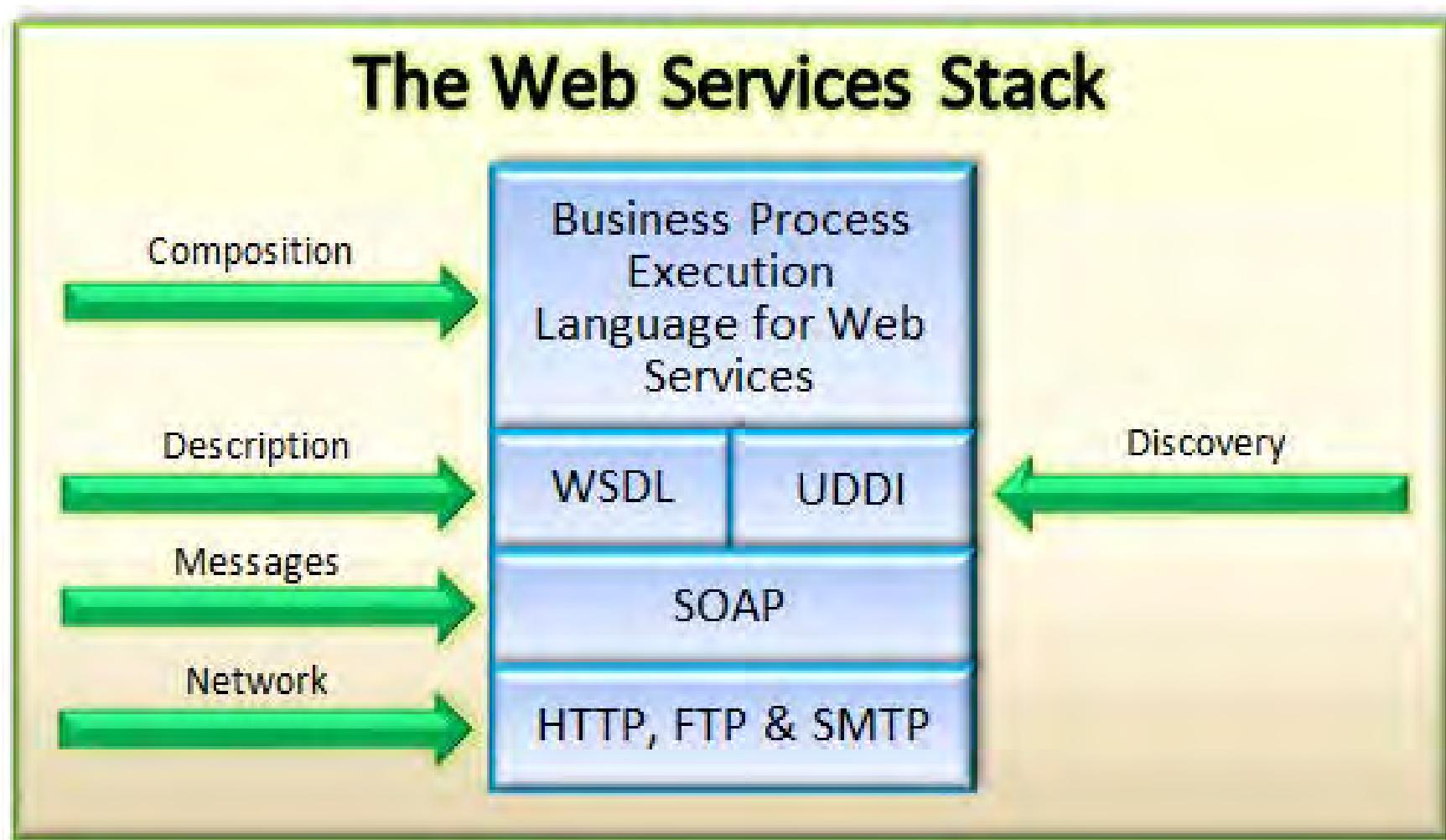
Status of Webservices and SOA in Practice

- ▶ SOA as an architectural style still important
- ▷ Webservice technology stack still in use, esp. enterprise integration
- ◀ Transport neutral text-based SOAP/XML is costly (heavy-weighted)
- ◀ WebService stack is complicated and not successfully standardized
- ◀ 'Holistic' approach and benefits are still hard to achieve
- ◀ Interface-based design via WSDL specification not 'mainstream'?

V.6. Interoperability and Standards

- ▶ **HTTP/1.1/HTTP/2** internet transport protocol (1999/2015)
more efficient interaction for webservers (pushing data) etc.
- ▶ **XML 1.1** (29.09.2006, <http://www.w3.org/TR/xml11/>)
 - self-describing, machine-processable documents
 - standardized formats for structured documents and schemata
 - transformation rules, sophisticated navigation, ...
- ▶ **SOAP 1.2** (27.04.2007, <http://www.w3.org/TR/soap/>)
Note: lot's of tools still work with version 1.1 today
 - message structure and XML mappings
 - **Nodes** and roles: Sender → Intermediary → Receiver
 - Interaction protocols and predefined '*msg exchange pattern*'
- ▶ **WSDL 1.2/2.0** (26.06.2007, <http://www.w3.org/TR/wsdl120/>)
Note: lot's of tools still work with version 1.1/1.2
 - defines data/msg types, interfaces, ops, mapping to XML

Webservice Stack – Overview: Lower Layers



project
raja.
com/
images/
pages/
Web
Services
Stack.
png

Webservice Standardization: A Lesson how to fail

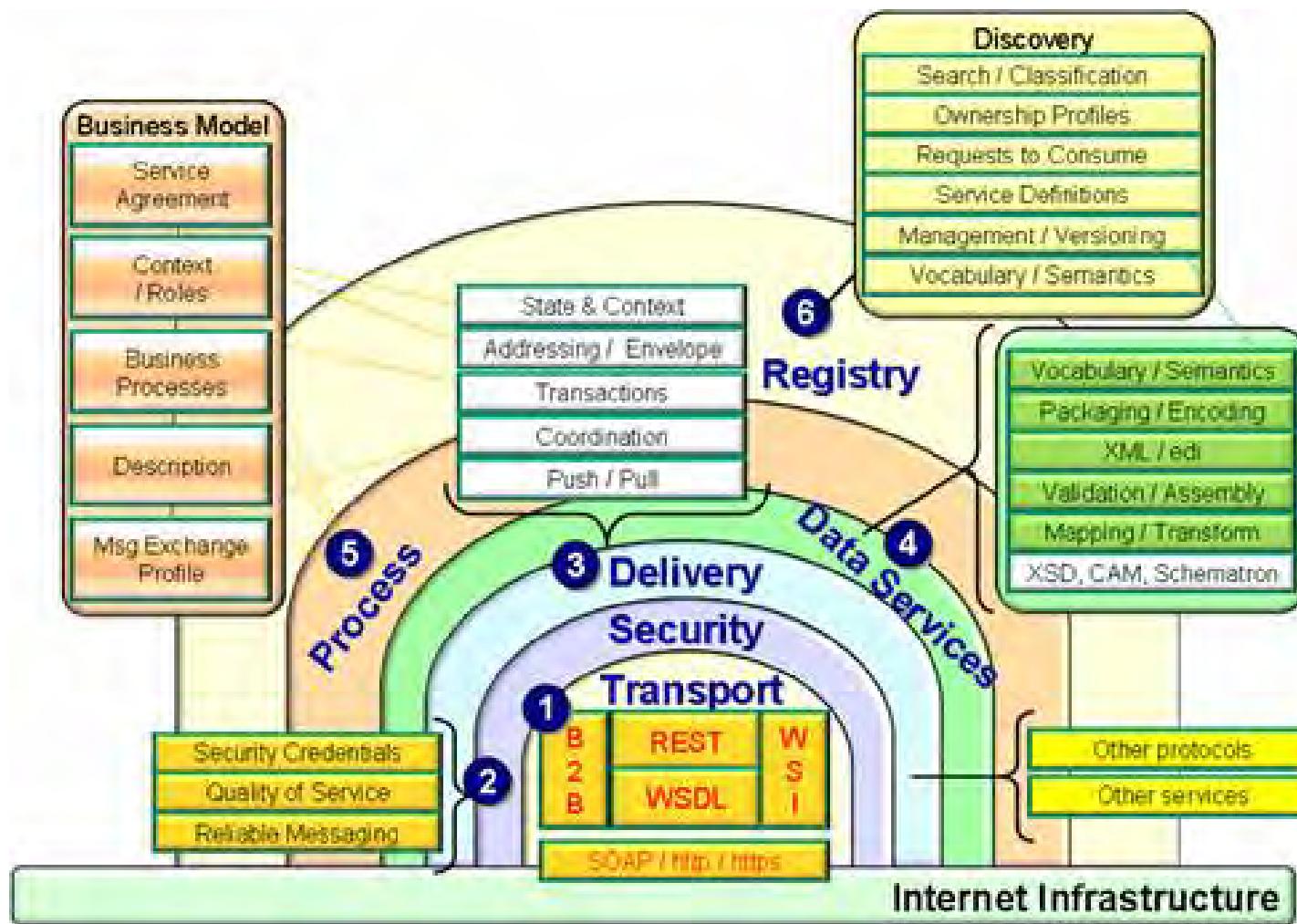


Web Service Interoperability Profiles 1.2/2.0

09.11.
2010

- ▷ standard compliance between standards of different organizations
- ▷ Additional rules eliminating standard inconsistencies etc.
- ▷ Use-Cases and test-suites for checking conformance
- ◀ Initiative was cancelled in 12/2017

Reasons for Standardization Failure: Complexity



- ▷ Big Picture: Handling lots of different architectural aspects . . .
- ◀ . . . requires lots of different inter-operability standards?

End
of
V.6

Summary: How to build Modern C/S Systems?

When to use which technology? No one-fits-all answer!

▷ WSDL-based Services

- * contract-based, high overhead, complex msg and environment
- * still ok for EAI in 'closed', 'statically planned' environments

► RESTful Services:

- * dynamic 'resources', overhead < WS, msg still 'clumsy text'
- * based on Web standards without too much a priori customization

► gRPC-based Services

- * contract-based, faster than REST due to binary msgs
- * more flexible interaction styles (esp. stream-handling)
- * higher requirements w.r.t. environment: HTTP/2, SSL, ...

First Humble gRPC \implies performance, inter-application, clouds

Guess: REST \implies inter-op., loose coupling, rapid change

Complex Standards - The Way of the World?



V.7.1 Appendix: Basic Standards – HTTP

HyperText Transport Protocol 1.1/2.0

- **Addressing** based on standardized rules

`http://<host> [:<port>] [abs_path [? <query>]]`

- **Data formats and encodings** for transmitting data

- * ASCII text and compressed sequences of bytes
- * Multipurpose Internet Mail Extensions: text/plain; image/gif

- Supported **interactions steps** and permitted formats

- * Request–Msgs allow for 9 pre-defined methods/formats
- * Response–Msgs: information (1xx), success (2xx), redirection (3xx), client errors (4xx), server errors (5xx)

- **Connection Model:** (*'philosophy' vs. 'efficiency'*)

- ▷ HTTP 1.0 connections-less: new connection for each message
- ▷ HTTP 1.1/2.0: persistent connection; better performance

HTTP Command 'Verbs'

GET

The `GET` method requests a representation of the specified resource. Requests using `GET` should only retrieve data.

HEAD

The `HEAD` method asks for a response identical to that of a `GET` request, but without the response body.

POST

The `POST` method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

PUT

The `PUT` method replaces all current representations of the target resource with the request payload.

DELETE

The `DELETE` method deletes the specified resource.

CONNECT

The `CONNECT` method establishes a tunnel to the server identified by the target resource.

OPTIONS

The `OPTIONS` method is used to describe the communication options for the target resource.

TRACE

The `TRACE` method performs a message loop-back test along the path to the target resource.

PATCH

The `PATCH` method is used to apply partial modifications to a resource.

HTTP – Properties of Command 'Verbs'

Fig.:
en.wikipedia.org/wiki/HyperText_Transfer_Protocol#Request_methods

HTTP method	RFC	Request has Body	Response has Body	Safe	Idempotent	Cacheable
GET	RFC 7231	Optional	Yes	Yes	Yes	Yes
HEAD	RFC 7231	Optional	No	Yes	Yes	Yes
POST	RFC 7231	Yes	Yes	No	No	Yes
PUT	RFC 7231	Yes	Yes	No	Yes	No
DELETE	RFC 7231	Optional	Yes	No	Yes	No
CONNECT	RFC 7231	Optional	Yes	No	No	No
OPTIONS	RFC 7231	Optional	Yes	Yes	Yes	No
TRACE	RFC 7231	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	No

- **Safe:** method is (by definition) guaranteed to have no side-effects on server side, i.e. reads or 'retrieval' only
- **Idempotent:** multiple *identical* requests have only one effect, i.e. repetition does not change server side
- **Cacheable:** if still valid, avoid re-transmit etc. (detailed control)

Main Issues: HTTP/2 ↔ HTTP/1.1

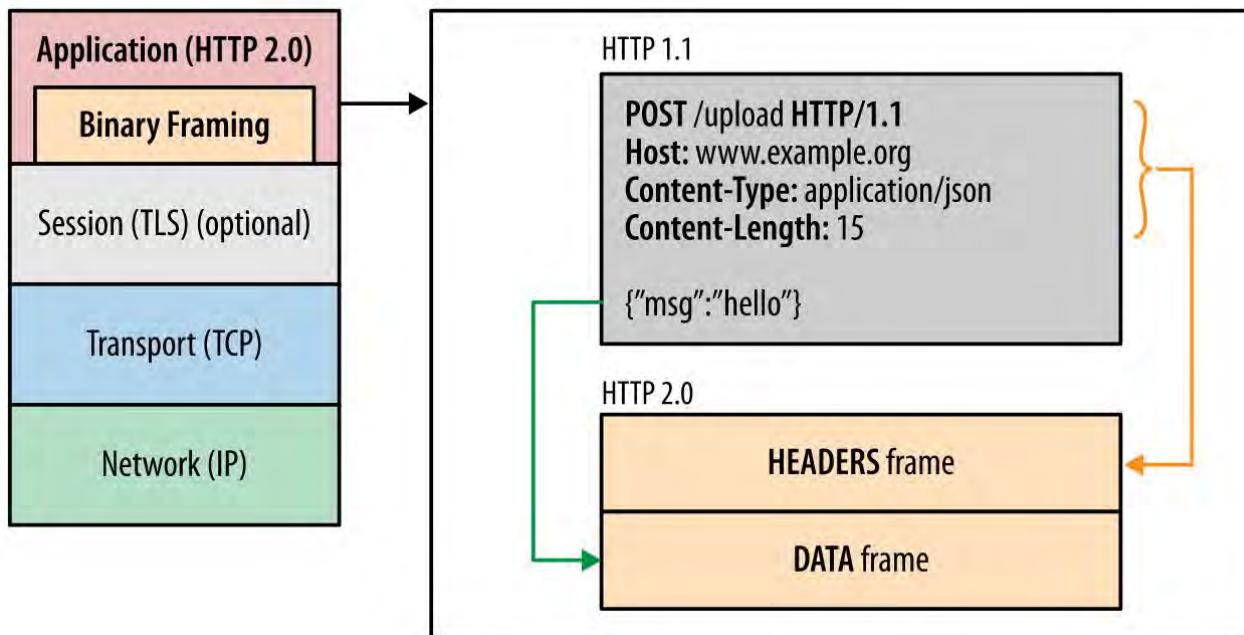


Fig.:
devel
opers.
google.
com/
web/
funda
mentals/
perf
ormance/
http2

- ▶ plain text vs. binary encoding
- ▶ multiplexing multiple streams on a single TCP connection in parallel vs. multiple TCP connections in case of multiple streams
- ▶ security overhead using TLS/SSL much less for single connection
- ▶ streams may be prioritized based on dependencies
- ▶ advanced *flow control* in order to avoid buffer overflows

Remark: Details also important for gRPC and esp. REST

HTTP/2 – Example Request/Response using Get

Connection

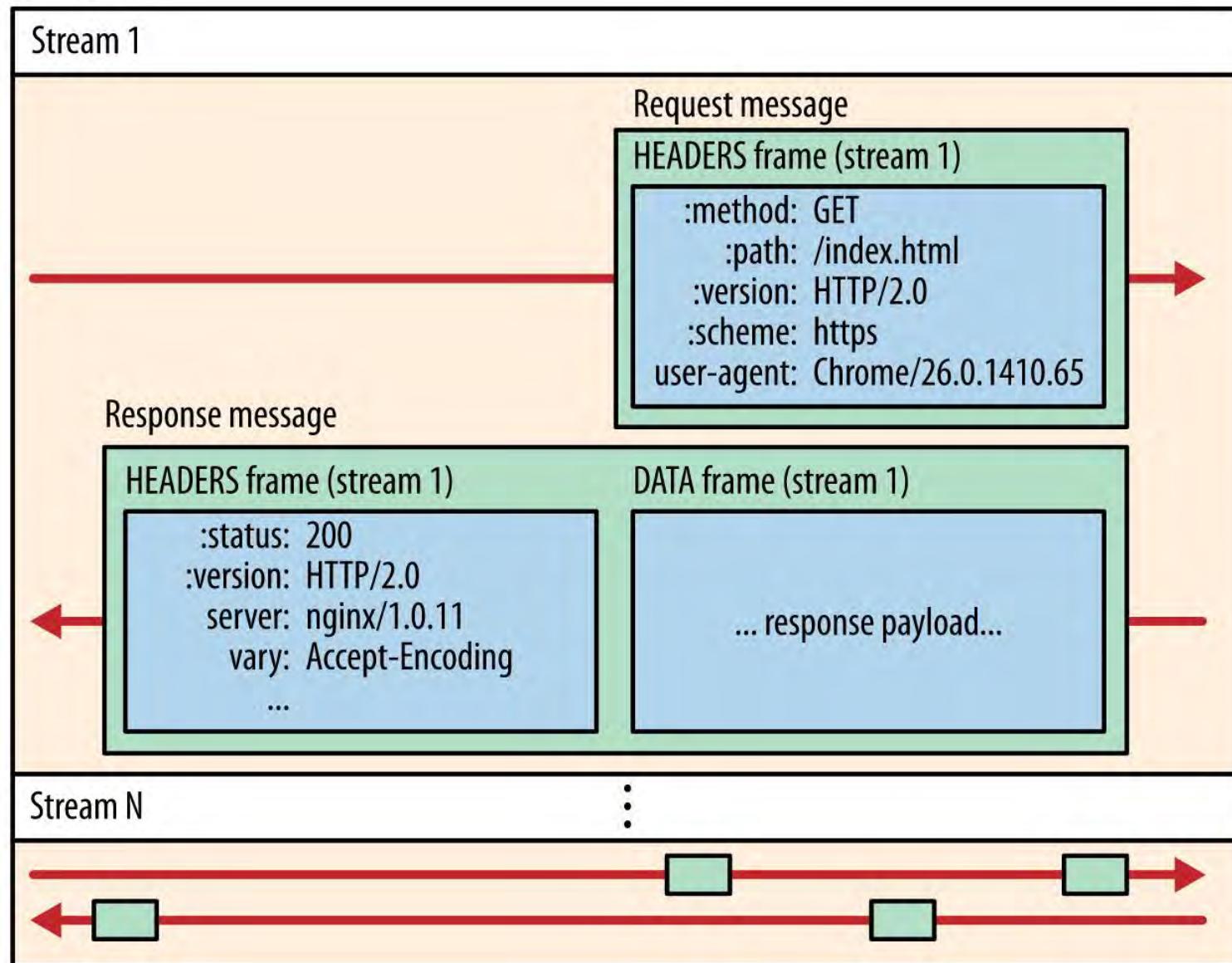


Fig.:
devel
opers.
google.
com/
web/
funda
mentals/
perf
ormance/
http2

V.7.2 Basic Standards – XML

eXtensible Markup Language: text-based language

► Basic Concepts . . .

- self-describing document structures holding meta-information
- separation of structure, content and form

. . . ease

- ▷ consistency checks: document vs. structural description
- ▷ automatic processing (parsing) based on format definitions
- ▷ handling the same content in different representations

► Background: long tradition of **hypertext** documents

- extends HTML (1989) by self-defined **Tags**
- Restricts the SGML ISO standard (1986)
- lots of additional standards and tools for XML processing

XML 1.0/1 standard defined by **W3C** still in use

(XML 1.0 1st 1998/4th 2006; XML 1.1 29.09.06)

Nelson
1950

XML Basis for structured messages

- ▶ **XML Vocabulary:** predefined set of elements/structures
 - nested complex structures via open/close markups \implies **Tree(s)**
 - **Structure** may be specified using different techniques:
 - ◀ **Document Type Definitions** (DTDs) similar to grammars and restricted to structure definitions (*out-dated*)
 - ▶ **XML Schemata:** structure specification in XML; supports type definitions, (nested) name spaces; extensibility
 \implies **Documents can be evaluated with respect to:**
 - * **well-formed-ness:** comprise to general **XML** specification
 - * **validity:** document matches predefined definition or schema
- ▶ Powerful **XML processing tools:**
 - **Simple API for XML/Doc Obj Model/Streaming API for XML**
 - JAXB: un/marshall Objects \leftrightarrow XML and JAX-WS
Annotations in Java-Code define attributes, . . . , webservices
 - **Transformation: eXtensible Stylesheet Language/Transformation**

c.f.
pg.
V-65c.f.
pg.
V-66DSG-
AJP-B

XML File complying to an XML schema definition

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

c.f.
www.
w3
schools.
com/
schema/
schema_
example
.asp

The corresponding XML schema definition

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="item" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="note" type="xs:string" minOccurs="0"/>
      <xs:element name="quantity" type="xs:positiveInteger"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="orderid" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

</xs:schema>
```

c.f.
www.
w3
schools.
com/
schema/
schema_
example
.asp

End
V.7

IT sucks . . . for Management and Financing

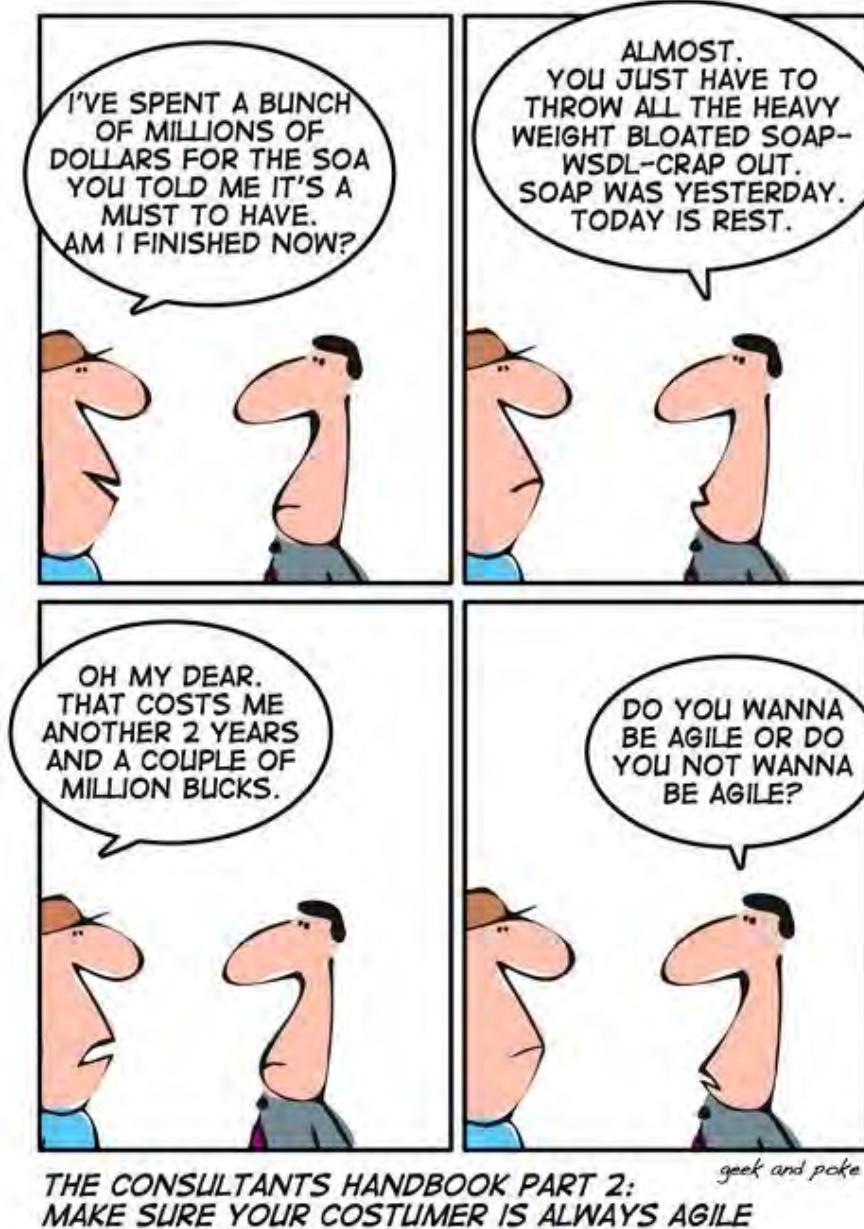


Fig.:
cdn.
crunchify
com/
wp-
content/
uploads/
2013/07/
REST-
Over-
SOAP-
Protocol
jpg

VI. Distributed Algorithms

Goal: Simulate useful properties of centralistic systems in more or less poorly ordered distributed systems.

c.f.
I-29

[Shingal et al. 1994]: *A distributed system consists of autonomous computers without any shared memory and without a global clock. Computers communicate using message-passing on a communication network with arbitrary delays.* ♦

⇒ **Suitable techniques do not use 'global knowledge'**

Essential: Assumptions w.r.t system model should be clear!

- Communication style:
 - * point-to-point between single processes
 - * broadcast to *all* processes (of a process group)
- Communication guarantees: lost messages, message order, ...
- Which node failures are acceptable for an algorithm?

Reason: *Algorithms do not work without these assumptions!*

Overview: Basic Distributed Algorithms

1. Time and Causality
2. Applications of logical time to message ordering
3. Applications of Time to Distributed Mutual Exclusion and Fairness
4. Consistent global snapshots and checkpointing
5. Determination of 'global' system states: Termination, Deadlocks
6. Distributed Coordination: Leader Election

⇒ **Characteristic techniques for solving distributed problems**

Note: *There are many algorithms/aspects we do not discuss here!*

- * Byzantine Agreement details ⇒ **MSc literature**
- * 2/3 Phase Commit protocols/transactions ⇒ **MSc literature**
- * Algorithms dedicated to unstructured Peer-to-Peer systems
- * Distributed Ledger Algorithms, Bitcoins, Etherium, ...

VI.1 Time and Causality

Properties of 'real' Time:

linear order: total order relation

Past (linear)/Present/Future (branching)

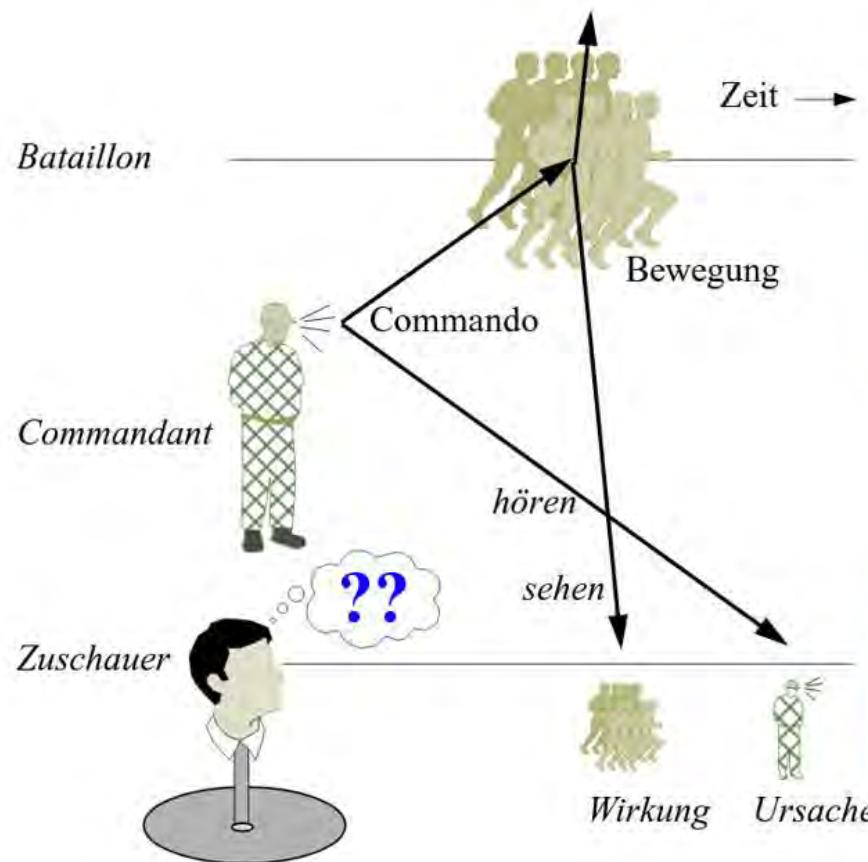
continuous; dense \implies **real** numbers as suitable basic model?

Fact: 'Full real time' is not always required

- ▶ traffic lights; backups based on time of day, . . . (points in time)
 - ▶ Accounting of resource usage (period of time)
 - ◀ Decision criteria and priorities for:
 - Resource allocation, e.g., CPU scheduling
 - Election algorithms ('Eldest')
 - ◀ Version control: text; source-code, e.g., build-organization
 - ◀ Synchronization: a_2 before a_1 ; not at the same time

\implies Typically not all properties of real time are really needed.

Example: Observations and Causality



Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, ehe er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der Causalzusammenhänge weiß er, daß die Bewegungen die Wirkung des gehörten Commandos sind, dieses also jenen objectiv vorangehen muß, und er wird sich sofort der Täuschung bewußt, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.

Christoph von Sigwart (1830-1904) Logik (1889)

Observer: Far away on a hill looking down on military exercising suddenly sees soldiers starting to run and shortly **after** that he also hears the commander who shouts 'RUN'
→ Effect is observable before the cause?

⇒ **Observer time line contradicts rule of cause and effect**

1. $P_{commander}$ sends msg_1 to soldiers (PS)
2. PS reacts, e.g. by running, which is observed by P_{observ} as msg_2
3. P_{observ} hears command msg_1 ⇒ Effect observed before cause?

Rule of Causality: Cause → Effect

- ◀ Observer observes effect before cause
- ◀ **Alibi principle:** Speed of light and the impossibility to be at two distant places at the same time
- ◀ Messages: send is always before the corresponding receive
- ◀ Time paradox in 'backward' time travel
Example: Kill the inventor of the time machine... ?

Note: internal relative order is important, not relation to real time.

- ==> **Simple linear orders are sufficient for most CS applications:**
- ▷ Position in FIFO-queue simulates relative arrival time
 - ▷ 'Age' of a process simulated by strict monotonous numbering
 - ▷ Version control for programs based on ordered Dewey-Notation
 - ▷ Mutual exclusion and fairness based on request ordering

==> **always use the most efficient but sufficient model**

Computer Systems: Real vs. Logical Time

1. 'Real' Time: Reference to external 'world' and time

Example: systems that control machines or traffic lights

Physical Clock \approx acceptable deviation from 'real time'

- ▶ internal physical clocks (quartz crystal oscillation)
- ▶ clock alignment within local/global networks
- ▶ external points of reference, e.g. **external time server**
 \implies costly in distributed systems if highly accurate

NTP

2. Logical Time: internal, relative causality-based order

\implies reference to real time not needed

Logical Clock \approx internally consistent, no reference to real time

- ▶ integer counter for logical steps (*ticks*)
- ▶ compare and align in the case of message exchange
- ▶ initial time is globally 0 via **reset**
 \implies cheap and efficient in almost all distributed systems

VI.1.1 Physical Clocks (= real time ?)

- ◀ **Astronomical Time:** $\frac{\text{solar day}}{24*60*60} = \frac{\text{solar day}}{86400} \approx \text{solar second}$
 Earth/Sun rotation constant, but earth rotation slows down
 \implies solar days/seconds become 'longer' \implies mean value GMT
- ◀ **Atomic time (TAI):** 01.01.1958
 1 solar second = 9.192.631.770 Caesium 133 transitions
- ▶ **Universal Coordinated Time (UTC):** compensates for Drift
 currently 3 millisec/day \implies TAI \oplus leap seconds (approx. 0.9 sec)
 \implies lots of problems in IT due to 'repeated second'
- ▶ **Distribution:** accuracy of source \oplus transfer time !
 - Radio-based signals, e.g., DCF77 $\approx \pm 1 \text{ msec} \oplus \pm 10 \text{ msec}$
 - Satellites, e.g., GPS or GNSS (multiple satellites)
 $\approx 10 \text{ nanosec} \rightarrow 1 \text{ microsec}$

Caution: 1 nanosec \approx 750 instructions in a 749.070 MIPS processor

Effects of Errors: www.theregister.com/2016/02/03/decommissioned_satellite_software_knocks_out_gps/

Savage
CACM
09/2015
see vc

10^{-9}
 10^{-6}

AMD/
Rizen 9
3950X
(2019)

Hardware Basis: Timer Chip

- Quartz ticks as source for time steps; counter decrement; reset;
- Software clock: **clock ticks** counted via interrupts based on counter
- Deviation ≈ 1 second in approx. $11\frac{1}{2}$ days
clocks diverge in opposite 'direction' \implies approx. 2 seconds max

Algorithms: consistent initialization *(How?)*
keep deviation below threshold by msg exchange

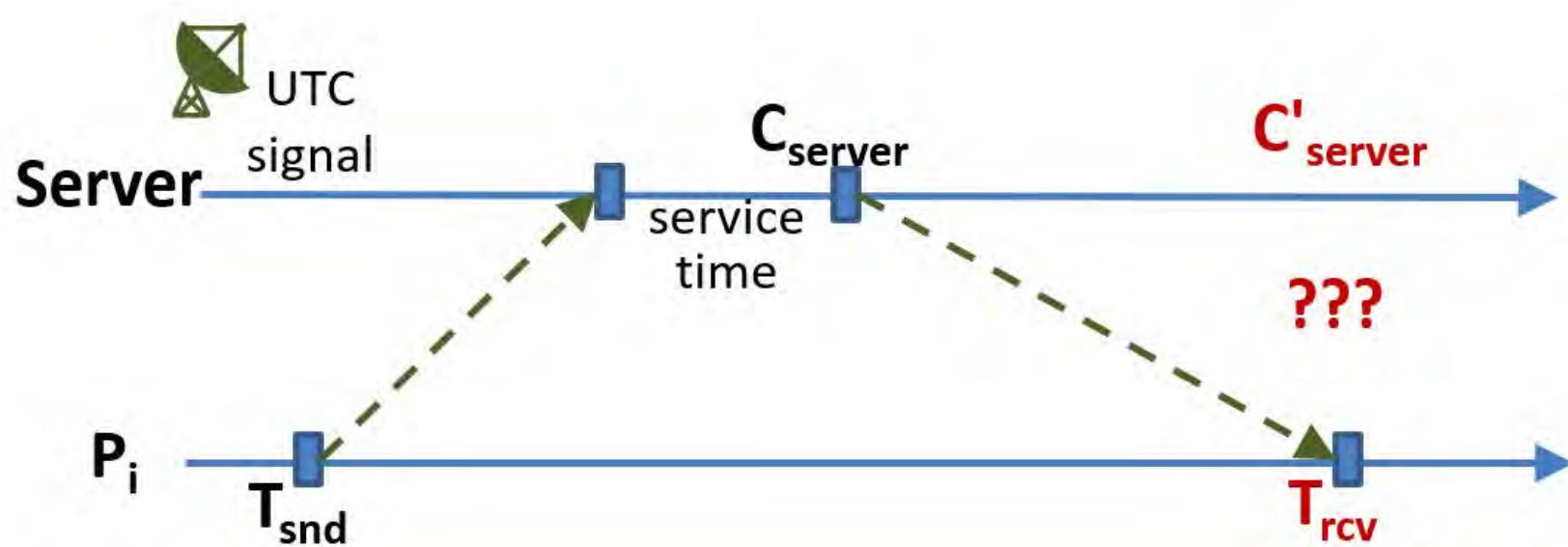
Problems: (in all algorithms)

- ◀ **Transfer time:** Source \longrightarrow Destination
different routes, number of hops, load levels etc.
 \implies measure transfer times and use them in calculations
- ▶ **Suitable methods for adjustment:**
never put clock back; small adjustment instead of abrupt change
 \implies slowdown/accelerate counter for local clock
Expl.: 100 IR/sec \implies 9/11 ms instead of 10 ms as compensation

Approximation of real time – 1

1. Passive Time Server

- (a) Time-Server holds 'external real' time in clock C
- (b) P_i sends Request at T_{snd} and receives C_{server} at T_{rcv}
- (c) $C_{server}^{rcv} \approx C_{server} + \frac{T_{rcv}-T_{snd}}{2}$ (plus service time on server)



- ◀ Same time for both messages? (e.g. uni-directional ring)
- ◀ Server and network may have varying loads

Approximation of real time– 2

2. Active Time Server, e.g., Berkeley UNIX

- (a) initially: correct setting of server clock
- (b) **Protocol:** in fixed intervals
 - i. Server sends T_{server} to all (local) network nodes
 - ii. $\forall P_i \in PS$: compute and send $\Delta_i := T_i - T_{server}$ to server by respecting transfer times
 - iii. mean value of divergences is used to correct local clocks

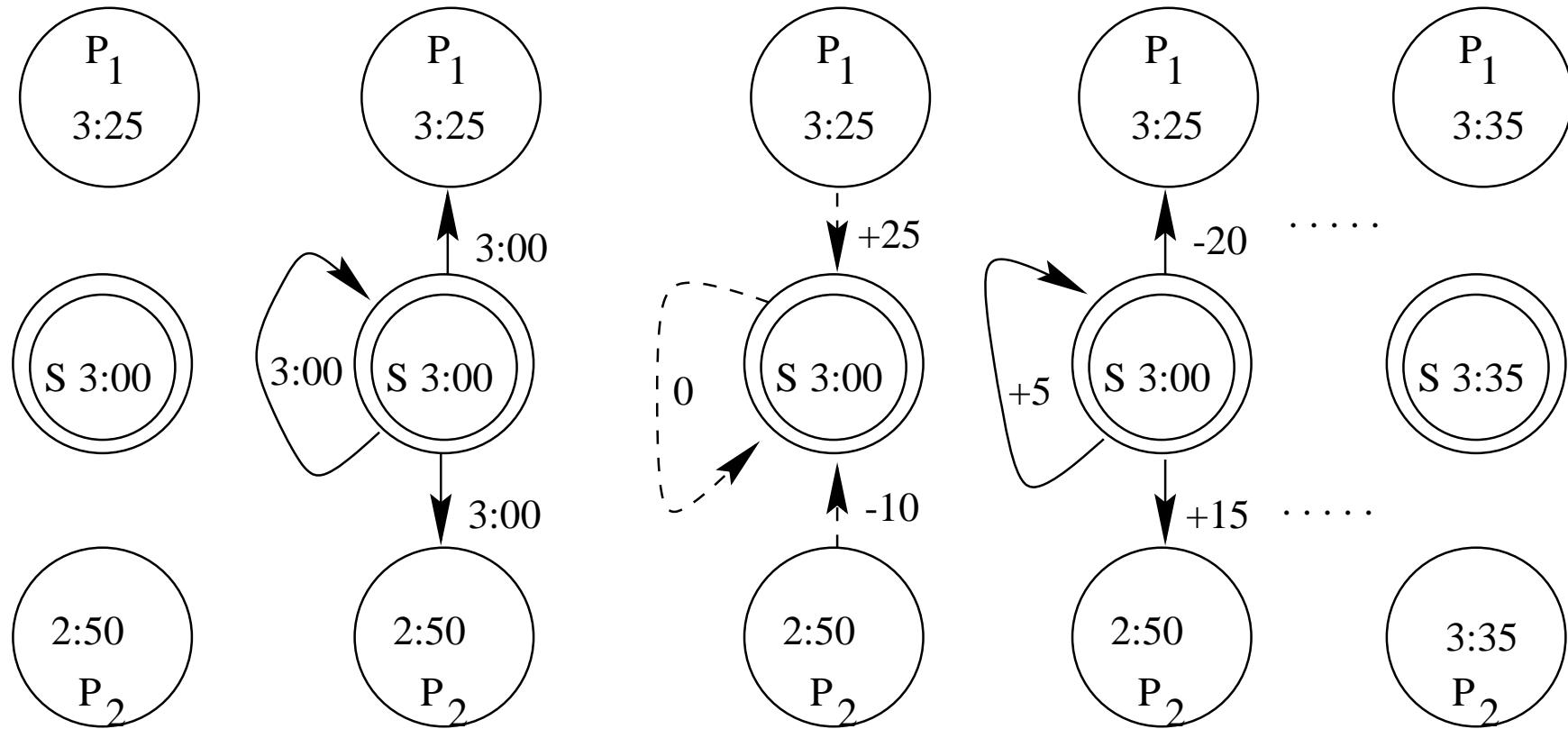
3. Distributed Adjustment (active)

- Re-Synchronize in fixed (local) intervals
- broadcast local time for calibration to all nodes
- compute local mean values for correcting the local clock
 - requires minimal number of answers; ignores extreme outliers

4. Multiple external clock sources (in different nodes)

Gusella et al. 1989
man timed c.f. pg. VI-11

Approximation of real time – 3: active Timeserver

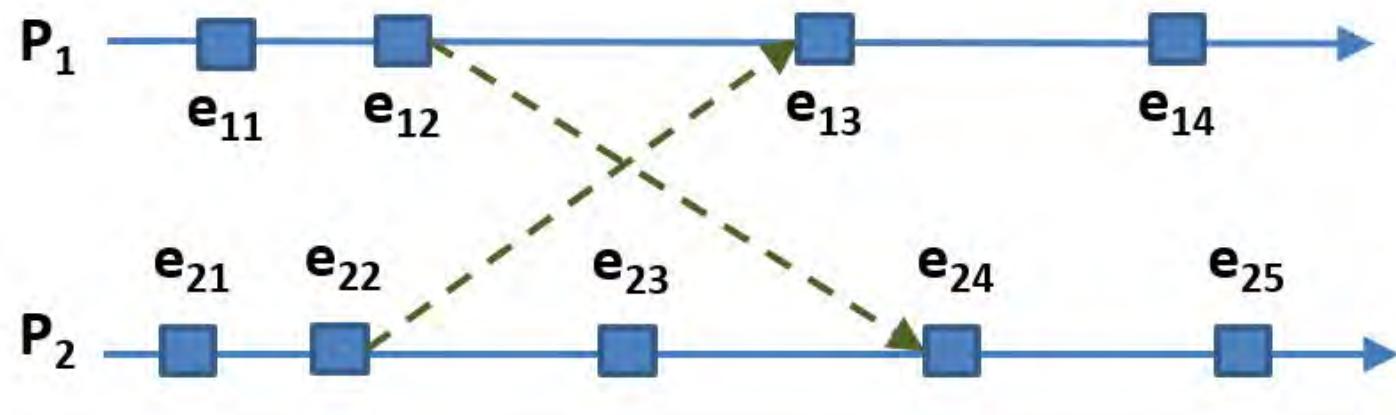


Algorithm: compute 'correct' time via arithmetic mean
 send computed deviations
 correction via slowdown/acceleration of local counter

VI.1.2 Logical Clocks - Virtual Time

Cause – Effect relation among

- ▷ Actions or **Events** of the same process
- ▷ Events of different processes due to sending/receiving messages
- ▷ Transitivity



Induced order: (causality relation)

- **local:** $e_{11} \xrightarrow{\sqsubseteq} e_{12} \xrightarrow{\sqsubseteq} e_{13} \xrightarrow{\sqsubseteq} e_{14}$
and $e_{21} \xrightarrow{\sqsubseteq} e_{22} \xrightarrow{\sqsubseteq} e_{23} \xrightarrow{\sqsubseteq} e_{24} \xrightarrow{\sqsubseteq} e_{25}$
- **Communication:** $e_{12} \xrightarrow{\sqsubseteq} e_{24}$ and $e_{22} \xrightarrow{\sqsubseteq} e_{13}$
- **Transitivity**, e.g., $e_{21} \xrightarrow{\sqsubseteq} e_{14}$
- **concurrent**, e.g., $(e_{11}, e_{21}), (e_{23}, e_{12})$

c.f.
III.1

Definition VI.1: (happened-before Relation)

Let a, b, c be events of a set of events E and P_i, P_j processes.

The relation $a \sqsupseteq b$ holds : \iff

1. $a, b \in P_i$ and $a \sqsubset b$ in P_i , or (sequential order)
2. $a \approx \text{snd}(\text{msg}, P_j)$ in P_i and $b \approx \text{rcv}(\text{msg}, P_i)$ in P_j , or
3. $a \sqsupseteq c$ and $c \sqsupseteq b \implies a \sqsupseteq b$

The events a and b are

- in a **causality relation** : $\iff (a \sqsupseteq b) \vee (b \sqsupseteq a)$
- **concurrent** : $\iff \neg(a \sqsupseteq b) \wedge \neg(b \sqsupseteq a)$



P and P' **without** communication \implies all events are concurrent

- no problem for program logic as there is no interaction
- ◀ no global time among all processes of PS achievable

Lamport's logical clocks

CACM
21(7),
1978

Idea: global time $C : (E, \sqsubseteq) \rightarrow (\mathbb{N}_0, <)$ for entire PS
 respects \sqsubseteq **without extra messages**

- $\forall P_i \in PS$ exists a **local counter** C_i initial 0
- $\forall a \in P_i$ exists a local, unique **time stamp** $C_i(a)$
 derived from local clock value C_i when executing action a in P_i

Two Rules for global time C ...

C1: $\forall P_i \in PS: a \sqsubset_{PS} b$ local in $P_i \implies C_i(a) < C_i(b)$
 Time respects local, internal order in each single process

C2: $\forall (a_{snd}, b_{rcv})$ where $a_{snd} \approx \text{snd}(\text{msg}, P_j)$ in P_i and
 $b_{rcv} \approx \text{rcv}(\text{msg}, P_i)$ in P_j
 $\implies C_i(a) < C_j(b)$

Time respects causality between corresponding send/receive

... **ensures:** $a \xrightarrow{\sqsubseteq} b \implies C(a) < C(b)$

Lamport Clocks – Implementation of C1 and C2

IR1: Increment local clock C_i **before** each new action

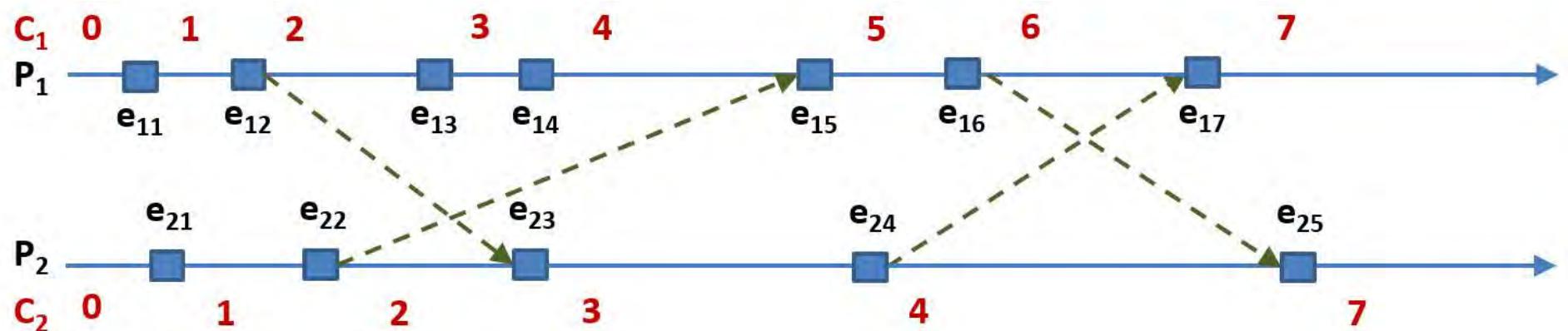
$\forall P_i \in PS \ \forall a \in P_i$ assign $C_i := C_i + d$ where ($d > 0$)
 before executing action a
 i.e. $a \sqsubset_{PS} b \implies C_i(b) = C_i(a) + d \implies C_i(a) < C_i(b)$

IR2: Propagate local time information with each message

- ▶ Let $a_{snd} \approx \text{snd}(\text{msg}, P_j)$ in P_i
 1. C_i is incremented locally to $C_i(a)$ in P_i
 2. a_{snd} is **extended** by $\text{snd}(\text{msg}, t_{msg}, P_j)$ where $t_{msg} = C_i(a)$
- ▶ Let $b_{rcv} \approx \text{rcv}(\text{msg}, t_{msg}, P_i)$ in P_j
 1. C_j is incremented locally to C_{temp} in P_j
 2. $C_j := \text{MAX}(C_{temp}, t_{msg} + d)$ where $d > 0$ (transfer time)
 i.e. $C_i(a_{snd}) = t_{msg} < t_{msg} + d \leq C_j(b_{rcv})$

Lamport Clocks – Example

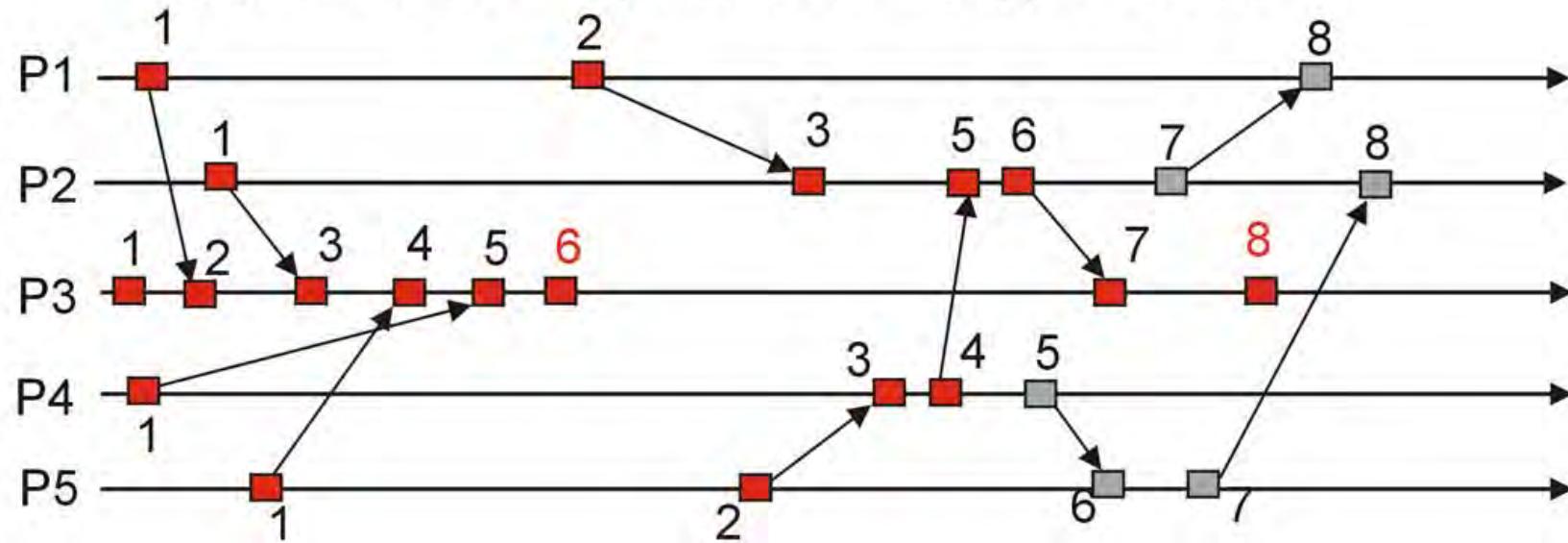
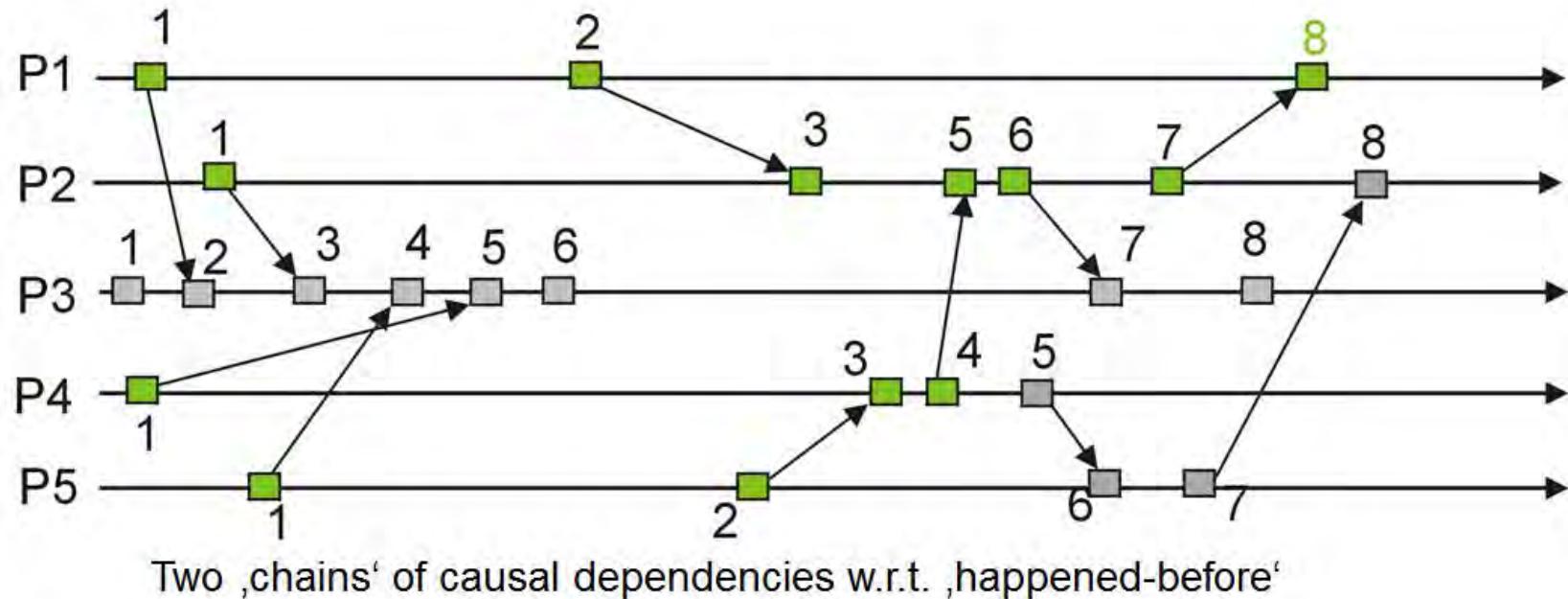
- ▷ $e_{12} \xrightarrow{E} e_{23}$: $\text{Max}(\overbrace{2+1}^{rcv}, \overbrace{2+1}^{t_{msg}+d}) = 3$
- ▷ $e_{16} \xrightarrow{E} e_{25}$: $\text{Max}(\overbrace{4+1}^{rcv}, \overbrace{6+1}^{t_{msg}+d}) = 7$



Observations:

- ◀ result is a partial order, e.g., (e_{14}, e_{23}) are concurrent
- ◀ $P_i \neq P_j$ use same clock value for different events, e.g., e_{17}/e_{25}
- ◀ concurrent events may have different clock values
Example: (e_{23}, e_{16}) concurrent, but $C(e_{23}) = 3 < C(e_{16}) = 6$

Expl.: 'History' of selected Events in the same PS



How to define a total order among events?

- ◀ Mapping $C: (E, \sqsubseteq) \rightarrow (\mathbb{N}_0, <)$ is not *injective*
- ▶ Based on a unique numbering scheme for all processes $|PS|$:
 $a \xrightarrow{\text{total}} b \iff C_i(a) < C_j(b) \text{ or } (C_i(a) = C_j(b)) \wedge i < j$
- ▶ new mapping from $(E, \xrightarrow{\text{total}}) \rightarrow (\mathbb{N}_0 \times \mathbb{N}_0, <)$ is injective
- ▶ lexicographical order based on Lamport time and process indices
 \implies **mapping** $(E, \xrightarrow{\text{total}}) \rightarrow (\mathbb{N}_0, <)$ is injective!

Problem: C does not uniquely denote causality!

$$C(a) < C(b) \implies \neg(b \xrightarrow{\sqsubseteq} a) \quad (\text{not } \iff)$$

- ◀ Increment may be caused locally or by send/receive?
- ◀ Most recent information only w.r.t. sender and receiver

Loss of structural information: $\xrightarrow{\sqsubseteq} \mapsto <$ and $\xrightarrow{\sqsupseteq} \mapsto >$
but: $\parallel \mapsto \{<, =, >\}$

Extending Lamport Time to Vector Time

Mat-
tern
1987
Fidge
1988

Idea: Propagate more detailed and also indirect information

- ▶ $|PS| = n \implies \forall P_i \in PS \quad \vec{C}_i = \langle C_i[1], \dots, C_i[n] \rangle$
- ▶ $\forall P_i \in PS$ is \vec{C}_i initialized by $\vec{0}$
- ▶ $(i \neq j) \implies \vec{C}_i$ and \vec{C}_j almost always **different**
 - $C_i[i] \approx$ local time C_i in P_i
 - $C_i[j]$ where $(i \neq j) \approx$ **most recent** information in P_i about the local clock value of P_j
 - * a is the most recent action in P_j where $a \sqsubseteq^* b$ holds and
 - * b is the most recent action in P_i $\implies C_i[j] = C_j(a) + \Delta$ where $\Delta > 0$
- ▶ **Update:** Messages in PS are extended by so-called **time vectors**
 - \implies only moderate additional overhead w.r.t. Lamport time
 - Information dissemination much faster
 - Processes **without** interaction are not ordered (as before)

Implementing C1 and C2 using Vectors:

c.f.
pg.
VI-14

IR1: Increment the local clock C_i before each new action

$$\forall P_i \in PS \quad \forall a \in P_i \text{ assign } C_i[i] := C_i[i] + d \text{ where } (d > 0)$$

$$a \sqsubset_{PS} b \implies C_i[i](b) = C_i[i](a) + d \implies C_i[i](a) < C_i[i](b)$$

IR2: Propagate time vector with each message

► Let $a_{snd} \approx \text{snd}(\text{msg}, P_j)$ in P_i

1. $C_i[i]$ is incremented locally to $C_i[i](a)$ in P_i

2. a_{snd} is extended by $\overrightarrow{\text{snd}}(\text{msg}, t_{msg}, P_j)$ where $t_{msg} = \overrightarrow{C_i}$

► Let $b_{rcv} \approx \text{rcv}(\text{msg}, \overrightarrow{t_{msg}}, P_i)$ in P_j

1. $C_j[j]$ is incremented locally in P_j

2. $\forall k \in [1 : n] \quad C_j[k] := \mathbf{MAX}(C_j[k], t_{msg}[k])$

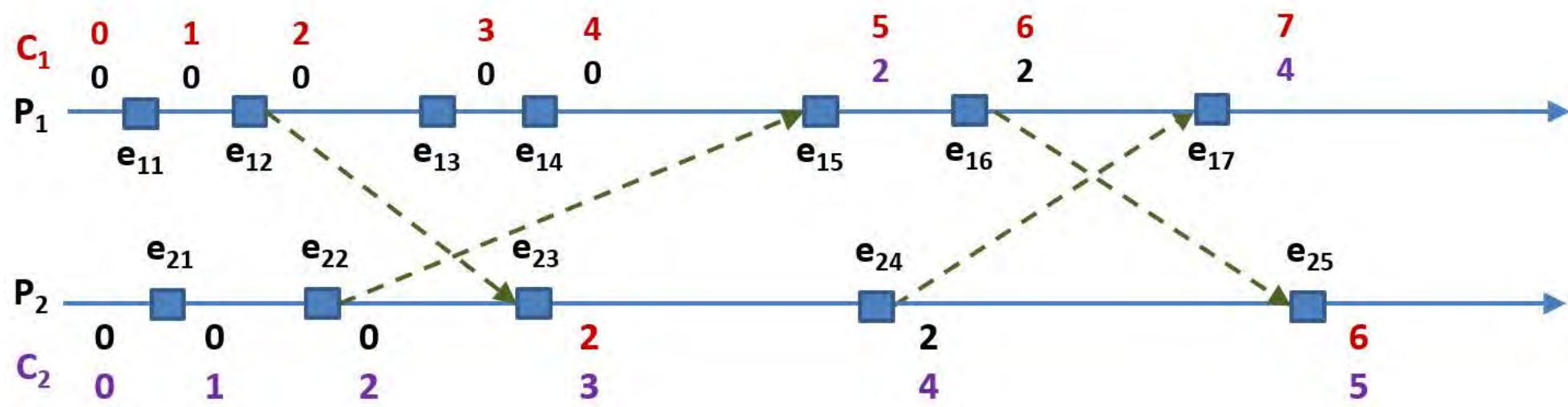
i.e. corresponding elements of vector $\overrightarrow{C_i}(a_{snd}) \leq \overrightarrow{C_j}(b_{rcv})$

Predicate: $\forall i \in [1 : n] \quad \forall j \in [1 : n] \text{ holds } C_i[i] \geq C_j[i]$

local time in P_i always more recent than its approximation in P_j .

Example: Vector Clocks – 1

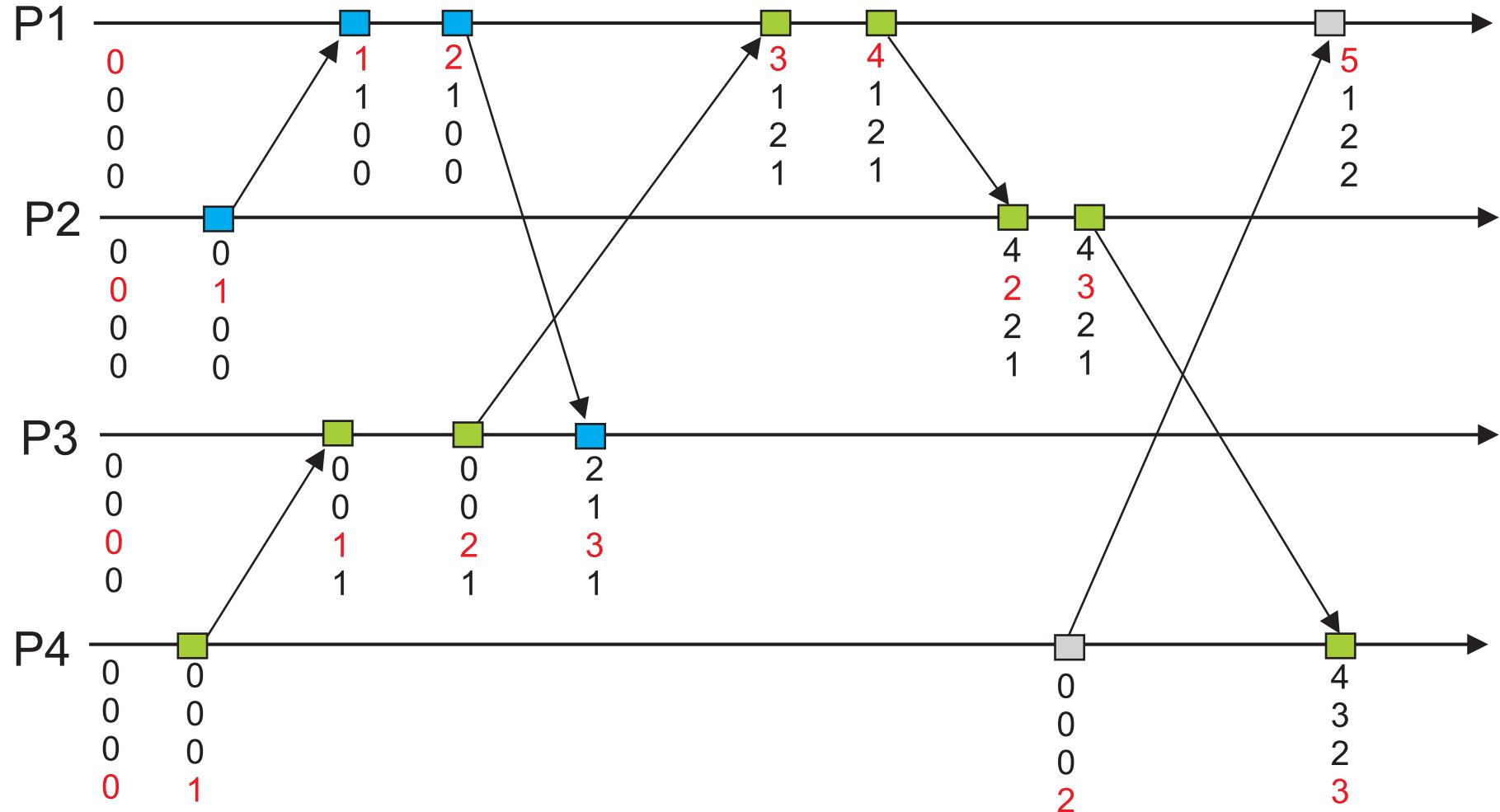
- $e_{16} \xrightarrow{\sqsubseteq} e_{25}$ and $C(e_{16}) = < 6, 2 > < < 6, 5 > = C(e_{25})$
- (e_{23}, e_{16}) not ordered and
 $C(e_{23}) = < 2, 3 >$ concurrent $< 6, 2 > = C(e_{16})$



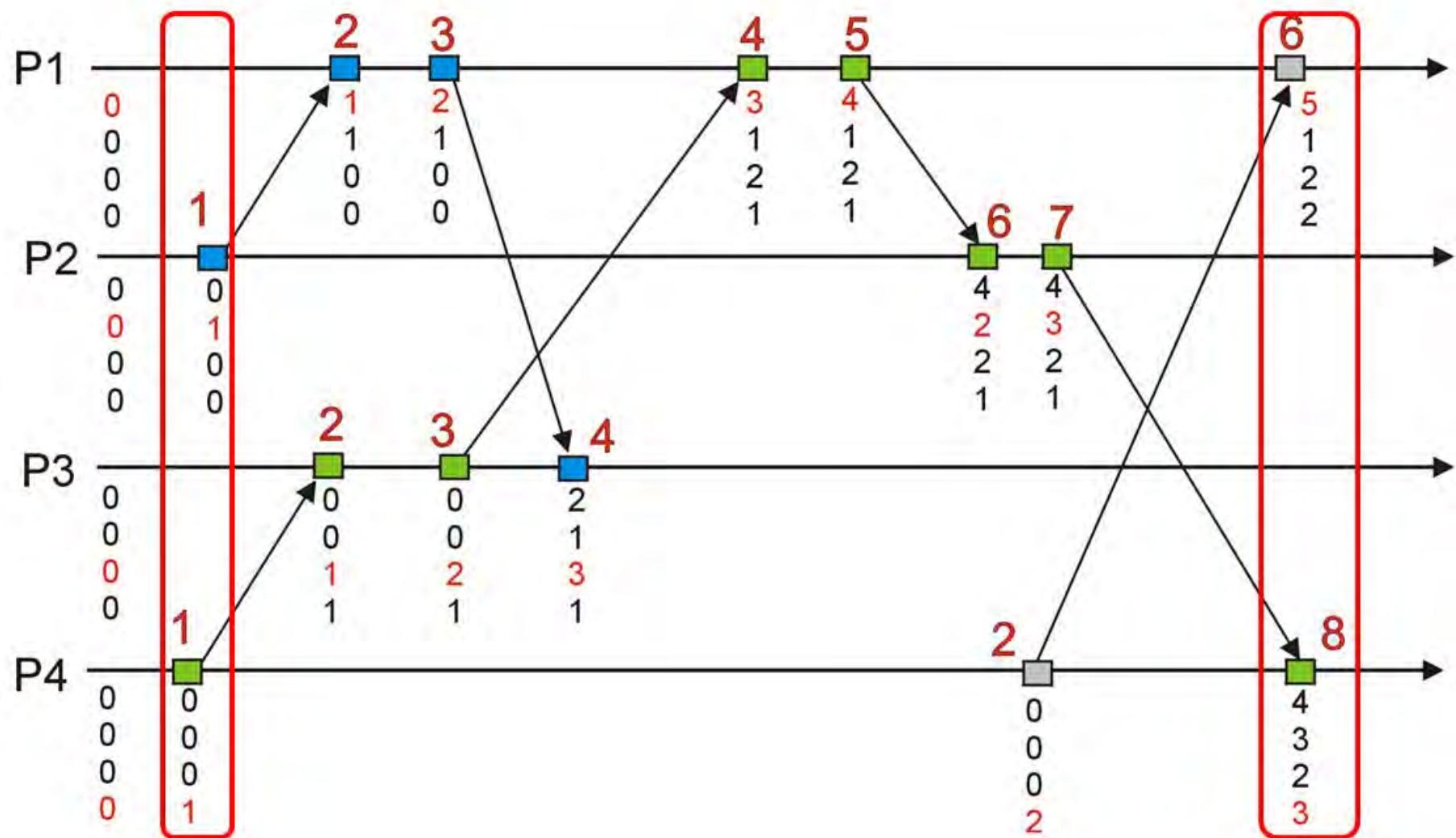
Comparison of vector clocks: (for actions a and b)

1. $t^a = t^b \iff \forall k \in [1 : n] \quad t^a[k] = t^b[k]$
2. $t^a \leq t^b \iff \forall k \in [1 : n] \quad t^a[k] \leq t^b[k]$
3. $t^a < t^b \iff (t^a \leq t^b) \wedge \neg(t^a = t^b)$
4. t^a concurrent $t^b \iff \neg(t^a < t^b) \wedge \neg(t^b < t^a)$

Example: Vector Clocks – 2



Example: Comparison of Lamport and Vector Time



Vector time: More information than Lamport

Advantage of vector clocks: $a \xrightarrow{\sqsubseteq} b \iff t^a < t^b$

\implies Causality can be derived from **time-stamp** alone!

Reason:

- $t^a < t^b \implies \forall k \in [1 : n] \text{ holds } t^a[k] \leq t^b[k] \text{ and}$
 $\exists l \in [1 : n] \text{ where } t^a[l] < t^b[l]$
 - **<-Entries:** local increment distinguishable from message
 \implies **step-wise backtracking** of message transfers
terminates (**finite** number of processes **and** a-cyclic chain)
-

Important: 'Logical time' definition depends on algorithm

- *Only events important to an algorithm are time-stamped.*
e.g., R/W in mutual exclusion, msg snd/rcv in msg ordering, ...
- Size of counters and message overhead is reduced.

Applications of logical time models

c.f.
VI.2/3

Origin: Lamport introduces 'logical time' for fairness in distributed mutual exclusion algorithms. (in: CACM 21(7), 1978)

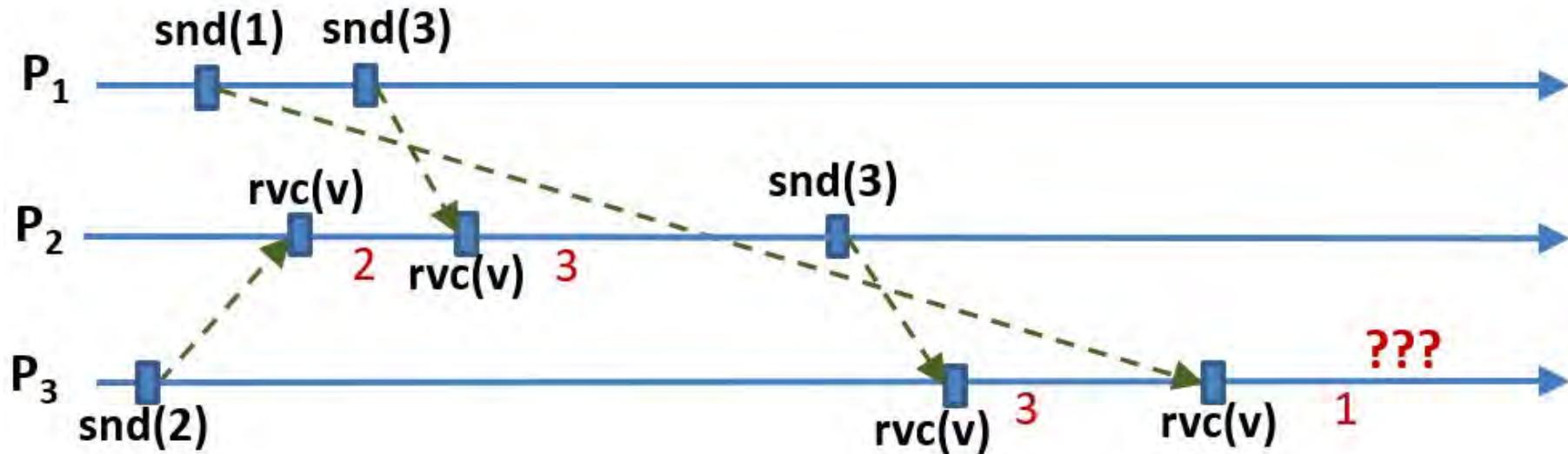
1. **Message ordering** for point-to-point and broadcast messaging VI.2
2. **Distributed mutual exclusion** VI.3
 - ◀ Request ordering for preventing deadlocks
 - ◀ Request ordering for ensuring fairness
3. Optimistic Concurrency Control in distributed database systems
 - conflicts based on mutual dependencies
 - optimistic ≈ conflicts are assumed to be rare
 - in case of conflict: choose 'victim' process and reset process
 - transactions are time-stamped for victim selection
4. . . .

End
of
VI.1

VI.2 Message Ordering

Problem: messages do not arrive in the order they were sent

- ◀ Effect: race conditions as in write-write conflicts
- Example: data base **updates**; the last update 'wins'
- ◀ Program logic based on causality may fail

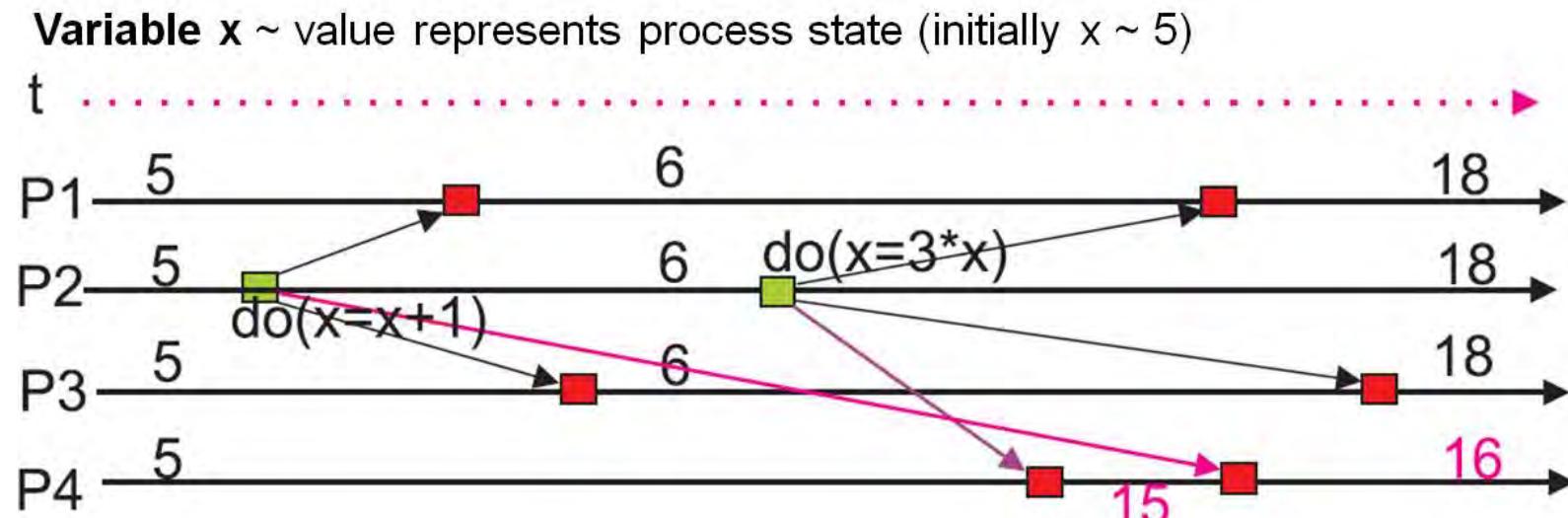


Recall: Isolated ordering between exactly two processes is simple in the context of lossless message transfer.

Ordering of Broadcast-Messages

- **FIFO broadcast:** All Broadcast-Msg of a **single** sender P_{snd} arrive for all receivers P_{rcv} in the order as sent.

Example: Problems with a broadcast without FIFO semantics



- **Causal broadcast:** P_{rcv} accepts a broadcast msg from P_{snd} **only after all** messages accepted in P_{snd} **before sending** this broadcast msg are also received and accepted in P_{rcv} .

Note: Causal broadcast \implies FIFO broadcast

Causal Broadcast: Birman/Schiper/Stephenson

1991

Preconditions

1. Broadcast message transfer is lossless
2. Each $P_i \in PS$ uses a vector for counting message send events

Idea: Time stamps VT for all broadcast send actions of all processes

1. Increment $VT_i[i]$ and send updated vector as part of message t_{msg}
2. Use MAX function to update local vector VT_i based on t_{msg}

Test in P_{rcv} : Compare local VT_i to time stamp vector t_{msg}

- If **all** local VT_i entries are equal or higher than those in t_{msg}
 - ⇒ all messages known in P_{snd} have also been accepted in P_{rcv}
 - ⇒ **accept** incoming message at once
- ◀ If VT_i **misses** messages ⇒ **buffer** arriving message in P_{rcv} until local vector VT_i is up to date w.r.t. t_{msg} , i.e., P_{snd}

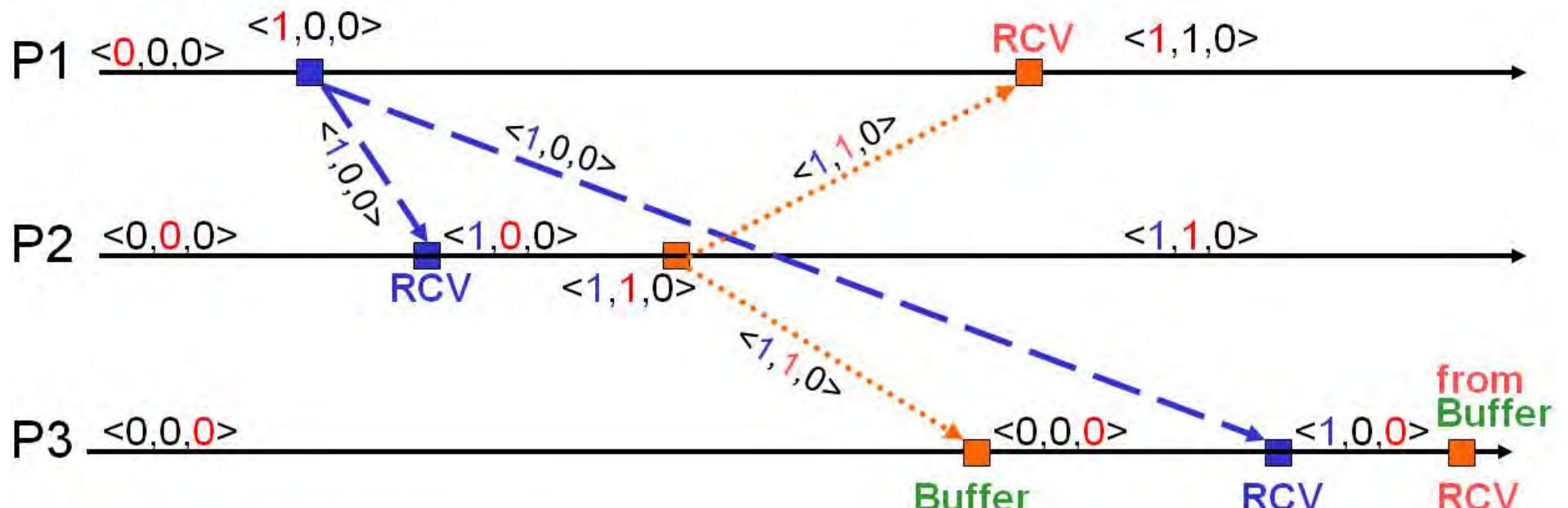
Note: logical time 'counts' broadcast events only

Birman/Schiper/Stephenson-Protocol

1. $\forall P_i \in PS$ initialize \overrightarrow{VT}_i by $\overrightarrow{0}$
2. **Before** a broadcast in P_i \approx increment $VT_i[i]$ locally
 $\implies VT_i[i] = |\text{Messages from } P_i|$
and send \overrightarrow{VT}_i as part ($\overrightarrow{t_{msg}}$) of the original message
3. **All** P_j where ($i \neq j$) receive message including $\overrightarrow{t_{msg}}$
 P_j **buffers** message **until**:
 - (a) $VT_j[i] = t_{msg}[i] - 1$ \implies message is most recent one from P_i **and**
 - (b) $\forall k \in [1 : n] \setminus \{i\}$ holds: $VT_j[k] \geq t_{msg}[k]$
 $\implies P_j$ knows at least all messages known by sender P_i at the time of sending.
4. **Accept**: Increment $VT_j[i]$ and execute **Buffer-TEST** (3.a/b)

Note: logical time \approx Number of messages sent
acyclic time stamps \implies no Deadlocks possible in (3.b)

Example 'Run': Birman/Schiper/Stephenson



RCV in P2:

- $\langle 0, 0, 0 \rangle[1] = \langle 1, 0, 0 \rangle[1] - 1$
- Other entries in P2 \geq Msg items

Buffer in P3: $\langle 1, 1, 0 \rangle$

- $\langle 0, 0, 0 \rangle[2] = \langle 1, 1, 0 \rangle[2] - 1$
- BUT: $\langle 0, 0, 0 \rangle[1] < \langle 1, 1, 0 \rangle[1]$
i.e. Msg from P1 not known in P3

RCV in P3:

- $\langle 0, 0, 0 \rangle[1] = \langle 1, 0, 0 \rangle[1] - 1$
- Other entries in P3 \geq Msg items

RCV from Buffer: $\langle 1, 1, 0 \rangle ?$

- $\langle 1, 0, 0 \rangle[2] = \langle 1, 1, 0 \rangle[2] - 1$
- and $\langle 1, 0, 0 \rangle[1] \geq \langle 1, 1, 0 \rangle[1]$
afterwards: $\langle 1, 1, 0 \rangle$

Note: RCV/Buffer events in P_2 and P_3 based on run

Point-to-Point Msg Ordering: Schiper/Egli/Sandoz

1989

Preconditions:

1. **Point-to-Point** message transfer is lossless
2. Each $P_i \in PS$ maintains vector time for all events

Idea: Send local state w.r.t. sending **to all other** processes
 \implies information 'simulates' **broadcast** effect for receiving processes.

Data structure: $PS = \{P_1, \dots, P_n\}$

- **local vector time** \vec{V}_i for all processes $P_i \in PS$
- **Message-List:** Each $P_i \in PS$ uses ML_i to store pairs $(P_j, \vec{v_m})$ holding knowledge about messages sent **to** P_j !!
 $\vec{v_m} \approx$ 'most recent' known vector time when sending in P_j
- **Messages:** $(P_i, \text{msg}, \vec{v_{msg}}, ML_i, P_j)$
 $(P_{snd}, \text{msg}, P_{snd} \text{ time stamp}, ML_i \text{ without current msg}, P_{rcv})$

Schiper/Eggli/Sandoz–Protocol – 1

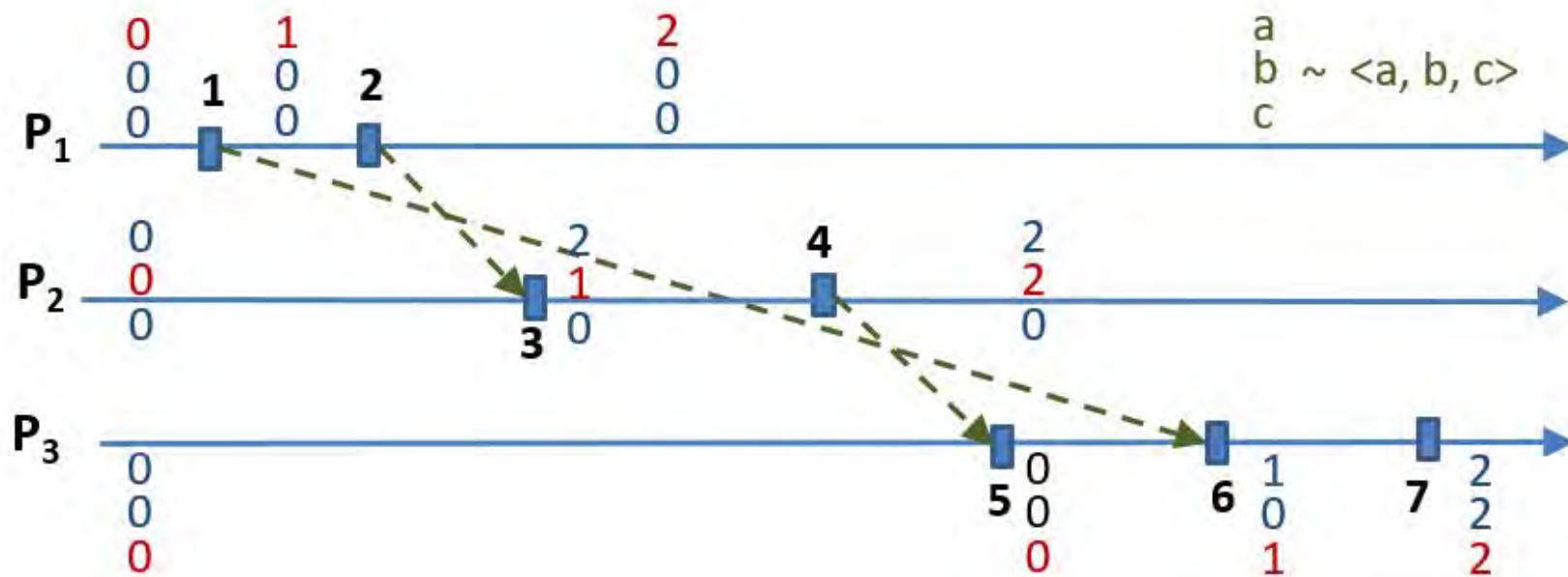
1989

- **Sender** P_i : $\vec{V}_i[i] := \vec{V}_i[i] + 1;$
 $\text{snd}(P_i, \text{msg}, \vec{V}_i, ML_i, P_j);$
 $ML_i := \text{insert}(ML_i, (P_j, \vec{V}_i));$
- Order !
- \implies current message **not** contained in ML_i message list
- ◀ **Receiver** P_j : on arrival of $(P_i, \text{msg}, v_{msg}^\rightarrow, \mathbf{ML}, P_j)$ (1)
 - TEST: if $\nexists (P_j, \vec{v})$ in \mathbf{ML} then ACCEPT; (2) $\in \text{Msg}$
 - else if $\vec{v} \not\prec \vec{V}_j$ then BUFFER(...); (3)
 - else ACCEPT; fi; (4)
 - fi;

(2) (P_j, \vec{v}) in $\mathbf{ML} \approx$ knowledge of P_i about messages to P_j
pair not found \implies **first message to** P_j (known in P_i)

(3) Condition holds \implies state in P_j is **not** more recent than
knowledge in P_i w.r.t. msgs to P_j
when sending \implies Buffer !

Schiper/Eggli/Sandoz–Protocol – 2



ACCEPT: a message $(P_i, \text{msg}, \vec{v}_{msg}, ML, P_j)$ from P_i in P_j

1. Combine $ML \oplus ML_j$ via insert/maximum for entries with $(k \neq j)$
 \implies only the most recent pair (P_k, \vec{v}) remains in ML_j
2. Increment local time \vec{V}_j and compute maximum based on \vec{v}_{msg}
3. Repeat TEST for all messages buffered using new time stamp analogous to steps (2/3) (may result in consecutive ACCEPT steps)

Result: Causal Point-to-Point message ordering

Example – Schiper/Egglı/Sandoz–Protocol

c.f.
pg.
VI-33

1. **snd in** P_1 : increment; $\text{snd}(P_1, \text{msg}, <1, 0, 0>, \emptyset, P_3)$
afterwards: $ML_1 := [(P_3, <1, 0, 0>)]$
2. **snd in** P_1 : increment; $\text{snd}(P_1, \text{msg}, <2, 0, 0>, [(P_3, <1, 0, 0>)], P_2)$
afterwards: $ML_1 := [(P_2, <2, 0, 0>), (P_3, <1, 0, 0>)]$
3. **rcv in** P_2 : the message $(P_1, \text{msg}, <2, 0, 0>, [(P_3, <1, 0, 0>)], P_2)$
 $(P_2, ?) \notin [(P_3, <1, 0, 0>)] \implies \text{ACCEPT}$
 $ML_2 := [(P_3, <1, 0, 0>)]; \text{increment and MAX} \implies V_2 := <2, 1, 0>$
VI-32
(2)
4. **snd in** P_2 : increment; $\text{snd}(P_2, \text{msg}, <2, 2, 0>, [(P_3, <1, 0, 0>)], P_3)$
afterwards: $ML_2 := [(P_3, <2, 2, 0>)]$
5. **rcv in** P_3 : the message $(P_2, \text{msg}, <2, 2, 0>, [(P_3, <1, 0, 0>)], P_3)$
 $(P_3, <1, 0, 0>) \in ML$, but $<1, 0, 0> \not< <0, 0, 0> = V_3 \implies \text{BUFFER}$
VI-32
(3)
6. **rcv in** P_3 : the message $(P_1, \text{msg}, <1, 0, 0>, \emptyset, P_3)$
 $(P_3, ?) \notin \emptyset \implies \text{ACCEPT}$
Combine \emptyset and $\emptyset \implies ML_3$ remains empty; $V_3 := <1, 0, 1>$
VI-32
(2)
7. **Buffer check in** P_3 w.r.t. $(P_2, \text{msg}, <2, 2, 0>, [(P_3, <1, 0, 0>)], P_3)$
 $(P_3, <1, 0, 0>) \in ML$, but $<1, 0, 0> < <1, 0, 1> = V_3 \implies \text{ACCEPT}$
 ML_3 remains \emptyset ; $V_3 := <2, 2, 2>$

VI.3 Distributed Mutual Exclusion

Goals:

- Safeness, i.e., correct critical section (csd) implementation
- Deadlock freedom, starvation freeness, fairness
- Efficiency: not too much overhead

Def.
III.8
III-25

Model: PS with Interaction using (asynchronous) message-passing

Costs:

- ▶ auxiliary data structures, vector clocks etc.
- ◀ additional processes and/or **messages**

Caution: single point-of-failure → multiple point-of-failure ?

processes do not react, lost control messages . . .

⇒ **only feasible under strong preconditions**

Logical Time: Fairness and avoiding cyclic waits (Deadlocks)

Approaches to Distributed Mutual Exclusion

- **Centralized approach:** Client/Server model, **but** server may be *bottleneck* \implies hierarchical systems, specialized server
single point of failure \implies replicated, redundant servers
- **Distributed approaches:** Permission to enter *csd* via ...

c.f.
chapter
III-66 ff.

III-71

c.f.
pg.
VI-41

... Inquiry among processes:

- P_i uses message passing to get permission from other processes
- Permission is considered granted iff majority accepts
- Inform other processes about own requests and answer requests

... Exclusive ownership of a control token

- Wait or ask for Token using message passing
- Handover of token after usage or via answering token request

Variants: different pre-assumed topologies for message or token exchange with varying overhead etc.

Token-based Algorithms

Basic Idea: Access to csd is granted to $P \iff P$ owns **token**

Variants: Organization of token circulation through PS
underlying communication structure of process system PS

Precondition: secure message transfer without loss of control token !!

1. **Simple Algorithm:** PS organized as a logical **ring** $P \mapsto P_{next}$

`rcv(token) \implies`

IF (request(self)) THEN csd ELSE $snd(P_{next}, token)$; FI;

Problems: 'unused' token circulates around the ring
node **crashes**; **lost messages**

2. **Suzuki-Kasami Broadcast Algorithm (1985)**

- ▷ P_i wants $csd \implies P_i$ sends **broadcast** with REQ for token
- ▷ Owner of token reacts only on incoming REQ
- ▷ REQ messages are prioritized based on logical REQ counter
 \implies Token handover after csd acts fair based on time stamps

Suzuki-Kasami Broadcast Algorithm

Data structures: (REQ -counter as vector time)

- local $RN_i[i]$ counter for most recent request REQ in P_i
- Vector $RN_i[1 : |PS|] \approx$ most recent, known REQ of other P_j
- **Token: Queue Q holding requesting processes and**
Vector $LN[1 : |PS|]$ holding numbers of granted REQ

Problems to be solved:

- ◀ **Outdated REQ** should not be answered
Example: All processes receive REQ from P_1 that is granted by P_2 ; afterwards, other processes should not hand over to P_1 again
 \implies **granted requests LN** are part of the token
- ▶ **Decide for next process to hand over:** (Fairness)
Queue Q of all processes with pending REQ is part of the token and updated after each csd access

Suzuki-Kasami Broadcast Algorithm

► Request in P_i :

```

 $RN_i[i] := RN_i[i]+1;$                                 /* Prologue */

broadcast(i,REQ, $RN_i[i]$ ); rcv( $Q, LN$ );    /* to all procs including  $P_i$  */

    csd                                         /* blocking rcv: wait for token ... csd */

 $LN[i] := RN_i[i];$                                 /* Start of Epilogue */

FORALL  $j \in [1 : n]$  DO                            /* Token-Q update */

    IF ( $RN_i[j]==LN[j] + 1$  AND  $P_j \notin Q$ ) THEN  $Q.\text{enqueue}(P_j)$ ; FI;

    OD;

IF ( $Q.\text{notempty}()$ ) THEN snd( $Q.\text{frontdequeue}()$ , $Q, LN$ ); FI;

```

► Reactions in P_i :

```

rcv(j,REQ, $RNr$ ) ==>                                /* always able to react */

 $RN_i[j] := \text{MAX}(RN_i[j], RNr);$                 /* update REQ counter */

IF (token() AND (NOT in csd/Epilog) AND ( $RN_i[j] == LN[j] + 1$ )
THEN snd( $P_j, Q, LN$ );

FI;                                                 /* empty Q due to Epilogue ! */

```

Properties of Suzuki-Kasami Broadcast Algorithm

- ▶ **Safeness:** At any time, token is owned by at most one process
entering csd only possible when owning token
- ▶ **Fairness:**
 - $(n + 1)$ -th REQ enters Q only if n -th request has succeeded
 \implies **maximum** number of $(|PS|-1)$ processes in **FIFO**-Queue Q ;
 - P hands token to next process after csd if there is a REQ in Q
- ◀ If P_i owns token and there are **no** REQ in Q
 - $\implies P_i$ may use csd repeatedly
- ▶ **Blocking:** Token is owned by P_i after a finite time (c.f. Fairness)
but: crashes, lost messages, non-terminating csds
- ▶ **Cost:** $0 \dots |PS|$ messages ($REQs$ and token)

Simulation of central knowledge (server):

Current owner of token acts as the server for the csd.

Non-Token Algorithms

Preconditions:

- ▶ global, unique time stamps for all messages
- ▶ message transfer respects message ordering !

Structure: each process $P_i \in PS$ 'knows' about it's process sets

1. **Request set** $R_i \subseteq PS$: **ask** for permission to enter *csd*
2. **Inform set** $I_i \subseteq PS$: **notify** if local state w.r.t. csd changes

Idea: **minimize** sets for each P_i

- R_i : ensure that at most one process is able to enter csd
- P_i is part of a sufficient number of inform sets I_j of other processes $P_j \implies P_i$ has a solid basis for it's own decisions
- in case of conflict: minimal time stamp \implies highest priority

Variants: Size of R_i and I_i

Number and content of messages required

Lamport Algorithm

CACM
21(7),
1978

Remark: First application of logical time concepts

- ▶ Simulate 'global server' through **all** processes of PS
 - **Request set:** $\forall P_i \in PS$ choose $R_i = PS \setminus \{P_i\}$
 - **Inform set:** $\forall P_i \in PS$ choose $I_i = PS \setminus \{P_i\}$
 \implies **global message exchange in** PS
- ▶ **Data structures** in each P_i : **Priority-Queue** Q_i holding request pairs
 - $< t_j, j >$ lexicographically ordered by **time stamps** in Queue
 - process numbers and Lamport clocks $\approx \text{clock}()$
 \implies globally ordered time stamps t_i
- ▶ **Message types:** $REQ \approx \text{request}$
 $REPLY \approx \text{acknowledgement for } REQ$
 $REL \approx \text{release}$

c.f.
pg.
VI-18

Algorithm Prologue/Epilogue and Process Behavior

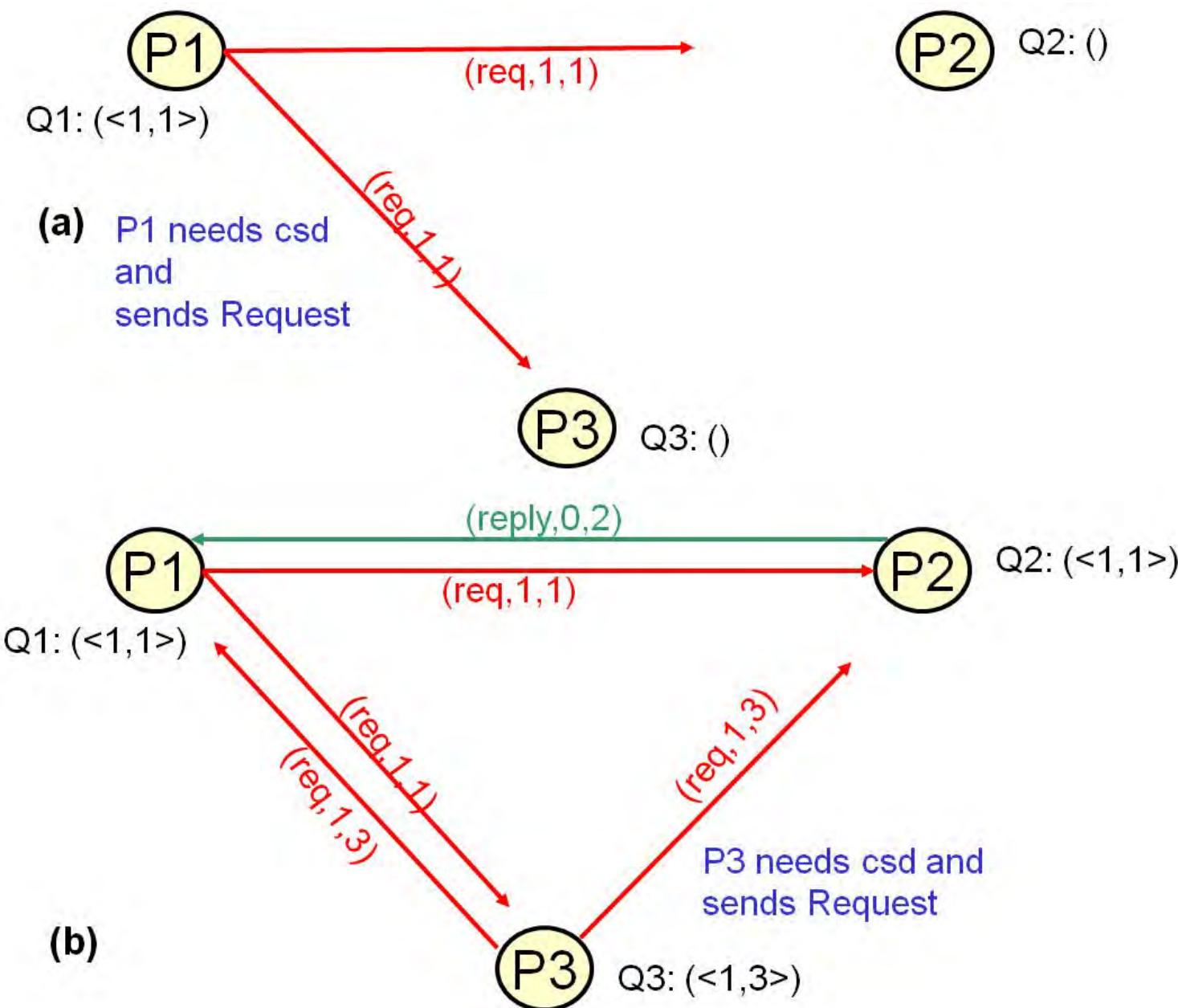
- **Request in P_i :**

```

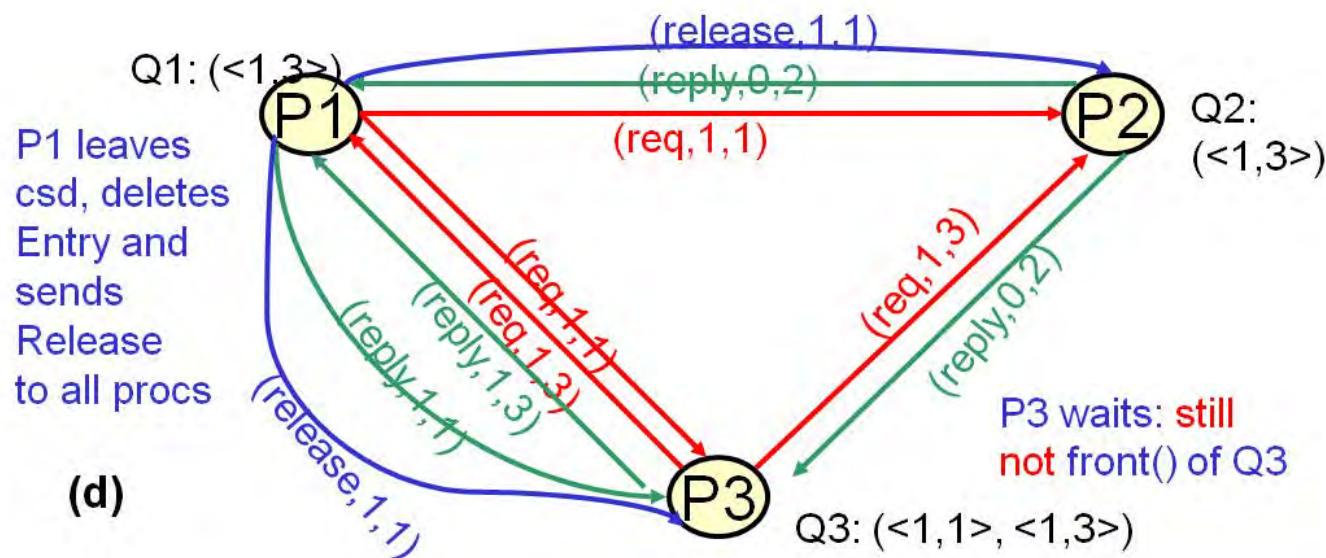
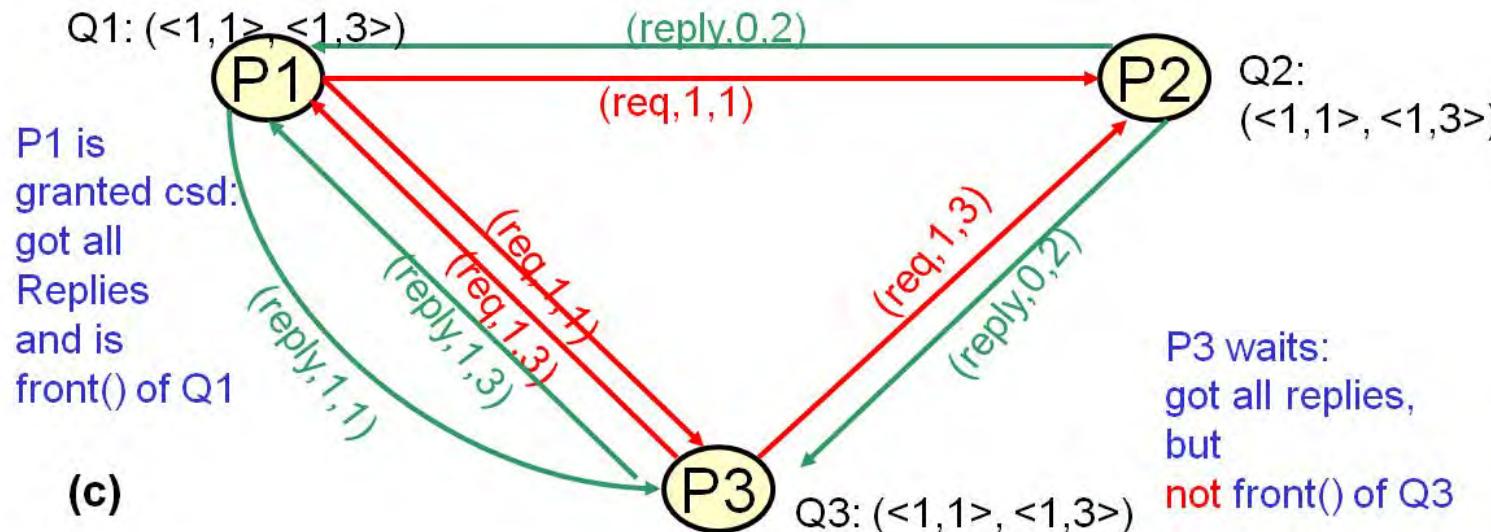
 $t_i \approx \text{clock}();$  /* Prologue */
 $\text{FORALL } P_j \in R_i \text{ DO } \text{snd}(P_j, \langle REQ, t_i, i \rangle); \text{ OD};$  /* inform */
 $Q_i.\text{enqueue}(\langle t_i, i \rangle);$ 
 $\text{WAIT UNTIL}$  /* periodic test */
 $\text{FORALL } P_j \in R_i \text{ rcv}(P_j, \langle REPLY, t_j, j \rangle);$  (1)
 $\text{AND } (Q_i.\text{front}() == \langle t_i, i \rangle);$  (2)
 $\text{csd}_i;$ 
 $Q_i.\text{dequeue}();$  /* Epilogue */
 $\text{FORALL } P_j \in I_i \text{ DO } \text{snd}(P_j, \langle REL, t_i, i \rangle); \text{ OD};$ 
```
- **Reactive behavior in P_i at all times**
 $\text{rcv}(P_j, \langle REQ, t_j, j \rangle) \implies \text{snd}(P_j, \langle REPLY, t_i, i \rangle);$ (3)
 $\quad Q_i.\text{enqueue}(\langle t_j, j \rangle);$
 $\text{rcv}(P_j, \langle REL, t_j, j \rangle) \implies Q_i.\text{remove}(\langle t_j, j \rangle);$

Example: Lamport Algorithm - Two Requests

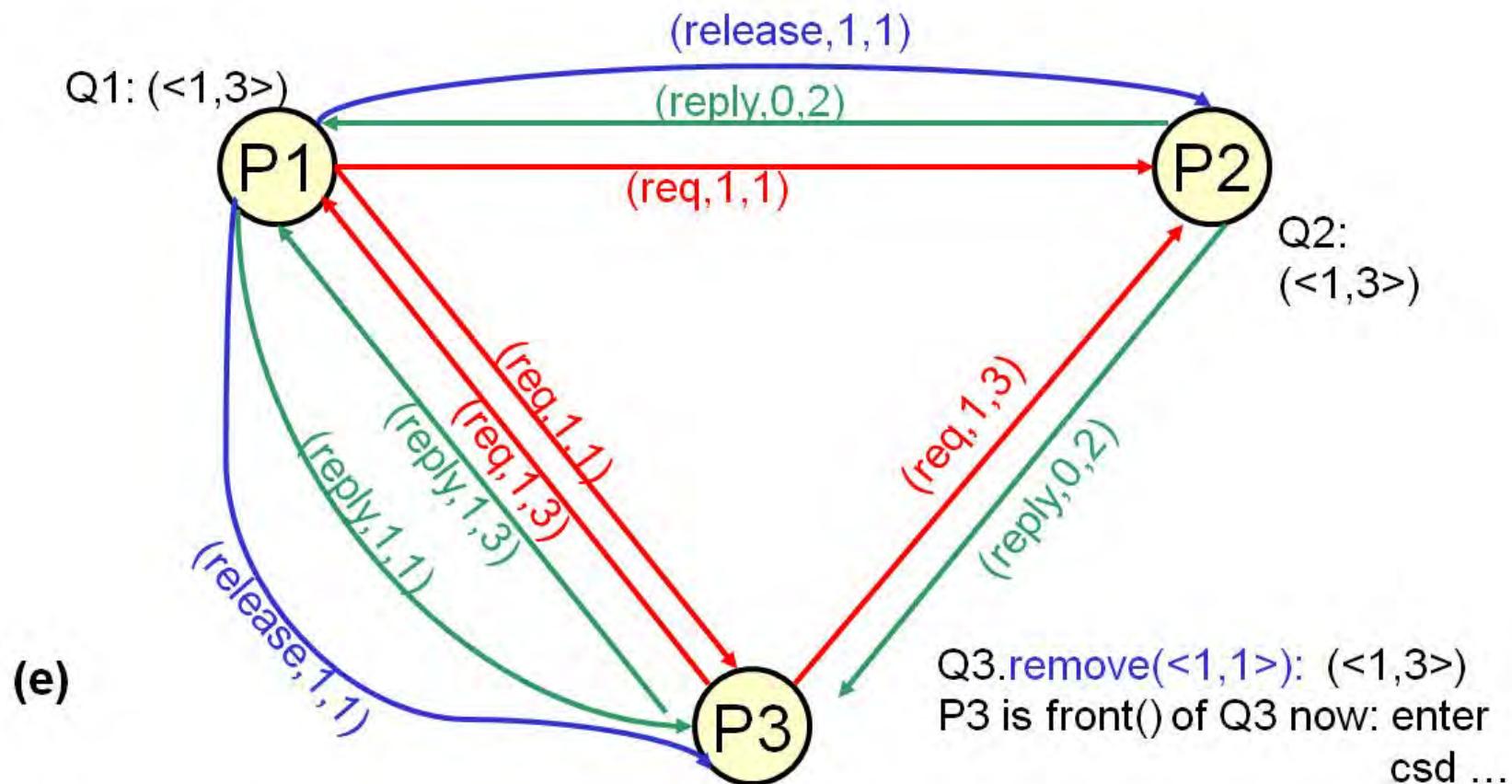
Le-
gend
c.f.
pg.
VI-46



Example: Lamport Algorithm - Permission & Wait

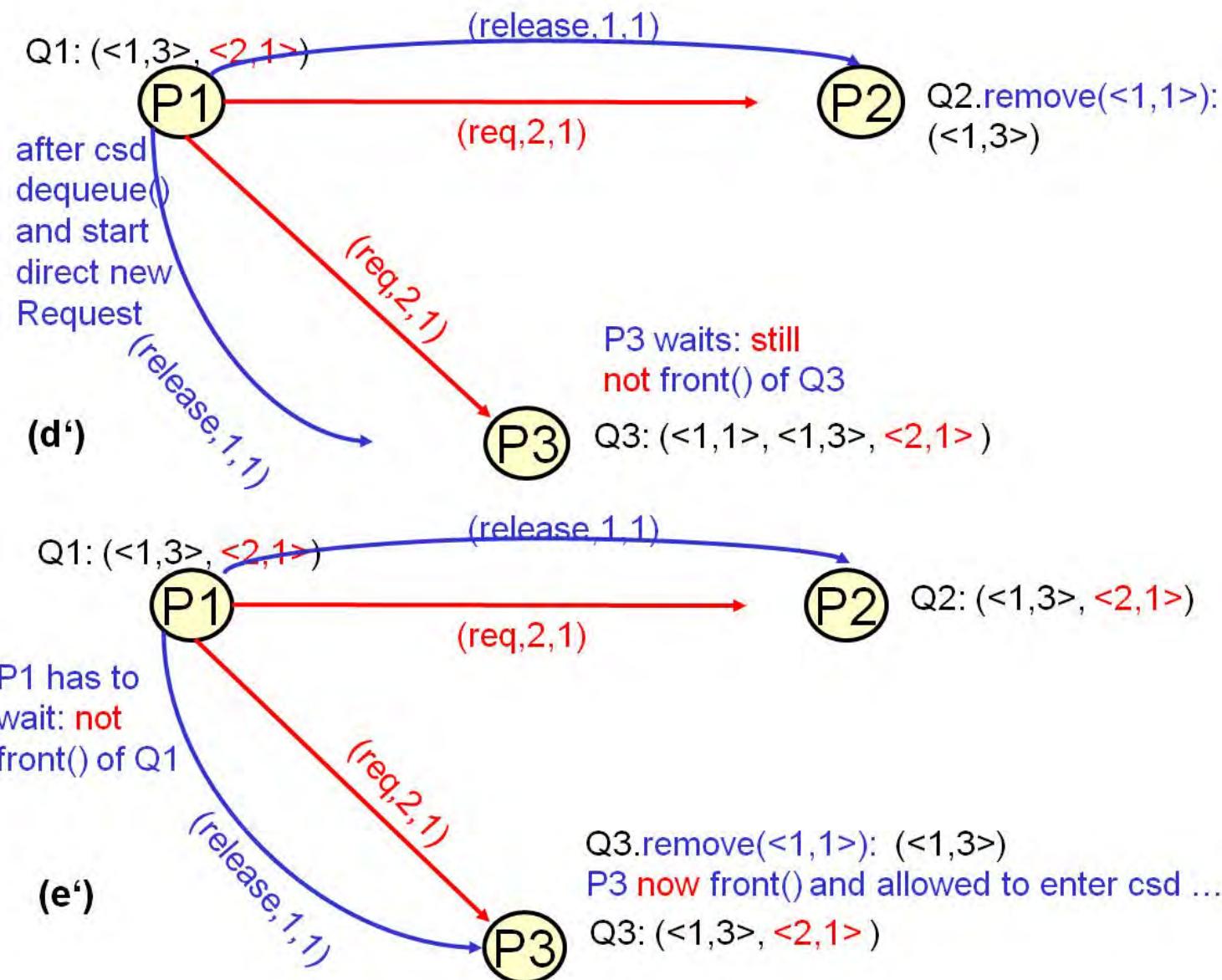


Example: Lamport Algorithm - Handover



Note: Request counter initial 0 in all processes;
 before sending Request: increment by 1
 All queues are empty initially
 Messages carry 3 items of information ($\langle \text{Type} \rangle$, ReqNr , ProcID)
 where $\langle \text{Type} \rangle ::= \text{req} \mid \text{reply} \mid \text{rel}$

Example: Lamport Algorithm - Unfairness ?



Assessment of Lamport Algorithm

- ▶ Safeness: P_i in $\text{csd}_i \implies$ Wait actions (1) and (2) executed
 - (1) REQ ; wait for $REPLY \implies$ processes know about REQ_i
 - (2) own request is front \implies other requests are more recent
 \implies queues of all other processes wait for REL from P_i
- ▶ No permanent blocking: (3) ensures that all $REPLY$ s are sent
- ▶ No deadlock: globally ordered time stamps prevent from cycles
- ▶ Fairness: conflicting requests are handled fair by REQ order queues
- ◀ **Cost:** $3 * (|PS| - 1)$ message for each csd permission
 \implies **optimization advisable**
- ◀ **Reliability:** *realistic preconditions?*
 - entire algorithm does not work if a single process fails to answer
 - lost $REQ/REPLY$ blocks P_i ; lost REL blocks csd
 \implies Use **time outs** to detect faulty processes

no
lost
msgs

Optimization: Ricart-Agrawala Algorithm

CACM
1981

Basic Idea: combine *REPLY* and *REL* messages

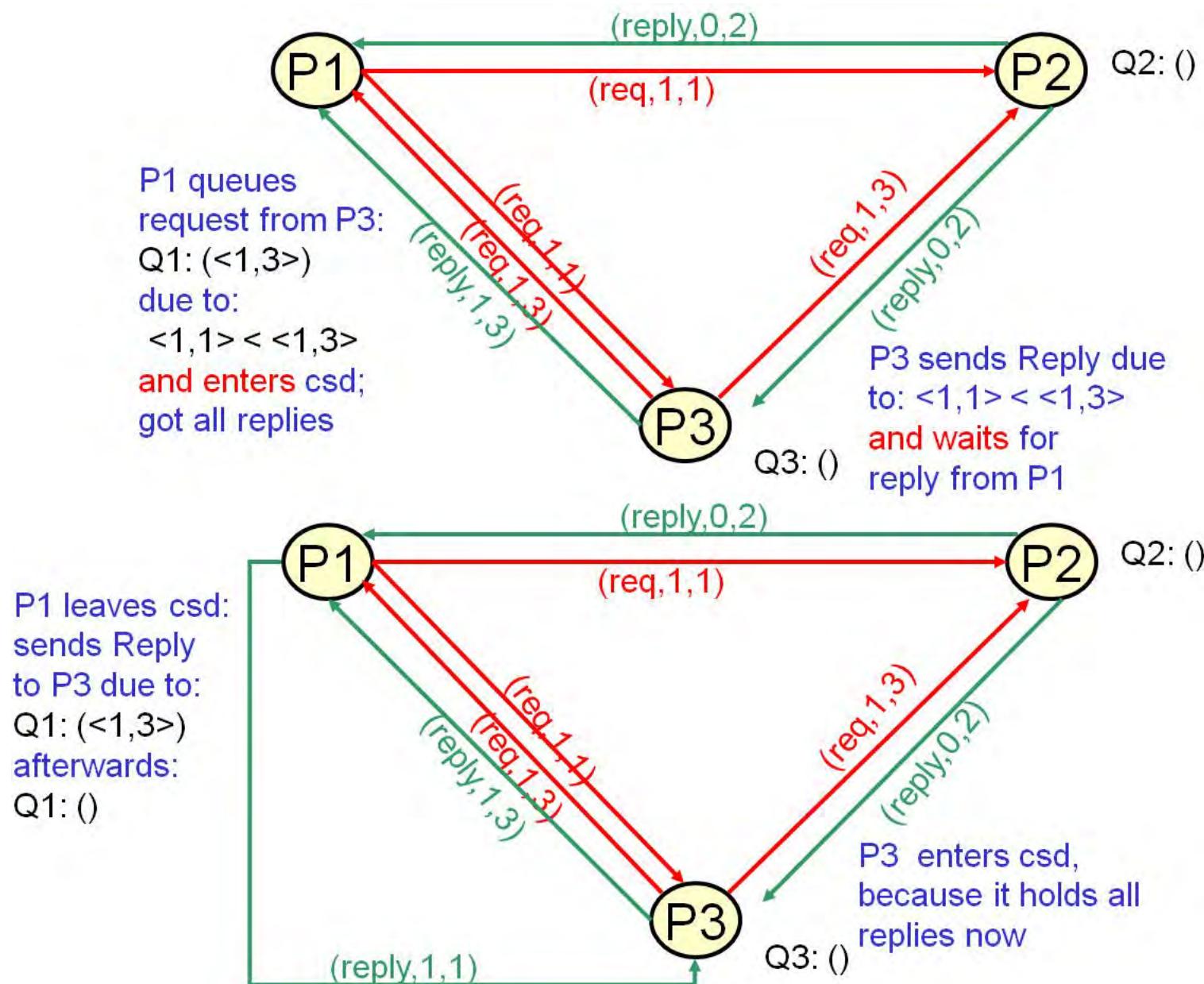
► **Request in P_i :**

```
FORALL  $P_j \in R_i$  DO snd( $P_j, <REQ, t_i, i>$ ); OD; /* inform */
WAIT FORALL  $P_j \in R_i$  rcv( $P_j, <REPLY, t_j, j>$ );
csdi;
FORALL  $P_j \in Q_i$  DO snd( $P_j, <REPLY, t_i, i>$ ); /*Epilogue*/
 $Q_i.remove$  OD;
```

► **Reactive behavior in P_i at all times:**

```
rcv( $P_j, <REQ, t_j, j>$ )  $\implies$ 
IF [ ( $P_i$  not in csd) AND ( $t_j < t_i$ ) ] /* csd not needed */
THEN snd( $P_j, <REPLY, t_i, i>$ ); /* or 'older' request */
ELSE  $Q_i.enqueue(<t_j, j>)$ ;
```

Example: Ricart-Agrawala Algorithm



Assessment of Ricart-Agrawala Algorithm

- Process stores requests only if it owns the *csd*
 - ▷ unburdens processes that don't need *csd* at all
 - ◁ not much redundancy for storing requests
- all processes have to react and send *REPLY* messages
- when leaving cs_i all pending reply messages are send

Safeness: P_i waits for **all** *REPLY* messages (as before)
only processes P_j that are not inside the *csd*
or have only a lower priority request will answer

Cost: $2 * (|PS| - 1)$ messages for each *csd* permission

Note: There are a lot more algorithms and optimizations dealing with distributed mutual exclusion. An overview can be found in P. C. Saxena and J. Rai: A survey of permission-based distributed mutual exclusion algorithms. Computer Standards & Interfaces, 25(2), 2003

VI.4 Global Snapshots and Consistency

Motivation: Fault Tolerance

- ▶ **Why:** many compute nodes and complex networks
⇒ **high probability of (partial) failures**

- ▶ **How:**

1. Store **local states** of processes
Examples: *MEM*, Register, *PC*, Resources
2. Store the content of **message channels**
3. **Combine** distributed local states to global **Recovery Points**
4. Crash ⇒ **Roll-back** based on most recent recovery point(s)

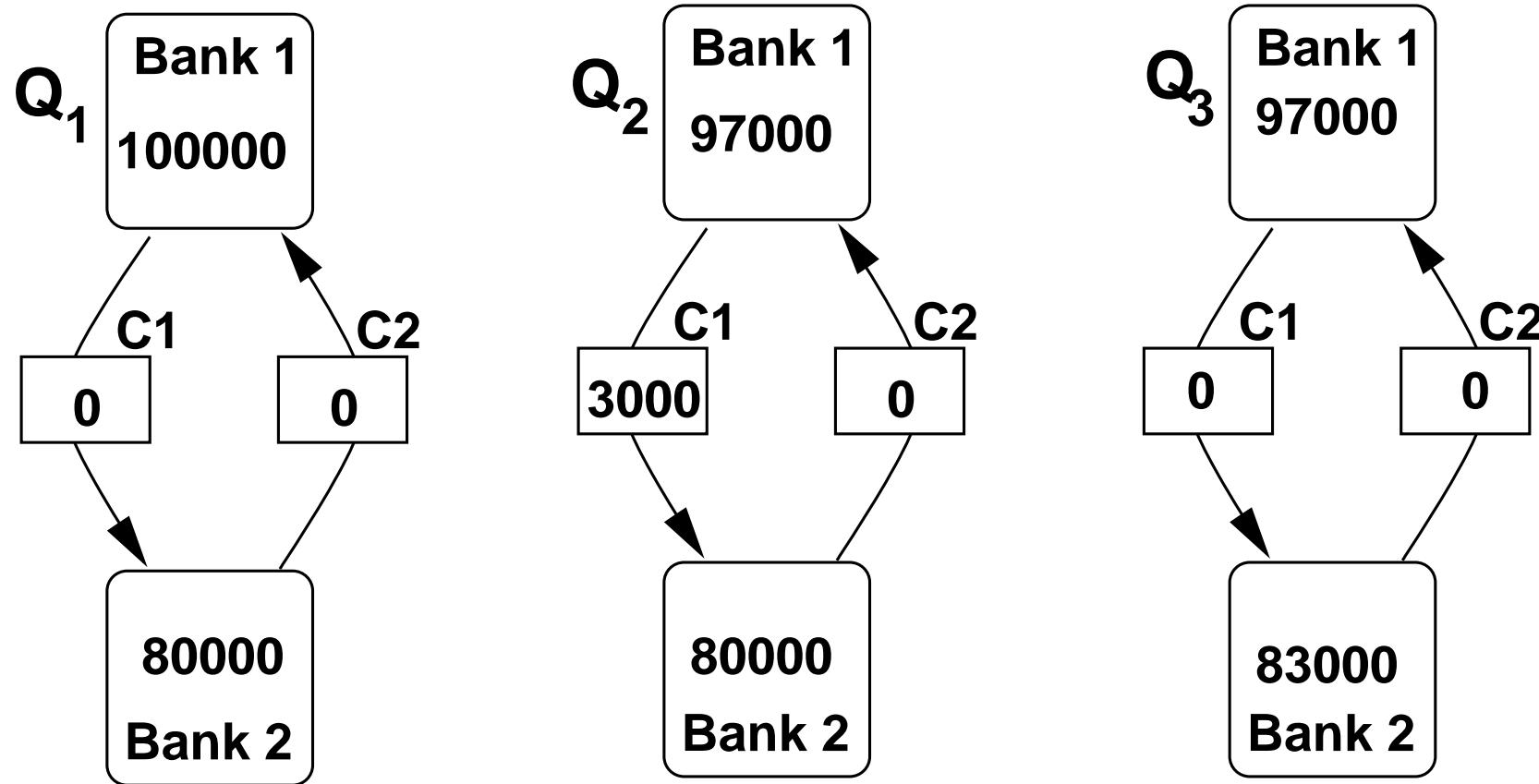
PCB
cf.
II-17

- ◀ **Problems:** additional overhead

- Trade-Off: Memory/Communication vs. Rollback benefits
- **Consistency** of stored 'global' states!

Overhead at runtime ⇒ **Crash easier to handle**

Example: Consistency is an Important Issue



- global state \approx amount of money in (bank1, C1, C2, bank2)
- **inconsistent state recording without coordination, e.g.,**
 1. Bank1 in Q_1 , channels/bank2 in Q_2 : $(100000, 3000, 0, 80000) \approx 183000$
 2. Channels in Q_1 , bank1/2 in Q_2 : $(97000, 0, 0, 80000) \approx 177000$

Reasons for Consistency Problems

◀ Processes:

- uncoordinated recording of local states is not sufficient
- coordination based on 'global clock' usually not feasible

Combination: internal state **plus** view on external system

◀ Message channels: How to get content?

- record content of channels (before/after sending) **or**
- wait for channel clearance based on maximum transfer time

Important: (P_1, C_{12}, P_2) observe $|SND_{P_1}| = |C_{12}| + |RCV_{P_2}|$
without FIFO channels \implies explicit msg ordering required

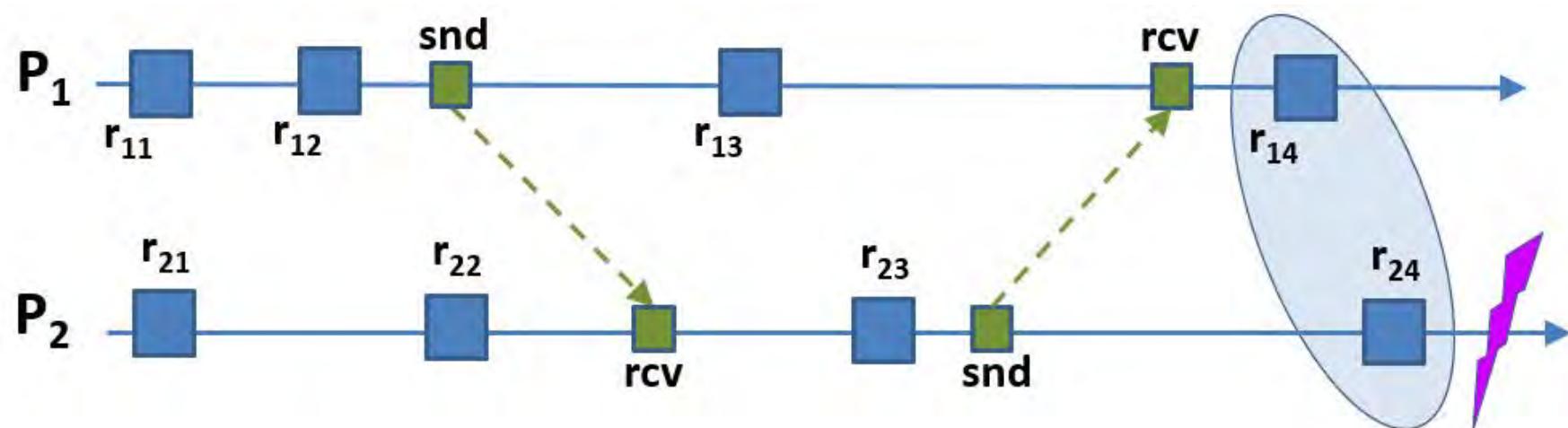
Elementary Problem: $P_i \neq P_j$ record **different** external views

- ◀ wrong assumptions about messages sent
- ◀ wrong assumptions about messages received
- ◀ how to detect messages lost during transfer?

Problems using local Recovery Points only – 1

- Each process P_i records its own local recovery-points r_{ik}
- Recording is **not** coordinated among processes
- Crash $\implies PS$ resets each P_i to it's *most recent* recovery-point
- Messages are re-send if sent after recording recovery-point

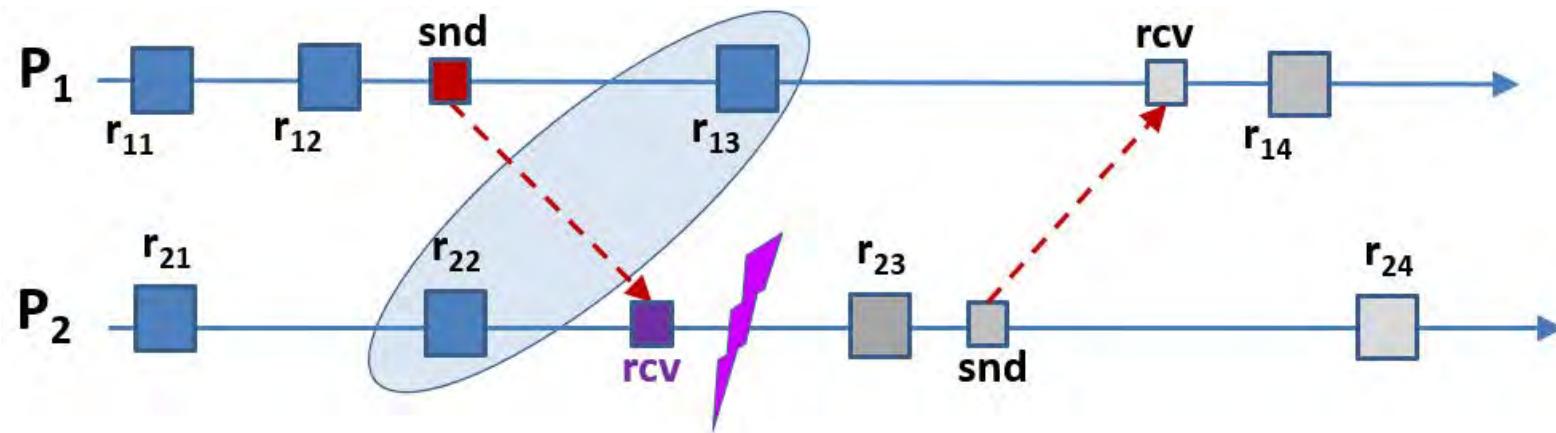
1. Crash in $P_2 \implies$ **roll-back** to state (r_{14}, r_{24})



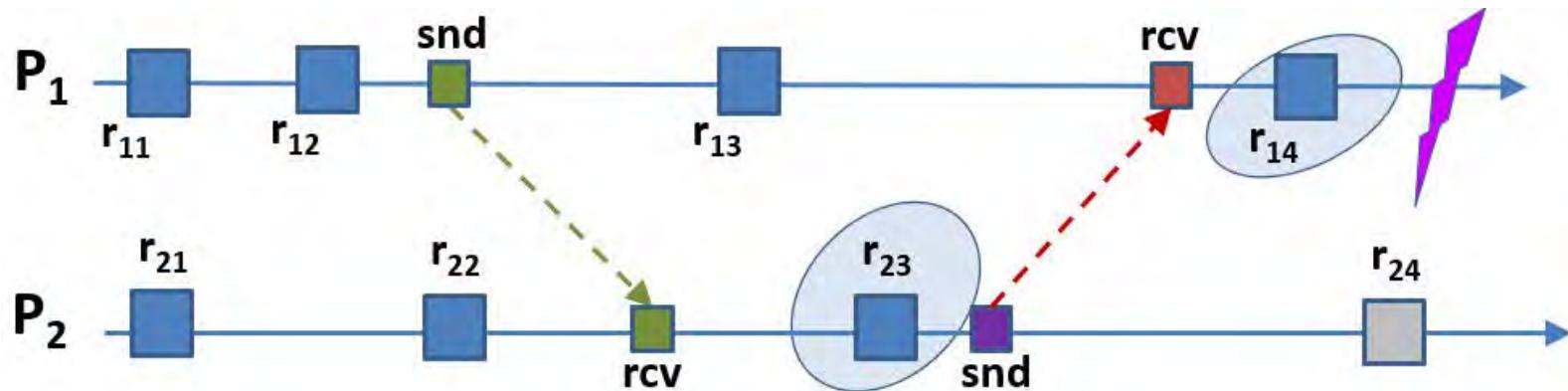
\implies Useful Procedure that keeps lot of information? Not always!

c.f.
pg.
VI-57

Problems using local Recovery Points only – 2



2. Crash in P_2 : (r_{13}, r_{22}) ? \Rightarrow **lost message** (will block P_2)
 Reset P_1 to r_{12} \Rightarrow **roll-back** to state (r_{12}, r_{22})

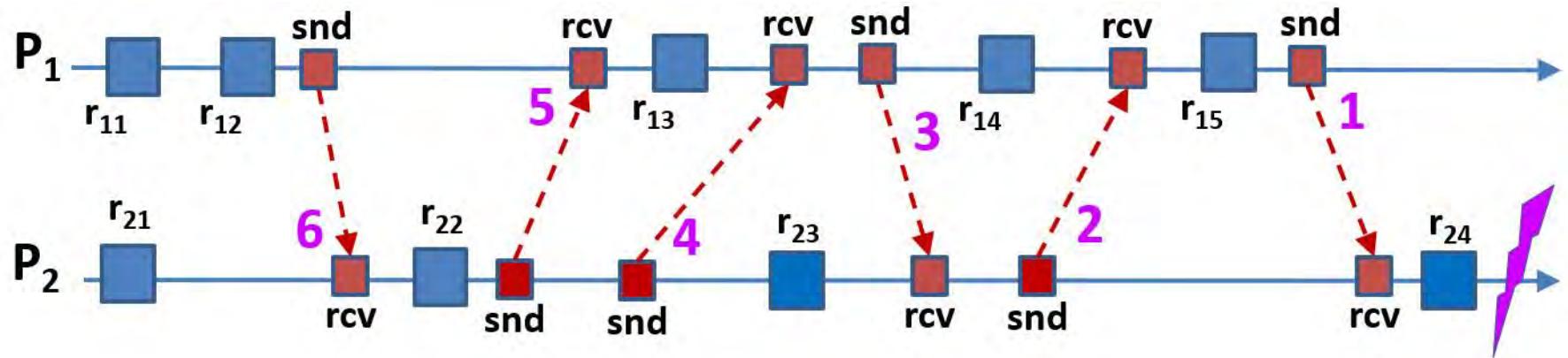


3. Crash in P_1 : (r_{14}, r_{23}) ? \Rightarrow **orphan message** (repeated snd)
 Reset P_1 to r_{13} \Rightarrow **roll-back** to state (r_{13}, r_{23})

Uncoordinated Procedure leads to Domino Effect

$\forall (P_i, P_j)$ check whether $\approx |\text{SND}_{ij}| = |\text{RCV}_{ji}|$?

- **Lost Messages** $|\text{SND}_{ij}| > |\text{RCV}_{ji}| \implies$ Reset SND process
 - **Orphan Messages** $|\text{SND}_{ij}| < |\text{RCV}_{ji}| \implies$ Reset RCV process
- until global recovery-point with *consistent information* is reached.



- Crash in $P_2 \implies (r_{15}, r_{24})$ includes Orphan 1 \implies reset P_2
- (r_{15}, r_{23}) includes Orphan 2 \implies reset P_1
- (r_{14}, r_{23}) includes lost message 3 \implies reset P_1
- (r_{13}, r_{23}) includes lost message 4 \implies reset P_2
- (r_{13}, r_{22}) includes Orphan 5 \implies reset P_1 (worst-case)
- (r_{12}, r_{22}) includes Orphan 6 \implies reset $P_2 \implies (r_{12}, r_{21})$

**Domino-
Effect**

Coordinated Procedure – General Considerations

⇒ Detect and avoid inconsistencies when recording

1. Lost Messages: Record contents of message channels

- ▶ fault-tolerance in network protocols and OS network interface
 - e.g., store-and-forward until successful acknowledgement
- ▶ repeating a SND is no big deal!
 - ⇒ Do not roll-back if $|SND_{ij}| > |RCV_{ji}|$

2. Orphan Messages: repeated SND may disrupt receiver process avoid this kind of *inconsistency*

⇒ Do rollback if $|SND_{ij}| < |RCV_{ji}|$

Coordinated Approach:

- Coordinate local state recordings in order to avoid *domino effect*
- Provide **secure message channels**
 - ▶ System **active**: $\forall (P_i, C_{ij}, P_j)$ holds $|SND_{P_i}| = |C_{ij}| + |RCV_{P_j}|$
 - ▶ System **inactive**: $\forall (P_i, C_{ij}, P_j)$ holds $|SND_{P_i}| = |RCV_{P_j}|$

empty
channels

System Model for Snapshot Algorithm – 1

- ▶ Process system $PS = \{P_1, \dots, P_n\}$
- ▶ **Complete Connectivity:** among each pair (P_i, P_j) of processes there exists a message channel C_{ij}
 1. Messages are **not** lost: middle-ware layer for lossless messaging
 2. Message channels have FIFO property, i.e., all channels (P_i, P_j) support proper message ordering.
- ▶ All processes and channels are *observable* (in theory)

Notation:

- $\forall P_i \in PS$ let LP_i be the local state of P_i
- **Observed Actions:** Send/Receive/Record state
 - * $\text{snd}(m_{ij})$ sending the message m_{ij} in P_i
 - * $\text{rcv}(m_{ij})$ receiving the message m_{ij} in P_j
 - * LP_i recording the local state LP_i in P_i
- $\text{time}_i(event) \approx$ local time in P_i when $event$ occurs

System Model for Snapshot Algorithm – 2

Basic Principles:

1. Which snd/recv events are part of recording a local state:

- (a) $\text{snd}(m_{ij}) \in LP_i : \iff \text{time}_i(\text{snd}(m_{ij})) < \text{time}_i(LP_i)$
- (b) $\text{recv}(m_{ij}) \in LP_j : \iff \text{time}_j(\text{recv}(m_{ij})) < \text{time}_j(LP_j)$

\implies local state recording respects **causality** due to sequential execution order in P_i .

2. Compare local states of different processes P_i and P_j ($i \neq j$):

- (a) **TRANSIT**(LP_i, LP_j) :=

$$\{ m_{ij} \mid \text{snd}(m_{ij}) \in LP_i \wedge \text{recv}(m_{ij}) \notin LP_j \}$$

msgs sent but not received w.r.t. compared states of P_i and P_j

- (b) **INCONSISTENT**(LP_i, LP_j) :=

$$\{ m_{ij} \mid \text{snd}(m_{ij}) \notin LP_i \wedge \text{recv}(m_{ij}) \in LP_j \}$$

msgs received but not sent w.r.t compared states, i.e. **Orphans**

System Model for Snapshot Algorithm – 3

Characteristics of global states:

1. **Global State** $GS := \{ LP_1, LP_2, \dots, LP_n \}$ such that holds:

$$\forall P_i \in PS \exists \text{ exactly one } LP_i \in GS$$

isolated recording of one local state for each process in PS

2. GS is **consistent** : \iff GS contains **no Orphans**, i.e.,

$$\forall i \in [1 : n] \forall j \in [1 : n] \text{ holds: } \text{INCONSISTENT}(LP_i, LP_j) = \emptyset$$

3. GS is **strong consistent** : \iff GS consistent **and**

$$\forall i \in [1 : n] \forall j \in [1 : n] \text{ holds: } \text{TRANSIT}(LP_i, LP_j) = \emptyset$$

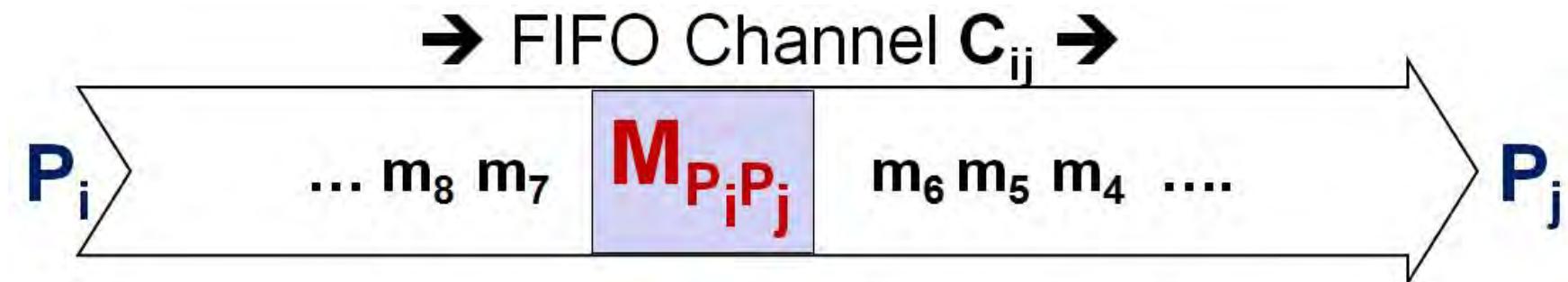
combined local states hold all information due to empty channels

Note: If GS is consistent **and** there is an upper bound for all msg transfer times t_{max} :

Record GS ; Update GS after waiting t_{max}
 \implies strong consistent GS' is achieved easily.

Chandy-Lamport Snapshot Algorithm – 1

- ▶ **Precondition:** lossless FIFO channels
- ▶ **Idea:** Isolate Phase ^{k} ; Record states; Phase ^{$k+1$}
 1. Initiate state recording **locally** in any process P_i spontaneously
 2. **Propagate** global recording via **marker** msg on all channels
FIFO eases isolation between different phases by distinguishing among messages sent **before/after** local state recording.
⇒ Processes are allowed to proceed including `snd/recv` actions.



3. Each P_i stores all msgs received after most recent recording LP_i
4. At the end of state recording → send state to Coordinator

Chandy-Lamport Snapshot Algorithm - 2

- **Start Snapshot Algorithm:** in an arbitrary process P_i

$\text{record}(LP_i);$ (stores LP_i locally)

$\text{FORALL } C_{i,j} \text{ DO } \text{snd}(M_{i,j}) \text{ OD;}$ (propagate execution)

- **Reaction** on a $\text{rcv}(M_{i,j})$ in process P_j (react at all times)

$\text{IF (NOT recorded}(LP_j)\text{) THEN}$

$\text{STATE}(C_{i,j}) := \text{empty}; \text{record}(LP_j);$ (store locally)

$\text{FORALL } C_{j,k} \text{ DO } \text{snd}(M_{j,k}) \text{ OD;}$ (propagate)

ELSE (update local state by all $msg_{i,j}$ received after local recording)

$\text{STATE}(C_{i,j}) := \text{STATE}(C_{i,j}) \cup$

$\{msg_{i,j} \mid time(LP_j) < time(\text{rcv}(msg_{i,j})) < time(\text{rcv}(M_{i,j}))\};$

FI;

- **Combine to global state:** all processes send LP_i to coordinator after all marker have arrived in P_i (counter for $|\{C_{i,j}\}|$)

Observations – Algorithm Properties

- ▷ **Result:** consistent global state
no: strong consistent state, **but:** states of channels are known
- ◁ **complete global state** $\implies \exists P_s$ connected to *all* processes in PS
- ▶ **Process Structure:** Initiation
 - **Diffusion** phase \approx when first marker arrives in a process P_j
forward marker $M_{j,k}$ to all processes P_k directly connected to P_j
 - **Contraction** phase \approx update local state for 'late' messages
no more forwarding of markers
 - **Collection** phase \approx send local state(s) to coordinator process
- ▶ **Distinguishing 'runs'** for different 'spontaneous' initiations:
 1. Extend marker $M_{j,k}$ by initiator process number P_i : $M_{i,j,k}$
 P_i acts as coordinator \implies decentralized, independent 'runs'
 2. Additional counter in P_i discriminates different runs by P_i

impor-
tant
w.r.t.
termi-
nation

Properties of Snapshot Algorithm – 1

1. Correctness: Resulting state is consistent \implies **no Orphans**

Proof: (indirect)

Assumption: global state holds (at least) one **Orphan** message

$$\implies \exists msg_{i,j} \text{ s.t. } \text{snd}(msg_{i,j}) \notin LP_i \wedge \text{rcv}(msg_{i,j}) \in LP_j$$

P_i sends msg **after** recording local state LP_i and

P_j has received msg **before** recording local state LP_j

\implies **Contradiction:** Processes do not respect protocol:

(a) $msg_{i,j}$ received **before** marker $M_{i,j}$ in P_j \implies

i. execution order: $\text{snd}(msg_{i,j}) ; \text{snd}(M_{i,j})$ in P_i **or** (illegal)

ii. Channel $C_{i,j}$ is not FIFO (precondition not met)

(b) $msg_{i,j}$ received **after** marker $M_{i,j}$ in P_j

execution order: $\text{rcv}(M_{i,j}) ; \text{rcv}(msg_{i,j}) ; LP_j$ in P_j (illegal)

\implies **No Orphans iff algorithm is executed as programmed.**

Properties of Snapshot Algorithm – 2

2. Termination: for a single, distinguishable run

Precondition: isolated steps in P_j terminate; finite transfer times

- Diffusion phase initiated by P_i at most $|PS|*(|PS|-1)$ markers (maximum connectivity) as each P_j store and propagates exactly once (THEN)
- Contraction phase ends after a maximum of $(|PS|-1)$ received markers at each process (full connectivity)
- Collection phase: exactly $(|PS|-1)$ messages LP_j go to P_i .

c.f.
pg.
VI-63

3. Costs: local memory, compute time and extra messages

- (a) $|PS|$ local states LP_i and $|PS|*(|PS|-1)$ channels
- (b) $|PS|*(|PS|-1)$ markers and $(|PS|-1)$ local states

Option: Record $B \subset PS$ of 'important' processes only,
e.g., *Server* for data bases, long-running computations

Importance of Message Channels for Snapshots

- ◀ Initiator does **not** reach all processes \implies **incomplete snapshots**
direct connection or connectivity via transitive hull of **msg** channels
- ◀ explicit control messages needed to avoid incomplete snapshots
otherwise: input data and actual process run may not contact all processes \implies not all processes will get markers.

Tradeoff: *Additional control messages vs. quality of result*

- ◀ **Channel properties** are essential:

1. No FIFO property \implies simulate message ordering
2. No *piggy-backing* of messages or extra messages permitted?

c.f.
VI.2

Without (1) and (2) \implies Algorithm has to **freeze entire system**
until snapshot is recorded. (*How?*)

Taylor
1989

at least: prevent processes from sending messages during recording!

Note: Determining *TRANSIT* is easy for FIFO channels with
known maximum transfer times, otherwise hard to get.

c.f.
pg.
VI-60

Global States and Causality

c.f.
chapter
VI.1

- **Cut** of process system $PS \approx$ exactly one event for each $P_i \in PS$
 \implies Recording $LP_i \forall P_i \in PS$ results in a *cut* of PS
 Cut is consistent \iff all events are pairwise **concurrent**
 i.e. are not ordered w.r.t. *happened-before* relation $\xrightarrow{\sqsubseteq}$
 \implies **consistent global state is also a consistent cut**

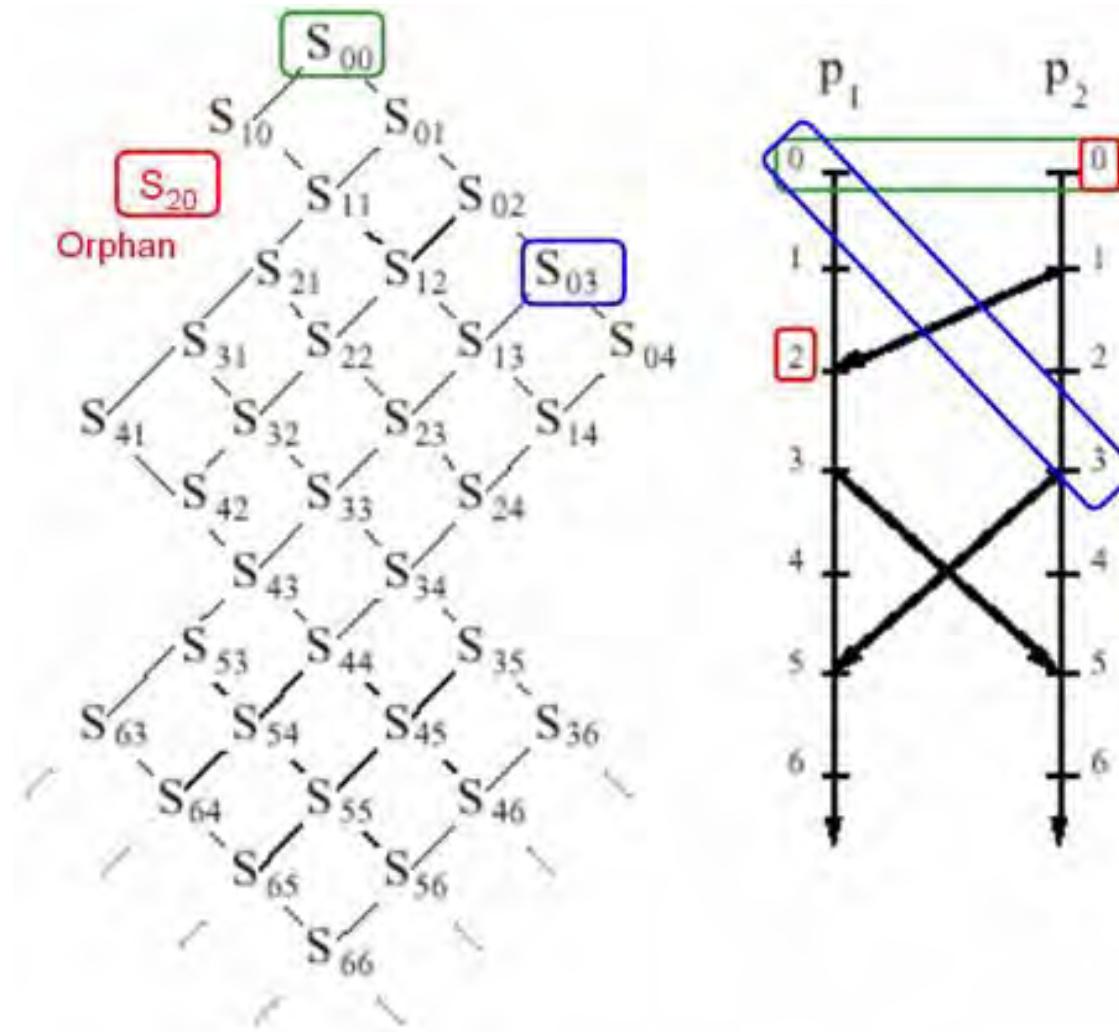
Orphan: `rcv` without `snd` implies causal chain through channels

- **Alternative Formalization:** (*Petrinet/process theories*)
 $\begin{array}{l} \text{c.f.} \\ \text{pg.} \\ \text{VI-69} \end{array}$
 1. $e_{ik} \in P_i$ subsumes its **history**(e_{ik}) := $e_{i1}e_{i2}\dots e_{ik}$, i.e., is identified with the prefix of the causal chain that leads to e_{ik}
 2. $\text{Cut}(PS) := \bigcup_{i \in [1:|PS|]} \text{history}(e_{ik_i})$ (1 chain for each process)
 3. **Front** := $\{e_{1k_1}, e_{2k_2}, \dots, e_{nk_n}\}$ most recent events of processes
 4. $\text{Cut}(PS)$ is consistent \iff

$$(b \in \text{Cut}(PS) \wedge a \xrightarrow{\sqsubseteq} b) \implies a \in \text{Cut}(PS)$$
 2. ensures local order in P_i ; **msgs** are the 'critical' events

Example: Synchronic Distance among Processes

c.f.
Colouris
et al.:
Distrib-
uted
Systems
Fig.10.15



- 'Lattice' of all permitted, combined states for P_1 and P_2
- more de-coupling, e.g., S_{25} not permitted w.r.t. consistency

End
VI.4

VI.5 Detecting Global System States

Global Consistent View on a DS is always a challenge!

► **Non-volatile, permanent States:** Detection not time-critical

- **System Termination**
- **Deadlocks**

VI.5.1

VI.5.2

◀ **Volatile States:** How to obtain a consistent view?

VI.4

- local changes may occur spontaneously
- detection algorithm may change the system behaviour
- what about messages in *TRANSIT*?

Additional Dimension: *State reachable in every run of the system?*

► Situation in the 'running' PS at hand \implies Debugging

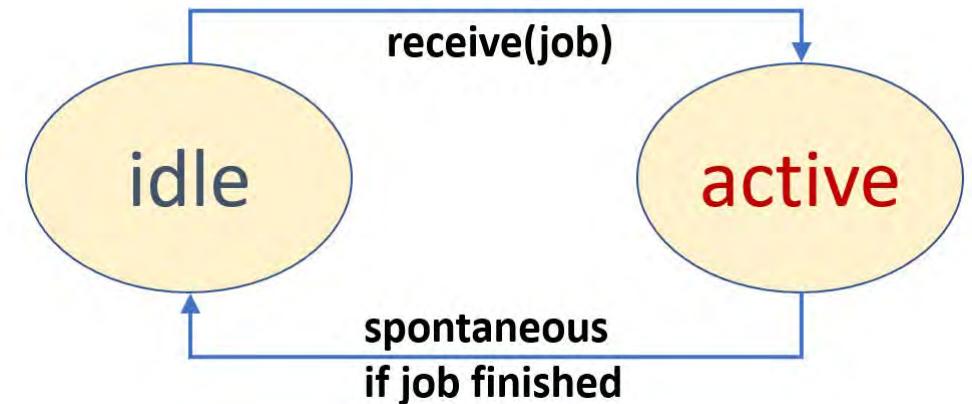
◀ **Property:** for all PS when running a program \implies Proof

- * *Safety*-Properties: nothing bad will ever happen
- * *Liveness*-Properties: something wanted is eventually reachable

VI.5.1 Detecting System Termination

System Model: $PS = \{P_1, \dots, P_n\}$

1. Message channel C_{ij} between each process pair (P_i, P_j)
channels are 'robust', i.e. no messages are lost
2. Every $P_i \in PS$ is always in one of the two states { idle, active }:
 - (a) active \implies Process may snd, rcv or compute
 - (b) idle \implies rcv of messages only (**NO** snd!)
3. All State Changes permitted but:
no spontaneous
change to
active based on
internal reasons only



Why Termination? always a problem in de-centralized algorithms
 \implies do not 'mix' termination detection with application

Basic Algorithmic Idea

► **Initial State:** Every process is idle, no messages

- $\forall P_i \in PS: STATE(P_i) = \text{idle}$
- $\forall P_i, P_j \in PS: TRANSIT(P_i, P_j) = \emptyset$

VI-60

► **Final State:** the same as initial state

► **Method:** unique **Monitor Agent** oversees all activities

- \exists **exactly one** $P_{mon} \in PS$ where $P.\text{monitor}() = \text{true}$
- **Weights** represent the level of work a process has to do

1. $\forall P_i \in PS$ holds $\text{Weight}(P_i) \geq 0$

2. **Algorithmic**

$$\sum_{P_i \in PS} \text{Weight}(P_i) = 1$$

Invariant:

- P_{mon} starts with the initial job and the $\text{Weight}(P_{mon}) = 1$
- Processes get part of the job along with part of the Weight
- If a job is done, the corresponding weight is sent back to P_{mon}

Result: $(\text{Weight}(P_{mon}) = 1) \implies \text{System is terminated}$

Termination Detection Algorithm (Huang)

```

STATE := idle; permitted_splits := S; W := 0;                                (0)
    /* fixed number S avoids non-terminating distribution */

DO
    (non-deterministic choice among matching alternatives)

    rcv( $P_{from}$ , WORK, Weight) → W := W+Weight;      /* may happen anytime */ (1)
        IF (STATE==idle) THEN STATE := active; FI;

    rcv( $P_{from}$ , CTRL, Weight) → W := W+Weight;      /* Monitor  $P_{mon}$  only */ (2)
        IF (W==1) THEN snd( $P_{out}$ , TERM); STATE := idle; FI;

    (STATE == active) → /* at least once, at most S times work distribution */ (3)
        IF (permitted_splits > 0) THEN permitted_splits := permitted_splits-1;
         $\langle W_1, W_2 \rangle$  := split(W);      /* where  $W_1, W_2 > MIN \wedge W_1 + W_2 = W$  */
        W :=  $W_1$ ; P := choose( $PS \setminus \{self\}$ ); snd(P, WORK,  $W_2$ );      FI;

    (STATE == active) → /* Do Real Work */ (4)
        perform(WORK);

    IF (NOT(monitor())) THEN snd( $P_{mon}$ , CTRL, W); W := 0; STATE := idle; FI; (4a)

OD;

```

Assumption: initial msg ($P_{out}, Work, 1$) is received by P_{mon} in (1)

Outline of Correctnes and Termination Proof

1. P_{out} initializes algorithm with weight 1 and activates P_{mon} in (1)
2. split is a loss-less division \implies no weights are lost
3. **Invariant:** $W_{mon} + W_{active} + W_{work} + W_{ctrl} = 1$ (after init)
 - W_{active} \approx Sum of weights from all P_i processes except P_{mon}
 - W_{work}, W_{ctrl} \approx Sum of weights of corresponding messages
4. $P_i \in PS \setminus \{P_{mon}\}$ active $\iff W_{P_i} > 0$
 - initially: $W = 0$ and STATE = idle
 - (idle \mapsto active): only in step (1) adds Weight > 0
 - (active \mapsto idle): only in step (4a) and $W := 0$ afterwards
5. Algorithm terminates $\iff P_{mon}$ sends TERM to P_{out}
 Invariant 3. where $W_{active} + W_{work} = 0$; $W_{ctrl} = 0$ after TERM

Termination: perform/msg-transfer times **finite** (Assumption)

- Only finite number of split steps in (3) due to ($MIN > 0$)
- No infinite delegation cycles in (3;1) due to limit S

VI.5.2 Distributed Deadlock Detection

Resources: '*Everything that processes compete for*'

- ▶ devices scheduled/data provided by the operating system
Transparency \implies local as well as remote devices, data, ...
- ▶ Synchronization: Access to critical sections
- ▶ Messages/RPCs: blocking wait for a message or reply

see
also
REST

VI.2.2

Resources also an issue in traditional OS:

- OS Usage Protocol: Request – Wait – Hold – Free
- Deadlock problem if scheduling is bad or too optimistic
Prevention vs. Avoidance, Detect and Resolve, Ignore

[A. Tanenbaum, 1995]:

Deadlocks in distributed systems are similar to deadlocks in single-processor-systems, only worse.

Goal: Efficient and Fair Resource Management

- ▶ **Global View** about state of all resources **required**
 - **centralized** is easy to control but an architectural **bottleneck**
 ⇒ all requests go through a **central Server/Manager**
 - **de-centralized** ⇒ problems w.r.t. **consistency of view**
 e.g., resource states change during message transfer times

◀ Problem for both models:

messages (requests, . . . , states) take too long, get lost, re-ordered

⇒ **General Problem requires handling of volatile states!**

c.f.
pg.
VI-70

Resource Model: (simplified for our algorithms)

- **non-consumable**, i.e. re-usable resources with **exclusive access**
- **no** multiple equivalent instances of resources (type vs. instance)

VI-78

System Models for Processes PS & Resources RS

System State assignment of internal/external resources to processes that are either internal or on remote nodes

1. Resource-Allocation Graph $(PS \cup RS, E_{req} \cup E_{assign})$

- $E_{req} \subseteq PS \times RS := \{(P_i, R_j) \mid P_i \text{ waits for } R_j\}$ (**requests**)
- $E_{assign} \subseteq RS \times PS := \{(R_j, P_i) \mid P_i \text{ holds } R_j\}$ (**assigned**)

Detailed: where are resources and who waits for which resource

2. Wait-For-Graph (PS, E) where $E \subseteq PS \times PS$ and $(P_i, P_j) \in E \iff \exists \text{ Resource } R \text{ s.t. } P_i \text{ is waiting for } R$ $\text{and } P_j \text{ holds } R$

Abstraction: hold \approx unambiguously
wait \approx maybe more than one process waits?

Deadlock $D \subseteq PS \iff (D, E \cap D \times D)$ contains at least one cycle

Note: Cycle implies deadlock iff there is only one instance/resource
 \implies *Deadlock Detection is done via Cycle Detection*

Problem: RAG vs. WFG – Multiple Instances

Legend:

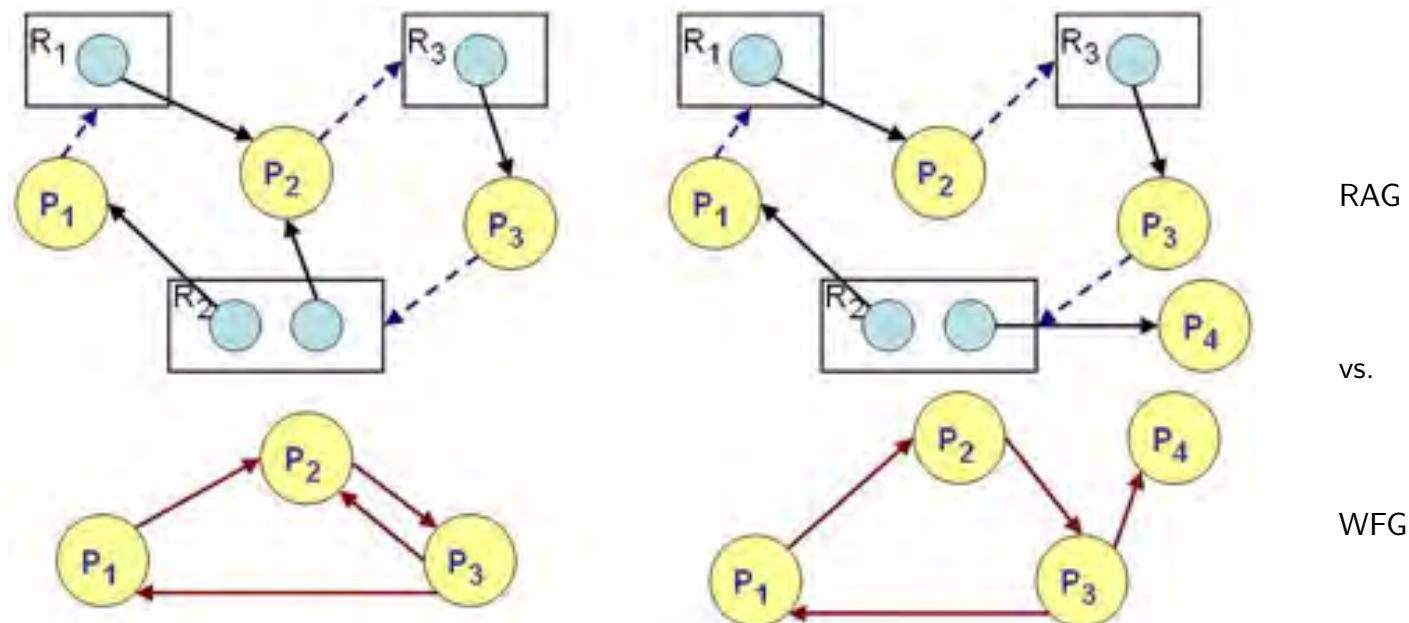
Circle $\in PS$

Square $\in RS$

Circle in Square
 $\approx RS\text{-Instance}$

Edge (PS, RS) request

Edge (RS, PS) assign



WFG describes Wait-Relation, but no detailed cause w.r.t. resources

- more compact \implies easier to find cycles
- ◀ cycles may imply **false deadlocks**
 - ▷ **left:** conflict establishes cycle and a deadlock
 - ◁ **right:** conflict establishes cycle but *no deadlock* (P_4 may end)

Reason: Multiple resource instances imply **OR**-Relation for wait

Distributed Deadlock-Handling: Detect & Resolve

◀ **Centralized Solution:** RAG/WFG in a global control process

► **Adopted Centralized Solution:**

1. **Local** Resource-Allocation-Graph in each process

VI-80

2. **Controller** combines local RAGs for detection; updates **either**:

(a) $P_i \in PS$ send always all changes to Controller

push

(b) $P_i \in PS$ send local RAGs in fixed intervals

push

(c) Controller requests local RAGs on demand

pull

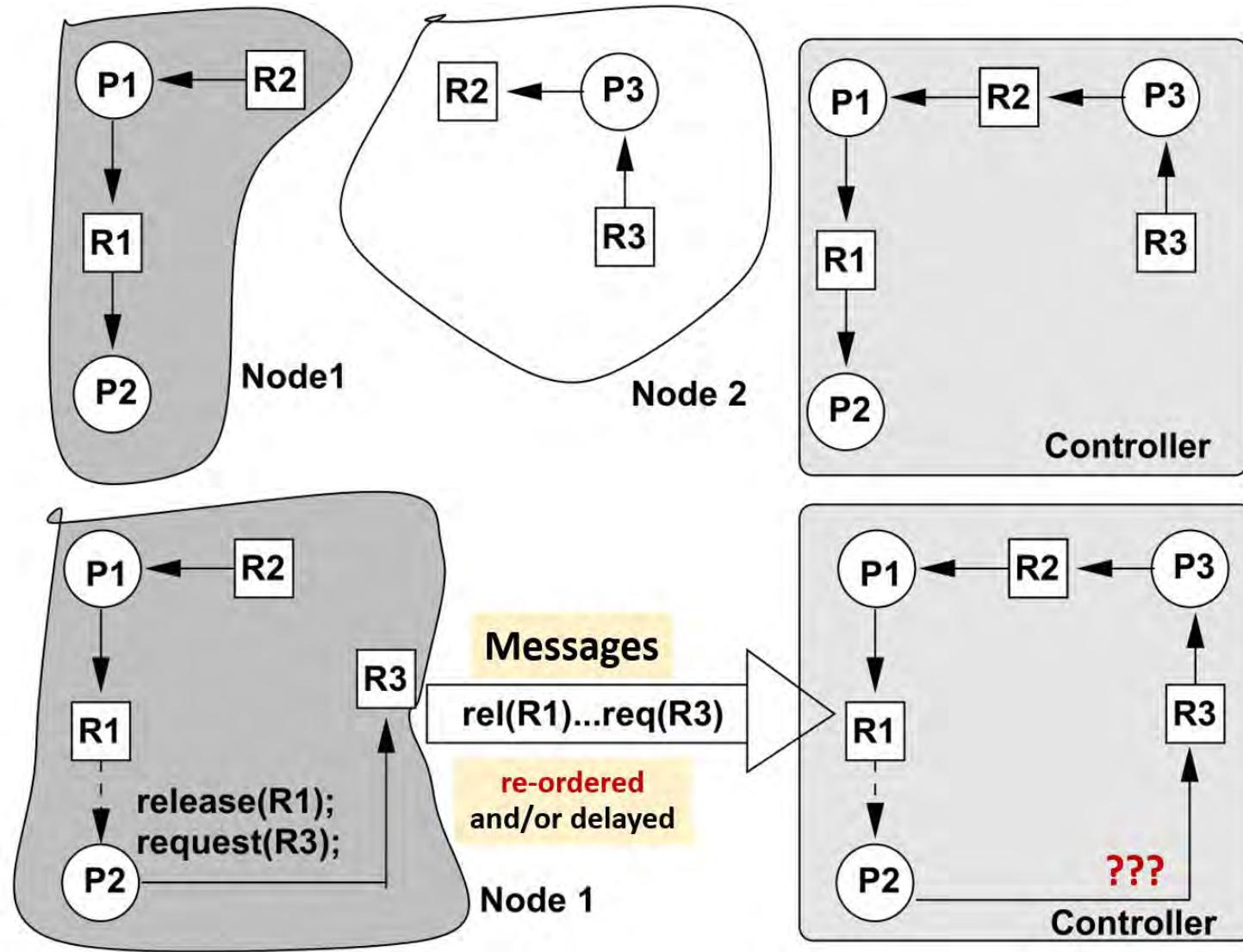
Problem: **false deadlocks** due to message timings \implies

▷ ordered msg channels or request updated infos (timestamps)

▷ Controller holds 2 global RAGs and computes AND for all edges

\implies 2-Phase Ho-Ramamoorthy (1982)

► **Distributed Solution:** distributed construction of Wait-For-Graph
out of local WFGs and **Edge Chasing Algorithm** VI-82
 \implies Chandy-Misra-Haas (1983)



False Deadlock identifiable: P_2 frees R_1 ; Termination Order $P_1 \rightarrow P_3 \rightarrow P_2$

Distributed Edge-Chasing–Algorithm

- ▶ Processes: request of multiple resources in a single request
 - ⇒ Process may wait for more than one other process
- ▶ Distinction: Waiting for **local** vs. **remote** resources
- ◀ No Controller ⇒ No global knowledge in a single process

Algorithmic Idea: How to detect cycles?

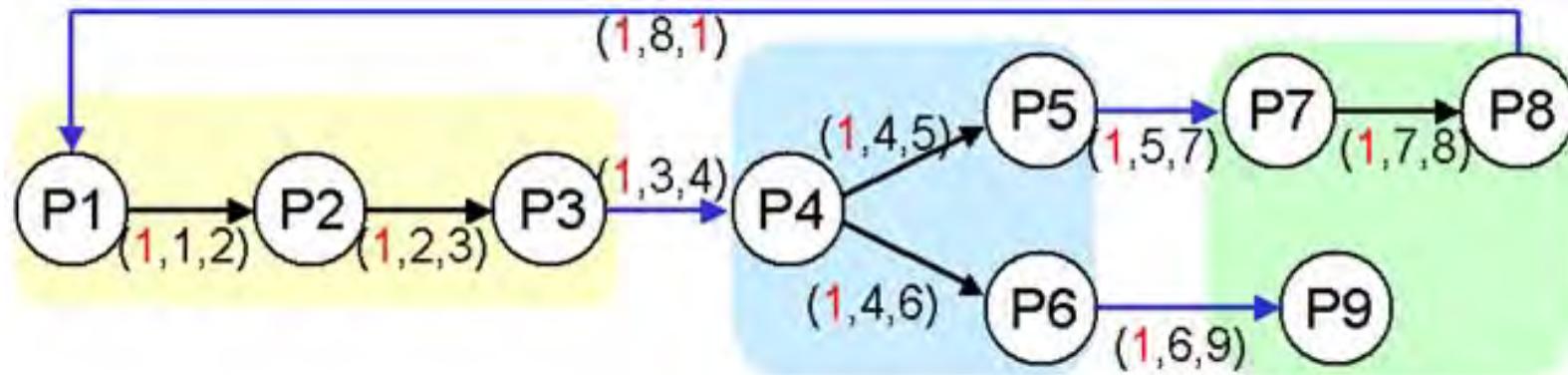
If process waits some defined time for a remote resource (blocked)
⇒ Deadlock is 'suspected' ⇒ Msg exchange starts

- ▶ **Probe Msg:** $< i, j, k >$ \approx $<$ Initiator, Sender, Destination $>$
- ▶ **Procedure:**
 - **Initiator** P_i sends Msg to all processes he **waits** for
 - also **waiting** destinations send msg to all those they wait for
 - If process P_i receives a message, it initiated ⇒ **Deadlock**
- ▶ **Optimization:** local waits are handled by local graph, not msgs
msgs store route as a hint for deadlock resolution

VI-82

Expl.: Edge-Chasing using three distributed Nodes

- local messages are avoided by looking into local Resource Graph
- msgs between nodes: explicit message-passing



Advantage: Construction of 'global knowledge' On Demand

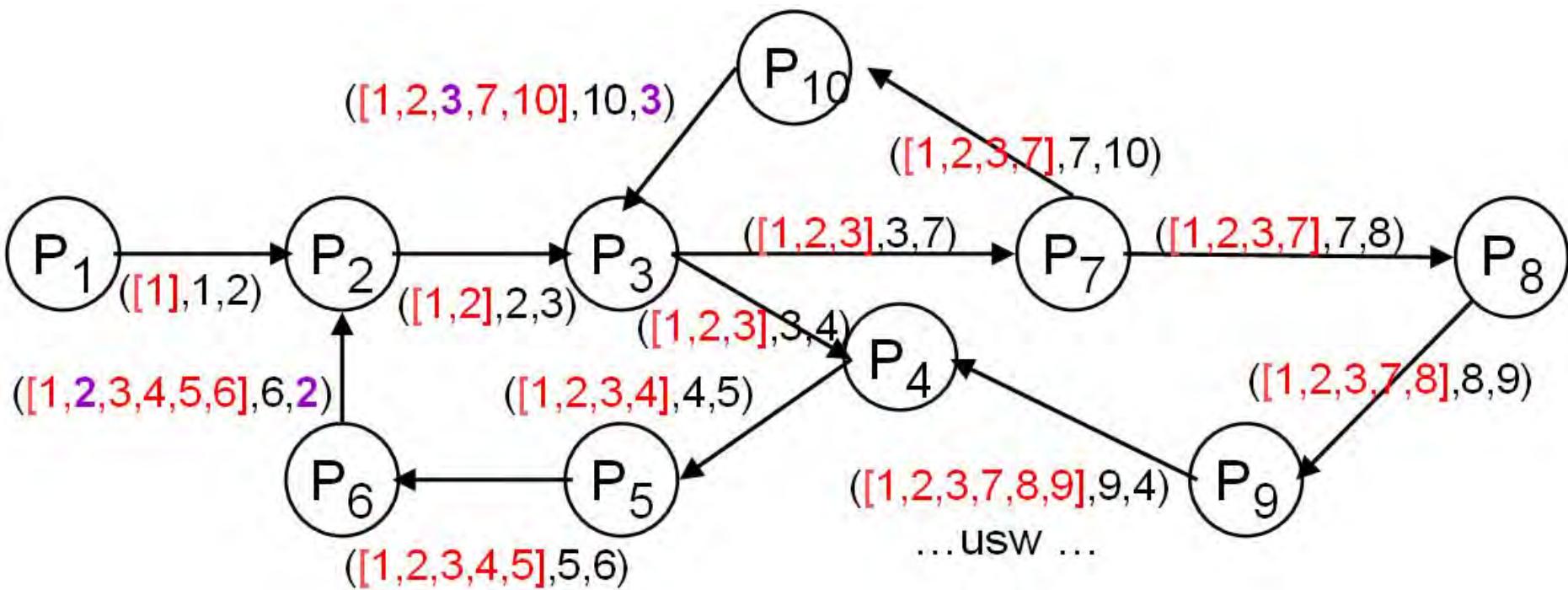
- Times for messaging not critical: processes are blocked anyway
- ◀ Waiting processes have to react to inquiry (watchdog)
- single local controller with local RAG per node

Thread

Remark: *lots of different algorithms in literature*

Shin-gal et al.

Expl.: Optimized Edge-Chasing stores Routes



Method: Propagate list of all nodes visited
Each node check whether it is the list

- Deadlocks are found independent of initiating process
- ◀ If multiple deadlocks are found \implies handling more complicated

Resolution: Preempt & Withdraw Resources

◀ **Problem:** preempted process loses work already done

$\text{recovery-points} \neq \text{process start} \implies \text{costly}$

Acceptable iff recovery mechanisms are required anyway

VI.4

► **Distributed Databases:** Transaction Management and Logging

- Precondition: unique time stamps used for prioritization
 $t_{P_1} < t_{P_2} \implies P_1 \text{ older than } P_2 \implies P_1 \text{ loses more work}$
 $\implies \textbf{Older Process will not be terminated!}$

• **How to avoid cyclic 'killings'?**

△ **Wait-Die:** wait only for **higher** time stamps

$P_{old} \rightarrow P_{young} \approx \text{wait} \mid P_{young} \rightarrow P_{old} \approx \text{kill}(P_{young})$

Problem: $(\text{kill}; \text{ restart})^+ \implies \text{thrashing}$ effect

▷ **Wound-Wait:** wait only for **lesser** time stamps

$P_{old} \rightarrow P_{young} \approx \text{kill}(P_{young}) \mid P_{young} \rightarrow P_{old} \approx \text{wait}$

Advantage: kill; restart; wait much less overhead

VI.5

Remark: Lots of literature in distributed OS and DBS.

VI.6 Distributed Coordination

- ▶ Arbitrarily distributed application system with *distributed*
 - * *control* to ensure robustness without bottlenecks
 - * *data* due to replication-based transparency
 - * *compute load* due to efficiency and feasibility reasons
- ◀ Basic level of agreement and consensus essential for, e.g.,
 - * mutual exclusion and consistency mechanisms for replicated data
 - * centralized/de-centralized organization of common decisions

Typical coordination problems in distributed systems

1. **Election:** Determine new unique coordinator after crashes
2. *Agreement* about common 'global' state in the context of errors
3. *Commitment* about executing collective actions, e.g. *transactions*

additional
MSc
topics

Note: Dynamic process systems/groups with ever changing process numbers are especially 'hard', e.g., mobile or P2P systems.

VI.6.1 Leader Election

Motivation: Leader Election is the basis for

- ▶ distributed algorithms that use some kind of centralized organization, e.g., mutual-exclusion, resource handling, deadlock or termination detection
- ▶ fault-tolerance in these algorithms by allowing for de-centralized replacement of crashed coordinators, servers etc.

⇒ **Requirements for any Election Algorithm:**

1. may be initiated by **any** process in the system
2. may be initiated **in parallel** by different processes

Reason: a priori unknown when and where a process crashes

3. terminates always appointing **exactly one Leader**
i.e, all processes know their state w.r.t. election: { leader, lost }

System Model for Leader Election

- ▶ **Lossless** message transfer
- ▶ Each $P_i \in PS = \{P_1, \dots, P_n\}$ knows about **all 'names'** in PS
- ▷ All processes $P_i \in PS$ are **a priori equal**
 - ⇒ **Each process** is able to act as a coordinator
- ▷ **Priorities** are defined by arbitrary **total, strict order** $<$ on PS
e.g., totally ordered process indices $P_1 < P_2 < \dots < P_n$
 $\forall M \in \wp(PS)$ is $P_i \in M$ assigned the highest priority : $\iff i = MAX(\{j \mid P_j \in M\})$ *

realistic ?

*

At any time, PS is partitioned into two process sets:

1. $UP(PS) \approx$ active processes
2. $DOWN(PS) \approx$ crashed processes

State changes are **spontaneous and not coordinated**

Caution: Even change from *crashed* → *active* is **not** predictable

Additional Challenges

- ◀ **Distinction:** crashed nodes vs. lost messages \implies lossless communication with **maximum response time** t_{max} (msg transfer plus time to react, e.g., via 'watchdog' thread) known
- ◀ **Elections may be initialized in parallel**
 - set of nodes waiting for a response governed by a **timeout**
 - e.g., access to a **critical section**: missing WAIT/OK messages
- ◀ **Formerly Leader is re-started** and becomes active again
 - Who is assumed to stay/become coordinator in this case?

System structure and fault tolerance

1. **complete** reachability \implies *broadcast*
2. **partial** reachability \implies special protocols needed
 - Example: Algorithms based on ring structures

Note: Partitioning due to channel crashes is most critical

c.f.
III-66

Challenge: Determine a **single**, unique coordinator, s.t.

all $P_i \in UP(PS)$ **agree in**

the name of the (new) coordinator process

Typical Algorithms: *Bully*-Algorithm

Ring-Algorithm

c.f.
pg.
VI-93

Bully Algorithm (Garcia-Molina 1982)

- ▶ **Precondition:** Each active process is able to reach **all** others
- ▶ **Idea:** Prioritize the process with the highest process index
- ▶ **Method:** timeout in a process \implies
 - initiate re-election for all processes with higher indices
 - stop re-elections from processes with lower indices
 - single process that has highest index \implies is elected

Note: start of multiple re-elections in parallel possible
after re-start, the 'former' coordinator is favored to become
the new coordinator again ('bullies' the current coordinator)

Each active process P_i reacts to the following 6 types of events:

1. **Time-out** of a msg to (last known) coordinator \rightarrow GOTO 2.

2. **Initiate re-election:**

```
FORALL { $P_j \in PS \mid j > i$ } DO snd( $P_j$ , VOTE) OD; /* higher up? */
ELECTION := true; RESPONSE :=  $\emptyset$ ; Coord := undef;
Wait( $T$ ) for EVENTS of { 3, 5 };                                (timeout T)
IF (RESPONSE ==  $\emptyset$ )                                         /* all higher procs down */
  THEN FORALL { $P_j \in PS \mid j < i$ } DO snd( $P_j$ , OK) OD; /* self */
  ELSE Wait( $T'$ ) for EVENTS OF { 4, 5 };                      /* higher elect */
    IF (Coord == undef) THEN 2. FI
ELECTION := false;
```

3. $rcv(P_j, WAIT) \rightarrow$ RESPONSE := RESPONSE $\cup \{P_j\}$;

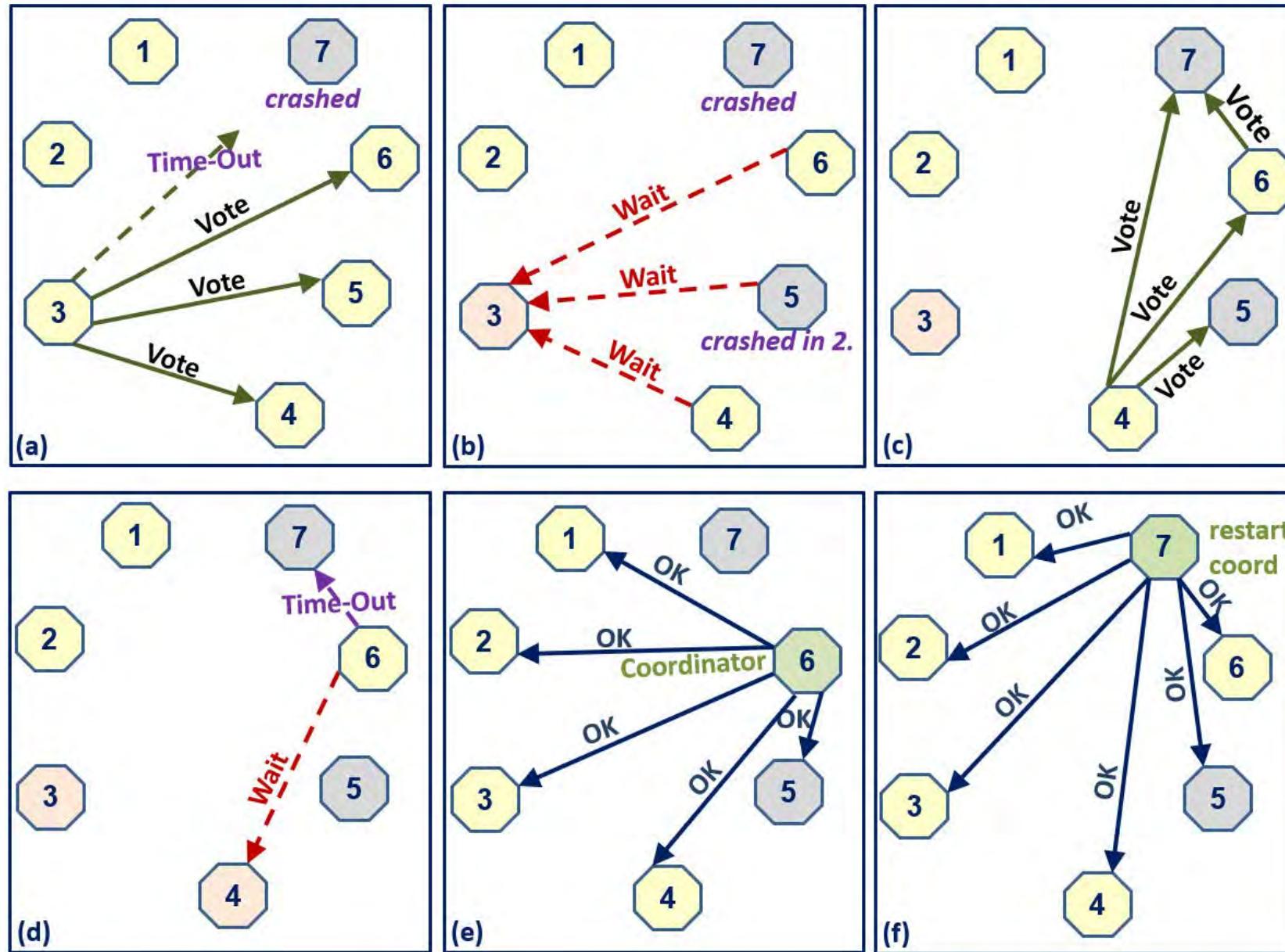
4. $rcv(P_j, OK) \rightarrow$ Coord := P_j ; /* new coordinator? */

5. $rcv(P_j, VOTE) \rightarrow$ snd(P_j , WAIT); /* stop 'lower' election */
 IF NOT(ELECTION) THEN 2. FI; /* vote */

6. **Recover** (from crash) \rightarrow GOTO 2.

Note: Process crashes after WAIT \implies never 4.: re-start election

Example: Bully Algorithm



Assessment of Bully Algorithm

- ▶ Nomen est Omen; very simple structure
- ▶ works despite multiple node crashes during election
- ▶ parallel elections cause no problems and will be stopped 'early'
 \implies processes with higher process indices determine overall result
- ▶ a single run terminates always due to finite number of messages

- ◀ Lots of crashes \implies triggers incessantly re-elections!
 \implies Determination of **timeouts** is critical for success

- ◀ **Costs** $\mathcal{O}(|PS|^2)$ messages (*worst-case*):
 UP = PS ; P_n crashes;
 P_1, \dots, P_{n-1} observe crash in parallel and start $(n - 1)$ elections;
 P_n is restarted again
 $\sum_0^{n-1} \text{VOTE} + \sum_0^{n-1} \text{WAIT} + (n - 1) \text{ OK}$ messages

Conclusion: rather costly w.r.t. number of messages exchanged

Ring Algorithm

1. **Naive** version using direct neighbors in a ring
2. **Efficient** version collecting information in a logical tree structure

Ring-Algorithm (LeLann 1977)

- **Precondition:** uni-directional ring for communication among active processes $P_1 \mapsto P_2 \dots \mapsto P_n \mapsto P_1$
 $P_i.next()$ determines current next **active** neighbor $\in UP$
 \implies strong precondition used here!
- **Idea:** Use maximum index in **list of processes** w.r.t. ring direction
- **Method:** timeout in process $P \implies$
 - initiate re-election; start with singleton list $[P]$
 - handover and extend list
 - after complete circle \implies propagate coordinator list

Note: start of multiple re-elections in parallel possible
restart of a former coordinator initiates new election

LeLann Ring Algorithm

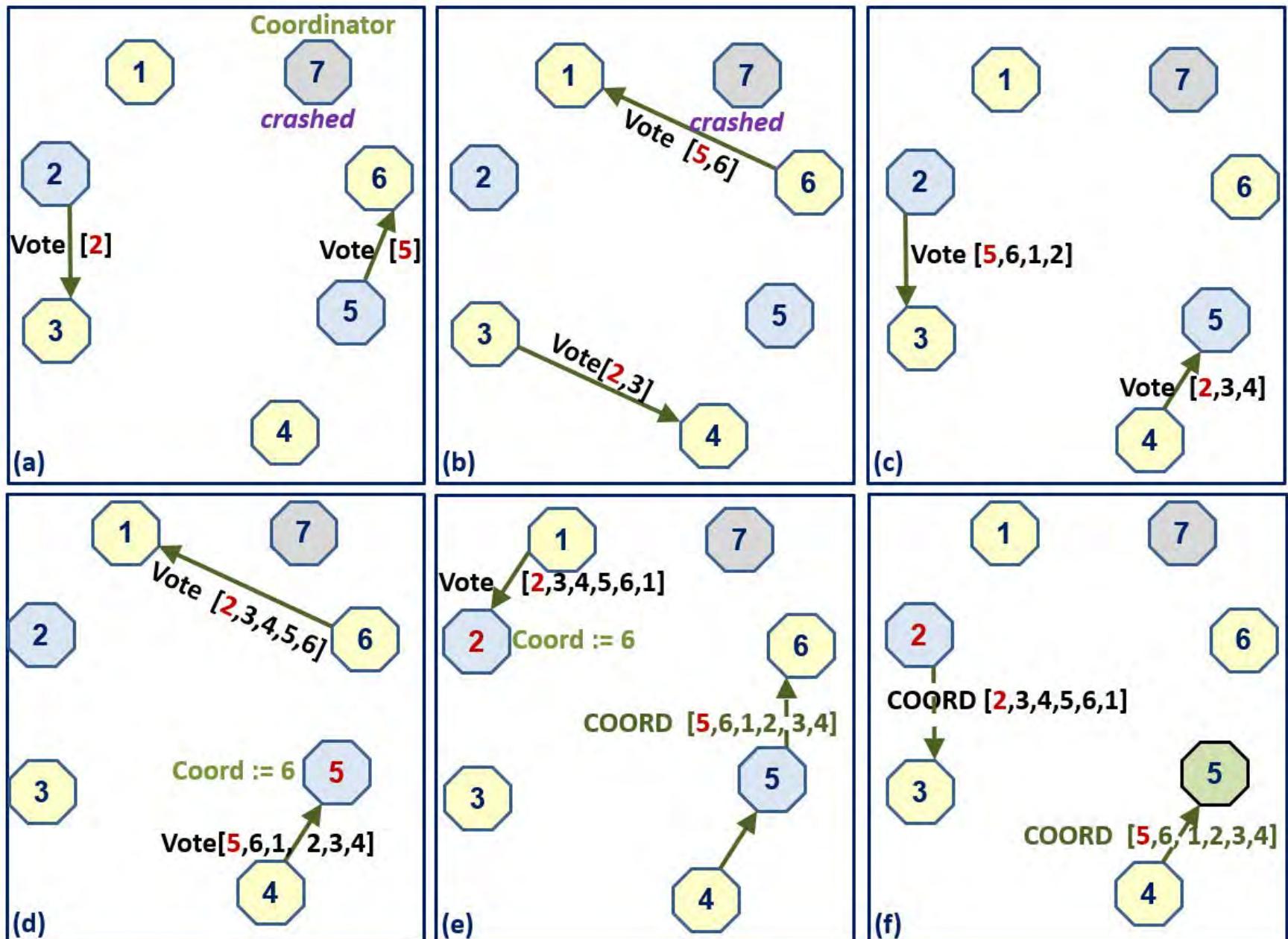
Each active process P_i reacts to the following events:

1. **Time-out** of a msg to (last known) coordinator \rightarrow GOTO 2.
2. **Initiate Election:** $\text{snd}(P_i.\text{next}(), \text{VOTE}[i]);$
3. $\text{rcv}(P_j, \text{VOTE}[List]) \rightarrow$ $\/* \text{transform} */$
 IF $i \in List$ THEN $\text{snd}(P_i.\text{next}(), \text{COORD}[List]);$
 ELSE $\text{snd}(P_i.\text{next}(), \text{VOTE}[i : List]);$
 FI; $\/* \text{extend} */$
4. $\text{rcv}(P_j, \text{COORD}[List]) \rightarrow$ $\/* \text{term/propagate} */$
 $\text{Coord} := \text{MAX}([List])$
 IF $i \neq List.\text{front}()$ THEN $\text{snd}(P_i.\text{next}(), \text{COORD}[List]);$
5. **recover** after crash \rightarrow GOTO 2. $\/* \text{no privileges} */$

Note: $VOTE \approx \text{ask around}$; $COORD \approx \text{propagate result}$

Termination: initiating process aborts further propagation

Example: LeLann Ring Algorithm



Assessment of LeLann Algorithm

- ▶ simple structure
- ▶ parallel elections cause no problems and achieve common result MAX
- ◀ **Costs:** each election requires $2*(|PS|-1)$ messages
 - worst-case $(|PS|-1)$ parallel election procedures
 - $(|PS|-1) * (2*|PS|-1))$ messages $\implies \mathcal{O}(|PS|^2)$

Conclusion: rather costly and requires optimizations

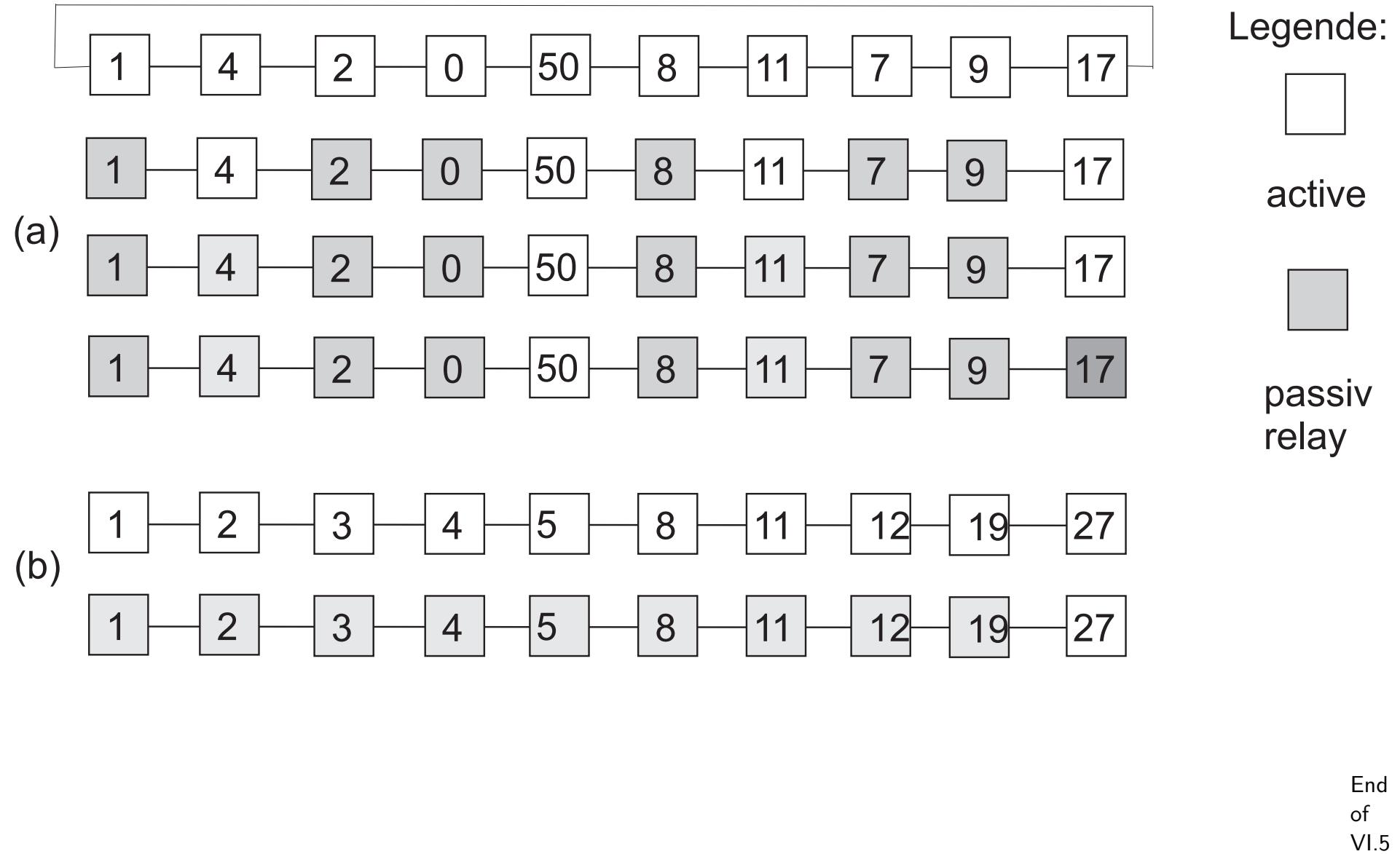
Worst-case Optimization: (Peterson 1982)

- tree-like reduction of ring structure for parallel elections
- active \approx participate in election/ relay \approx transfer messages only
- each step cuts number of active processes in half (at least) via
 - bi-directional ring: $\text{MAX}(\{P_{i\ominus 1}, P_i, P_{i\oplus 1}\})$ (left/right)
 - uni-directional ring: $\text{MAX}(\{P_{i\ominus 2}, P_{i\ominus 1}, P_i\})$ (2 predecessors)

ACM
ToPLaS,
(10)
1982

c.f.
pg.
VI-97

Example: Peterson Ring-Election Algorithm



VI.6.2 Agreement Protocols – Outlook

Simplified Process Model

- PS with n Processes (nodes)
- **robust, loss-less direct communication** $\forall (P_i, P_j) \in PS^2$
⇒ no special handling based on node where error occurs

Reason: Network Partitioning in central node, e.g., star network

Error in **name server** vs. user node

No subsequent errors due to transitive message routing

ERROR (Fault): *Deviation of expected system behavior* in

- ◀ **General** systems, i.e., synchronous as well as asynchronous:
no or unexpected reaction of nodes or msg channels
- ◀ **synchronous** systems additionally via **timing** errors, e.g.,
clock drift exceeds tolerable limit; msg transfer times
time for computing or replies exceeds limit

VI-100

c.f.

I-33

Why is Agreement important in DS?

- ◀ **Quality of Service** assumptions are not met
 - Expl.: E-commerce, online orders ...
guaranteed delivery time exceeded; order not processed
 - ◀ **Distributed Algorithms fail** due to preconditions not met
 - ⇒ distributed infrastructure no longer robust
- Expl.: Algorithms waiting for msgs will block
Bully–Algorithm using faked process indices
- ⇒ **Fault tolerance is important**

- ▶ **Replication** of active and passive components
- ▶ Saving states as recovery points; Logging (transactions)
- ▶ **Reduce impact of faulty components**
 - * Determine which parts are faulty and which are ok
 - * Secure functionality of system parts working correctly

chap.
VII
VI.4

Errors and Faults: Causes and Classes

- **Connections:** lost, duplicated, corrupted msgs
- **Nodes:** different levels of impact for errors, esp.
 1. **Fail stop:** P_i **identifiable** permanent **down**
 2. **Crash fault:** P_i **not identifiable** permanent **down**
 \implies Process does not send or reply in the future
 3. **Send Omission fault:** not all msgs (to all destinations) are sent
 4. **Receive Omission fault:** not all msgs arriving are accepted
 Impact: content loss plus blocking in synchronous interaction
 5. **Byzantine fault:** Unpredictable behavior of nodes !!
 sometimes correct, sometimes faulty behavior
 omits/**manipulates** some msgs, even generates unexpected msgs

AGREEMENT: Make sure that **all non-faulty nodes** get always the correct information

\implies *Prohibit or Reduce the effects of faulty nodes.*

Agreement Levels for Byzantine Faults

1. Byzantine Agreement (single-source broadcast)

A **single** P_i propagates a fixed value v_i and **all** non-faulty processes agree on a **single value** V (If P_i is not faulty $\implies V = v_i$)

2. General Consensus (multiple-source broadcast)

Each P_i propagates **it's own** fixed value v_i (values may differ) and **all** non-faulty processes agree on a **single value** V
(If all v_i are the same in all non-faulty processes $\implies V=v_i$)

arbitrary
?

3. Interactive Consistency (multiple-source broadcast)

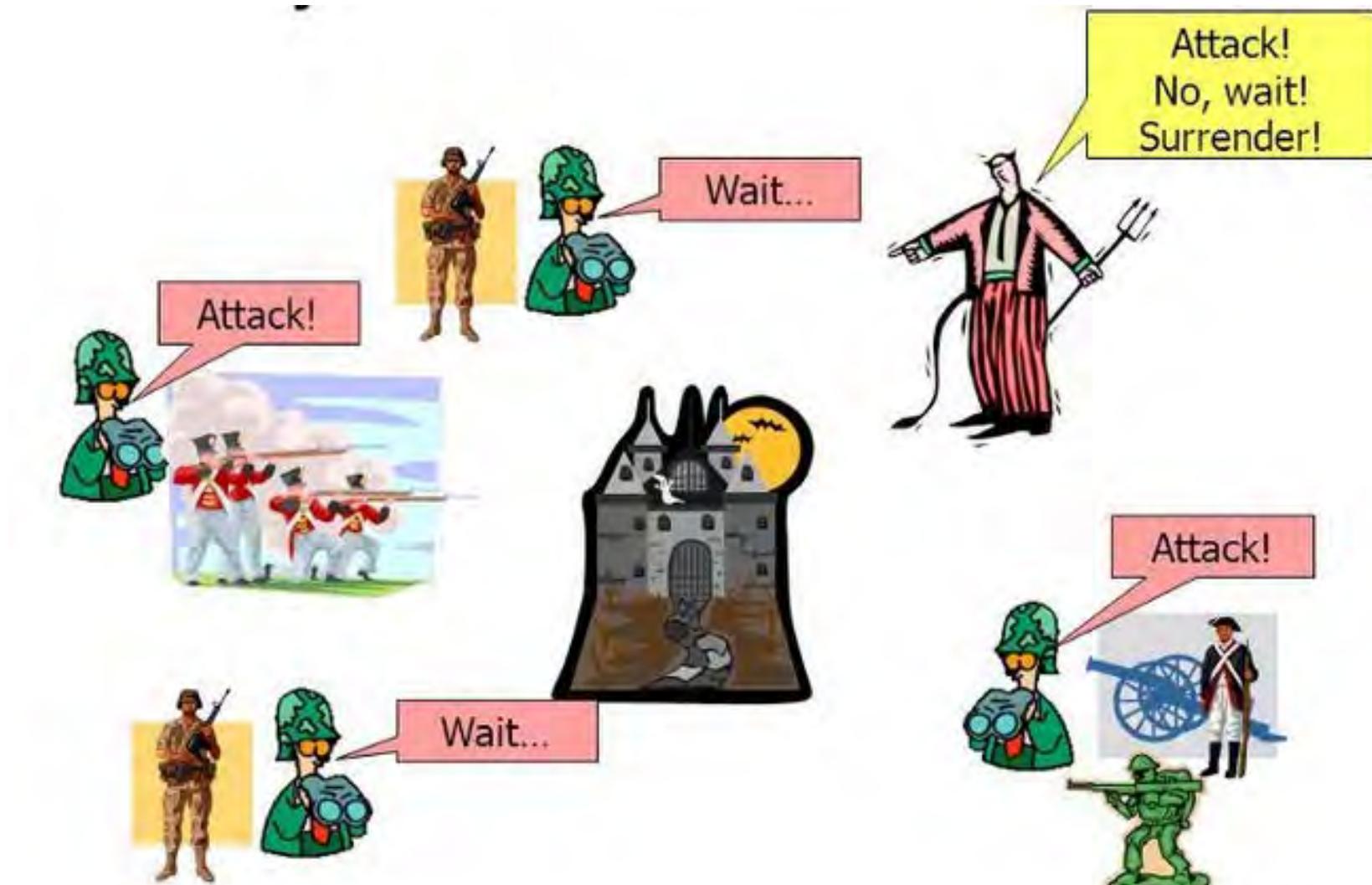
Each P_i propagates **it's own** fixed value v_i (values may differ) and **all** non-faulty processes agree on a **Vector** (V_1, V_2, \dots, V_n) of values (If P_i is not faulty \implies entry V_i of the vector V is v_i)

see
Colou-
ris
11.5

Note: Implementing 1. for all processes implies 3.

Implementing 3. plus a global majority function implies 2.

Byzantine Generals – 'History?'



From cs4410 fall 08 lecture

Fig.: gauthamzz.github.io/tendermint.html#byzantine-fault-tolerant

Byzantine Agreement - Basic Situation

- A single P_i wants to establish a consensus about value v_i in PS
- Arbitrary node failures are 'expected'; no messages lostVI-100
- All processes use the same symmetrical algorithm
- ◀ Messages may be manipulated/faked
- ▷ **No** use of authorization or signaturesVI-108

Algorithm: Exchange values to guarantee **forming a majority**

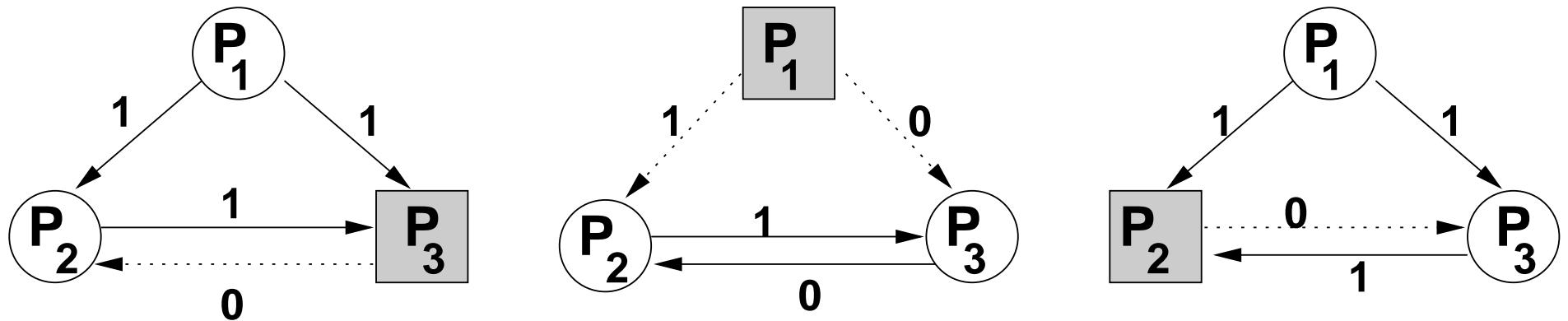
Problem:

- ◀ Nodes may manipulate msgs before forwarding
 - ⇒ multiple interaction (rounds) used to detect faked msgs
 - ⇒ message exchange is very costly
 - ◀ **Only** $m \leq \frac{n-1}{3}$ (*ceiling*) of **faulty processes** in n nodes are tolerable, i.e., $n = 3m + 1$ processes may compensate for m faultsVI-104
- Example: 1 out of 4 is ok; 1 out of 3 not;
2 out of 7 processes etc.

Example: 3 Processes with 1 Faulty Process

P_2 : P_1, P_2 ok, P_3 fault $\implies P_2$ has to choose P_1 value

P_2, P_3 ok, P_1 fault $\implies P_2$ has to choose P_1 value (same algorithm)



P_3 : P_1, P_3 ok, P_2 fault $\implies P_3$ has to choose value of P_1

P_2, P_3 ok, P_1 fault $\implies P_2$ has to choose value of P_1 (same alg.)

\implies **No Agreement** among P_2 and P_3 if P_1 is faulty

Generalization: ($m > 1$) reduces to 1-of-3 problem by *contradiction*

\implies general algorithm would work also for ($m = 1$)

Oral-Message-Algorithm: Lamport/Shostak/Pease

Method: Recursive Exchange as Remote Procedure Calls

► **Arbitrary Call** $\text{om}(m, \text{PROCS}, p)$

$m \approx \text{Max-Faults}$; $\text{PROCS} \subseteq PS$ participate; $p \approx \text{starts call}$

$\text{om}(m, \text{PROCS}, p) ::= \text{PROCS} := \text{PROCS} \setminus \{p\};$

FORALL $P \in \text{PROCS}$ DO $\text{snd}(P, \text{val}_p)$ OD; (1)

IF ($m > 0$) THEN

FORALL $P \in \text{PROCS}$ DO $\text{RPC}(P, \text{om}(m-1, \text{PROCS}, P))$ OD; (2)

FORALL $P \in \text{PROCS}$ DO $\text{RPC}(P, \text{majority}(P, \text{PROCS}))$ OD; (3)

FI;

$\text{majority}(P, \text{PROCS}) ::= \text{val}_p := \text{Majority value from all } P' \in \text{PROCS};$

► $P \in PS$ **reacts:** after $\text{rcv}(p, \text{val}_p)$ start own RPC call

► **initial Call:** $\text{om}(m, PS, P_{init})$ of Initiator P_{init} using $\text{val}_{P_{init}}$

Costs: $\text{om}(m, PS, p) \approx |PS|-1 \text{ RPCs } \text{om}(m-1, PS \setminus \{p\}, P) \dots$

Expl.: $m = 1$; 4 processes $PS = \{P_0, P_1, P_2, P_3\}$: $\text{om}(1, PS, P_0)$

1. Case: Initiator P_0 is non-faulty

(a) Let $\text{val}_{P_0} = 0$

P_0 sends 0 to $\{P_1, P_2, P_3\}$

$\text{val}_{P_1} = \text{val}_{P_3} = 0$

(b) in $P_1 \text{ om}(0, \{P_1, P_2, P_3\}, P_1)$

in $P_2 \text{ om}(0, \{P_1, P_2, P_3\}, P_2)$

in $P_3 \text{ om}(0, \{P_1, P_2, P_3\}, P_3)$

forward correct in P_1 and P_3

forward faked in P_2

(c) majority

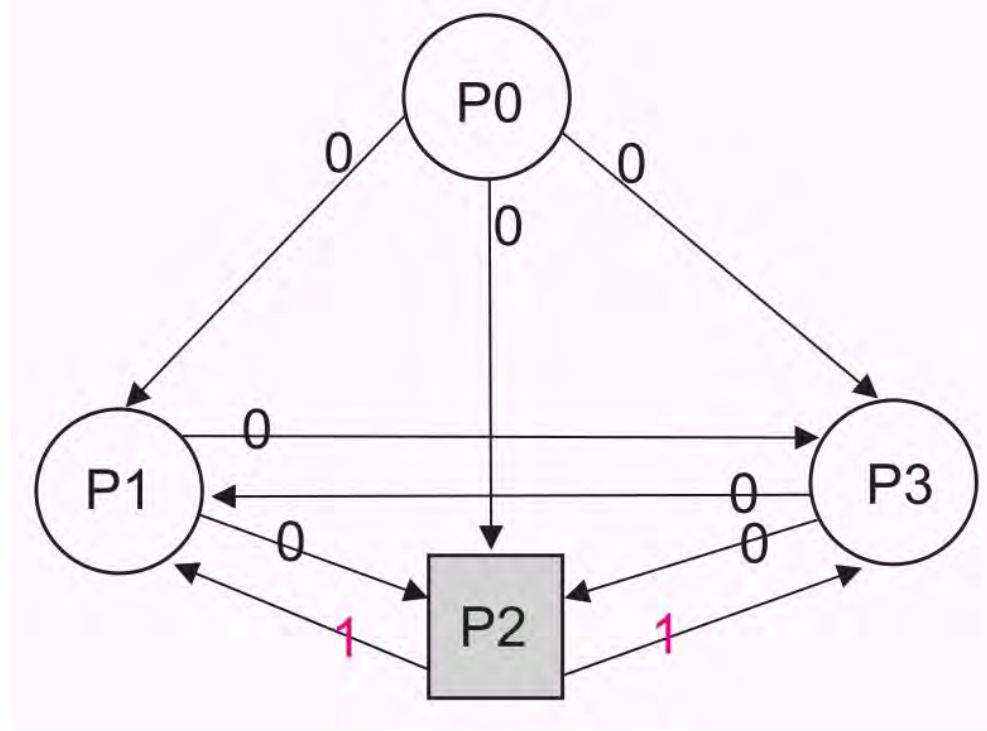
in P_1 using $<0, 1, 0> \implies 0$

/* values from $\{P_1, P_2, P_3\}$ */

in P_2 using $<-, -, -> ??$

in P_3 using $<0, 1, 0> \implies 0$

$\implies \{P_0, P_1, P_3\}$ agree on the **same value 0**



Expl.: $m = 1$; 4 processes $PS = \{P_0, P_1, P_2, P_3\}$: $\text{om}(1, PS, P_0)$

2. Case: **Initiator P_0 itself is faulty**

(a) arbitrary (unknown) value in val_{P_0}

P_0 sends 1 to $\{P_1, P_3\}$, but 0 to P_2

$\text{val}_{P_1} = \text{val}_{P_3} = 1$ but $\text{val}_{P_2} = 0$;

(b) in $P_1 \text{ om}(0, \{P_1, P_2, P_3\}, P_1)$

in $P_2 \text{ om}(0, \{P_1, P_2, P_3\}, P_2)$

in $P_3 \text{ om}(0, \{P_1, P_2, P_3\}, P_3)$

forward correct in P_1 , P_2 and P_3 ,

but different values val_{P_i}

(c) majority

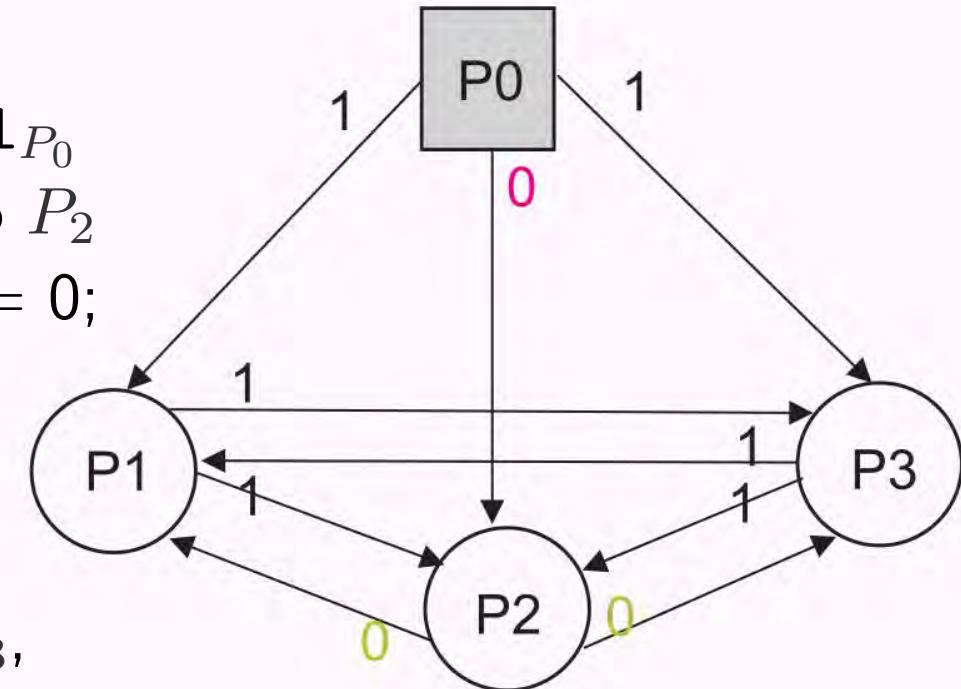
in $P_1 \text{ mit } <1, 0, 1> \Rightarrow 1$

/* values from $< P_1, P_2, P_3 >$ */

in $P_2 \text{ mit } <1, 0, 1> \Rightarrow 1$

in $P_3 \text{ mit } <1, 0, 1> \Rightarrow 1$

$\Rightarrow \{P_1, P_2, P_3\}$ agree on the **same value 1**



Algorithms: $3m + 1$ Processes – m tolerable faults

► Lamport/Shostak/Pease 1982

$m + 1$ Exchange rounds; overall $\mathcal{O}(|PS|^m)$ Msgs

► Dolev/Reischuk: $2m + 1$ Rounds; $\mathcal{O}(|PS|*m + m^3)$ Msgs

► Gary/Moses (1993): $m + 1$ rounds; number of msgs polynomial

⇒ m tolerable faults require $m + 1$ deterministic exchanges

⇒ Trade-Off: Number of messages vs. Number of Rounds

⇒ *Algorithms are too costly for most situations*

ACM
ToPLaS
VI-105

JACM
1985

Fischer/
Lynch
1982

More Efficient Solution ⇒ Advanced System Model

► Use forgery-proof **Signatures** for all messages

► Protect message channels from eavesdropping and manipulation

⇒ *Fakes when forwarding messages can be detected*

Expl.: Signatures allow even for 1-out-of-3 solution

Exchange $P_2 \longleftrightarrow P_3$ exposes inconsistent msgs from P_1

VI-104

End
VI.6.2

VI.6.3 Outlook: There are a lot more Algorithms

- Coordination in flat and nested transactions: 'Commit Protocols'
 - Detecting and guaranteeing globally valid predicates
 - Distributed Ledger Algorithms (BitCoin et al.), ...
-

Some Issues for further Studies

Complexity: Trade-off between preconditions, robustness and costs.

Correctness: Important issue for safety-critical distributed systems,
but especially hard to tackle due to *State-Space explosion*.

Other System Models: Normally do not meet such favorable pre
conditions, e.g.,

- * Indirect, even non-transitive, connectivity among nodes, e.g., in
mobile or very heterogeneous systems
- * Modern P2P systems: no stable structure and high churn rates

Conclusion – Distributed Algorithms

- ▶ **Overall Goal:** *Compensate typical deficits of distributed systems through additional algorithmic layers.*
- ◀ **To some extent achievable but there are limitations:**
 - ◀ Asynchronous systems with unpredictable error rates are barely manageable for practical applications.
 - ◀ Erroneous message channels are hard to overcome at all.
 - ◀ Constantly crashing nodes (processes) render productive work more or less impossible.
 - ⇒ both result in a '*trade-off*' between
 - **blocking** and even deadlocks due to long waiting periods
 - **life-locks** due to timeouts and permanent re-start of algorithms
 - ◀ Algorithms often costly, esp. number of msgs to be exchanged.

Practical Distributed System Development:

- * 'Hide' preconditions by using *Middleware* with QoS guarantees.
- * Confine algorithms to 'stable' settings, i.e., server environments.

VII. Replication and Transparency

Transparency: *Abstraction from ...*

I-36

I-37

- ◀ peculiarities and deficits of underlying hardware
- ◀ hard to handle characteristics of geographical distribution
- ▶ methods to ensure portability and standard conformance of systems
- ▷ methods to ensure reliable message transfer or encryption etc.

... through the use of intermediate software layers that

- ▶ provide a more comfortable 'system/programming model'
- ▶ let the user rely on features without worrying about implementation

\implies *Lots of arguments are in favor of transparency*

Disadvantages:

- ◀ Implementing 'good' and portable middleware itself is hard.
- ◀ Systems may get 'slower' and costlier at runtime due to overhead.
- ◀ Most middleware systems provide programming models of their own and require more discipline among users and developers.

Two Main Reasons for Replication

- **Reliability:** *Failures and malfunctioning components of a DS are not visible to the user but will be compensated internally.*

- ▶ Replication of components is indispensable for any robust system.
- ▶ Heterogenous Replication allows for diversification for hardware
- ▶ Cross-Region Replication reduces network partitioning impacts

⇒ *Abstraction* from detailed hardware combined with migration & relocation transparency allows for exchanging components.

VII-3

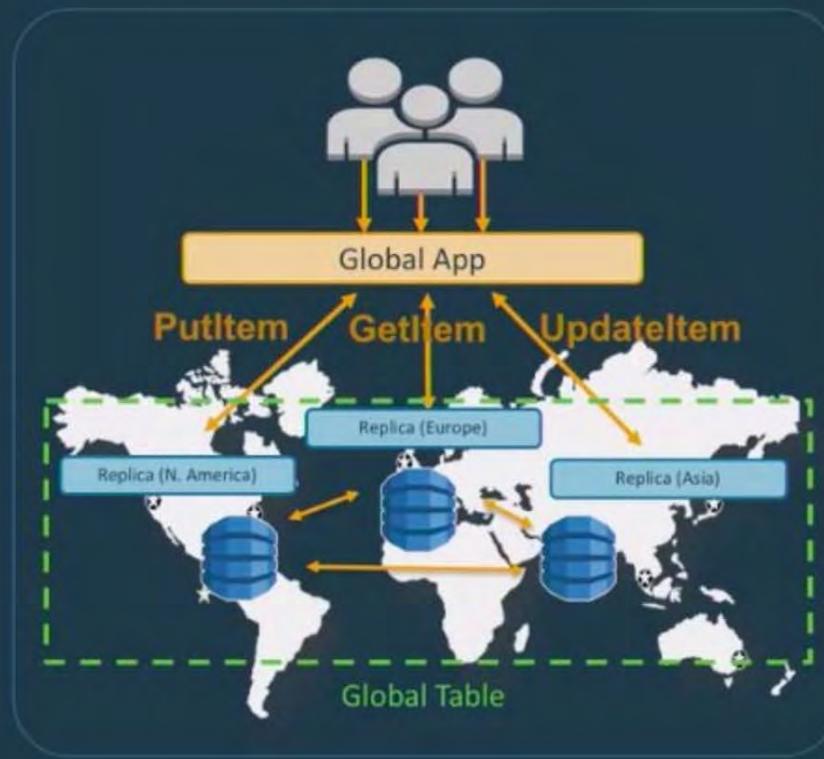
VII-5

- **Performance and Scalability:** *Distribution does not impede the perceived performance for users. A distributed system should scale for high loads as well as for load variations.*

- ▶ **Replication** is also the key technique for performance:
 - * Data replication and *Caching* for fast access
 - * Service replication based on current system load
 - * 'Service Pools' using on demand activation

Example: AWS Replication in three Regions

Global Tables



Replicate table across multiple regions

Read and write any items in any region

Eventual convergence

Conflicting updates resolved with “last writer wins”

- Load-Balancing for different regions locally
- Hand-Over in the case of network problems
⇒ Copying of data and handling of changes

Fig.:
pbs.
twimg.
com/
media/
DmTn
4aVW4
AA8
dNM
.jpg

Example: 2 Replicated Service Regions - Running

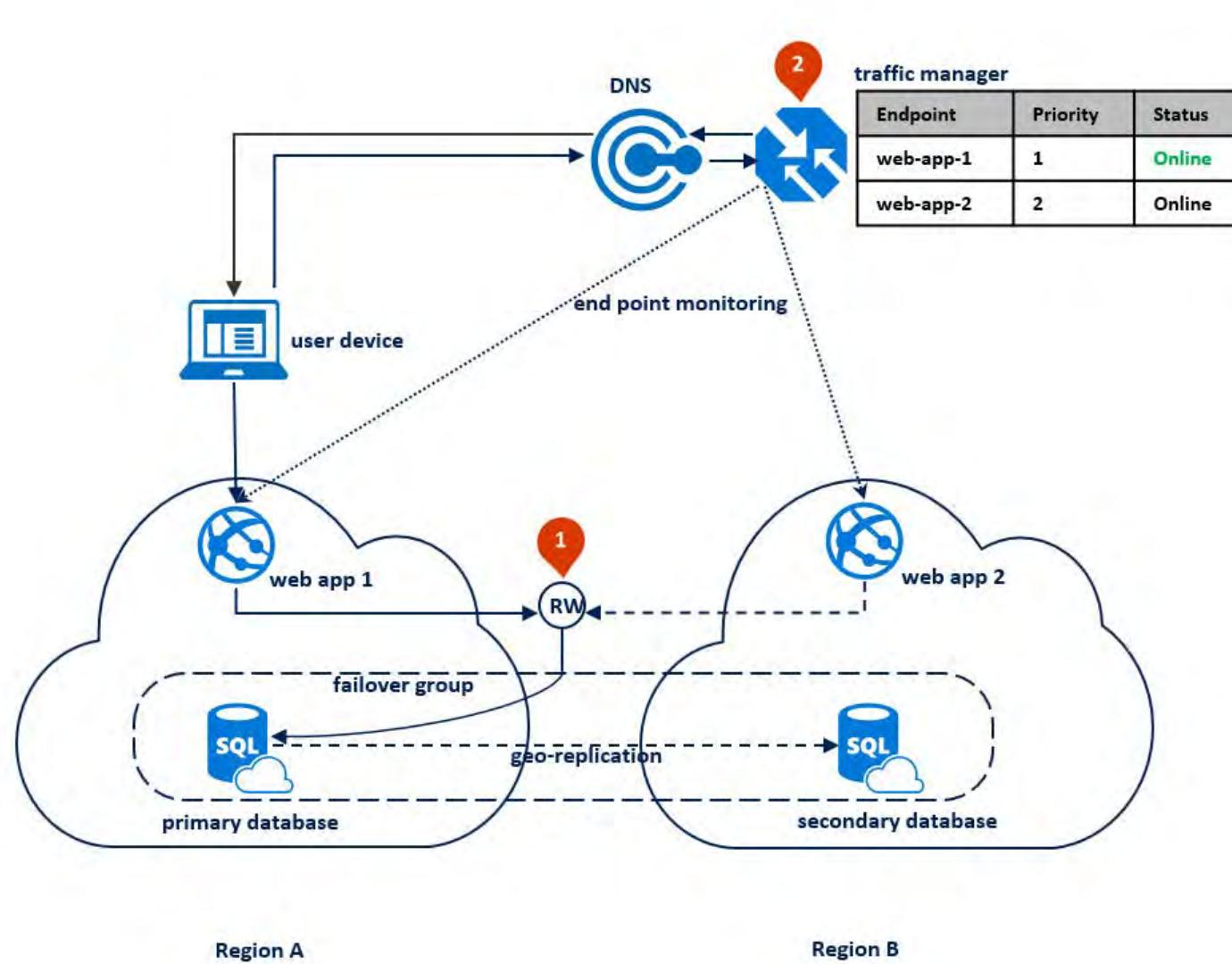
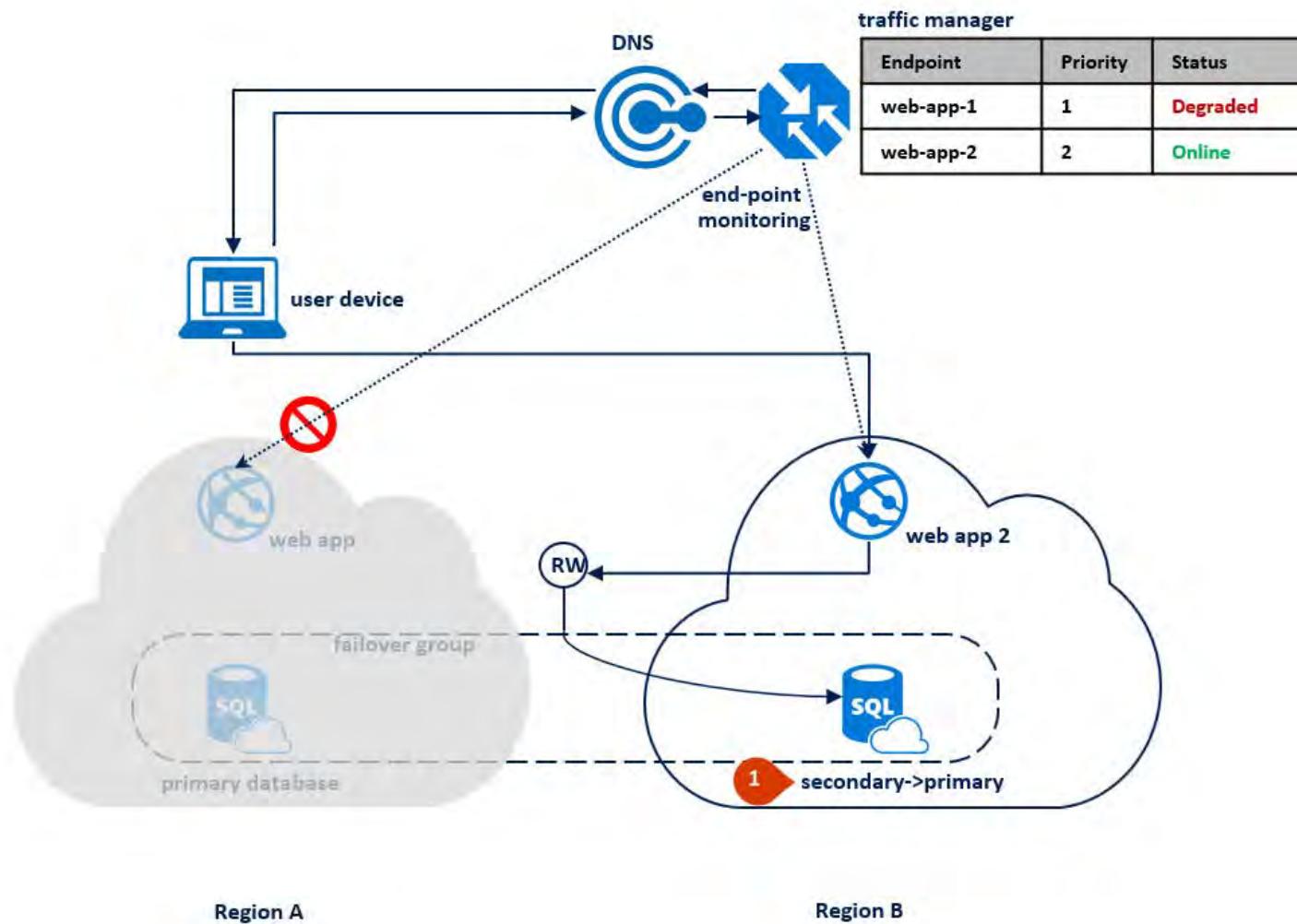


Fig.:
docs.
micro
soft
.com/
en-us/
azure/
azure
-sql/
data
base/
desi
gning-
cloud-
solu
tions-
for-
disaster-
recovery

Example: 2 Replicated Service Regions - Handover

Fig.:
c.f.
pg.
VII-4



Different 'Target Directions' for Replication

1. **Active** components, e.g., compute nodes, communication, server
2. **Passive** components, e.g., 'information', data, database tables

Common characteristics: *Resources* exhibit typical problems

Availability, competition, bottlenecks

\implies Usage discipline needed (Synchronization) **and**
Management (deadlocks, fairness etc.)

Common abstract system model: Client–Server

v-1/2

- C initiates compute task; S computes; C waits for result
- C initiates read; S transfers data; C waits for result

Different Approaches to make Replication transparent:

VII.1 *Models for Managed Active Servers*

VII.2 *Techniques for Data Replication*

VII.1 Replication of Active Components

- ▶ Potential for Replication: All hardware and software layers, e.g., processor, OS components, **services (daemons)**, . . . , applications
 ⇒ *Containers and Cloud technologies support replication*
- ▶ Different transparency levels w.r.t. **external client view**:
 - **Broker**-Models: communication partner information for clients
 - Server Addressing: IDs vs. service properties, e.g., *yellow pages*
 - Server internals: Server state relevant for availability?
e.g.: Buffer store empty/full; saturated vs. idle server
 ⇒ 'addressing' takes state of requested server into account
 - specialized vs. comparable servers (or worker)
- ▶ Server logging w.r.t. Client requests: **stateful** vs. **stateless**
Example: RPC semantics w.r.t. failures and repeated requests
- ▶ **Load-balancing** essential for performance transparency

Levels of Replication

Levels of Replication cont'd

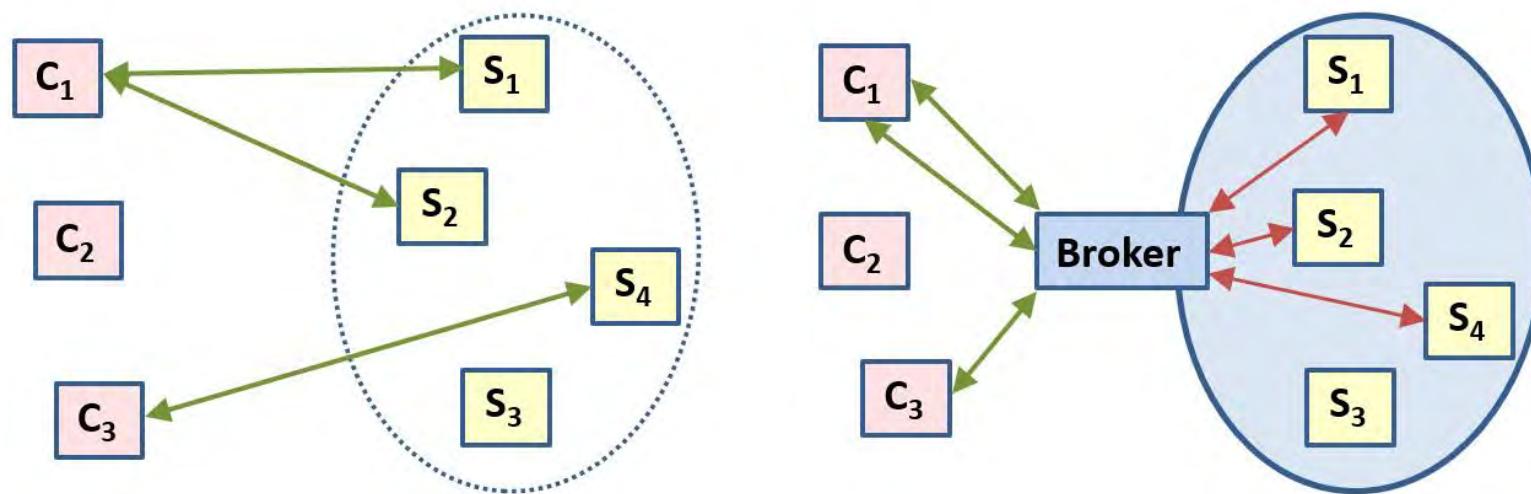
- **Concurrent Server:** *monolithic replication*
Example: Multiprocessor workstation; centralized DB server
 - ▶ no problems w.r.t. *consistency* on server level
 - ◀ single-point-of-failure
- **Distributed, concurrent Servers:** *Always 'true' replication?*
 1. **Co-operative Server:** multiple servers required for a single job
e.g., distributed file system (NFS) *no replication*
 - ▶ Load balancing ensures performance transparency
 - ◀ multiple single-points-of-failure (**AND**-Model)
 2. **Replicated Servers:** > 1 autonomous server **true replication**
i.e. each **single** server is able to process complete jobs
Example: redundant name services, mail server, DB systems
 - ▶ Performance as well as failure transparency (**OR**-Model)
 - ◀ costly w.r.t. hardware and software, esp. **Consistency**

I-28

I-28

Broker Models: Architectures for True Replication

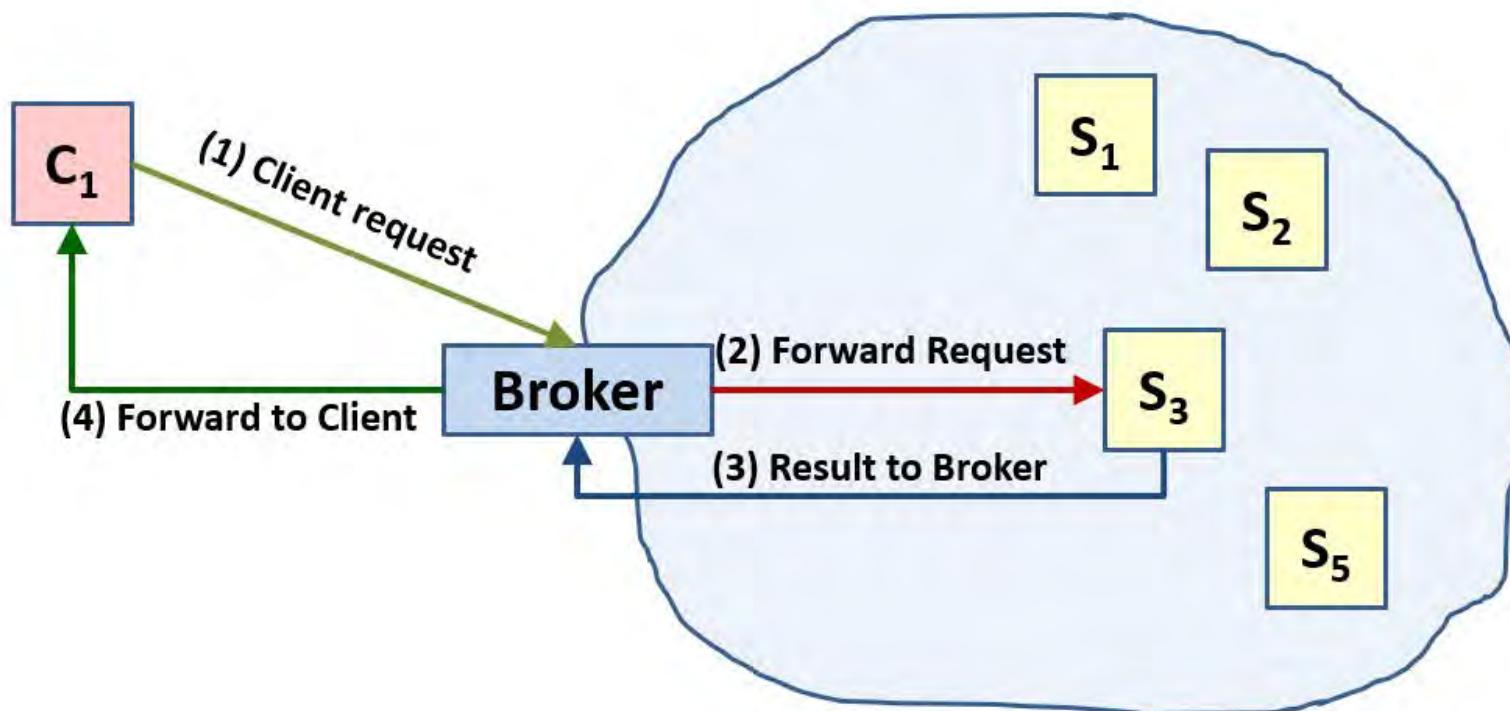
- ◀ **Without Broker:** Client has to know about all servers (needed)
 - ◁ transparency missing, distributed load-balancing impossible
 - ▷ no overhead due to broker interposition



- ▶ **Using a Broker:** abstracts from concrete naming schemes
 - ▷ **Naming transparency** (except Broker or **broadcast** search)
 - ▷ Broker controls load \implies load-balancing much easier
 - ◁ Overhead due to broker interposition

Broker – Server Organization

- **Forward Design:** *internal delegation*
only broker visible/known to client; broker handles request/result
Problem: additional overhead for copying and communication

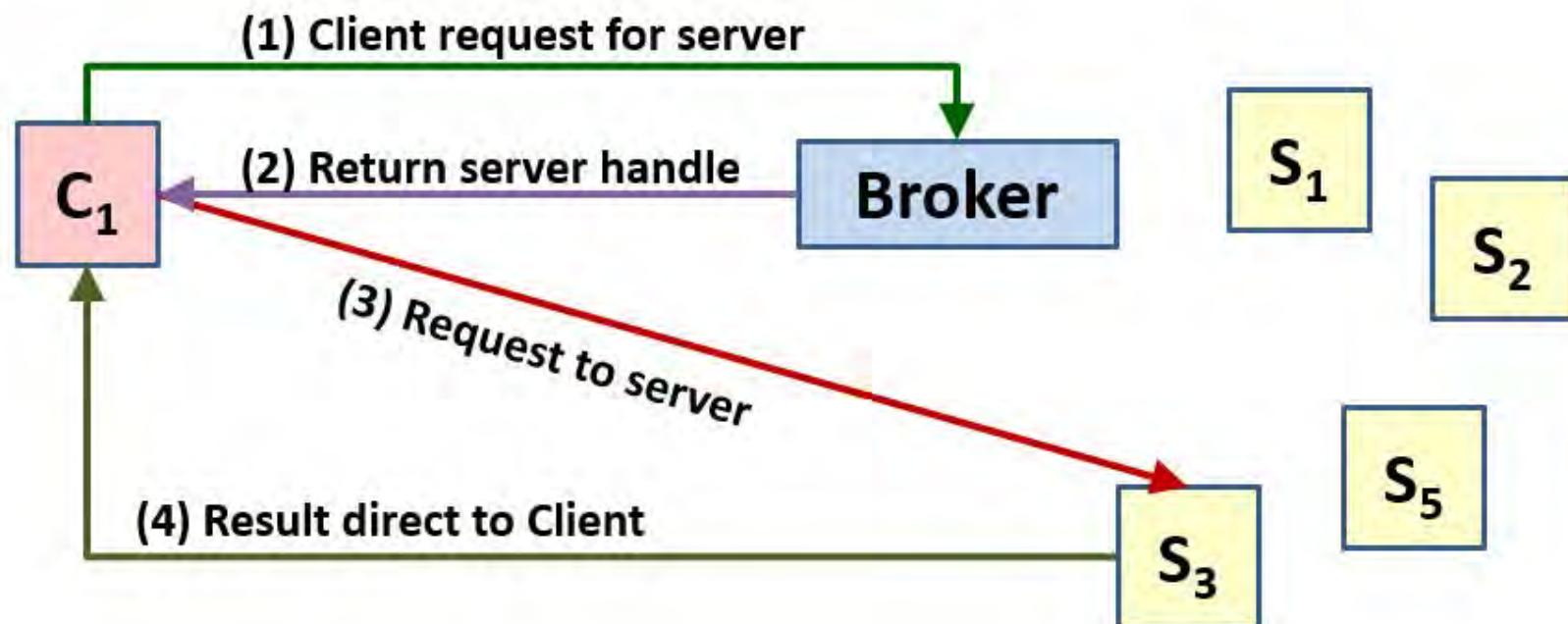


⇒ Client cannot re-use Server without Broker

Broker – Server Organization cont'd

- **Handle-driven Design:** *externally visible delegation*
 - Client sends inquiry to broker; broker sends *server handle* to client
 - direct server–client interaction for request/reply (Security?)
- Problem:** Client side caching of addresses annuls broker decisions

cf.
Web
Services



⇒ Avoids Broker Bottleneck to some extent

Broker – Server Organization cont'd

► **Hybrid design:** *combination of other models*

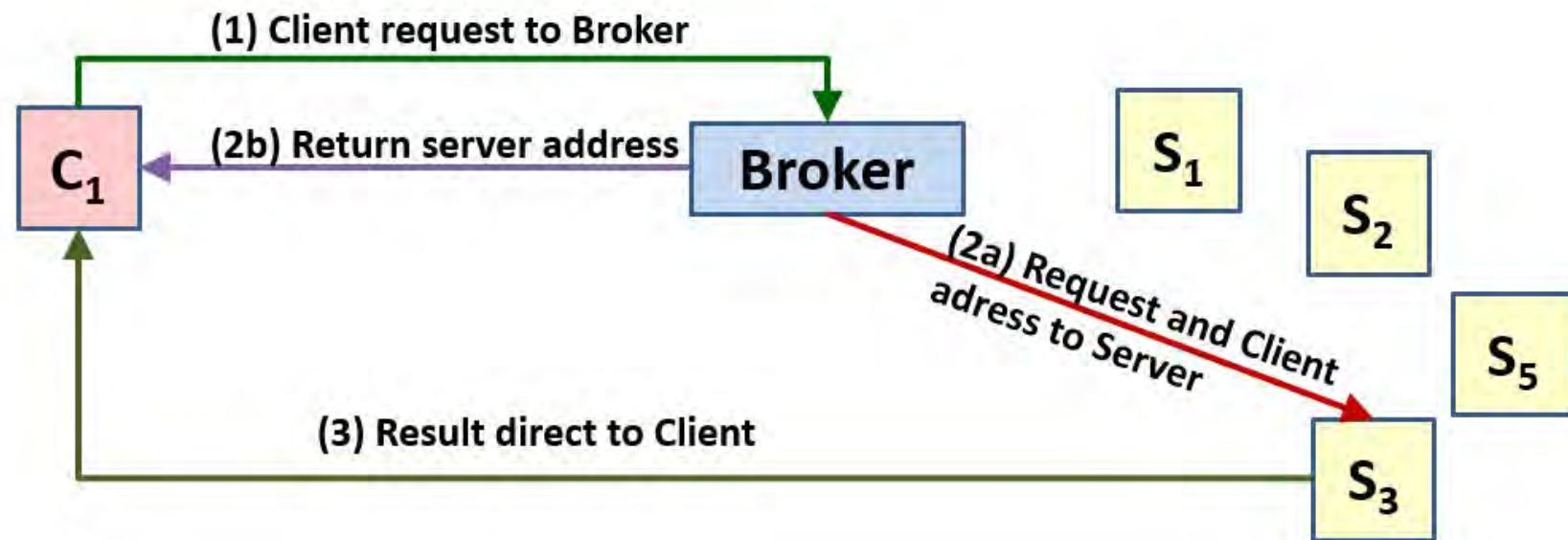
Client sends (full) request to broker;

Broker hands over client address plus request to suitable server
and server address back to client

direct server–client interaction used for result only

Problem: costly requests are sent even if no server is available

v-13
call
back



Trade-Off: Broker bottleneck vs. dissemination of system knowledge

Outlook: Load-Balancing required in any DS/PS

- **Models:** How to assign requests/jobs to servers?
 - **centralized:** Broker decides how to distribute load
 - Basis: Information via *forwarding/handle-driven*
 - Adaptive Pool:** start/terminate servers based on current load
 - e.g., avoid *cold-start* problem in container/cloud environments
 - **de-centralized:** Avoids broker bottleneck problem
 - * One role for *Server/Broker*: If a server has high loads, it acts as broker and *forwards* job to other server.
 - * **Bidding** and **Forwarding**: broadcast search for server first (or 'best' w.r.t. QoS etc.) answer wins
- ◀ **Problem:** Complex jobs that require a couple of (different) servers
 - ⇒ **Dependencies** complicate load distribution and introduce additional boundary conditions
 - ⇒ Hierarchical designs are easier: Broker, Server, Sub-Server

Stateful vs. Stateless Servers

- **Stateless service:** No state information about jobs processed
e.g., Time-Service, Name-Service need no client information
- **Atomicity-of-requests:** server stores jobs under processing
e.g., transaction models; Assignment: Job \longleftrightarrow Server
Avoiding duplicate processing for **at-most-once** RPC semantics
 - ▷ fault tolerance due to repeated execution of jobs
- **Stateful service:** processed jobs result in server state changes
e.g., DB data storages; File-Server
 - ▷ information re-use avoids communication and allows for efficiency
 - ◁ assumptions about 'global' system state among servers/clients?

Example Trade-off: performance vs. fault tolerance in a file system

- classical: open file pointers; **hot stand-by** replication
- stateless: each new request triggers 'full' overhead, e.g., http 1.0.
- **optimal:** external stateless – internal caching with replication

Perspective: Service and Cloud Eco Systems

c.f.
DSG-
SOA-M

- ▶ **Service Descriptions:** capture non-technical information, too !
 - **Interfaces** define operations and parameter types
 - external **protocols** use services via RPC, msg passing, . . .
 - internal **state information**: specifies which *ops* are available
 - QoS attributes define robustness, security, trustworthiness . . .
- ▶ **Infrastructure: Offer, Search, Find, Bid/Negotiate, Choose**
 - Compute, Storage and Communication Resources via Clouds
 - **Broker acts as a 'Trader'**: teams up Requester and Provider
 - free/with costs directory services . . . electronic markets
 - Search/Match based on functionality, state and attributes
 - Compensation for unavailable services, replicated execution etc.
- ▶ **Pricing, Negotiating and Billing of services**

⇒ **Long-term Goal:** Service Ecosystems with **on-the-fly** service replication and composition at run time.

Agent
plat-
forms

Example: Car Data Eco Systems

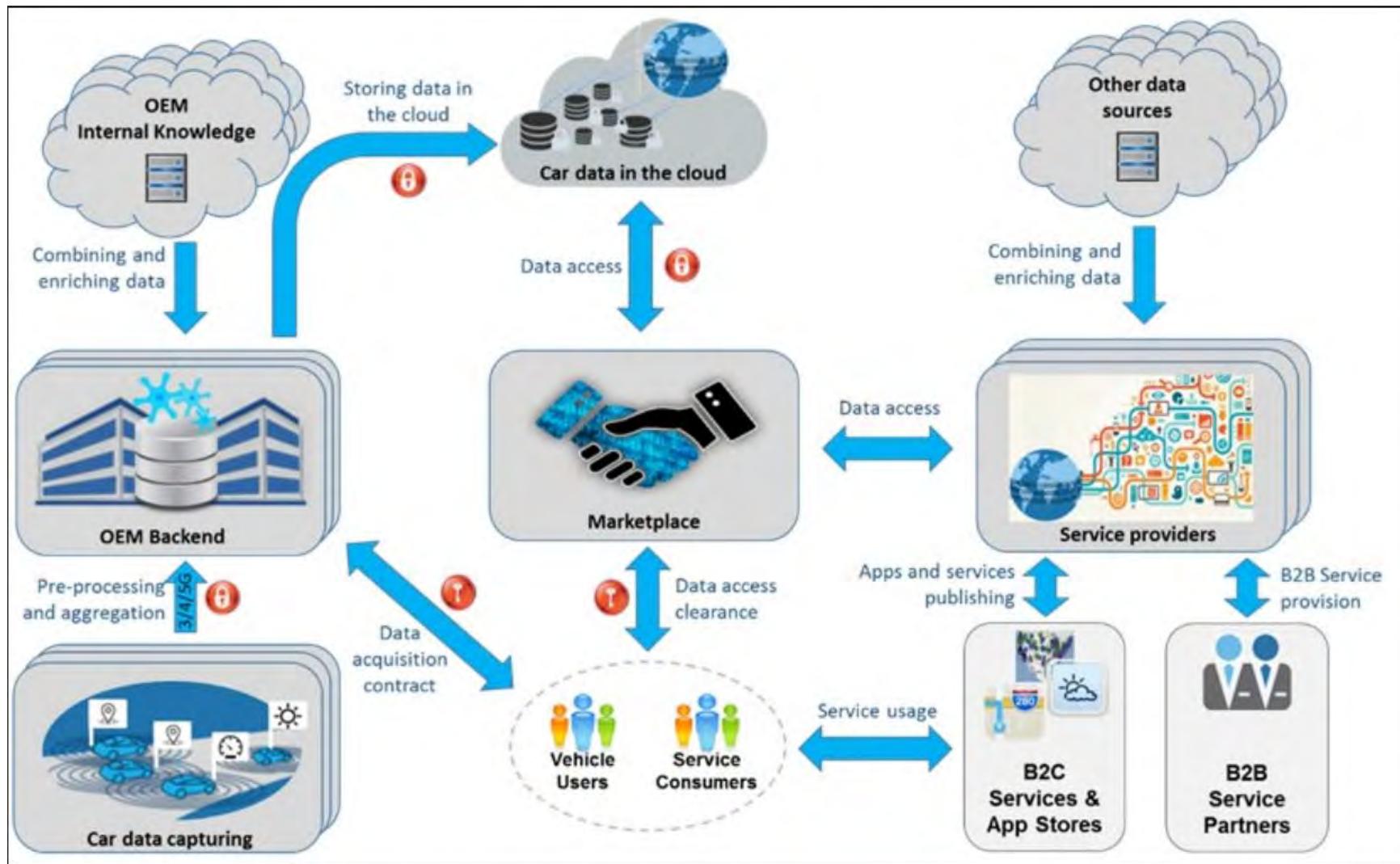


Fig.:
 cordis.
 europa
 .eu/
 docs/
 results/
 h2020/
 644/
 644
 657_PS/
 figure-1-
 automat-
 -eco
 system-
 -modules-
 -and-
 -actors-
 .jpg

VII.2 Replication of Passive Components

passive \approx Data: Text, Tables, Objects, DB entries,
Variable, Segment/Page/Word

Different Aims require different Measures:

► **Failure Transparency:**

- 'Duplicating' data mandatory for robustness
- 'Healing' via exchange protocols to recover data from copies

► **Performance Transparency:**

- local vs. remote access and *locality* principle \implies Caching
- distribute **peak loads** by means of distributed data access

► *Virtual Shared-Memory Programming Model*
remote data are loaded *on demand* \approx 'networked MMU'
 \implies **Location transparency** on programming model level

K. Li
1986

Choice of Techniques motivated by Usage Context

Trade-Off: Advantages of copies **vs.** Costs for consistency
⇒ **No strategy matches 'all' use cases!**

Wide range of applications ⇒ **Different profiles**

- **Example:** WWW client using a local cache:
 - ▶ Web presence of a company: **Write**-Ops are rare **optimistic** ⇒ Client initiates consistency checks (re-load) long time periods before expiration date
 - ▶ Stock exchange quotation: **Write**-Ops are frequent **pessimistic** ⇒ expiration date is always set to 'now'
- Client-side advantage: **centralized Write**-Ops
- Remark:** many **Clients** ⇒ **Proxy** used as an efficient cache
- **Example:** Parallel shared-memory programming: many **Writes** **pessimistic** ⇒ Wait and lock data before **Write**-Ops

Suitable Techniques for Transparency Aims

- **Performance Transparency:** wide scope of techniques
 - ▷ HW, system software, middle-ware, programming: **Caching**
 - ▷ Copies on platforms with comparable 'reliability' and performance
 - ▷ Distribute copies dynamically based on usage
 - ▶ *Optimistic* strategies are admissible
 - ▷ **Migration** may also be an alternative for sequential use of data at different locations
- **Failure Transparency:** highly restricted scope of techniques
 - ◀ **Multiple copies at different locations** are mandatory
 - ◀ Wide range w.r.t. performance: fast . . . persistent storage media
 - ◀ *Pessimistic* strategies are required
 - ◀ Strong synchronization combined with logical coupling is costly
Trade-Off: high costs vs. 'critical' periods in case of crashes

Remark: Similar Trade-Off as number of snapshots and rollback

and
VSM

VII-27

cf.
VI.4

Replication: Original(s) vs. Copies

'Naive' Consistency: Copies are always identical to 'original data'
⇒ *Strict definition not feasible in distributed systems?*

see
VII.2.1

Three Replication Levels:

0. *Single Read Single Write* ≈ No replication

Migrate original to location of usage (**trashing** problem)

1. **Multiple Read Single Write** ≈ **Read Replication**

only a few writes but lots of read accesses

Write is only allowed on a **single original**; Read on n copies

VII-23

2. **Multiple Read Multiple Write** ≈ **Full/Write Replication**

Writes at different locations ⇒ enhance write performance

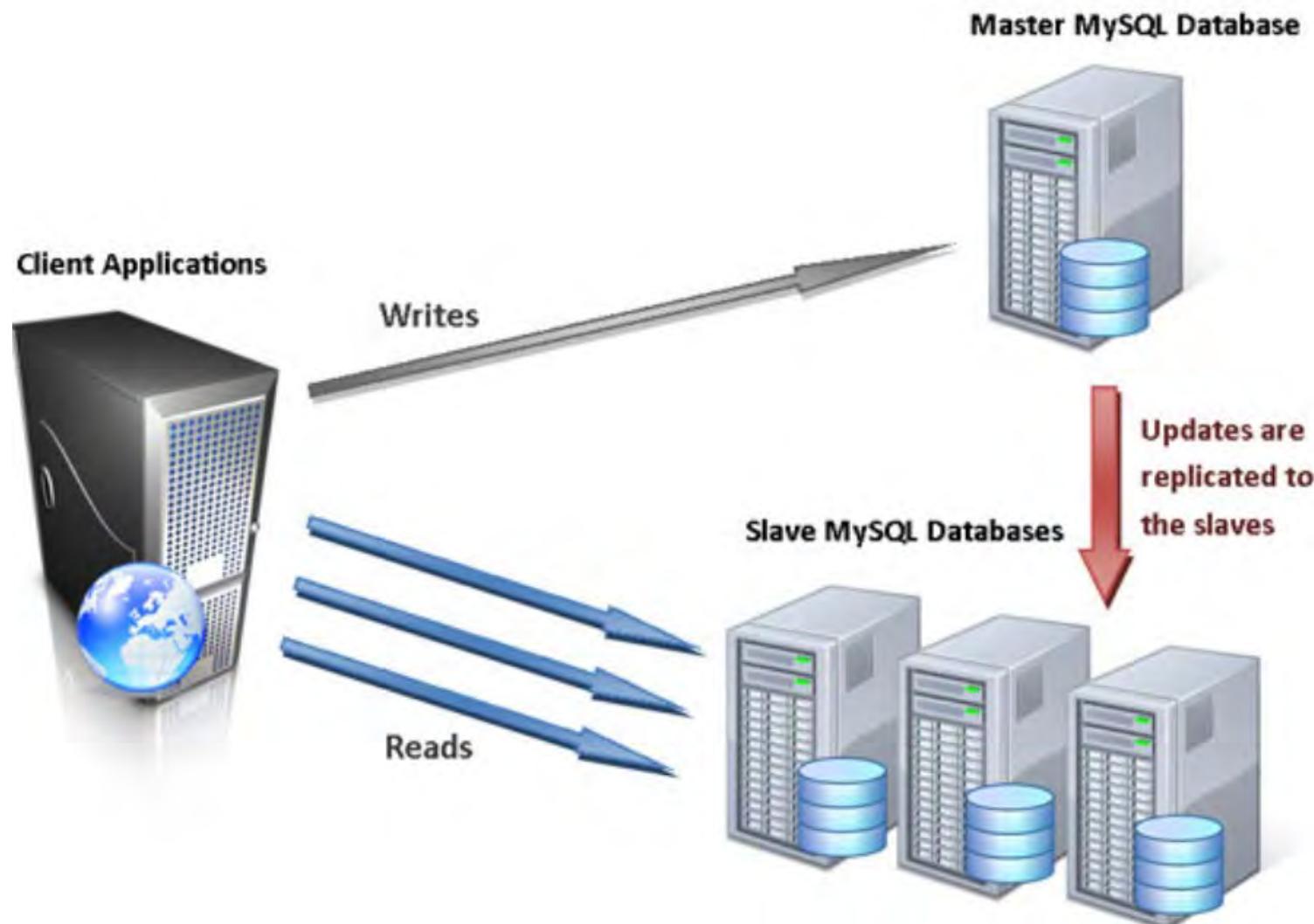
Writes are allowed on copies; **Consistency hard to achieve**

Essential decision: rely on a single 'original' vs. majority quorum

VII-24

Example: Non-Transparent MRSW Configuration

Fig.:
www.
learn
compute
com/
mysql-
repli
cation/



MRSW – Roles and Strategies

- ▶ **Unique Owner** ≈ holds the single 'original' that is written
- ▶ **Copy Set** ≈ n Processes hold (almost) identical copies
- ▶ **Manager** ≈ co-ordinates write accesses ⇒
Invalidation of Copies vs. **Propagation** of 'new' original

VII-25

Variants of Implementations: How to implement the **manager role**

c.f.
Broker

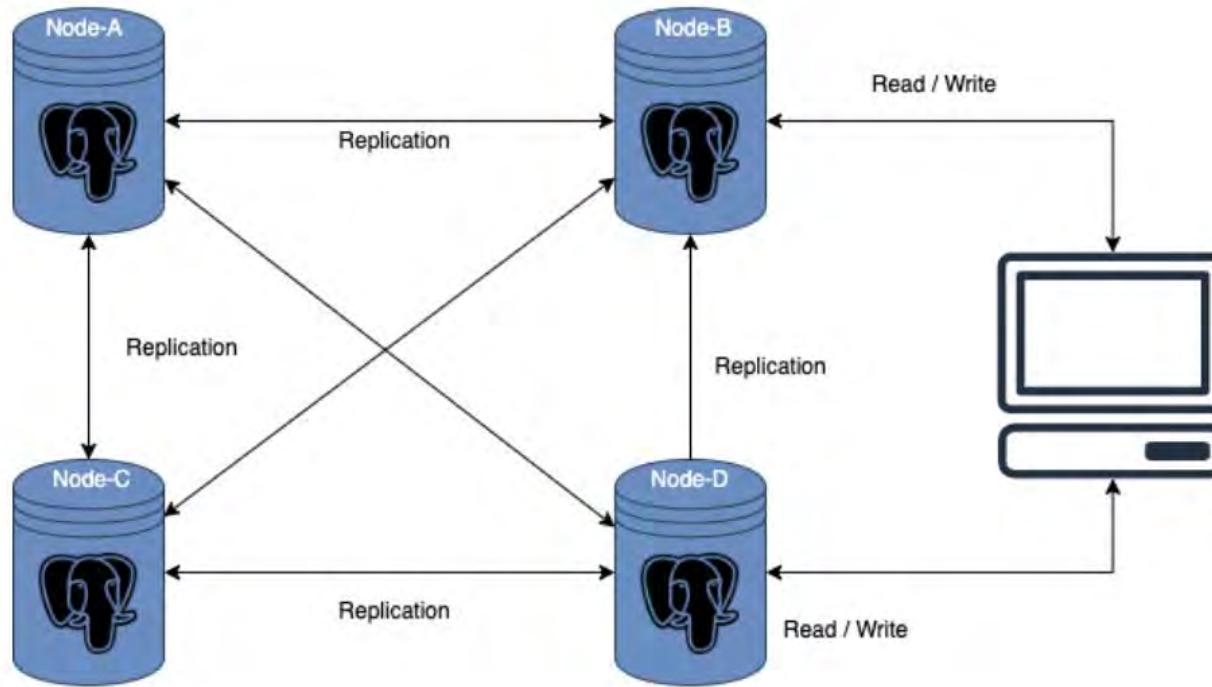
1. centralized: Manager organizes all accesses and delegates to owner
2. *Owner invalidates*: Owner = Manager
3. Distributed Manager:
 - (a) fixed distribution of Owner roles to different processes
 - (b) dynamic distribution: Owner role migrates with **Write** accesses
Owner information has to be retrievable; outdated ⇒ forward
 - (c) *Broadcast and (fixed) distribution*: Requests go to all processes
⇒ all processes check all requests, but only Owner responds

realistic?

Global knowledge assumption? critical for all other than 3.(c)

MRMW – Additional Role and Strategies

- **Owner Set** ≈ many 'Originals' that can be written



- Before or after Write ⇒ all processes of the **Owner Set** informed
 - ◁ Manager manages **Owner Set** as in centralized mutex setting
 - ▷ Agreement Protocol among **Owner Set** in case of changes
- Manager provides sequence numbers to sequentialize changes

Fig.:
www.
percona
.com/
blog/
2020/
06/09/
multi-
master-
repli-
cation-
solu-
tions-
for-
post-
gresql/

Basics for Handling Changes of Data

How to Propagate Changes: Original or a single MRMW copy is (*Coherence Policy*) written:

- ▶ **Write-Invalidate:** Invalidate all copies in *Copy/Owner Set*
 ⇒ avoids communicating complete, possibly huge objects for short read periods by using (ObjID,Flag)-Msgs
- ▶ **Write-Update:** Propagate new original to *Copy/Owner Set*
 ⇒ copies that are valid over long times due to infrequent writes

Note: **Leasing** of copies helps to reduce number of copies.

When to Propagate Changes: New value is published . . .

- ◀ *synchronous*: after acknowledged change from **all** copies
- ▶ *asynchronous*: directly and process to update or invalidate is triggered directly afterwards
- ▶ *semi-synchronous*: after acknowledged store at a predefined minimum set of nodes, e.g., Owner vs. Copy set

Trade-Offs for Performance Transparency

few vs. many copies \implies Costs for invalidate/update
single vs. multiple write \implies Costs for consistency mechanisms
write-Bottleneck vs. Overhead for sequential consistency

Size of replicated data chunks: (Units)

- Segments, Pages, Records, Words (technical view)
- Objects, DB entries, methods, variables (logical view)

c.f.
Page-
size
in
OS

Small Units:

- ▶ fewer conflicts \implies less write overhead
update strategies are suitable
- ◀ lots of Copy Faults \implies more read overhead (locality)

in
appli-
cation

Big Units:

- ▶ fewer Copy Faults due to locality
- ◀ more Write conflicts: **false sharing** due to 'shared units'
lots of copies and more write Overhead \implies invalidate suitable

Reduced Options for Failure Transparency

VII-20

Different types of Errors: (flawless communication pre-assumed)

1. **Copy** is lost (no longer accessible)

- ▶ Read Copy: similar to a **read fault** \implies no problem at all
- ◀ Write Copy: update current **Owner Set**

2. **Original** is lost

- ◀ MRSW or MRMW using a single Owner \implies Algorithm fails
- ▶ If there are copies: **Majority** determines 'new' Original

Voting

$\implies n > 1$ **distributed 'Originals' have to be kept consistent**

- More than one write copy (Originals)
- If Original is lost \implies Election Algorithms for new Original
- Majority vote among Write copies in case of inconsistencies

3. **Network Partitioning:** worst case because not all of the Originals/Copies are accessible!

see
VII.2.2

Example: Adaptable Algorithm for both Objectives

Idea: Boundary Restricted Multiple-Reader Multiple Writer where

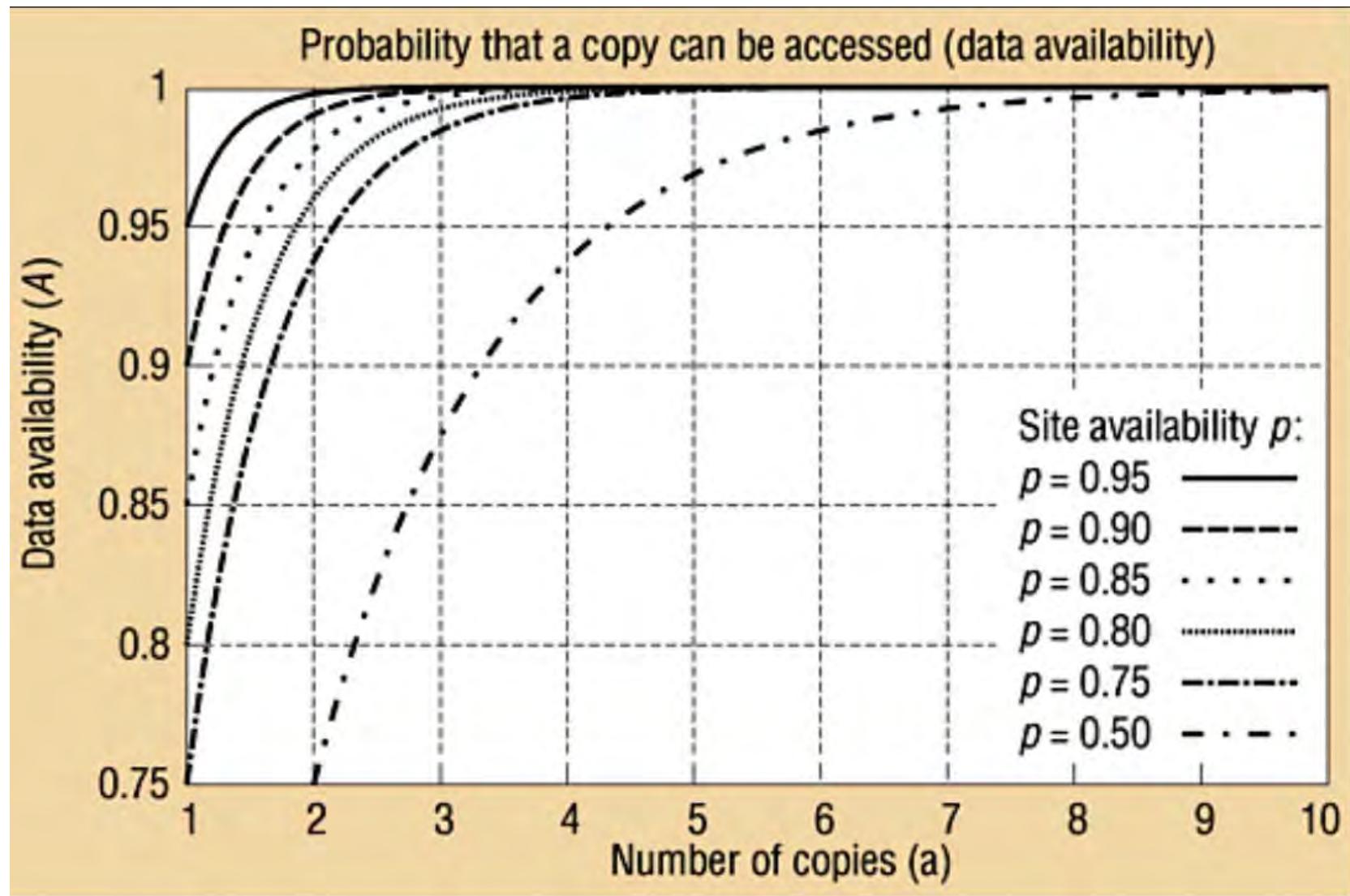
- ▶ Minimum R/W_{min} numbers of copies guarantee **reliability**
- ▶ Maximum R/W_{max} numbers reduce **consistency costs**

Algorithm: (*implements 'sequential consistency'*) VII-33

- **3 Levels of Access Rights** for nodes w.r.t. Read/Write:
 - ▷ (local read, local write) \implies 1 Original
 - ▷ (local read, global write) \implies Copy handling like MRSW
 - ▷ (global read, global write) \implies no copy available, i.e., locked
- Alternating **Phases** w.r.t. **global** Reads and Writes
 - Read** request: assign copy; ensure R_{min} copies are distributed
If R_{max} copies are distributed reduce (invalidate) copies first.
 - Write** request: assign copy; ensure W_{min} copies are distributed
Reduce the number when performing the update via **invalidate**.

IE³
Concur-
rency,
June
2000

Availability: a Copies vs. Node Availability



B. Fleisch, H. Michel et al.: Fault Tolerance and Configurability in DSM Coherence Protocols. IE³ Concurrency 8(2) June 2000, pg. 10-21

VII.2.1 Consistency Models

Objective: Approximate properties of centralized shared memory in a message-based widely distributed system.

► MRSW: 1 **Owner** \implies organization of changes **easy**

Delay: Write in one process vs. propagation via messages

◀ MRMW: additional overhead to localize the 'most recent copy'
even parallel writes may be allowed

Delay: Localization plus propagation times

Naive Goal: (*Single-Processor Strict Consistency*)

*Any read of a memory cell x results in exactly that value that has been written into x by the **most recent** write.*

◀ writes and reads 'almost' at the same time

◀ upper limit for message transfer is 'speed of light'

\implies **Strict consistency is no reasonable objective in DS!**

Realistic Approximations for Strict Consistency

Consistency is costly \implies Suitable model based on applications.

Note: Transparency allows **implicit** models only (no VSM).

► Client-centric Consistency:

VII-32

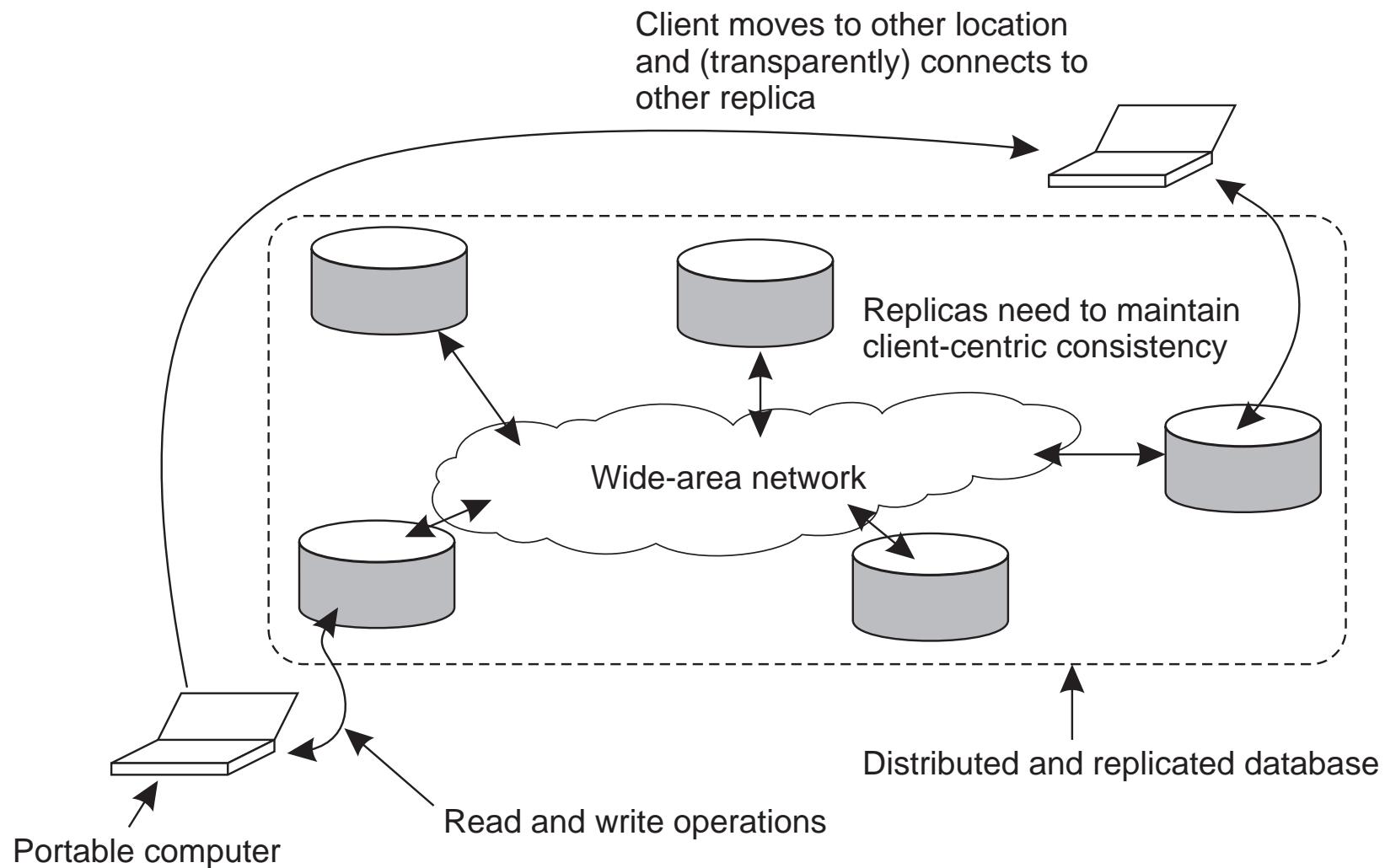
- * If a client uses the **same** copy, everything seems consistent
- * **Eventual Consistency**: global inconsistencies are tolerated, but after a long period without Write \implies Guarantee that after a 'finite time' all copies are up-to-date
- * If the client accesses a different copy:
 1. **monotonous Read**: copy is always the same or newer
 2. **monotonous Write**: Write propagated before a new Write

► Data-centric Consistency:

VII-33
ff.

- * based on a 'global view' on the overall state(s)
- * compares original(s) and replicated copies of data

Isolated Client View on Consistency



taken from: **Fig. 6.19** in: A. Tanenbaum/M. van Steen: Distributed Systems, pg. 318

Global Consistency – Transparent Models – 1

Sequential Consistency: *The result of any (parallel) execution is the same as if the operations of all the processors were executed in some (arbitrary) sequential order where the operations of each individual processor appear in this sequence in (exactly) the same order as specified by its program.*



c.f.
Lamport
1979

- **Weakened Condition:** No true parallelism among processors
Non-deterministic interleaving/**serialization** instead of parallelism
- Interaction **without** implicit assumptions about execution order works correctly; typical problems with dependencies and non-determinacy otherwise.

c.f.
III-6/8

Example:

```
a=0; b=0; parbegin { br=b; ar=a; } || { a=a+1; b=b+1; } parend;
```

correct values for (ar,br) are (0,0); (1,1); (1,0)

incorrect: (0,1) because b is incremented and updated before a

► Clear semantics, implementable and transparent

VII-34

Implementing seq. Consistency by Write-Invalidate

1. C wants to write O, but not in local memory \implies write-fault
2. Get copy from current Owner (broadcast)
3. Send Invalidate message to all processes in Owner Set (broadcast)
4. C reads/writes O until different Client C' start similar request (2.)
5. C'' tries to read O \implies read-fault because invalidated
6. Get copy from current Owner C ...
 - Write waits until all copies have been invalidated **and**
 - Lock first, write exclusively afterwards \implies Accesses are serialized \implies **sequential consistency** guaranteed

c.f.
paging
VII-23
3.b/c

Problems: (1) Processes try to start Writes/Updates in parallel
 \implies global **Sequencer** needed for serializing **Write-Requests!**
(2) Lost **invalidate** messages lead to outdated reads

Global Consistency – Transparent Models – 2

Causal Consistency: Write operations that are **potentially causally related** are seen by every node of the system in the same order. Concurrent writes that are not causally related, may be seen in different order by different nodes.

◆
Hutto
et al.
1990

- **Weakened Condition:** execution order matters only iff the operations are causally ordered
- Causality is induced via *Read/Write* dependencies
- ◀ **Implementation:** requires a causality analysis of the program
Each variable uses an attached vector clock for propagation

Example: T →

```
P1: write(1);           write(3);
P2:       read(1);    write(2);
P3:           read(2); ?read(1)?
P4:       read(1);      read(3); !read(2)!
```

incorrect: read(1) after read(2) in P3 because $\text{write}(1) \xrightarrow{*} \text{write}(2)$

correct: entire run without ?read(1)?; e.g., !read(2)! in P4 is ok
because write(2) and write(3) are concurrent.

Global Consistency – Transparent Models – 3

PRAM/Processor/FIFO consistency: All processes see writes from **one process** in the order they were executed in this process. Writes from different processes may be seen in a different order on different processes.

◆ Goodman
1989

- **Weakened Condition:** No causality **among** different nodes
- Easy to implement via **message ordered broadcast** for all updates and buffering on the receiver side for out-of-order messages.

Example:

```
a=0; b=0; parbegin
    { a = 1; if (b == 0) then kill(P2) }      /* P1 */
//    { b = 1; if (a == 0) then kill(P1) }      /* P2 */
    parend;
```

correct: Test in P1 (P2) before assignment in P2 (P1) \implies kill P2 (P1)

also correct here: both test occur before any update \implies both processes are killed

c.f.
seq.
consis-
tency

◀ hard to use practically!

Global Consistency – Transparent Models – 4

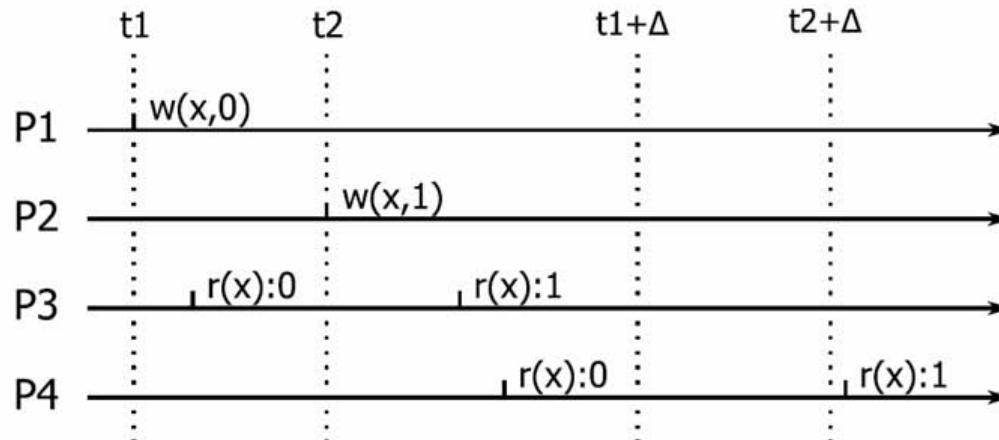
Delta Consistency: An update will propagate through the system and all replicas will be consistent after a fixed time period, i.e., the result of any read operation is consistent with a read on the original (copy) except for a (short) bounded interval of Δ **time units** after a write. ♦

- **Weakened Condition:** bounded delay of updates is tolerated where 'Delay-Models' differ based on the application at hand:
 - * Approximation of global virtual time with $\Delta[t]$ ticks delay
 - * 'Distance' among version numbers is kept below Δ
 - * Only number Δ of 'important' state changes counts
- Realistically implementable via:
 - * Use global, virtual time clocks
 - * Check 'freshness' and update outdated copies (pull/push)
- **Typically used in** *Read-centered applications*, e.g., Caching for web servers . . . Content Distribution Networks

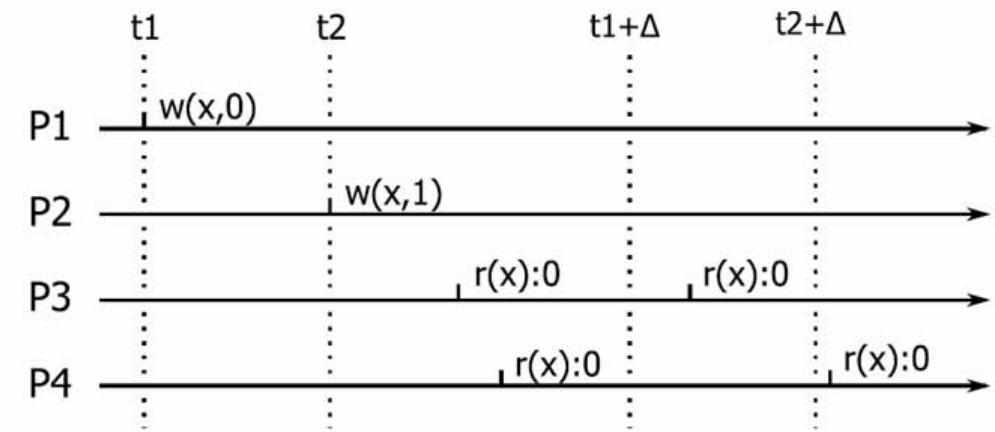
Singla
et al.
ACM
SPAA
1997

P2P
also

Example: Δ Consistency and Logical Time



(a)

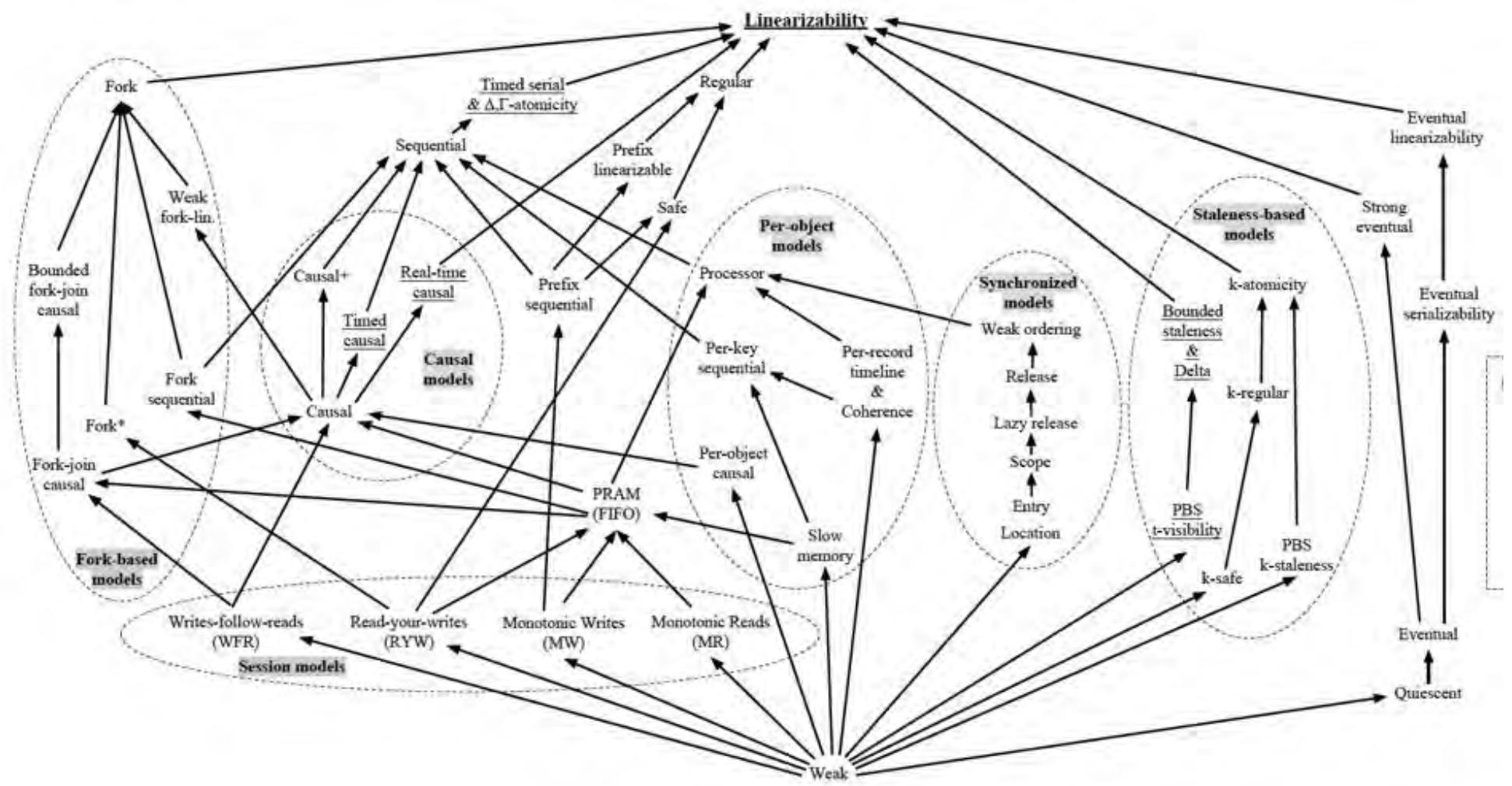


(b)

- (a) valid run despite outdated reads because delay is still within tolerated range Δ .
- (b) last access in P_4 is invalid because delay is out of range Δ .

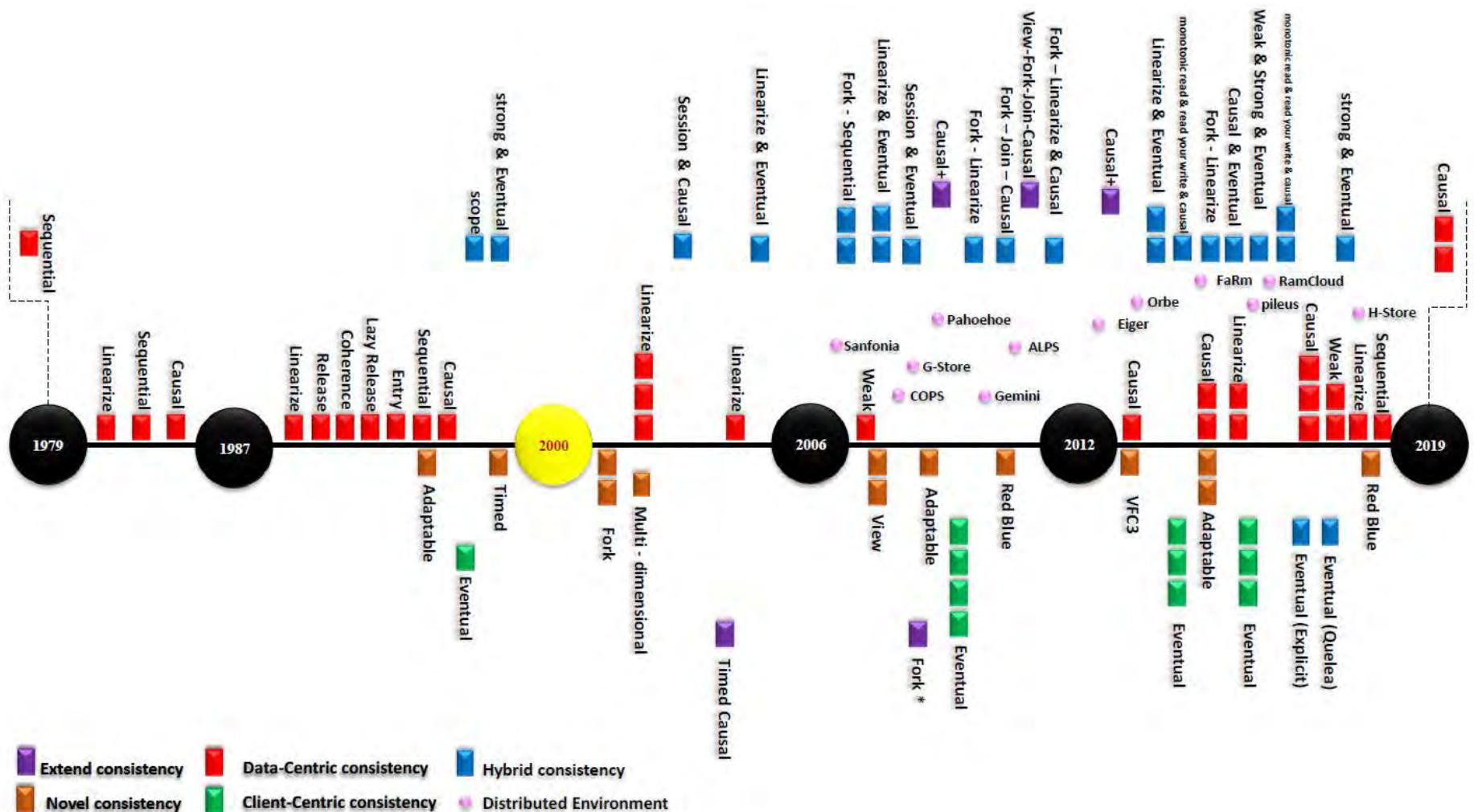
from:
C.
Simons
Context-
Aware
Appli-
cations
in
Mobile
Distribut
Systems
Diss.
2007,
pg. 84

Outlook - A whole Bunch of Consistency Models . . .



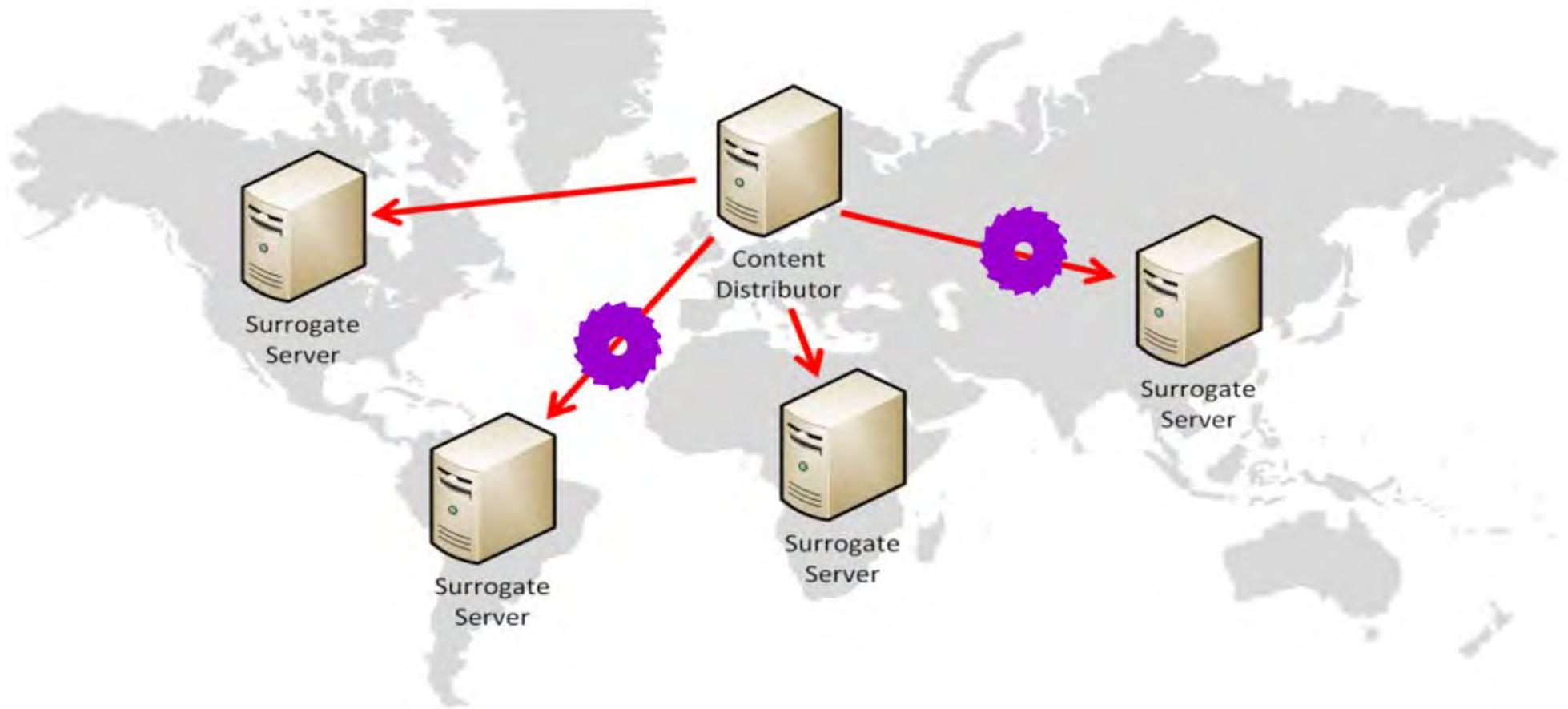
P. Viotti, Paolo, M. Vukolic: Consistency in Non-Transactional Distributed Storage Systems. in: ACM Computing Surveys (49)1; 07/2016

... and still evolving ...



H. Aldin, H. Deldari, M. Moattar, M. Ghods: Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications. in: deepai.org, 08/2019

VII.2.2 Replication and Network Partitionings



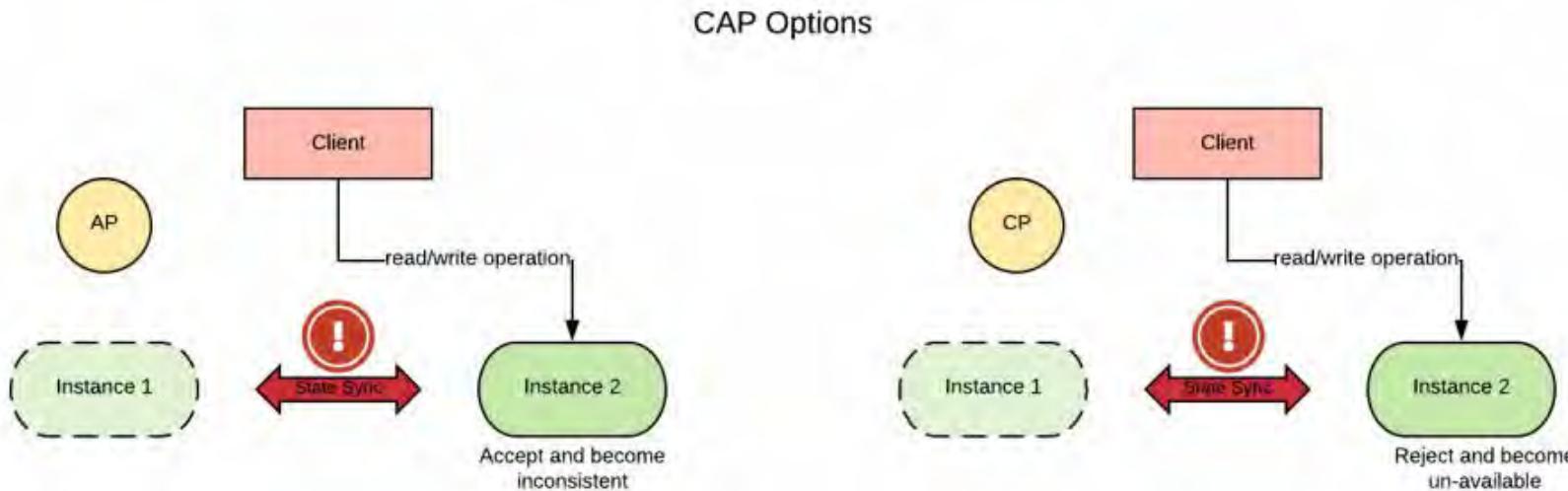
- ◀ Worst-Case for Availability Considerations
- ◀ Worst-Case for Replication Schemes w.r.t. Consistency

Trade-Off for Network Partitionings - 1

Problem for (all) protocols:

- Read-Fault: attempt to get a copy may **block**
 - Write: consistency mechanisms (**invalidate/update**) **block**
- Reason:** Consistency requires 'global' answers for **Writes**

Fig.:
www.
open
shift
.com/
blog/
state
full-
work
loads-
and-
the-
two-
data-
center-
conun
drum



⇒ **Modified Protocols** trade consistency for availability

Trade-Off for Network Partitionings - 2

Modified Protocols relax consistency to some extent

1. **Primary Copy** \approx MRSW or MRMW using a *lock manager*
part of system 'with' primary copy resumes work; rest is blocked
2. **Majority Vote** w.r.t. Write or Read copies
Network part that holds majority of copies goes on; rest is blocked
Example: $\frac{1}{3}$ required for Read; $\frac{2}{3}$ for Write
3. **Majority** is required for Write only \implies all processes may read
4. Accept **deviations** of values for better availability
 - ▷ record and log changes for later recovery
 - ◁ complex rules for combining diverging data:
Examples (from DB): latest, primary(n_k), Program, Notify

questi-
onable
!

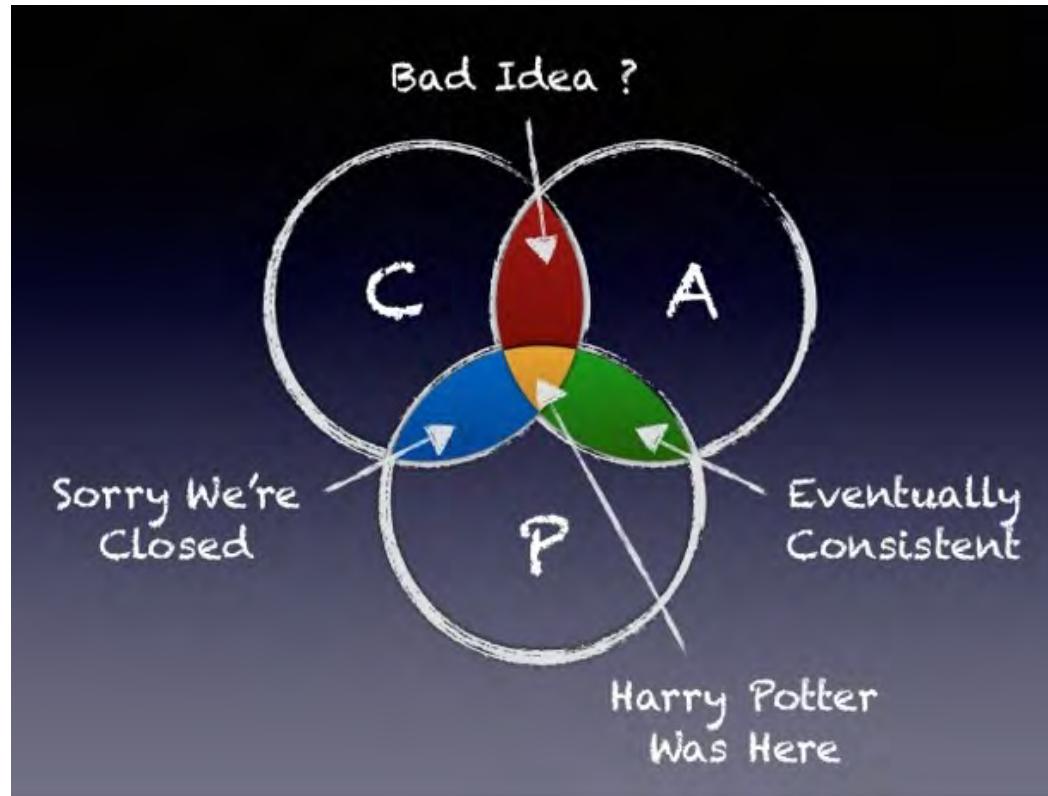
The C-A-PT Theorem aka CAP Theorem

CAP-Theorem: \exists Trade-Off among the following properties:

Consistency – Availability – Partitioning Tolerance (of network)

\implies **?You can't fulfill all three at the same time?** \Leftarrow

Fig.:
www.
slide
share.
net/
christoph
evg/
revis
iting-
the-
cap-
theorem



\implies **How frequent are Network Partitions in your application?**

Consistency for Real-life Scalable Applications – 1

- **Theory:** There exists a more general 'Trade-Off' between *Safety* vs. *Liveness* in an '*unreliable*' system that is suffering from faulty communication.
 - * **Safety** Property: Holds at all times in a system
 - * **Liveness** Property: Holds '*eventually*', i.e. after a finite amount of time, the system reaches a state where the property holds.
- **Real Life:** *What does C-A-P really mean?*
 - * **Consistency:** All nodes read the most recent value due to the chosen Consistency model
 - * **Availability:** Each active node reacts within a time frame that is acceptable due to the chosen Service Level
 - * **Partitioning Tolerance:** Even in the presence of network partitionings, the system works and respects its consistency model

Gilbert
Lynch
IEEE
Compu
ter
No.
45(2)
02/12
pg. 30
ff.

Consistency for Real-life Scalable Applications – 2

- **Practical Considerations:** Availability is not enough
 - * Maximum time of tolerated delays for 'availability' is critical
 - * Effect of '**Latency**' as an additional important aspect is missing

Connection: Arbitrary high '**Latency**' \implies no '**Availability**' at all!

Extended model that takes "**Latency**" into account:

"A more complete portrayal of the space of potential consistency tradeoffs for DDBSs can be achieved by rewriting CAP as PACELC: if there is a partition (P) how does the system tradeoff between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system tradeoff between latency (L) and consistency (C)?"

D. Abadi: Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. IEEE Computer, 45(2):37-42, 2012.

Example choices w.r.t. CAP- Trade-Off

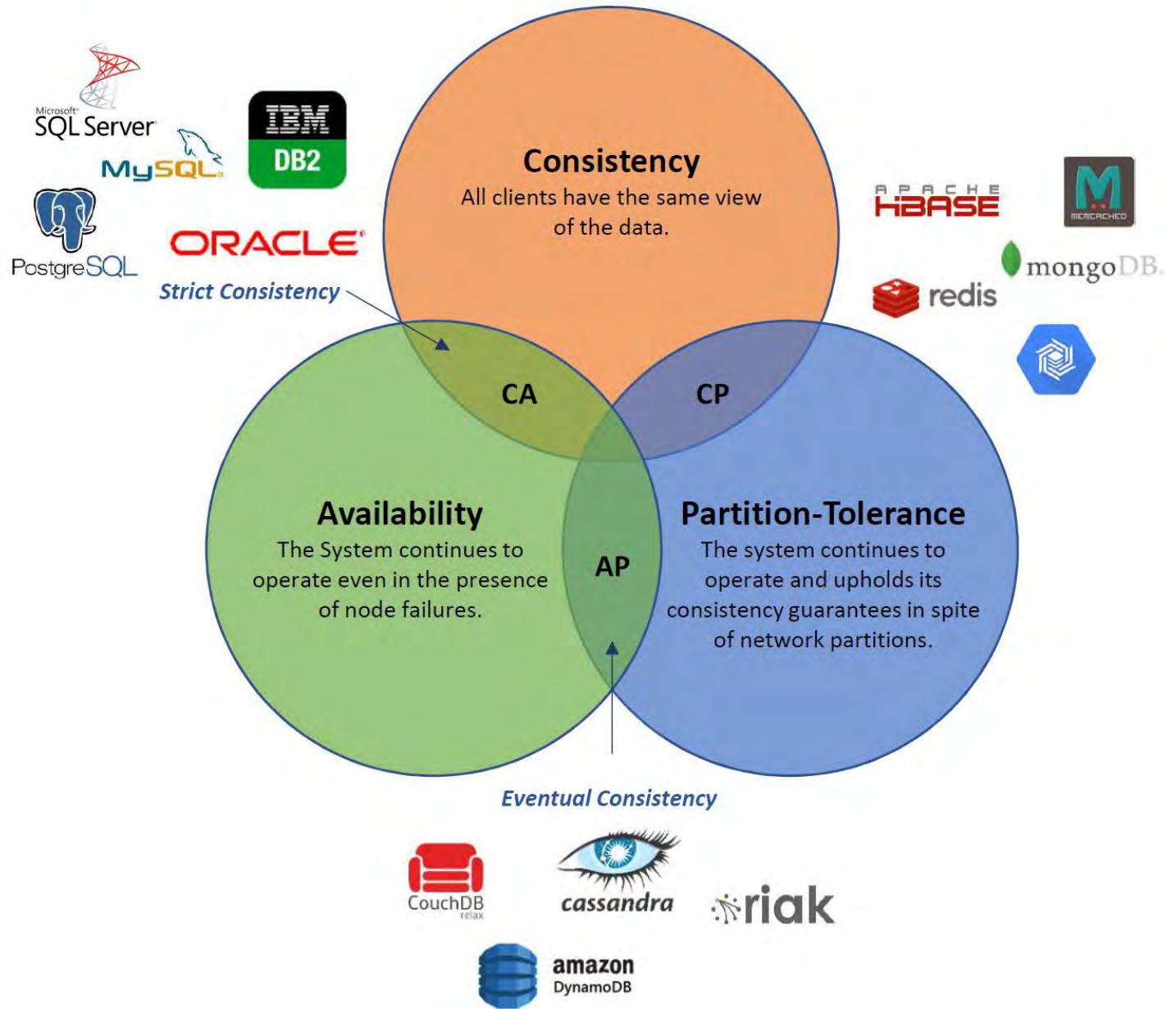


Fig.: G. D. Samaraweera and M. J. Chang, Security and Privacy Implications on Database Systems in Big Data Era: A Survey, in: IEEE Transactions on Knowledge and Data Engineering, doi: 10.1109/TKDE.2019.2929794.

Consistency and Availability – Outlook

c.f.
Papers
in
vc

Further Readings:

(*required for MSc students!*)

- W. Vogels: [Eventual Consistency](#). in: Comm. of the ACM 52(1):40-44, 2009
- Simon S.Y. Shim: [The CAP Theorems Growing Impact](#). in: IEEE Computer, 45(2):21-22, 2012.
- Eric Brewer: [CAP twelve years later: How the “rules” have changed](#). in: IEEE Computer 45(2):23-29, 2012
- IS. Gilbert and N. Lynch: [Perspectives on the CAP Theorem](#). in: IEEE Computer, 45(2):30-36, 2012.
- D. Abadi: [Consistency tradeoffs in modern distributed database system design: CAP is only part of the story](#). in: IEEE Computer, 45(2):37-42, 2012.
- Eric Brewer: [Spanner, True Time & The CAP Theorem](#). in: Google TR45855, February 2017

Summary – Replication and Transparency

- ▶ **No Performance/Failure-Transparency in Distributed Systems possible without Replication.**
- ▶ Techniques used are too complex to offer them on user level
 ⇒ *Typical issue for OS and Middleware levels.*
 Example: User → Caching → Replication Management
 → Snapshots and Logging → Backup-Server
- ◀ Overhead and additional resource usage is typically high
- ◀ High, in parts even contradicting, demands make solutions with specific 'Quality-of-Service' offers tailored to specific classes of applications the most promising proceedings.

Importance: *Real problem in all recent systems in the context of Online-Storage, Data-Clouds ...*

End
VII

End of DSG-(I)DistrSys-B/M Lecture Material

[Shingal et al. 1994]: *A distributed system consists of autonomous computers without any shared memory and without a global clock. Computers communicate using message-passing on a communication network with arbitrary delays.* ♦

- ⇒ Overview w.r.t. characteristics, types and problems of DS
 - ⇒ Basic mechanisms to overcome shortcomings of DS
 - ⇒ First 'ideas' how to architect and program such systems
-

Outlook: DSG Offerings on master level

- DSG-DSAM-M: *Distributed Systems and Middleware* (winter term)
- DSG-SOA-M: *Service-Oriented Architectures* (summer term)
- DSG-SEM-M: Seminar discussing advanced aspects, e.g.,
 - * Advanced Distributed Algorithms
 - * Consistency Models and Cloud Applications
- DSG-Project-M: winter term topics are tba.