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. ♦

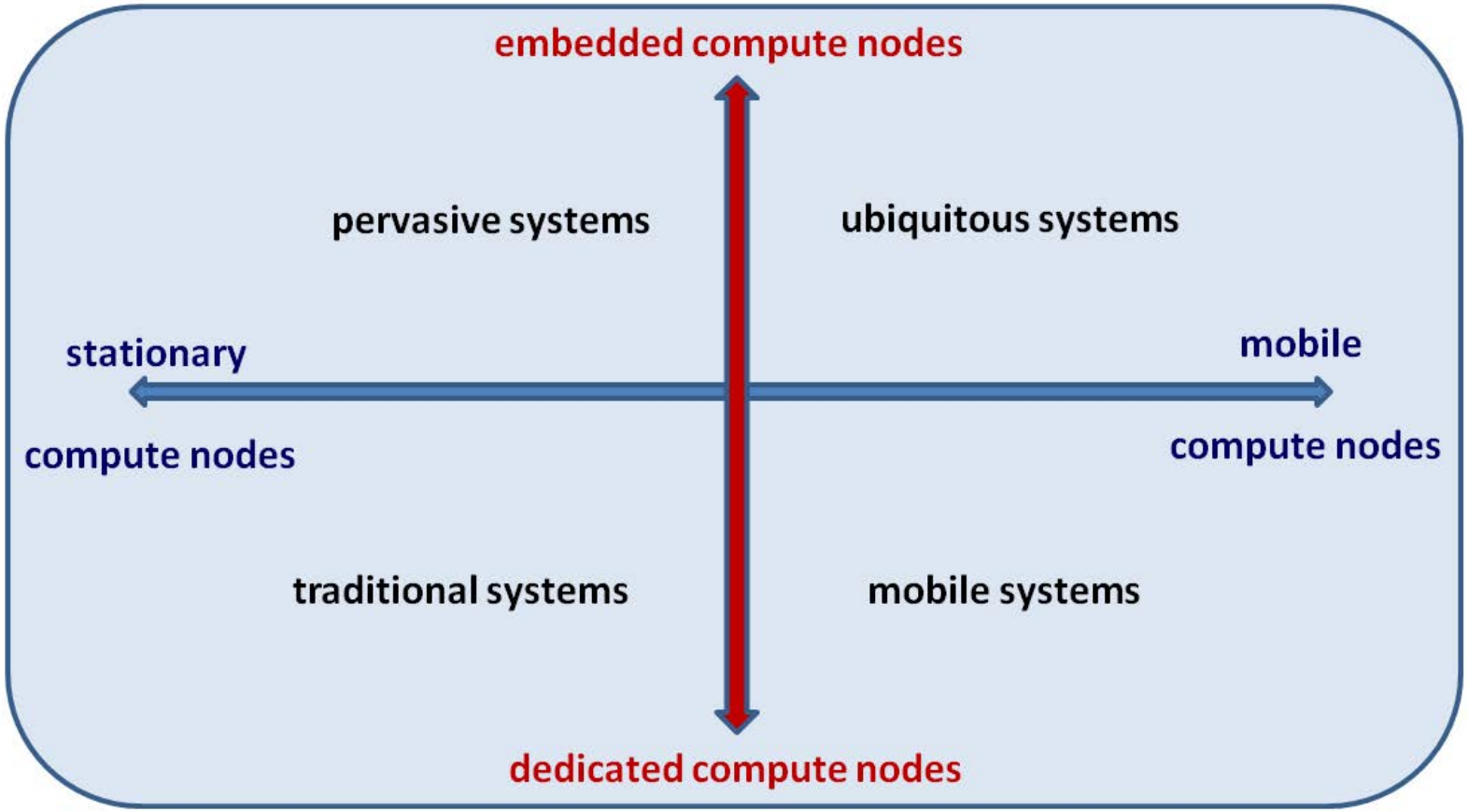
▶ There is more to a distributed system than infrastructure:
- software layers that allow for controlled information exchange and coordination among applications on distributed nodes
- mechanisms to get work towards a common goal done
- awareness of different *local vs. remote views* on the same DS

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
    $\Longrightarrow$ 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, ...
    $\Longrightarrow$ 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

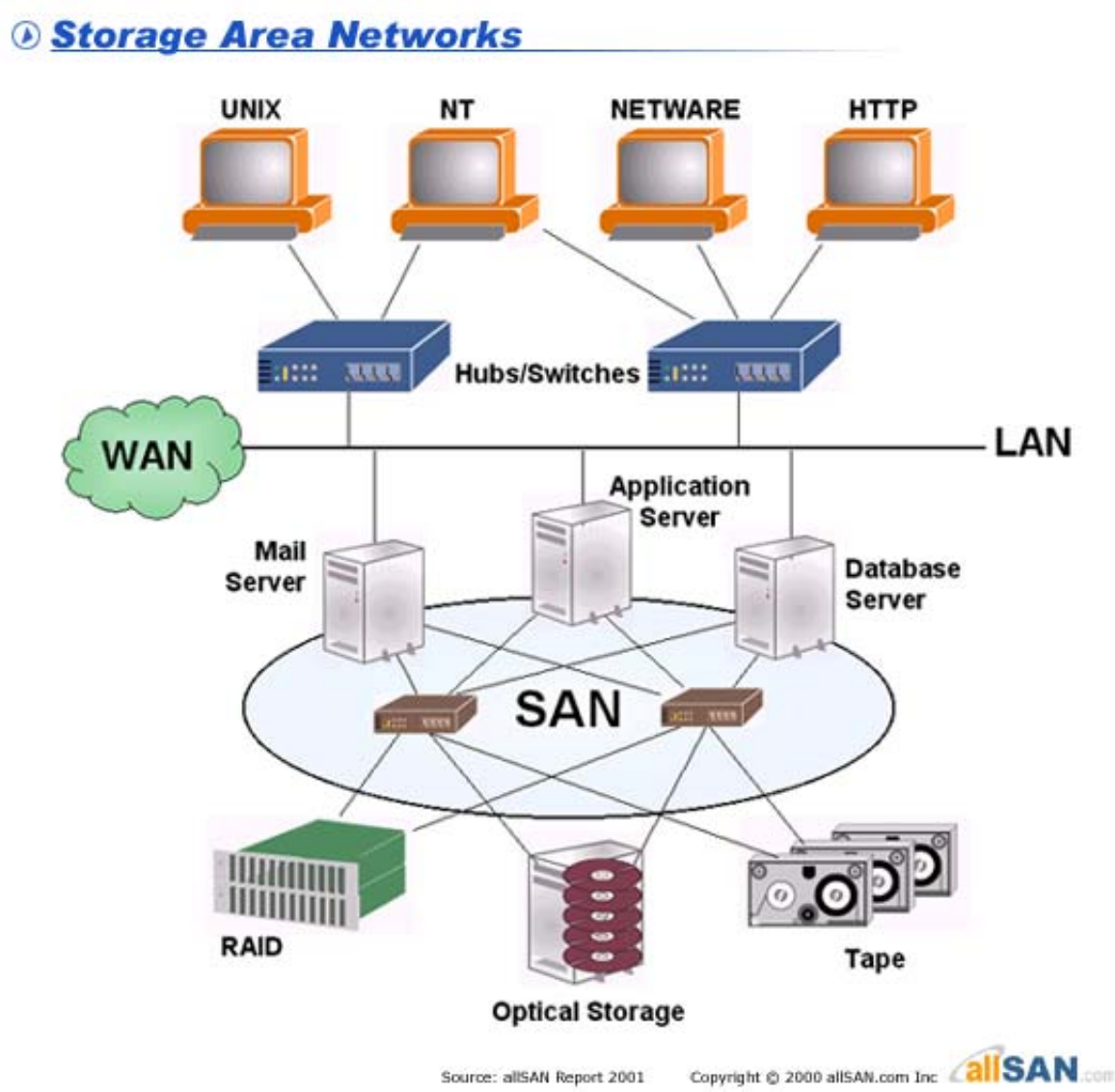
Example: Storage Area Networks as low-level DS

SANs abstract from:

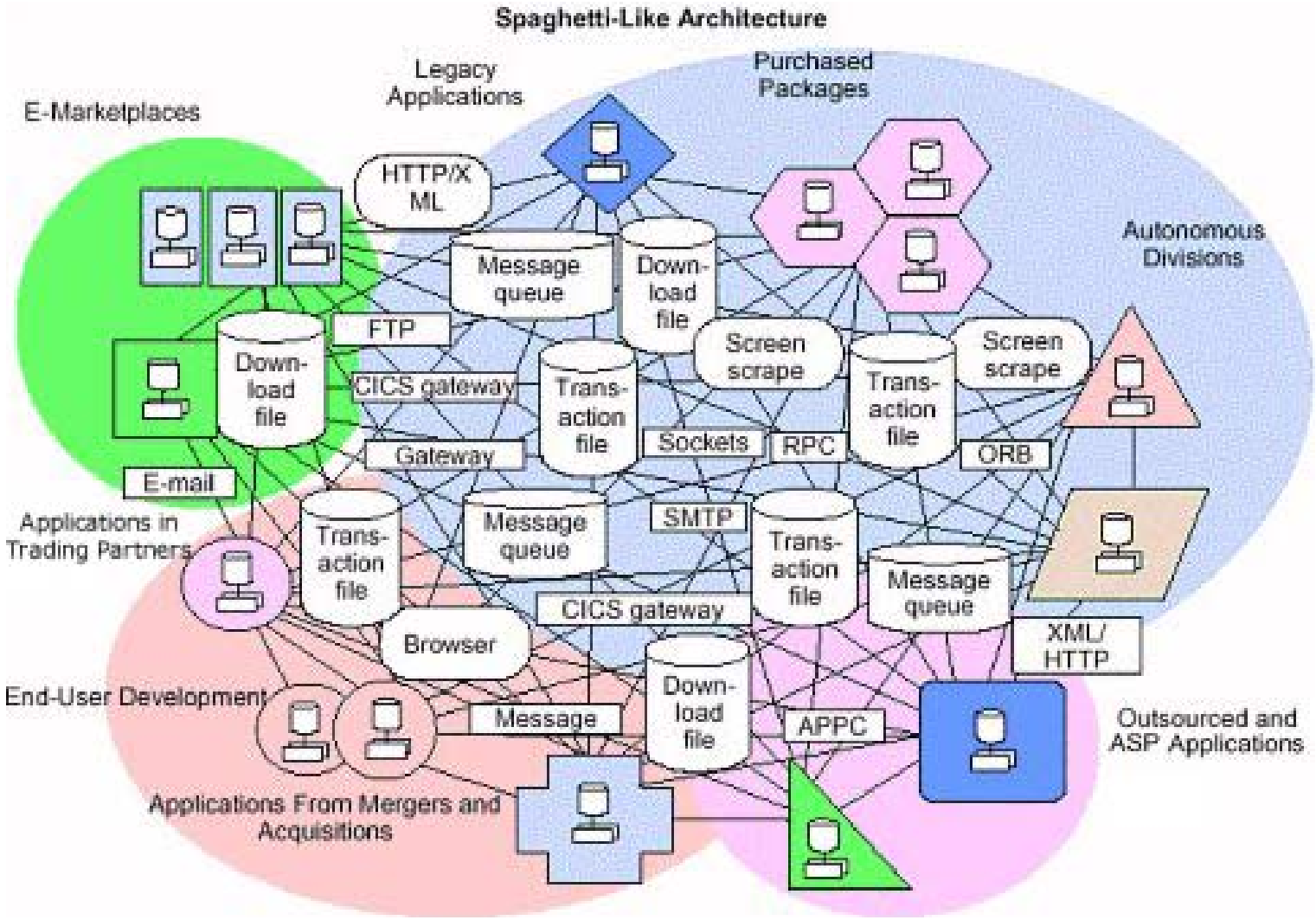
- network details
- heterogenous hardware
- heterogenous OS
- server/storage correlation

and allow for common

- access rules
- security policies
- backup strategies

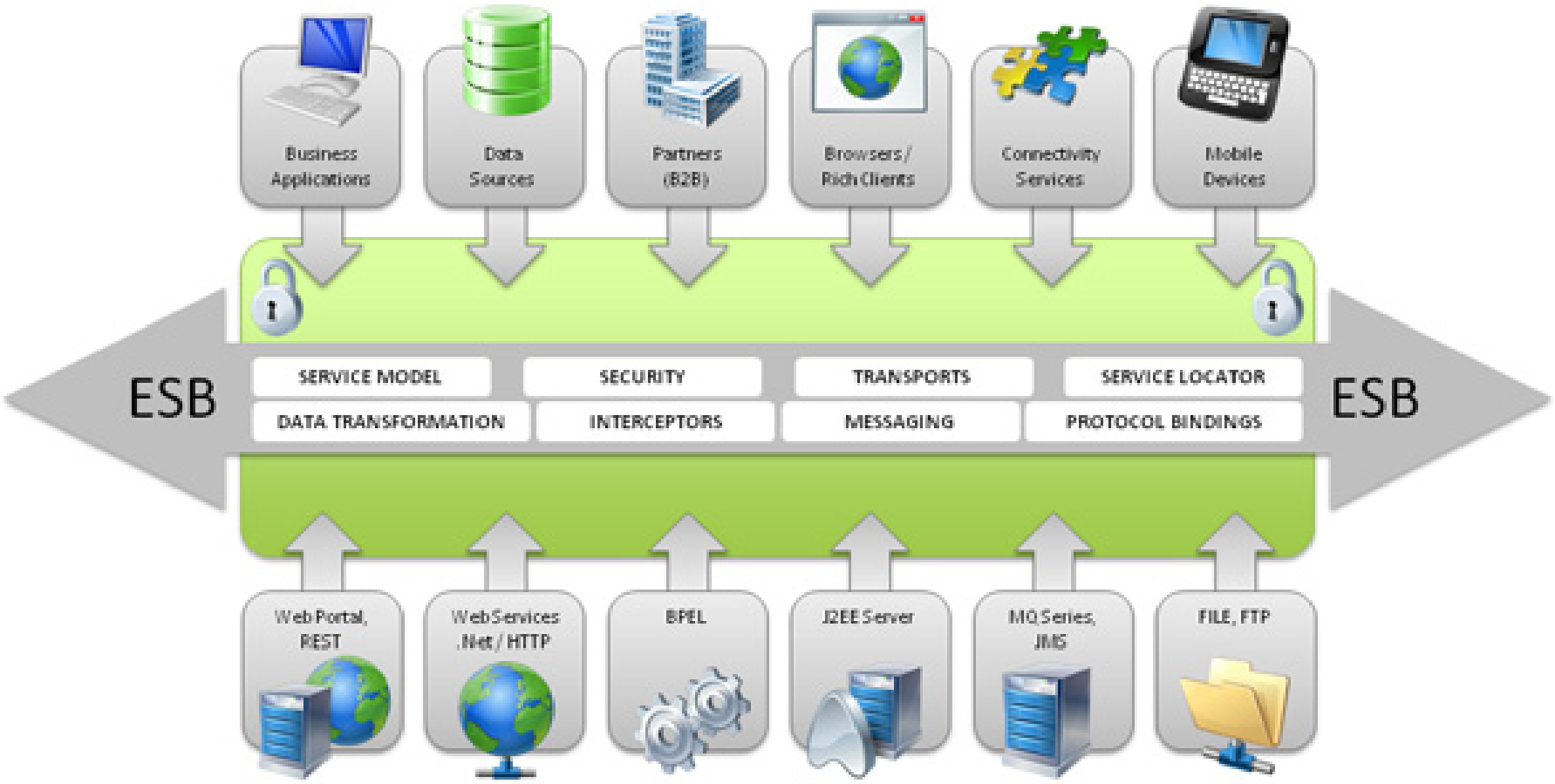


Example: Chaotic Enterprise Application Landscape



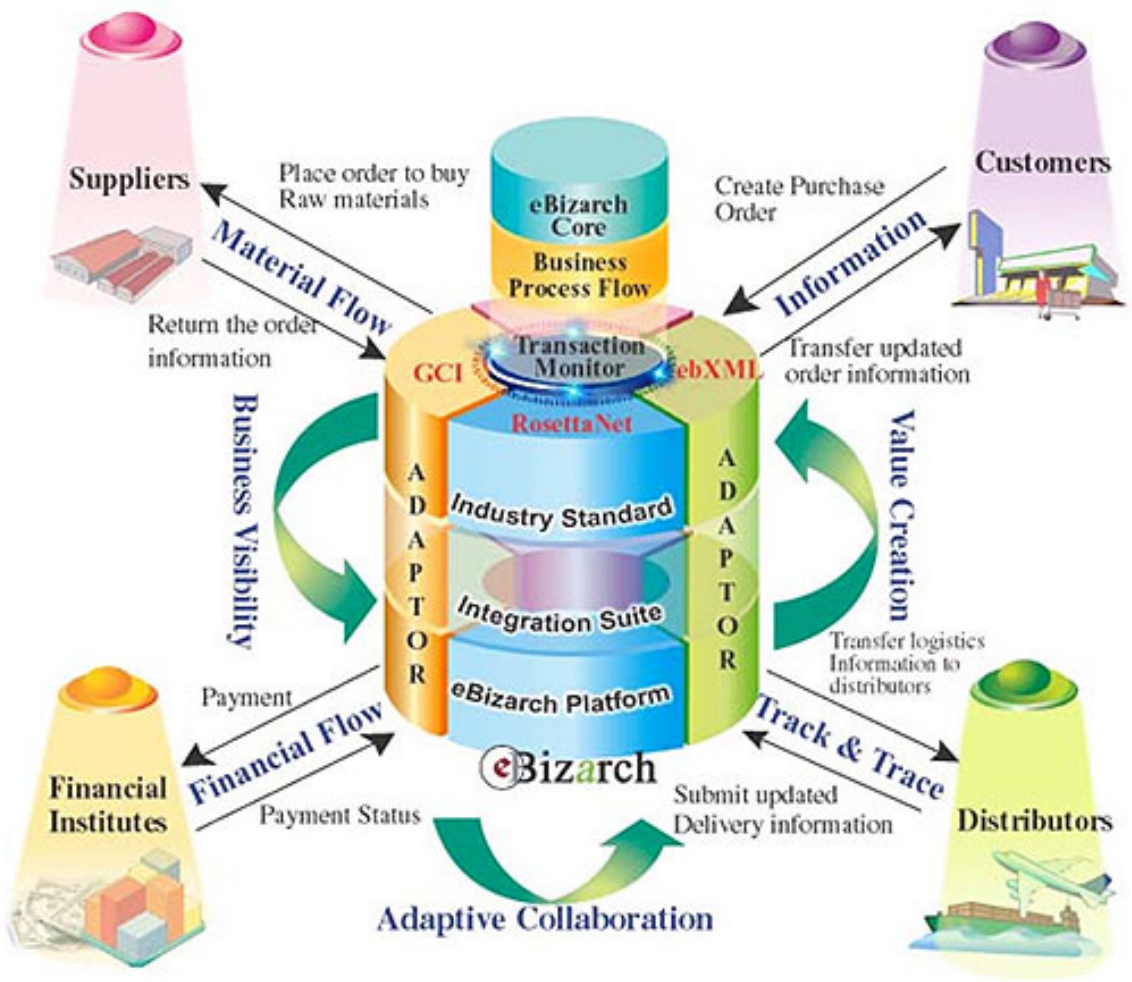
- How to manage such a system? Software Architecture matters!

Flavour 1: Structured EAI – Enterprise Service Bus



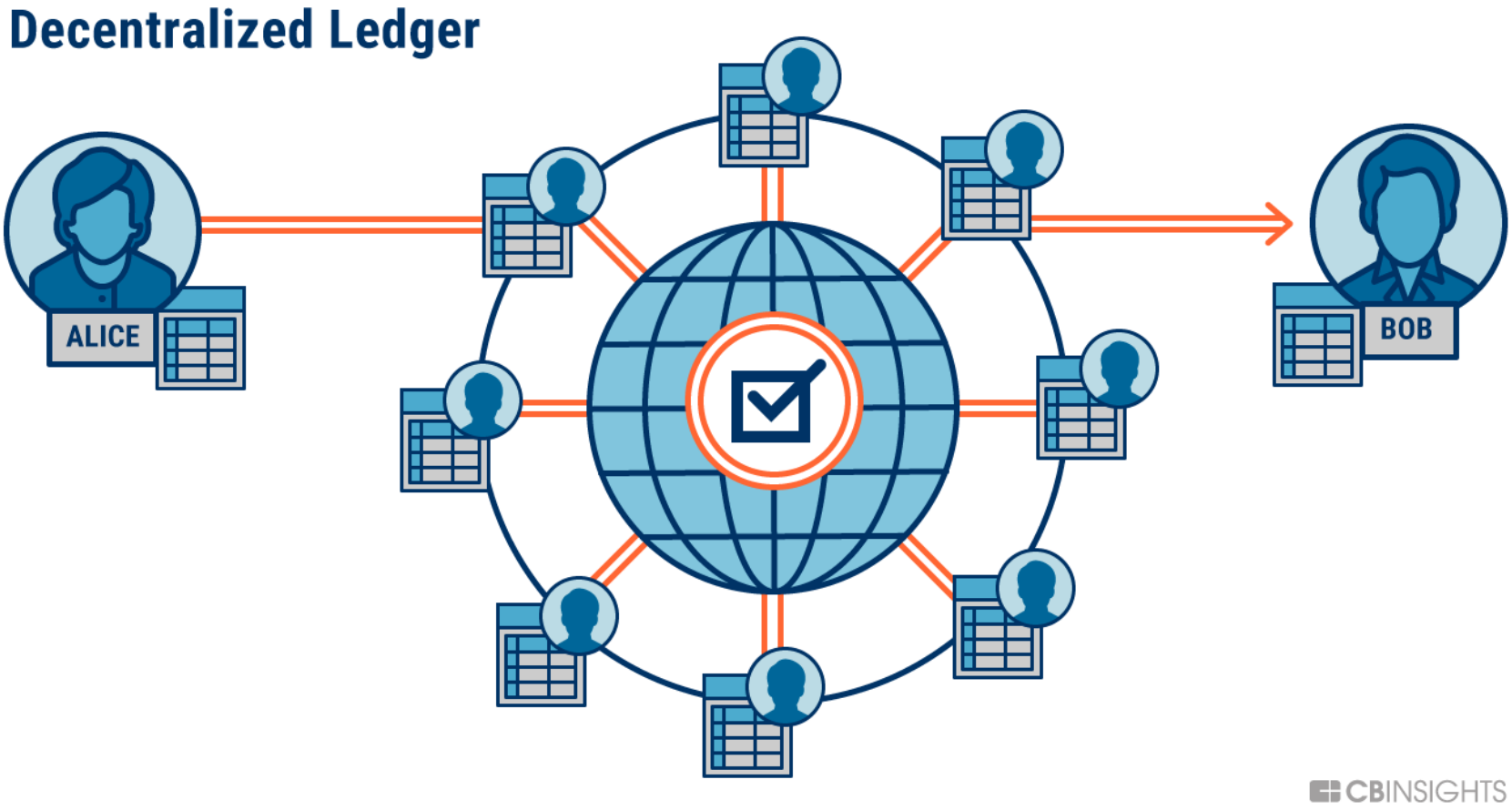
- Unified channel for linking heterogenous systems/technologies
- Single actor has global control $\Longrightarrow$ **'centralized'** distributed
- Well-defined interface to define and implement general policies

Flavour 2: DS Potential for B2B Integration



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

Flavour 3: Peer-2-Peer Networks for Transactions



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

Distributed vs. Decentralized Systems

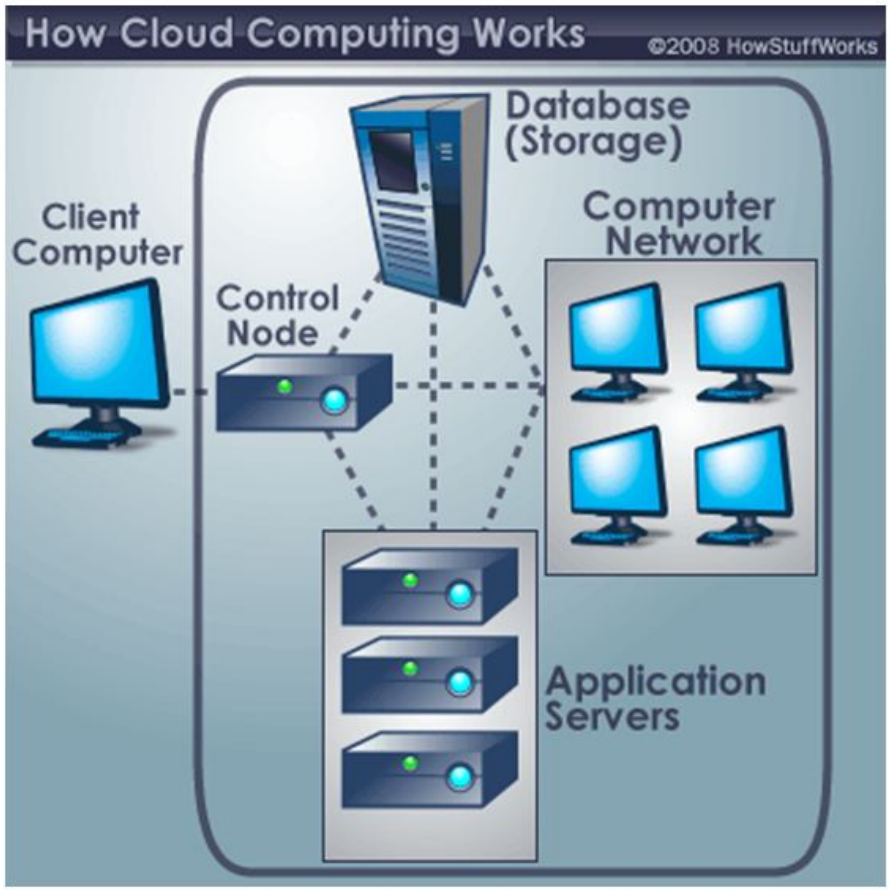
* distributed: matter of location, technology and architecture
* (de-)centralized: matter of control and organization

[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. $\Longrightarrow$ centralized
- Distributed Ledger like, e.g. Blockchain network $\Longrightarrow$ decentralized
- P2P network for file sharing $\Longrightarrow$ decentralized

Clouds – A Modern Prototype for DS Abstraction

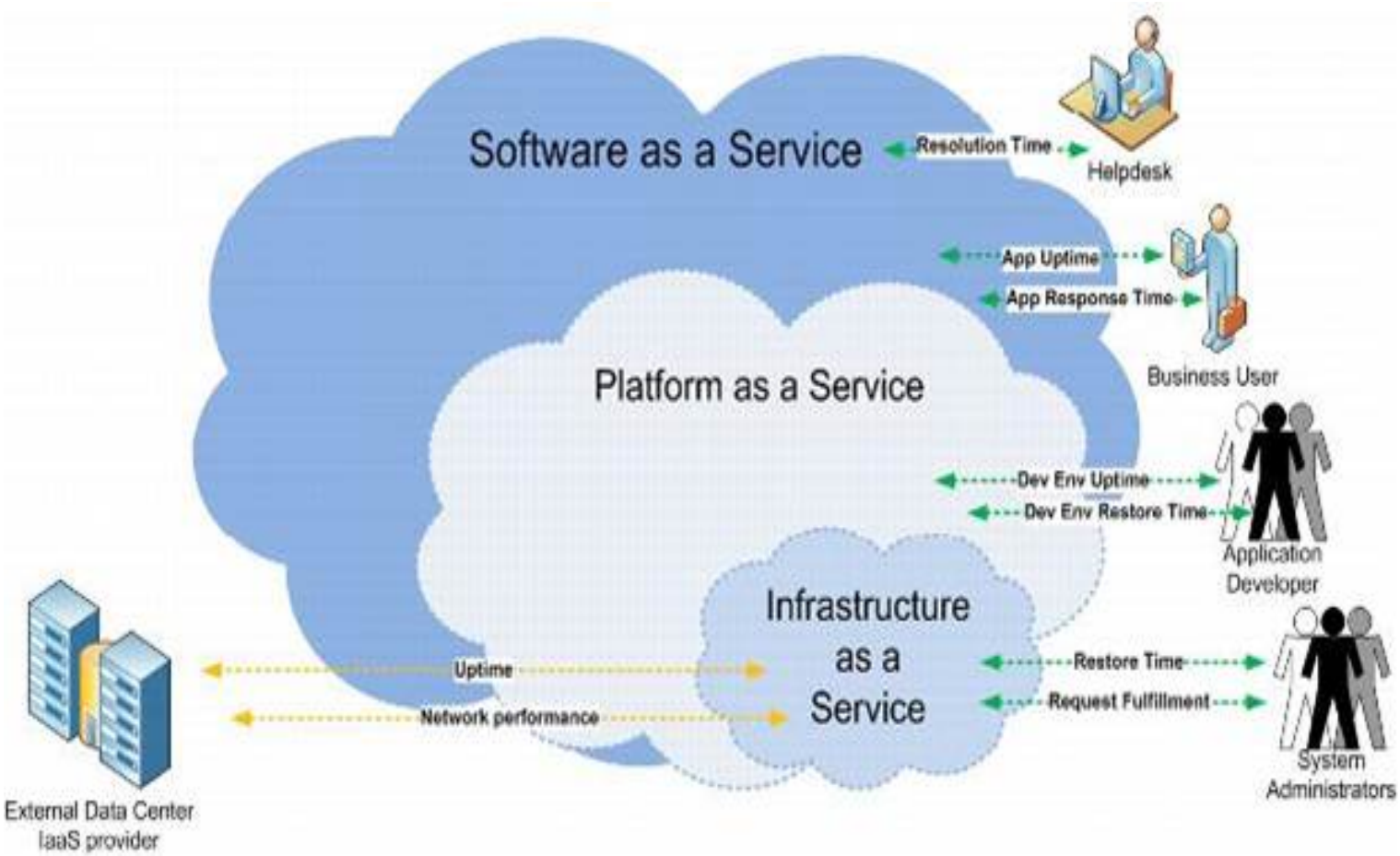


- **Cloud Usage** allows for a unified working environment despite the heterogeneity of the underlying infrastructure.
- **Cloud Implementation** details require state-of-the-art techniques w.r.t. reliability, scalability, access rights etc.

left

right

Cloud Computing – Different Abstraction Layers



Hardware **Host** $\longrightarrow$ **Build** $\longrightarrow$ **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)

    $\Longrightarrow$

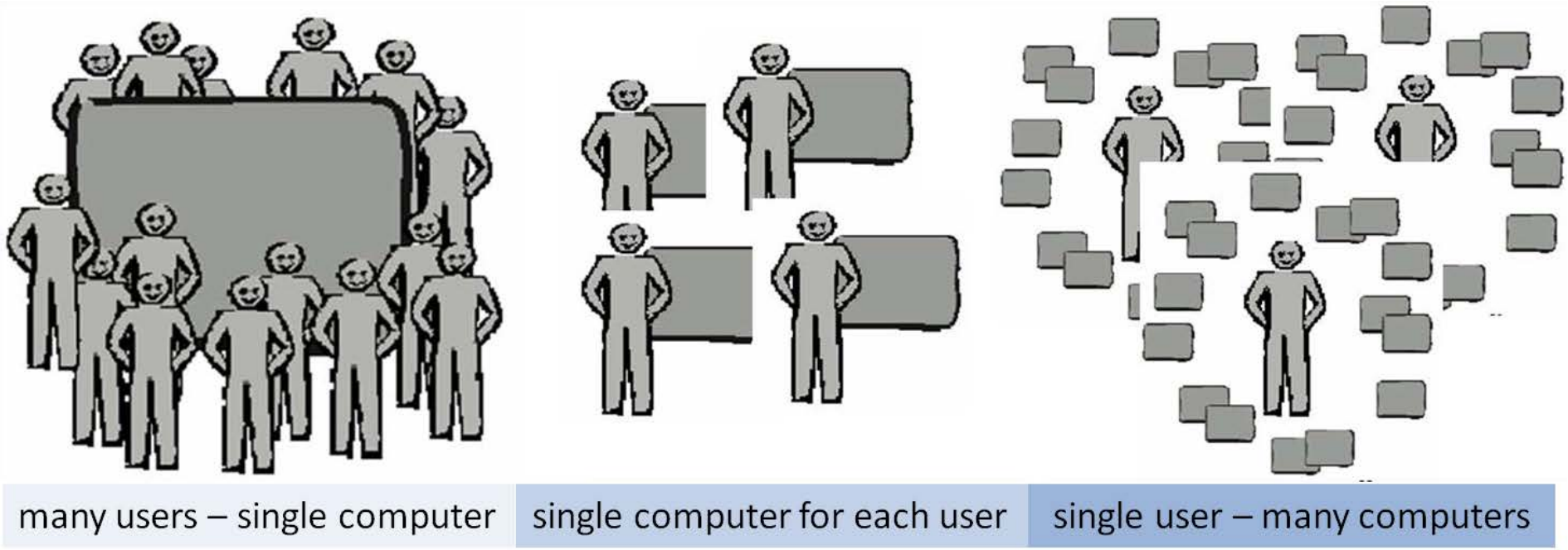
  * computers and computer support (more or less) everywhere
  * hard problem: find suitable, new modes of interaction

- **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

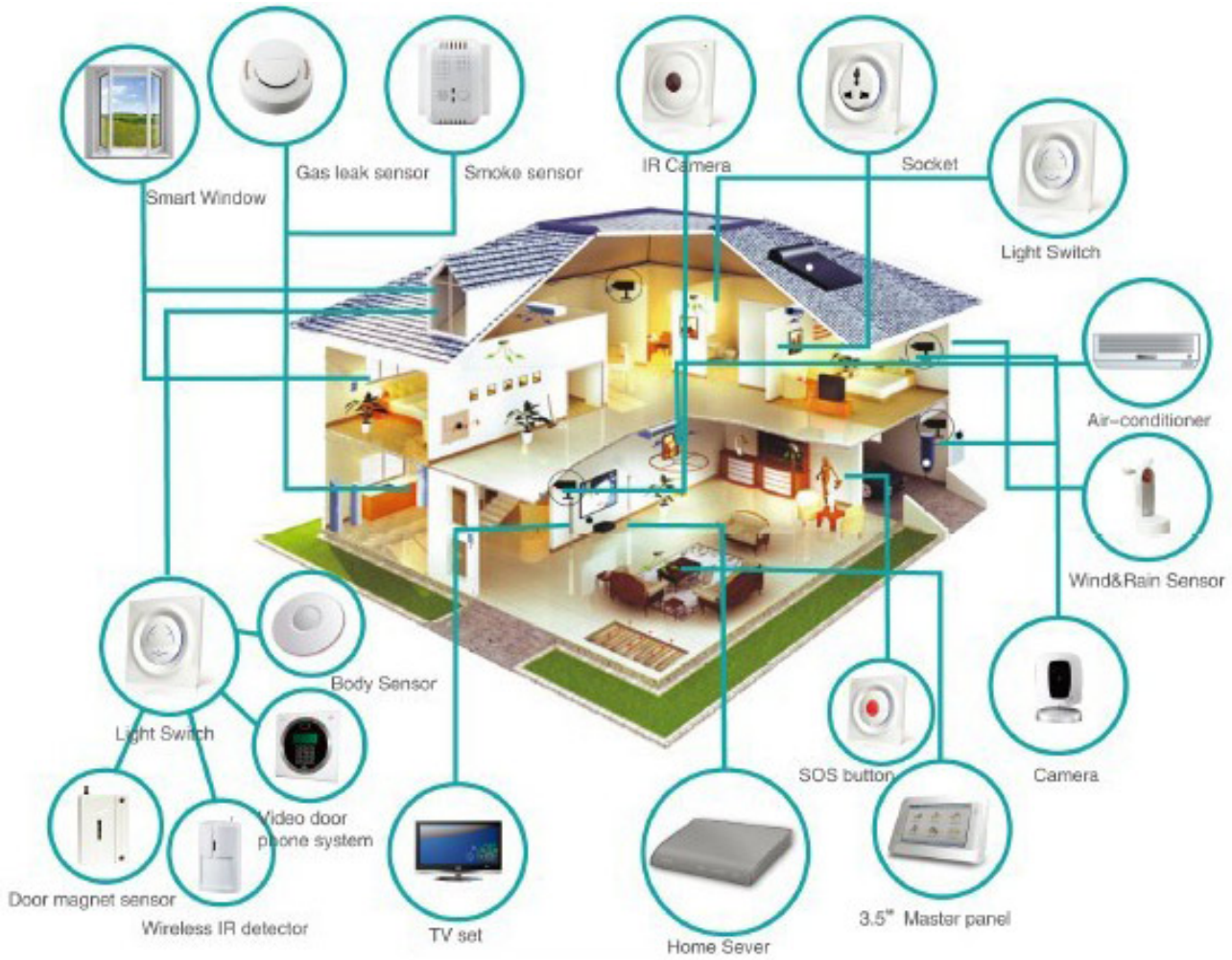
Mark Weiser 1952-1999



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

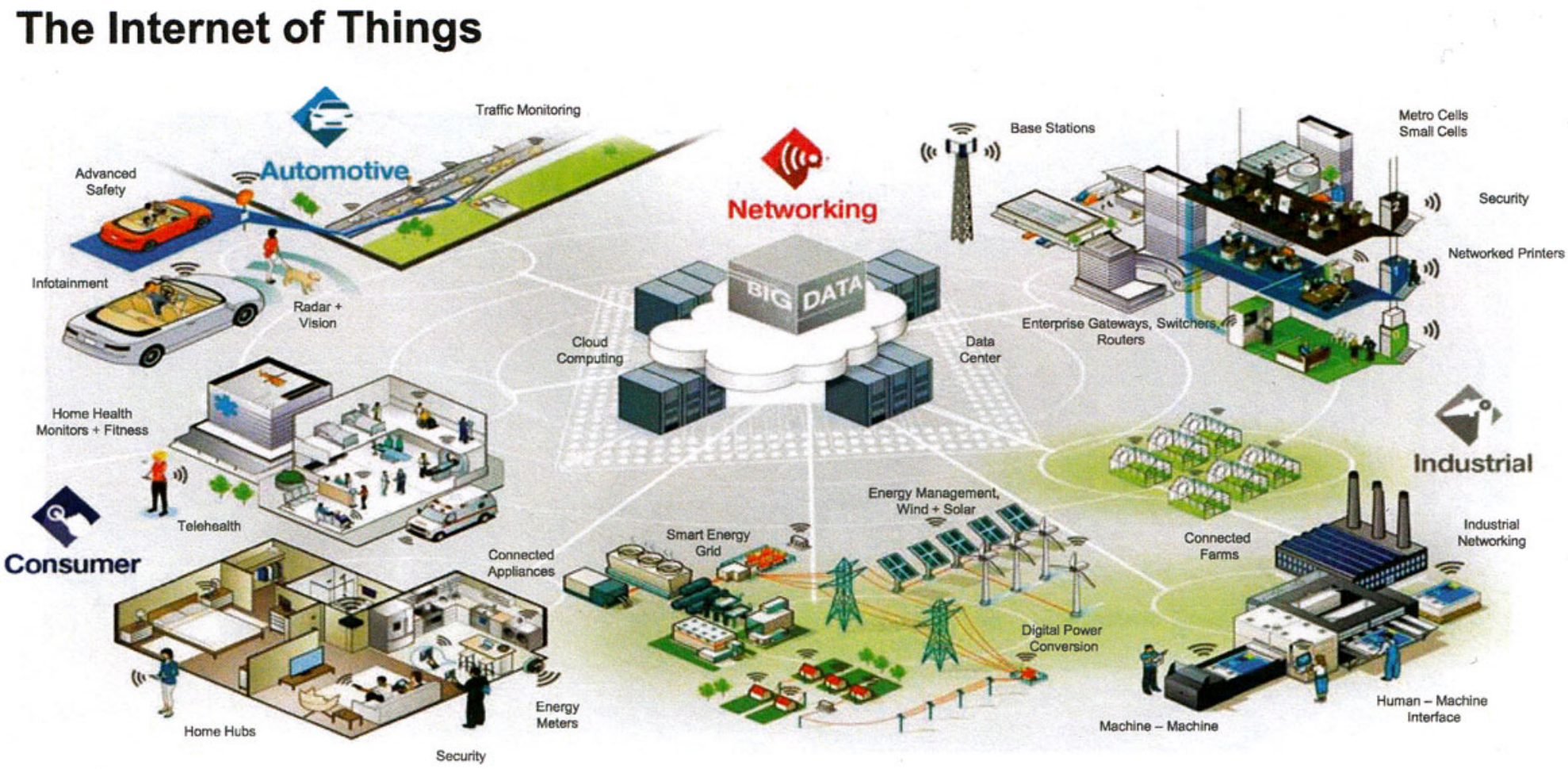
  Computers will act like books, windows, walks around the block, phone calls to relatives. They won't replace these, but augment them, make them easier, more fun. Dwelling with computers, they become part of the informing environment, like weather, like street sounds.

Example: Pervasive Computing at work



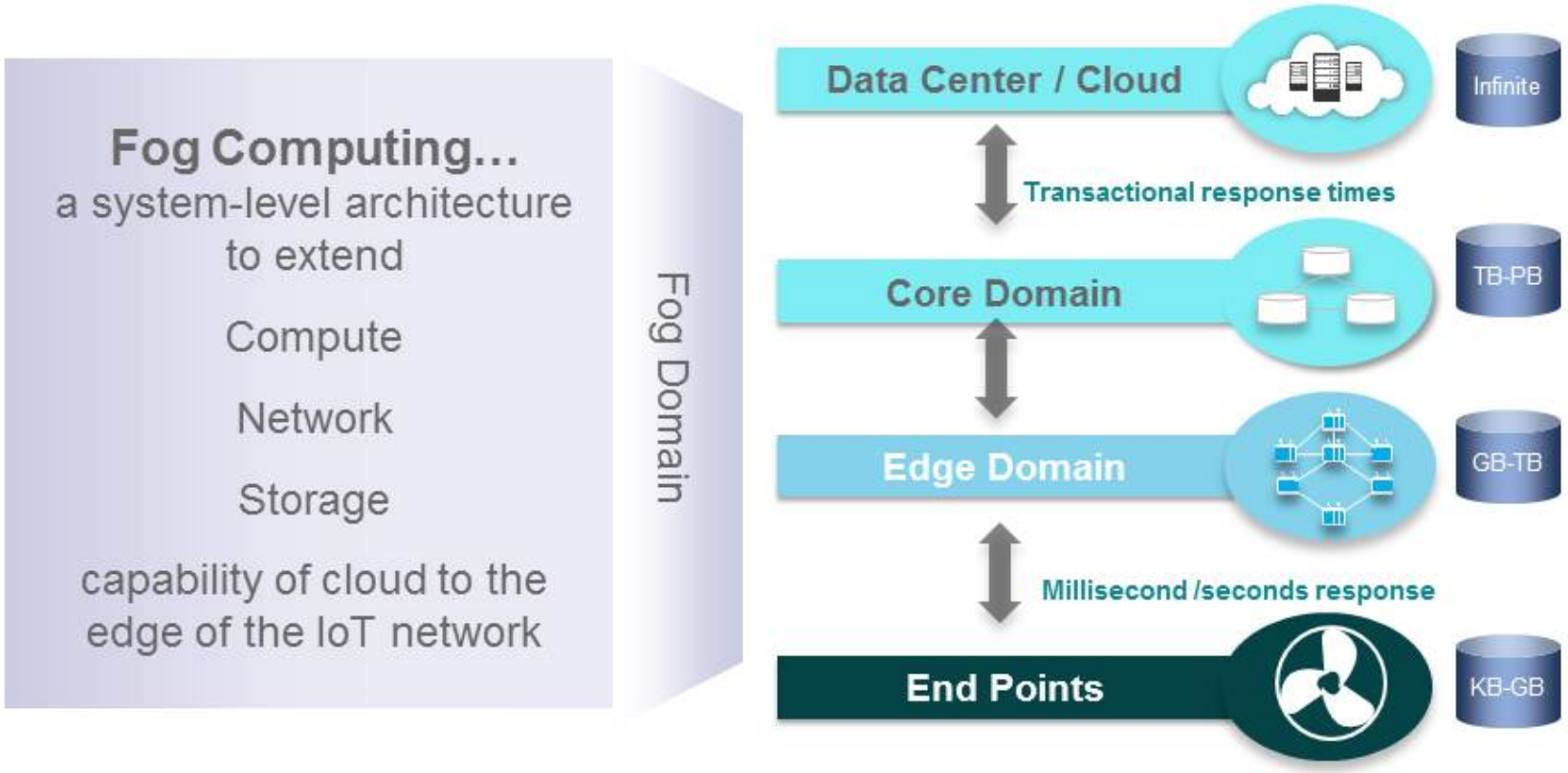
Applications: Energy management, security, . . ., more comfort
. . . *assisted living*, esp. for the elderly/handicapped

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



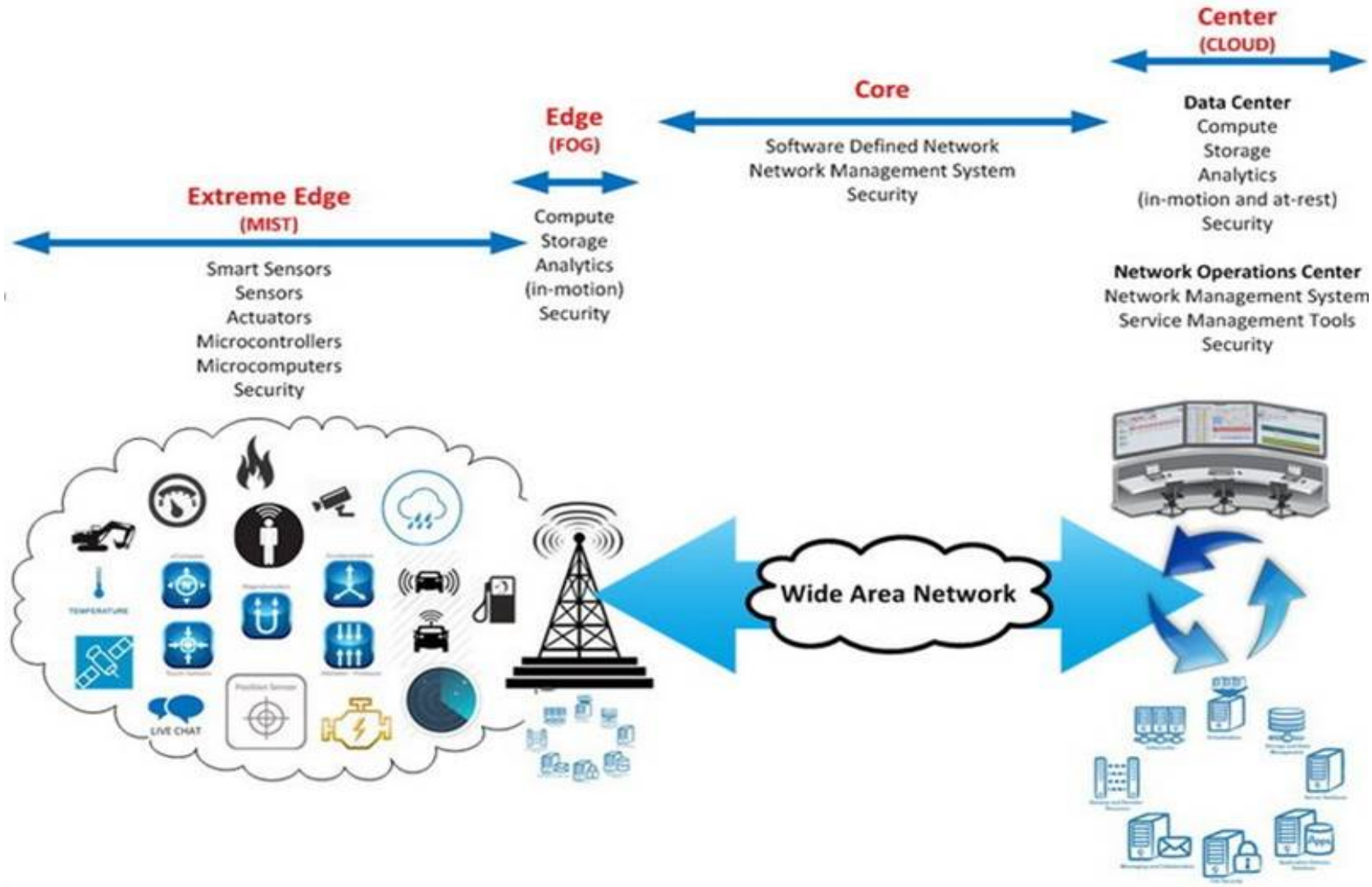
- Connectivity: Sensors – Ad-hoc Networks – LANs – WANs
- One single, global system? How to ensure reliable infrastructure?

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

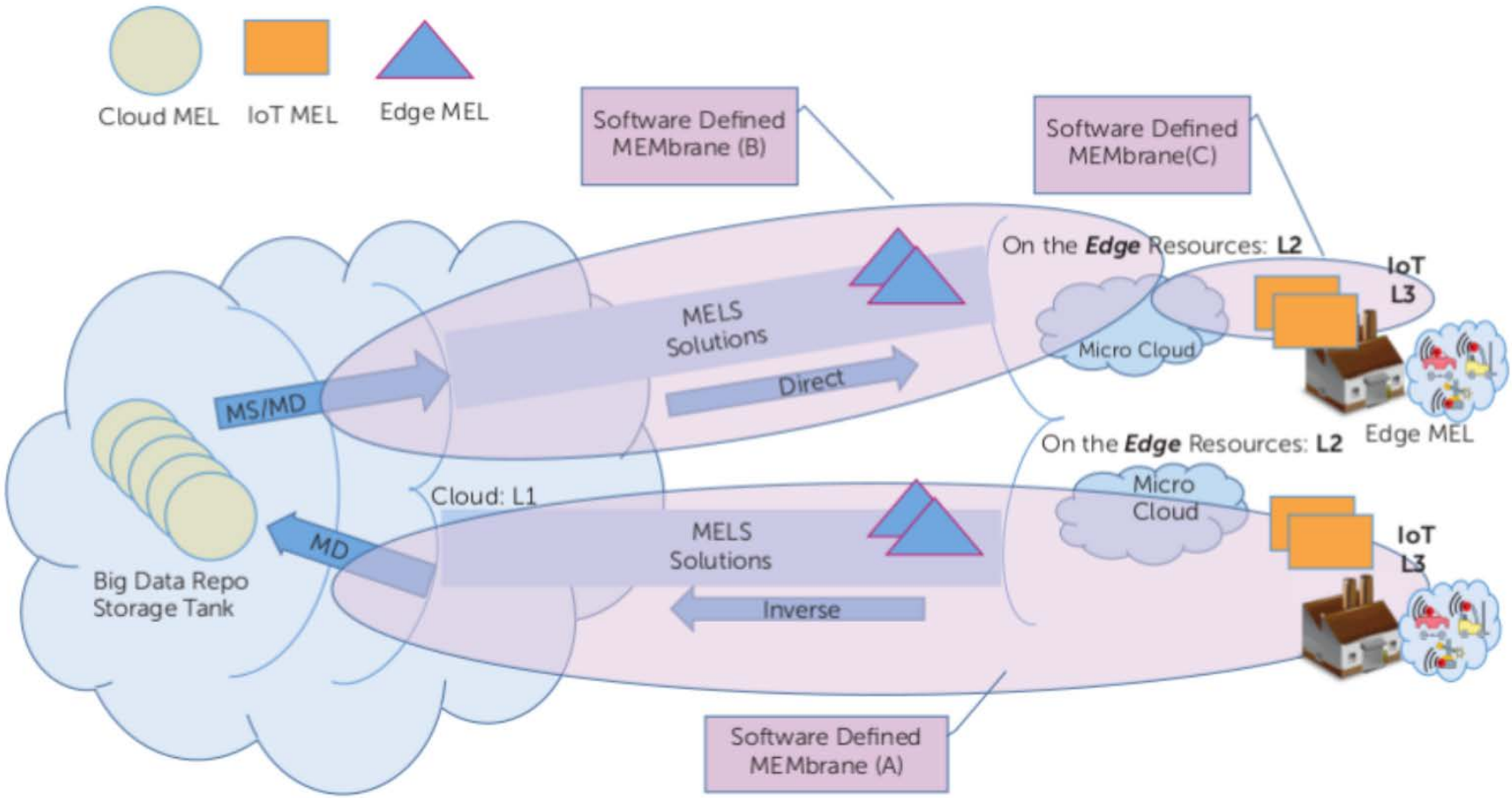
Where to Place the Compute-Load?



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 metaphore:

- 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 $\Longrightarrow$ **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 $\Longrightarrow$ **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

  $\Longrightarrow$ *Well-known, but now: always everywhere available*

- **Mobile Services:** available on smart phones and tablets

  $\Longrightarrow$ *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
  $\Longrightarrow$ *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?'

  $\Longrightarrow$ *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
  $\Longrightarrow$ a system may be as vulnerable as the weakest link of a chain
- DS consist of many nodes
  $\Longrightarrow$ there is a high probability of failures in single nodes
- DS do not fail as a whole, but are prone to **partial failures**

$\Longrightarrow$ The more essential components, the more possibilities for failure
$\Longrightarrow$ 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 $\Longrightarrow$ robust system
- **AND**-Distribution (much cheaper) $\Longrightarrow$ 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

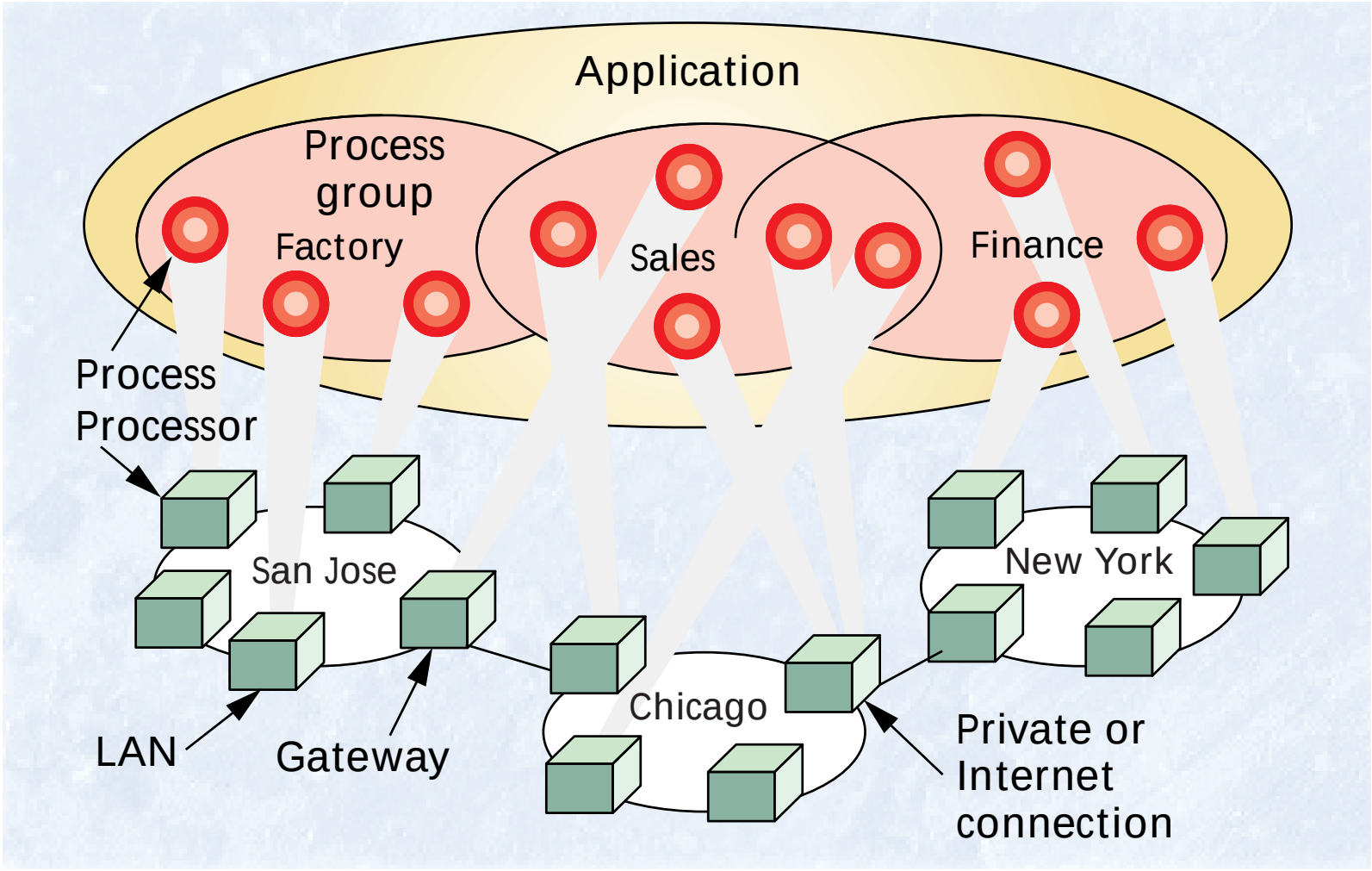
◀ DS comprise a *local vs. remote view*: sharing
- data may act as the basis for *data leaks* and data loss
- resources may lead to *resource abuse* or *unauthorized usage*

◀ **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

$\Longrightarrow$ Approximation requires distributed algorithms

Example: A simple consistency problem – 1

IE3 Computer 3/1998 pg. 63



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

Example: A simple consistency problem – 2

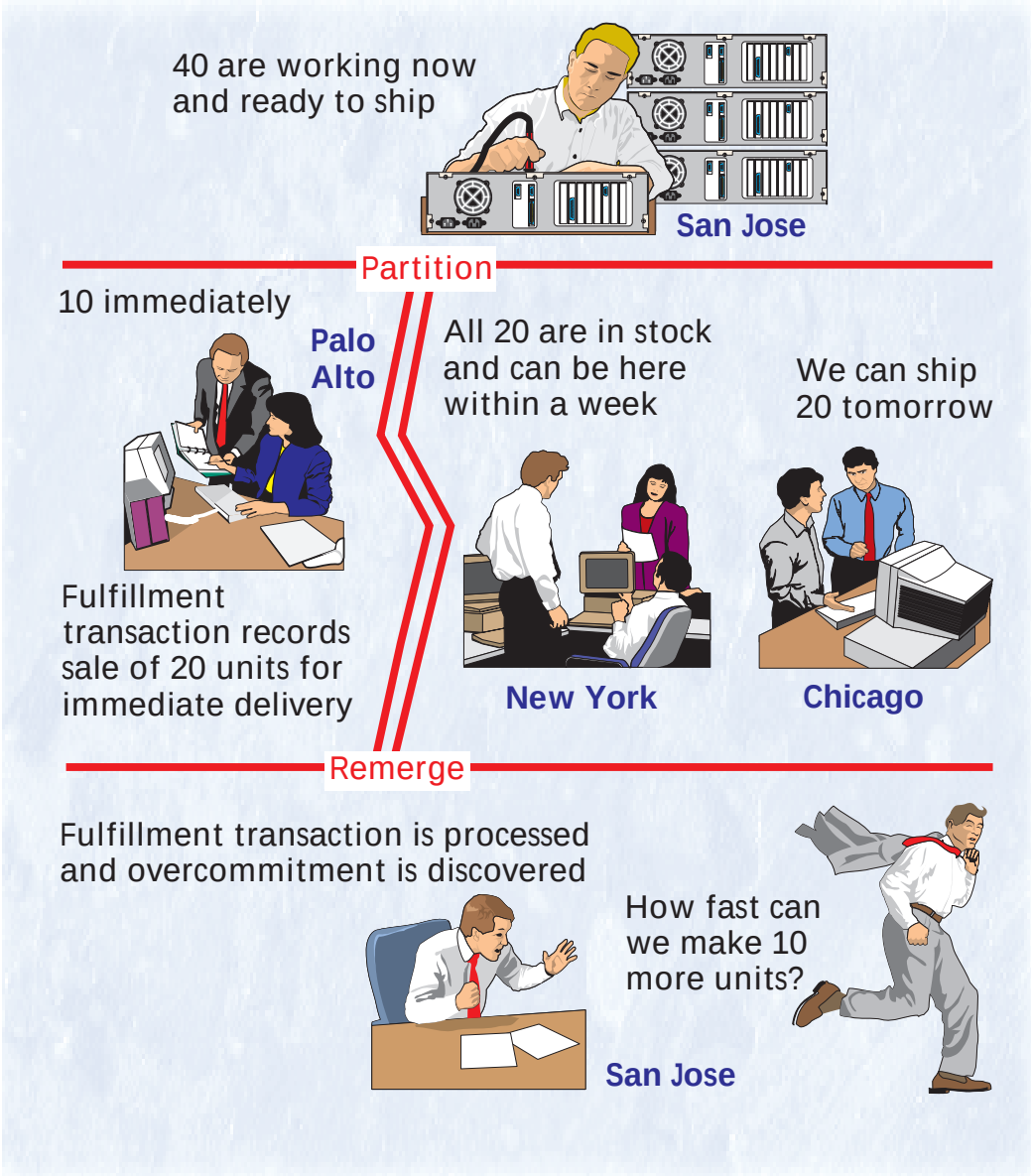
IE3 Computer 3/1998 pg. 63

Network Partitioning:

- ▷ 40 units produced
- ◀ network problem

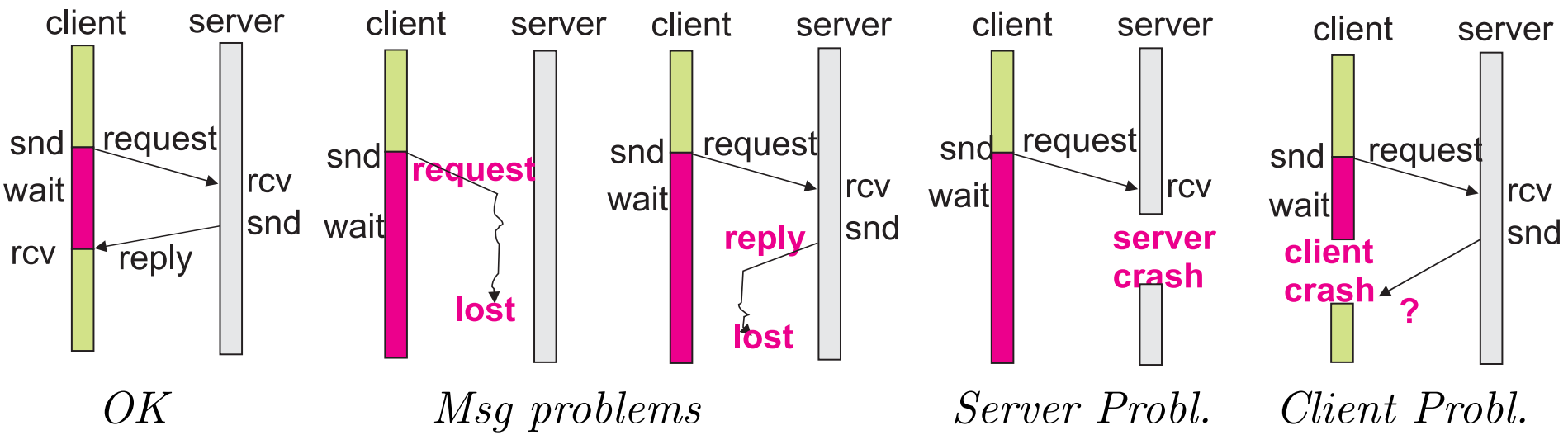
- 10+(20+20) sold
- all local views ok

- 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 $\Longrightarrow$ possible effect on server?

$\Longrightarrow$ *Time-Outs* are needed but: *For how long should a node wait?*

Synchronous vs. Asynchronous Systems

Hadzilacos/Tueg

1. A distributed system is called **synchronous** $:\Longleftrightarrow$
   (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** $:\Longleftrightarrow$
   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)

$\Longrightarrow$ **Synchrony is a fundamental prerequisite for building DS!**

Limits of DS even in synchronous settings

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

Brewer 2000
2012
2017
Lynch Gilbert 2002

1. **C**onsistency of states among all nodes in a strong sense
2. **A**vailability of data and services
3. **P**artition tolerance of underlying communication network

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

- In the $worst\text{-}case$ 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 $not\ available$?

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
    $\Longrightarrow$ 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
    worst case
    - 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.

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. ♦

$\Longrightarrow$ **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. ♦

$\Longrightarrow$ **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.

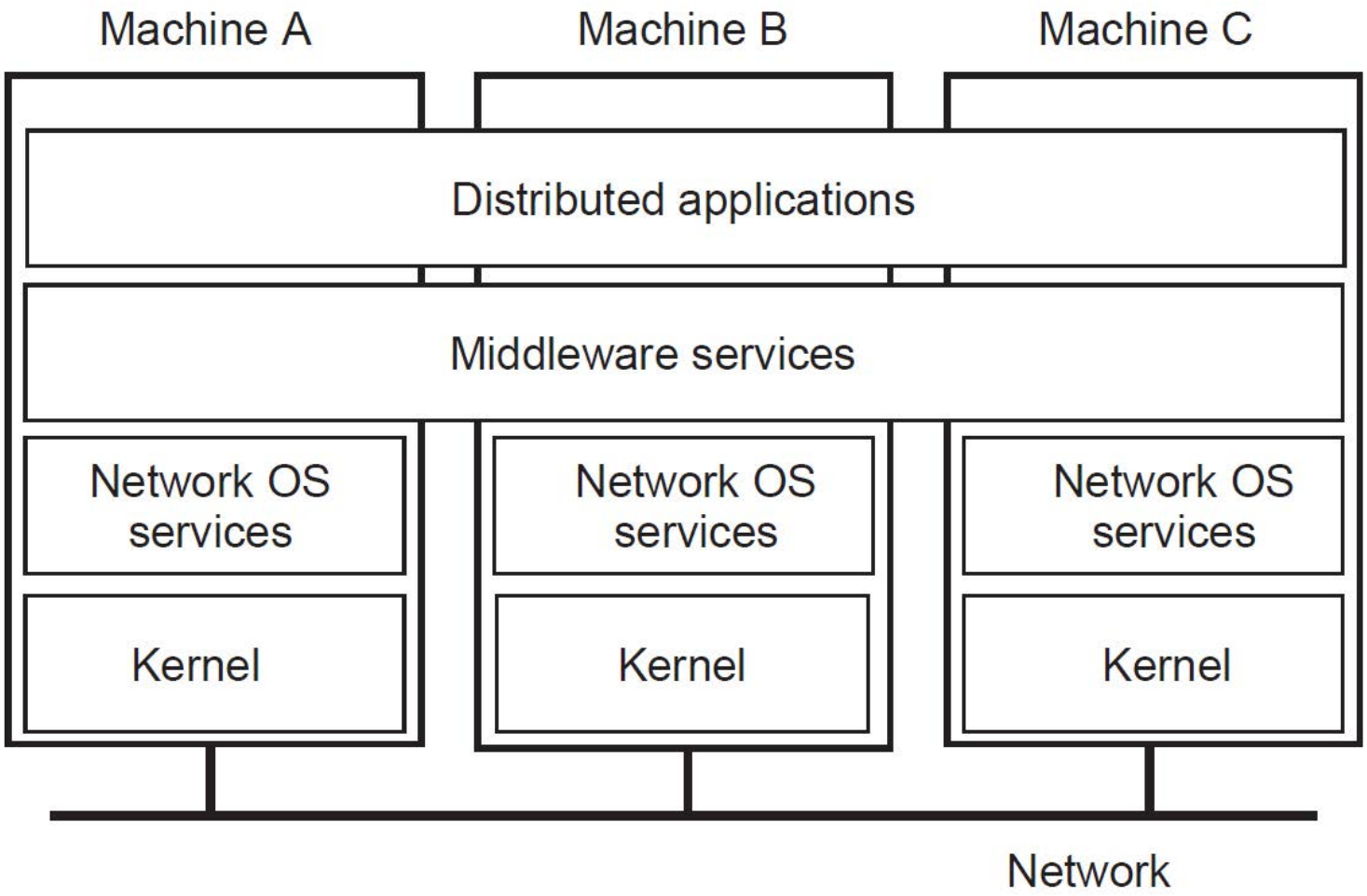
How to build 'good' distributed systems - 2

- Distributed systems are complex software systems
  $\Longrightarrow$ **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

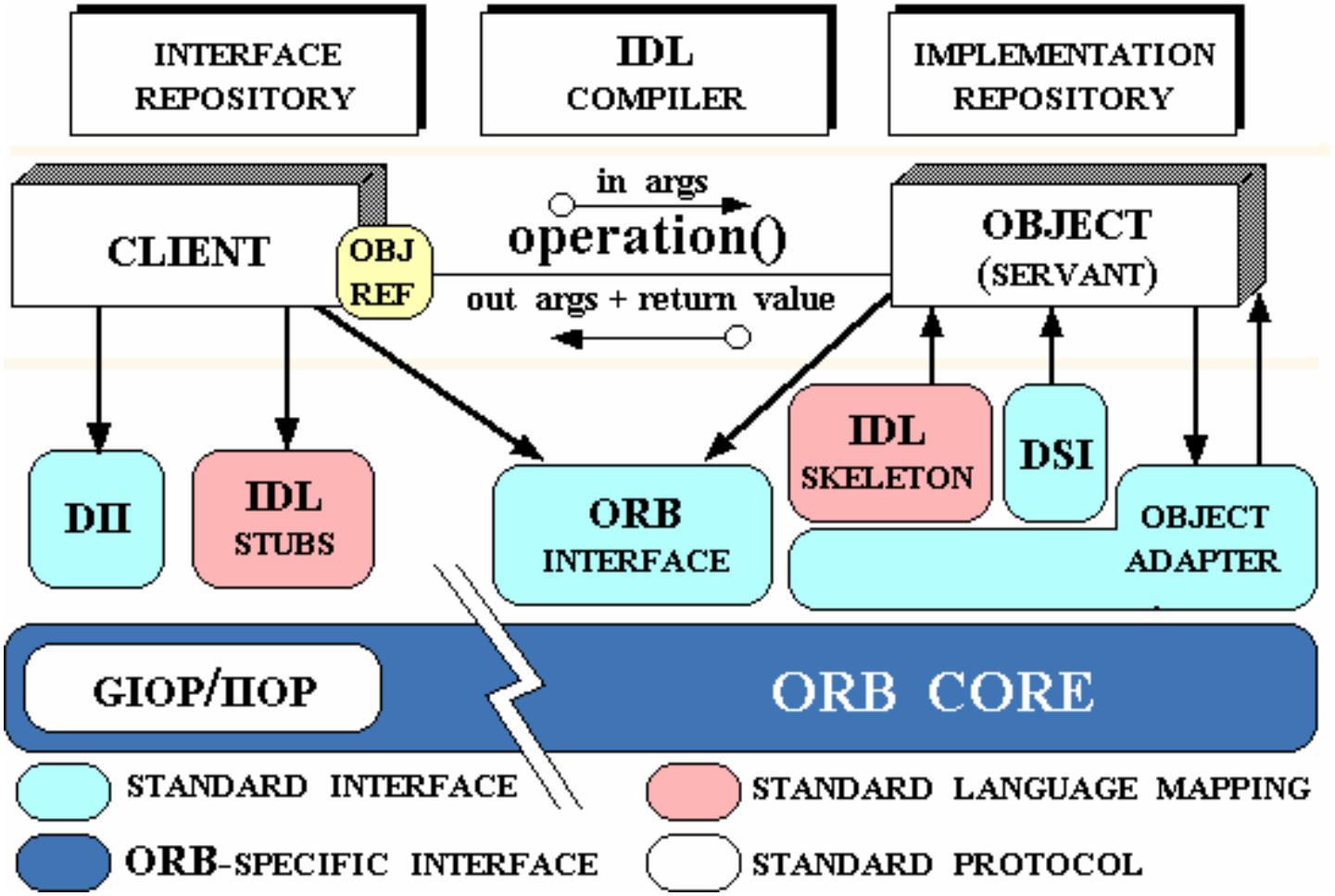
- Use standards and standard tools whenever available.
- Use available *middleware systems* iff they fit your needs instead of programming everything from scratch:
  - ◁ complex and requires time and efforts to get familiar with
  - ▷ in the end you get much more work done ...

Typical Middleware Layer Architecture for DS



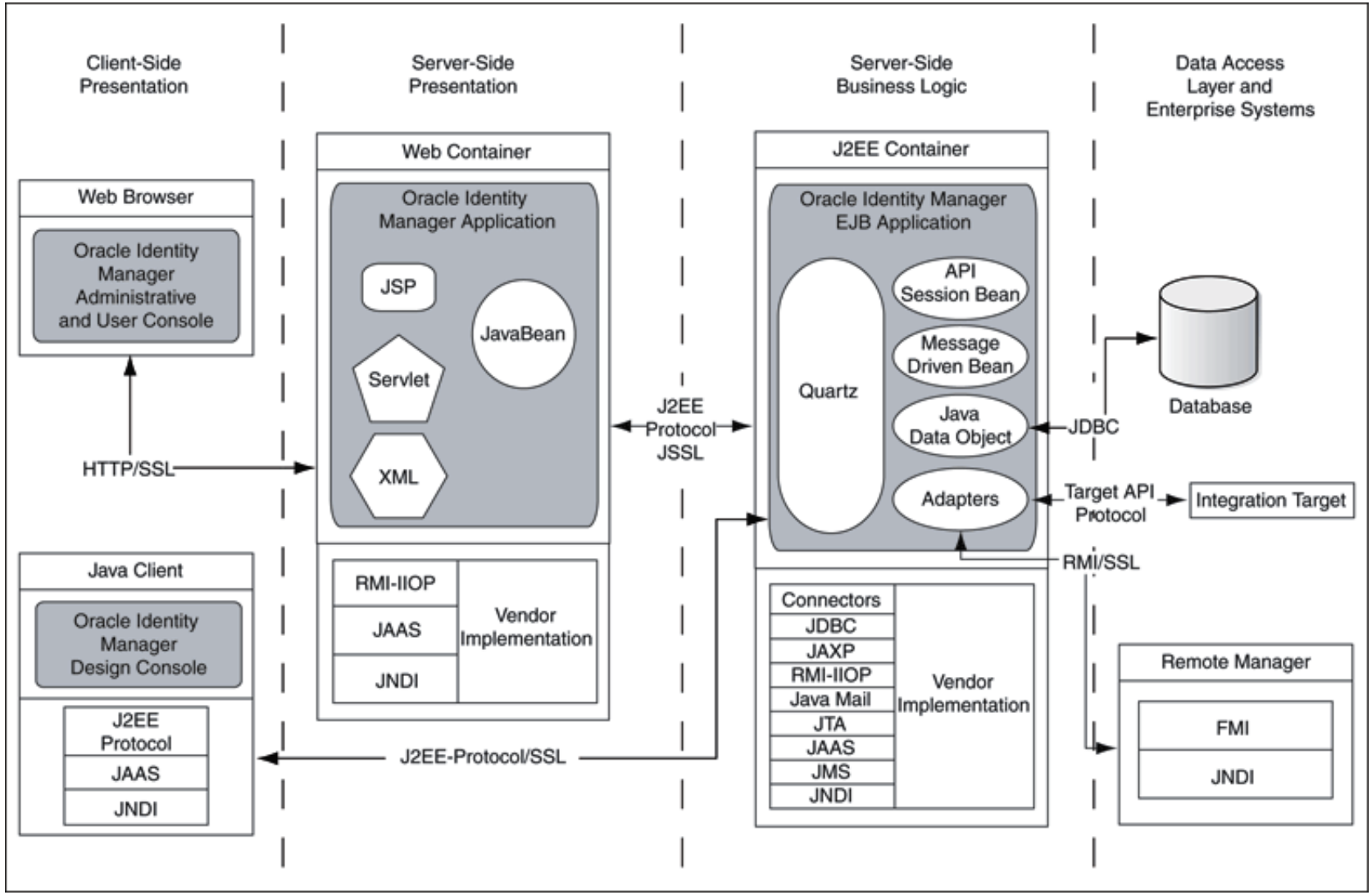
- abstracts from network and heterogeneity
- additional services, e.g., global naming, messaging, DB interface
- neutral w.r.t. applications: basis for many different applications

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

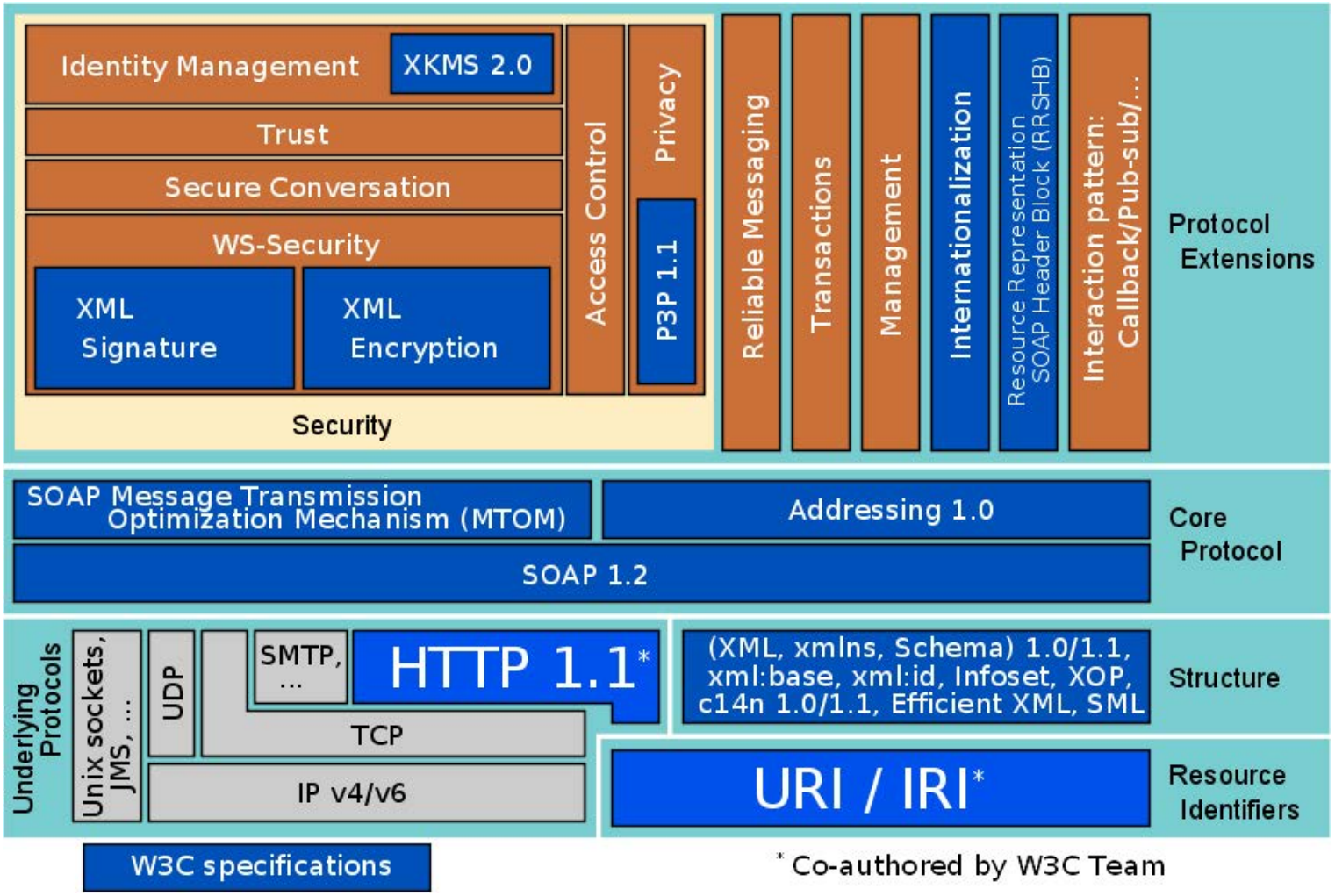


- Common Interface Definition Language for multi-language RPCs
- standardized messaging, event handling, service directories, ...

Example 2: J2EE multi-tier architecture (1999)

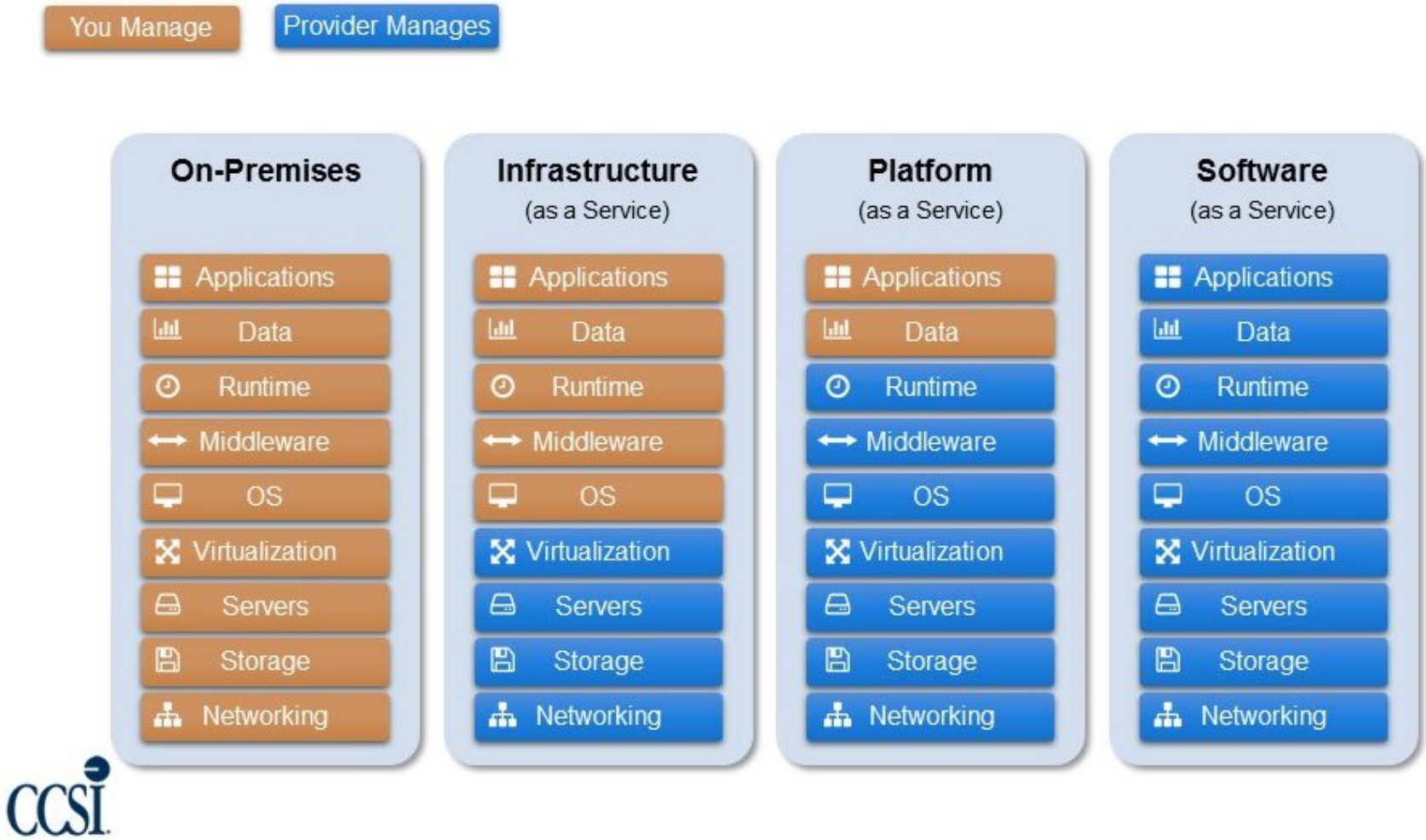


Example 3: Service Oriented Architecture Stack



- 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 $\longrightarrow$ 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

End of Chapt.I