Technische Universität Darmstadt

Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 1: Introduction

Lecturer: Dr. Benedikt Schmidt, Dr. Immanuel Schweizer, Prof. Dr. Max Mühlhäuser

Computer Networks: Recall NCS

Computer Network := interconnected collection of autonomous computers

CN := {AS} ∪ CSS

1. AS: Autonomous System (node) := pair (CPU, local memory) [+ net + 'addressability']
   - 1 AS may be multiprocessor (shared memory)
   - Note: AS able to run stand-alone
   - Note: a shoebox may contain a distributed system!
2. CSS: Communication Subsystem („the network")
   := „whatever enables the ASs to exchange messages"
   - Wired/wireless, arbitrary topology (ring, star, bus, [partly] meshed,...)
   - Note: AS may be part of CSS (Host & Router)

For **Ubiquitous Computing:**

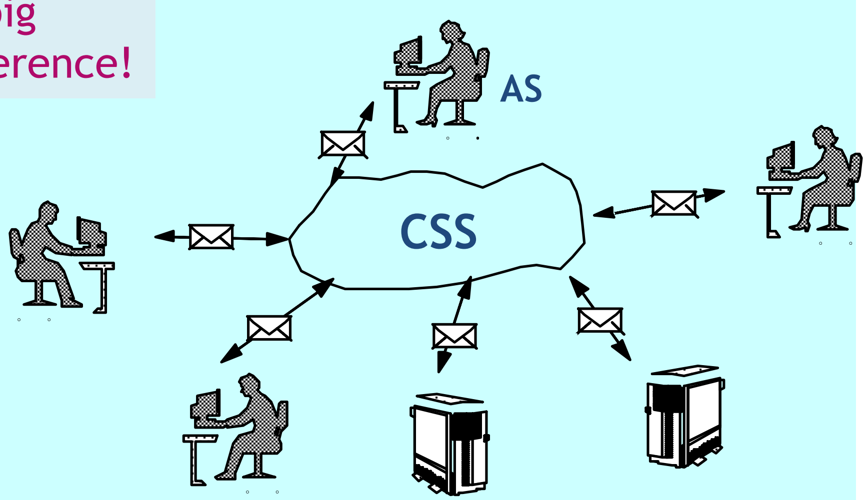
- Consider 4-tuple from above: (address ≈ identity, communication, CPU, memory)
- Ubiquitous computing: your shirt, suitcase may be an Internet node !
- But then: minimalistic node may consist of: (passive?) communication & identity → *not* an AS, no 4-tuple
- This definition of AS is not used here any further, reserved for TK3

Computer networks are everywhere: LAN/Intranet, Internet, in-car, on-body, …

A. "Definition" of Distributed Systems

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

no big difference!



„A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
-- L. Lampport

Summary of many definitions: In a **Distributed System ('DistSys')**, networked computers

– communicate and coordinate their actions **only by passing messages**
– may be spatially separated by **any distance**

Definitions suck! So, what's the **distinction** *Computer Network* ↔ *Distributed System*?
well… not precisely defined either, but:
Distributed Systems* establish some level of transparency atop *Computer Networks
… of locations, distribution, concurrency, performance … (details further below)

B. Basic Problems of DistSys

Basic problems in Distributed Systems form a dangerous *Bermuda Triangle*

Basic Problem #1: Global State Not Accessible (without unacceptable slowdown)

- no synchronized global variables, no global shared memory
- message / agent travelling A → B: out-dated state of A arriving at B

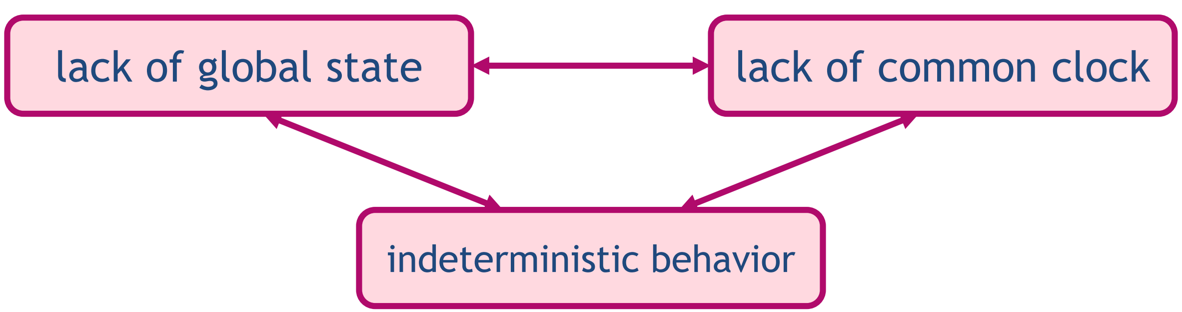
Basic Problem #2: Clocks Not 100% synchronized

Events E_A, E_B at A, B with recorded times $t(E_A) < t(E_B)$:

- May have happened at $t(E_A) > t(E_B)$!!
- When is it safe to „believe“ $t(E_A) < t(E_B)$?
- How to find out which is true? if undecidable: does distinction matter?

Basic Problem #3: Indeterminism – multiple execution of same system may yield different results

- Race conditions (messages from different senders, different threads) …
- … plus ‘erroneous’ underlying computer network →„correct program“ has unpredictable result!!



C. Requirements – selected (1)

1. Heterogeneity Support

Integration of different vendors → different …

- Networks, computer hardware, operating systems, programming languages
- Sub-requirement: support exchange of data & code despite heterogeneity
  - **Data:** in different HW architecture, OS, programming languages:
    - Known as „presentation“ issue
    - XML *is* not the solution (only syntax), may be *used* to build one
  - **Code:** mobile objects, mobile code

2. Openness

„Anyone may come along and participate“

- Remember, for computer networks: Open Systems Interconnection OSI
  - Standardized comm. Protocols → different implementations interoperate → global Internet
- For DistSys? Build application such that foreign processes may participate?
  - e.g., Reflection:... by means of application that „adapts itself“
  - e.g., Evolution: ... supporting „new versions“ of modules/parts
  - Publish/Subscribe paradigm (see later): plug in new process at any time

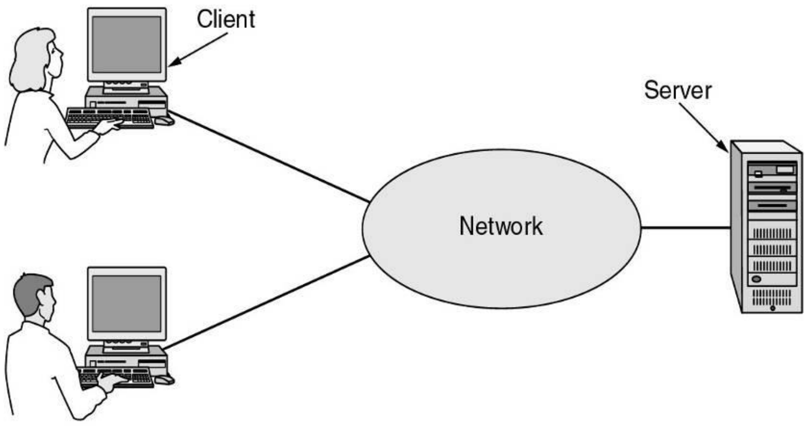
C. Requirements – selected (2)

3. Scalability
   - # of processes&nodes, # of users, # of transactions, ...
4. Security
   - “open system” → malicious users may come in
   - Authentication, authorization, trust
5. Failure tolerance
   - How to handle AS or CN failures?
6. Concurrency
   - Distributed applications are inherently “multithreaded”
   - Concurrent access to shared resources -> ensure consistency
7. Transparency
   - Abstraction of an aspect; many variants – see below

D. System Models (1)

1. Client/Server

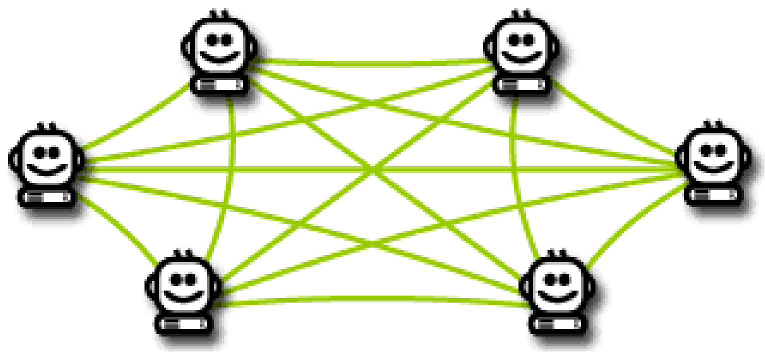
- Traditional model, easily comprehensible abstraction
  - Clients request service (initiate connection)
  - Servers provide service (answer requests)
  - Examples:
    - Web Client/Server
    - Mail Client/Server (well…)
    - FTP … ; well …
    - these are all using P2P (interprocess comm.) technologies
    - MAY be considered client-server IF: re-entrant behavior of server („many“ clients)
    - true C/S technology: remote procedure call (RPC)



D. System Models (2)

2. Peer-to-Peer

- Brand new paradigm?
  - no, the *oldest* one
  - appears to be new due to
    - MP3 piracy → overlay network (DHTs etc.)
    - new idea: resource client :=: resource server
    - Consideration of entire net, not just two peers
- *P2P-new* successful since 2000‘s
  - First tool: Napster (file sharing)
    - route around censorship
  - Other services *include*
    - streaming media
    - distributed computing



Other cool P2P technologies: ;-)

- Telephone
- Usenet
- DNS
- IP Routing

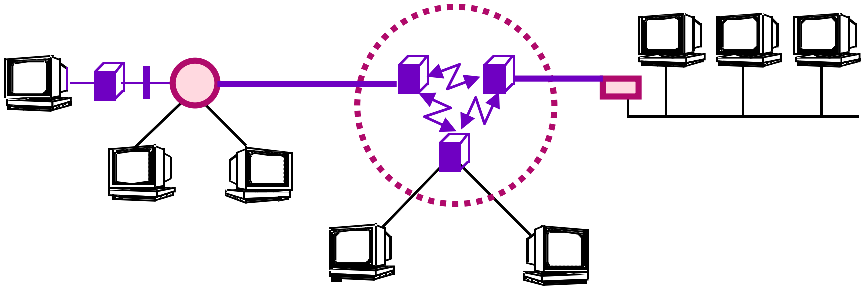
Actually, P2P = original Internet model
(symmetric network - all hosts are
(ftp, telnet, ..) clients and servers

E. Abstractions (1)

TECHNISCHE UNIVERSITÄT DARMSTADT

Level 1: physical configuration - seems irrelevant for DS:

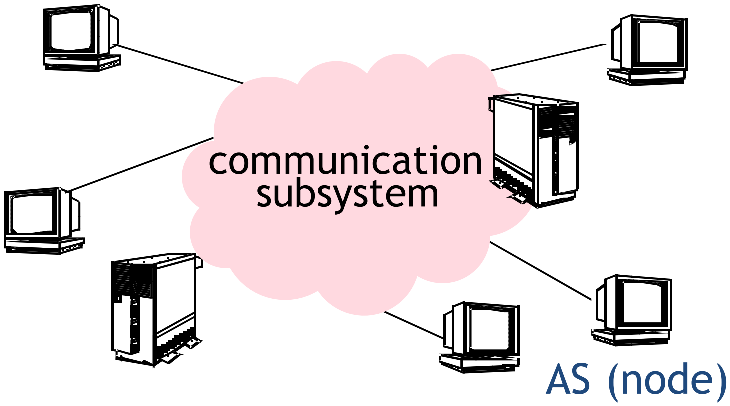
- Object of SysOp people
- For DisProg people, should be abstracted from



- but:
  - ownership → cost (public net?), security, ... !
  - bandwidth etc. → performance
  - reliability?

Level 2: logical configuration - „the“ CompNet abstraction!

- CSS = „cloud“, classes of ASs; AS may be part of CSS
- sometimes, abstraction too high (see above)
- usually, abstraction too low for DistProg



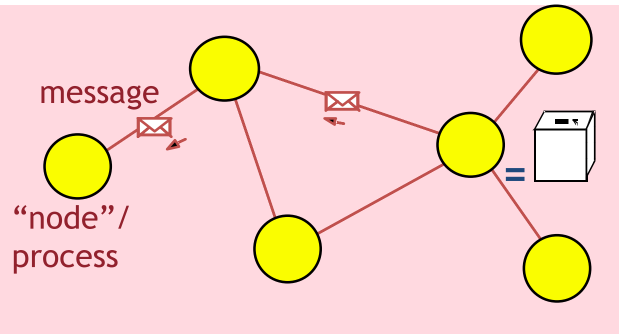
E. Abstractions (2)

Level 3: process network (logical distribution): abstracts from real distribution

(≈ Distributed Program DistProg):

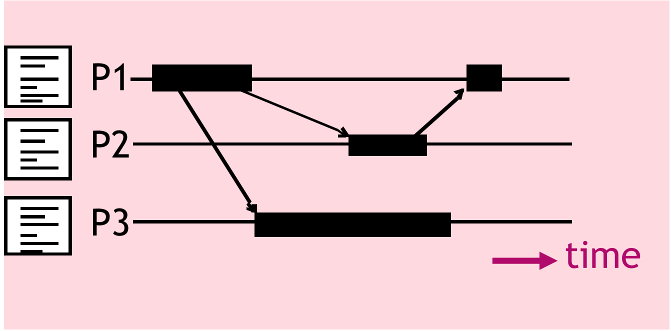
processes (objects, agents) exchange messages

- e.g., processes reached via mailboxes
- state distribution (no global view), see below
- no common time (no exact global "clock"), see below
- reliability, indeterminism assumed?
  depends further on abstraction & underlying support



Level 4: Distributed Algorithm - abstracts from

- Target environment (→ reliability, performance, ...)
  not necessarily from reliability / performance issues
- Target process configuration i.e. # of processes (well, ought to…)
  not necessarily from interconnection / topology issues!
- Implementation language, platform, lifecycle



F. Transparency (1)

- Definition:

  Transparency = „Concealment from the user and the application programmer of the **separation of components** in a distributed system, so that the **system is perceived as a whole** rather than as a collection of independent components.

  - Note: „transparency“ is about „hiding something“ in English
  - We introduce 8 forms of transparency below, but literature varies → there are more!

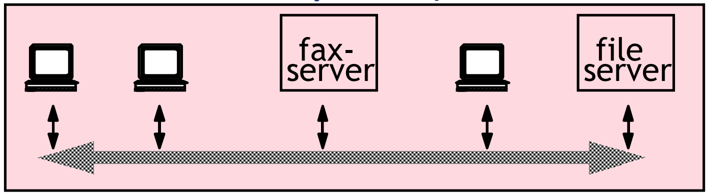
1. Access transparency
   - Local & remote resources accessed using identical op’s
2. Location transparency
   - Resources accessed w/o knowledge of their physical/network location
3. Concurrency transparency
   - Several processes operate concurrently using shared resources without interference between them

F. Transparency (2)

4. Replication transparency
   - Multiple instances of resources used (→ reliability, performance) w/o knowledge of replicas by users & programmers
5. Failure transparency
   - Concealment of faults, allowing users & programs to complete tasks despite HW/SW failure
6. Mobility transparency
   - Movement of resources/clients w/o affecting users & programs
7. Performance transparency
   - Local/remote op (exec, data access) don't differ by orders of magnitude (most persistent problem!)
   - allows system to reconfigure to improve performance as loads vary
8. Scaling transparency
   - Allows system/applications to expand in scale without change to system structure or application algorithms

G. Classification

1. According to *xIxD* classification, DistSys are classified as MIMD (Flynn's taxonomy)
   - Like multiprocessor systems (but w/o shared memory → big difference!)
   - remember: SISD = PC etc., SIMD = vector processor, MISD = <unused>
2. Federation classes: autonomous ASes access certain services (never 100% pure)
   - function federation (use particular function of "other" system)
   - load federation
   - data federation
   - availability federation: here, availability



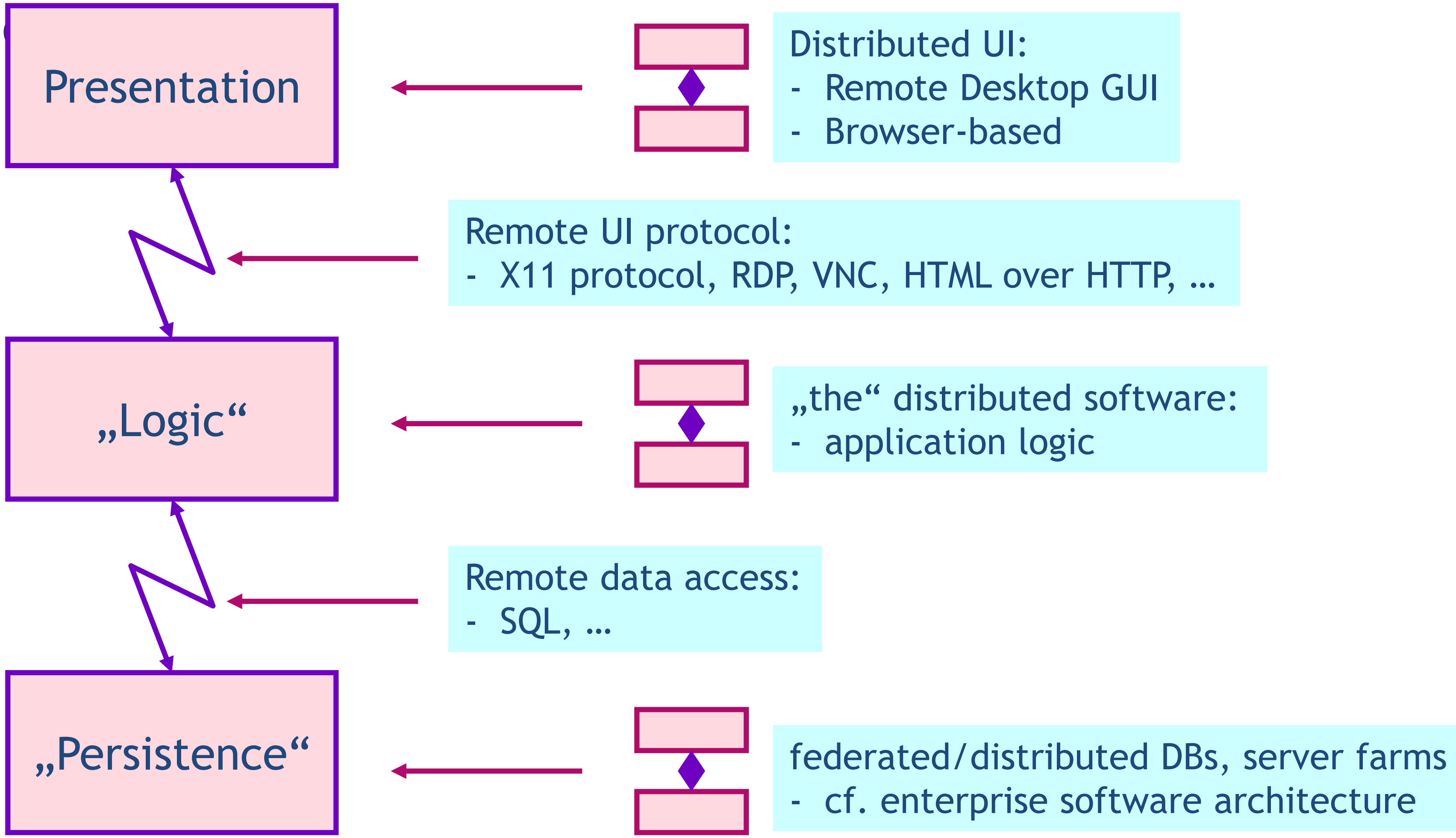
- 1: pretty useless, 2: helpful for understanding, but: exemplars quasi nonexistent
- many more classifications exist: not very helpful either

Important lesson: **Myths & Reality & Future of DistSys**

- Dream was: DistSys for parallel processing
- Reality is (~90%): DistSys due to "locality of data / CPU", central server
- Will central servers go away? … no! rather, importance may decrease
- Will server farms "explode" soon? … was envisioned since 20 years!

H. Tier Architectures (1)

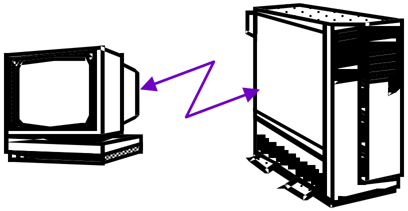
Classical (logical) SW Architecture is 3-tiered → 5 distribution



H. Tier Architectures (2)

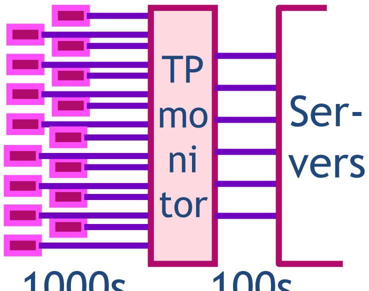
Note: physical tiers! (enterprise computing lingo, used by database people)

- 2-Tier: early Client-Server
  - tier 1: GUI & application
  - net: file I/O, protocol (http, ...), SQL/ODBC, ...
  - tier 2: DBMS, „Resources“



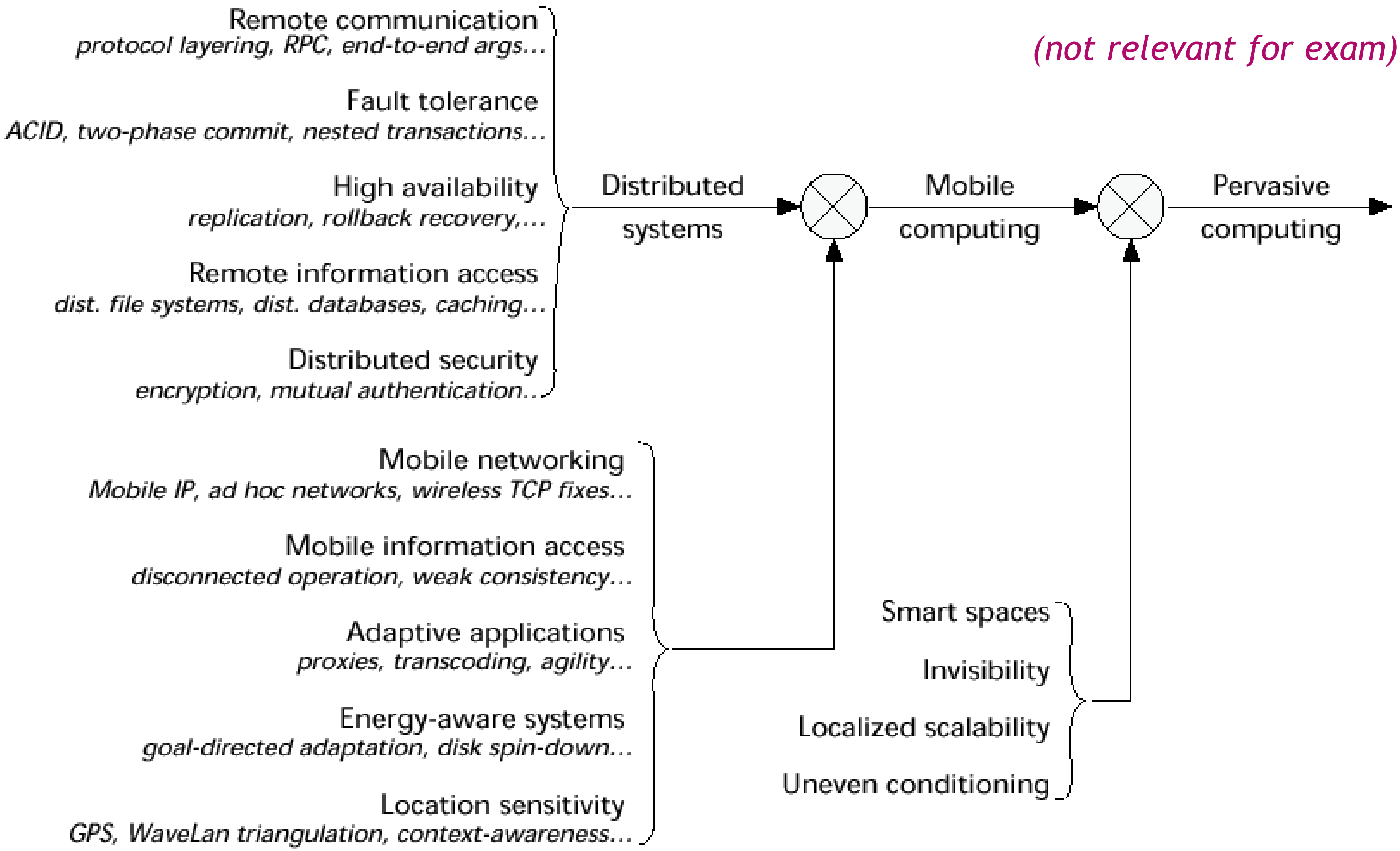
- 3-Tier:
  - different definitions exist, but mostly:
  - Client – Application Server – Database Server
- n-Tier: 2nd Tier expanded
  - specialized application servers (OrderProcessing, Accounting, Human..)
  - cf. ERP (enterprise resource planning) companies like SAP, Oracle,...
  - scalability considered to require „TP monitors“

- note: each tier may be „n nodes“
  - many clients anyway; server farms distribute load
  - server-calls-server is rather common



I. Challenges

(not relevant for exam)



J. SWE Approaches → Language Approach (1)

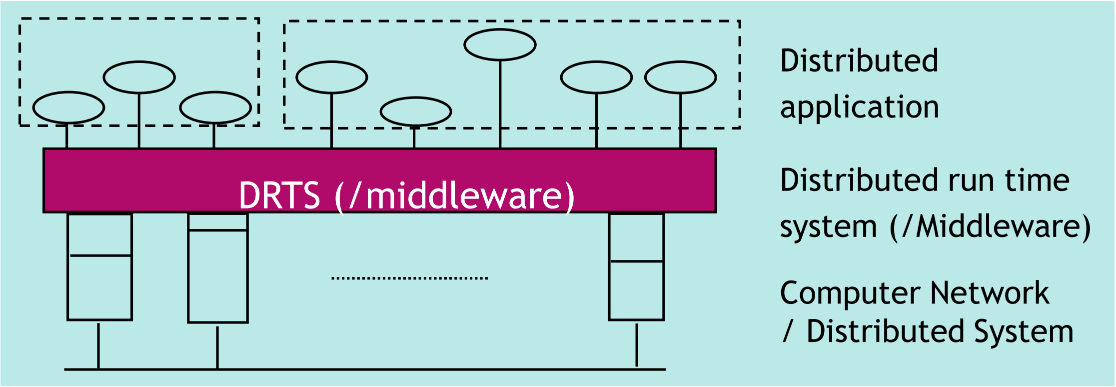
4 principles for abstractions wrt. Distributed Software Development

1. **Distributed operating system approach**
   - Support for distributed programming is part of operating system
   - Pro: Quite general solution, *parallel* programming paradigm
   - Con: Needs wide-scale adoption of the same system
     - Large systems always heterogeneous
2. **Distributed database approach**
   - Same as above, except OS replaced by a database system
   - Pro: Allows for all DB features (semantics, …), isolated *sequential* prog's
   - Con: Independent applications with shared database
   - Con: Many distributed algorithms hard to realize in this case
3. **Protocol approach** for dedicated purposes (Xwindow etc.)
   - Standardized protocols for connecting to servers (e.g., HTTP)
   - Pro: Open, global; ~ sequential prog's (+ callback threads)
   - Con: Limited to standard functionalities
4. **In this lecture: Distributed Programming 'Language' approach**
   - the only one that is wide spread – up to now!

J. SWE Approaches → Language Approach (2)

Distributed Programming Language Approach

- Arbitrary, heterogeneous OSes (and databases)
- Distributed programming language



- original approach (almost died out – this may change!):
  - "distributed programming language" + dist. runtime system
  - compiler can help a lot (like assembler vs. HPL in trad. systems)
- today's approach:
  - Sequential programming language + extensions + middleware
  - Compiler will not see the distributed program

syntactic difference may be small if dist. lge = seq. lge + extensions
semantic difference is big though (compiler sees / ignores dis. prog.)

J. SWE Approaches → Language Approach (3)

- Distributed Programming ‚Language‘ Approach - Pro:
  - DRTS intertwined with language (efficiency↑)
  - side-by-side competition of languages → competition of concepts
  - heterogeneity, migration, autonomy etc. may be addressed
  - encompassing support for DistProg (including „Platforms“) possible on a „global“ scale
- Distributed Programming ‚Language‘ Approach - Con:
  - *n* languages → *n* DRTSes
  - loosely intertwined w/ OS (efficiency?)
  - many problems to be addressed (→ extensible solutions?) ... but that‘s true for the other approaches, too
  - new language constructs ↔ programmers‘ acceptance / learning effort
  - „compiler sees everything“↔ we may not *have* everything @ compiletime

Conclusion: for the rest of this lecture...

- we‘ll stick to the language approach
- we‘ll dream of dist. lanugages, look at lge. extensions/middleware
- E2E argument → if you know what can be done, you can BYO

Language Approach: Requirements

...think of >100 processes, >10000 objects, dynamic changes (new users etc.)

Sequential language + extensions → distributed lge.

- **minimum** extensions: API + IDL (interface definition lge.); **better:** compiler support
- support for what?
  - structuring (hierarchy or 'network' of processes? 'ports' for different message types?)
  - communication interplay (‚compatibility' of msges sent/received, flow of msges over time, …)
  - management / administration support (installation, configuration)
  - dynamics (wrt. processes / connections added/removed)
  - flexibility wrt. the „model" applied (see further in this chapter)
  - safety in programming (remember: this is why we have prog.lge's!)
    - compiler & RTS „understand" your „entities" & how you cope with them
    - typing of comm. partners, connections, objects interchanged, ...
  - late distribution (see DOC subchapter)
  - openness wrt. cooperation of separately developed parts
  - integration of existing services
  - distribution transparency (mimic concurrent prg.)
  - all other kinds of transparency (see before)

Language Approach: Support

Language Approach: Support

Software production environments:

- methods & tools must be adapted; or even: need for new methods & tools?
- „integrated SPE“ wrt. UI, functional, & data layers
- integration along life cycle (today, even design → implementation ??)
- aspect oriented (e.g., performance, reliability, ...) across life cycle?
- extensible?
- distributed SPE (distributed-software eng. vs. distributed SW-eng.)

Platforms:

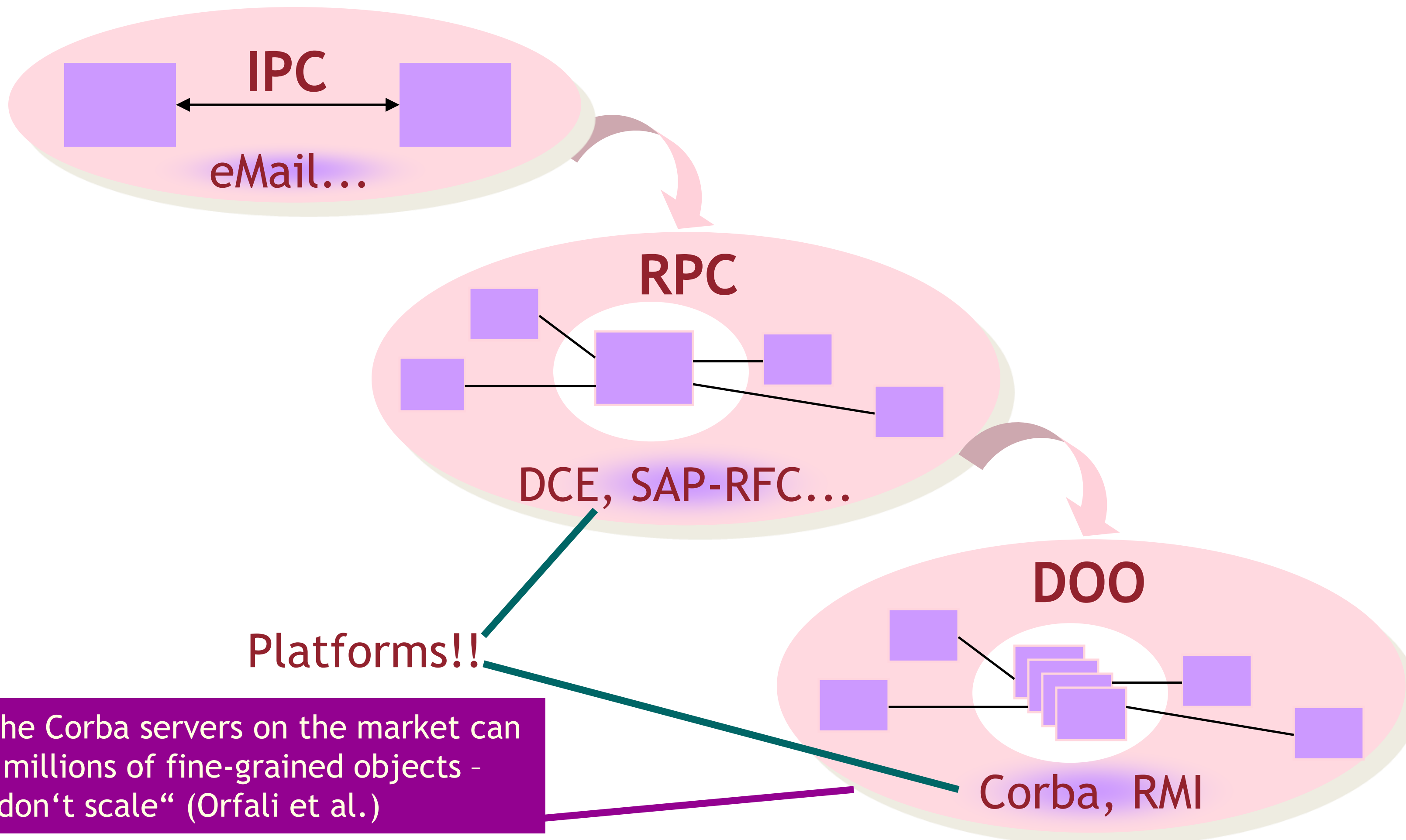
- middleware that includes numerous services, tools, ...
- „horizontal“: generally useful services
- „vertical“: intended for specific application domain / market

But recall End-to-End argument! … no feature-ladden middleware

→ at least, consider „plain TCP /UDP“ as an alternative! Rest: build your own BYO
(on the other hand, this is the ‚assembler approach‘ of distributed programming)

→ know many concepts from literature (/lecture) !!!

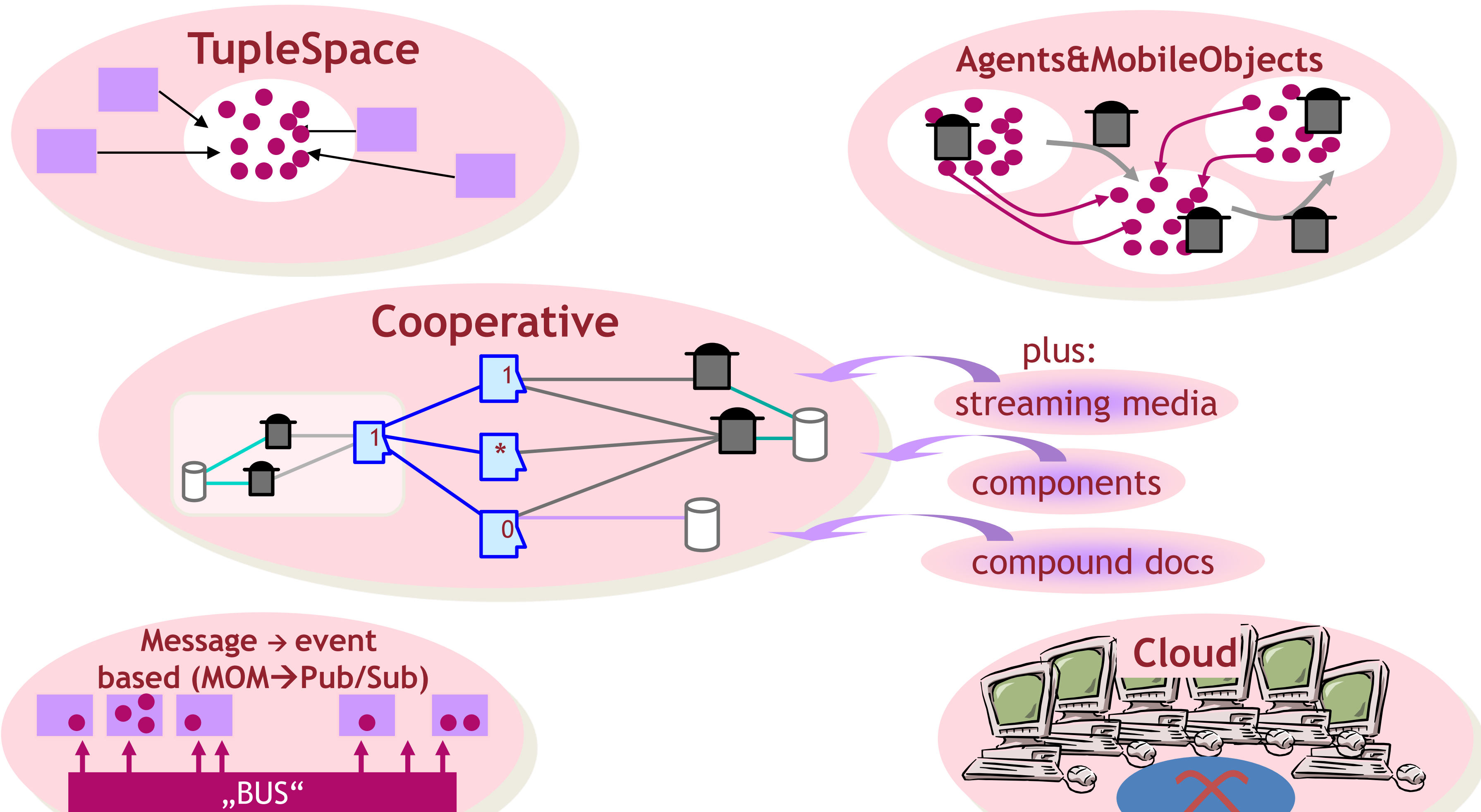
K. Distributed Programming Paradigms



„None of the Corba servers on the market can deal with millions of fine-grained objects – they just don‘t scale“ (Orfali et al.)

mainstream paradigms

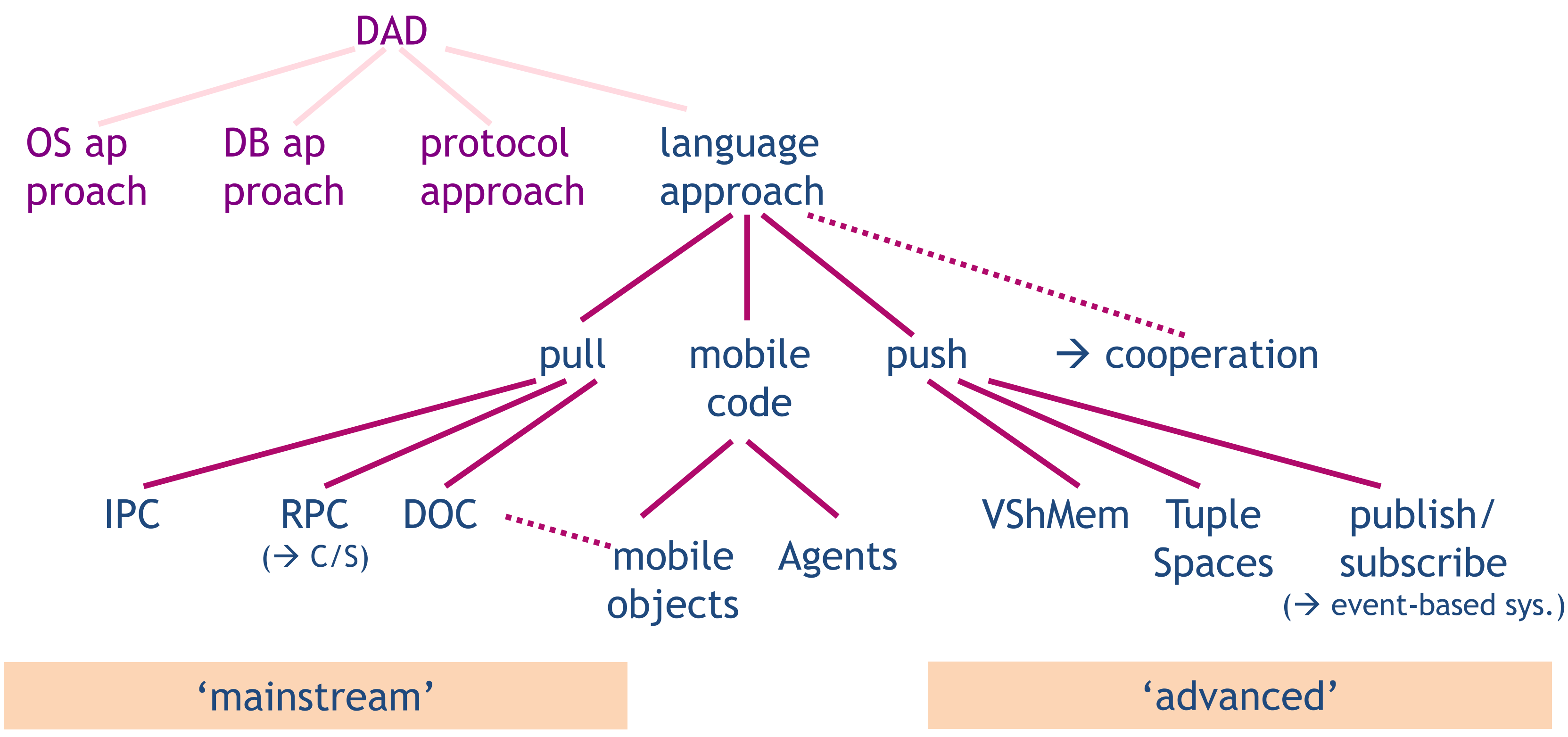
K. Distributed Programming Paradigms



advanced paradigms

Pragmatic Taxonomy

all in one, we get the following taxonomy for distributed application development (DAD):



Famous „Religious“ Disputes

(not discussed in lecture 1, but during all of course)

- Escape from Flatland („Flatland“: romance in 19th century)
  - „DS speed/cost depends on interconnection fabric, need new visions!“
  - but: nothing really changed for 40 years (well, trend went from central switches to shared medium LANs back to switches)
  - will wireless tech., will multiplexing schemes bring about change? (wavelength division multiplexing, CDMA/OFDM air interfaces)
- „End-to-End argument ...“ (J.Salzer et al., ACM ToCS 2(4), 1984)
  - „feature-rich protocols/middleware = big waste! Build-your-own (BYO)!“
    1. customized selections needed
    2. can‘t provide 100% reliability etc.! anyway
  - but: development cost! Will components do?
  - truth-in-the-middle: reentrant, highly optimized code not bad anyway
- „A note on distributed programming“ (J. Waldo et al., Sun TR‘94)
  - many systems have attempted to paper over the distinction between local and remote objects → they fail to support basic requirements of robustness and reliability!
  - truth-in-the-middle: consider SWE effort, new paradigms in middleware

Overview of this Lecture

1. Introduction
2. Distributed Programming
    1. **Mainstream Paradigms** for Distributed Programming
        - IPC: Interprocess Communication
        - Inlet: Distributed Programming Languages
        - RPC: Remote Procedure Call
        - Inlet: Concurrency
        - DOC: Distributed object-oriented computing
        - Web Services
    2. **Advanced Paradigms** for Distributed Programming
        - Event-Based & Publish/Subscribe Communication
        - Tuple Spaces
        - Distributed Shared Memory Approach
    3. **Mobile Objects**, Unified Objects, Mobile Agents
    4. **Cloud Computing**
    5. **Formal Approaches:** Process Calculi
3. Distributed Algorithms
    1. **Foundation:** Motivation, Properties, Characteristics
    2. **Synchronization:** Logical Clocks, Physical Clocks, Global States
    3. **Coordination:** Failure Detection, Mutex, Election
    4. **Cooperation:** Multicast (On Different Topologies), Consensus
    5. **Local Algorithms**