

Distributed Systems

2 Characteristics

Goals & Challenges, Types of DS

April-22-2009

Summer Term 2009

System Architecture Group



Schedule of Today

- Fundamentals
- Inherent Characteristics of a DS
- DS Goals & Challenges
- Types of DS



Fundamentals

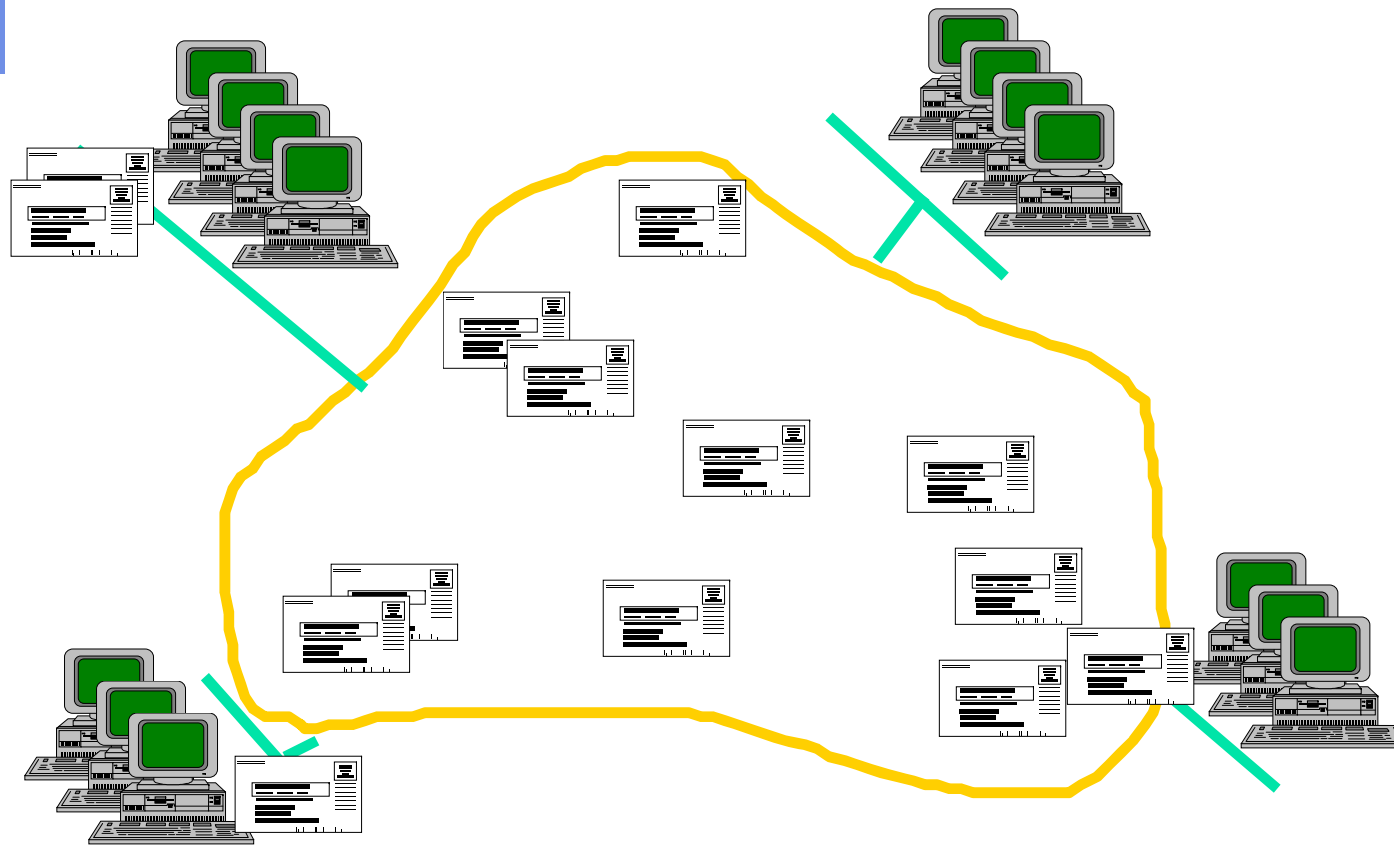


Review: Terminology

- **Program** is the code you type in
 - **Process** is what you get when you run it as a single activity
 - **Task** is what you get when you run it as multiple activities
- **Message** is used to do IPC between processes/tasks. Arbitrary size.
- **Packet** is a fragment of a message that might travel on the wire. Variable size but limited, usually to 1400 bytes or less.
- **Protocol** is an algorithm by which processes cooperate to do something using message exchanges.
- **Network** is the infrastructure that links the computers, workstations, terminals, servers, etc.
 - It consists of routers
 - They are connected by communication links
- **Network application** is one that fetches needed data from servers over the network
- **Distributed system** is a more complex application designed to run on a network. Such a system has multiple processes/tasks that cooperate to do something.



Network ~ "mostly reliable" Post Office





Why isn't it totally reliable?

- Links can corrupt messages
 - Rare in high quality ones on the Internet "backbone"
 - More common with wireless connections, cable modems, ADSL
- Routers can get overloaded
 - When this happens they drop messages
 - As we'll see, this is very common
- But protocols that **retransmit lost packets** can increase reliability



How do DSs differ from Network Appl.?

- DSs may have many components but are often designed to **mimic a single, non-distributed process running at a single place.**
- “State” is spread around in a DS
 - Networked application is free-standing and centered around the user or computer where it runs, e.g. your “web browser”
 - A DS is spread out, decentralized, e.g. the “**air traffic control system**”

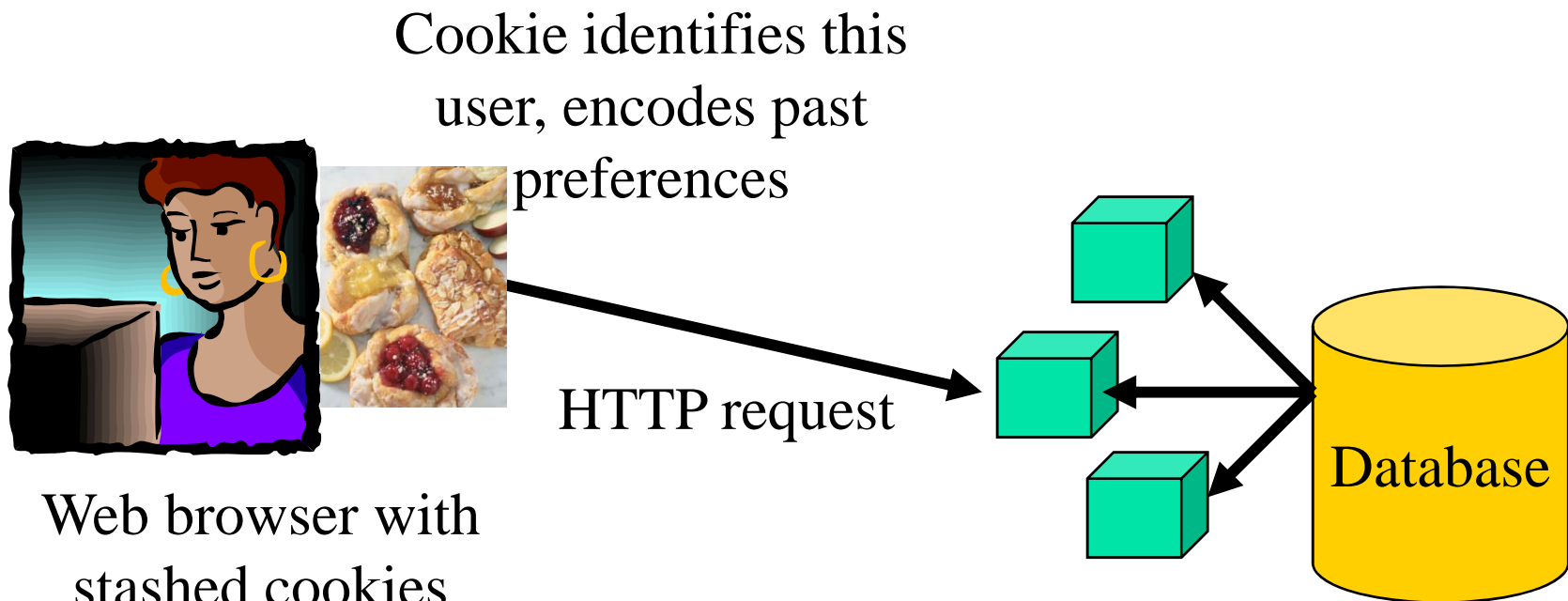


What about the Web?

- Your web browser is independent: it fetches the data you have requested when you have asked for it.
- Web servers don't keep track of who is using them. Each request is self-contained and treated independently of all others.
 - Cookies don't count: they sit on your machine
 - And the database of account info doesn't count either... this is "ancient" history, nothing recent
- Web has 2 network applications that talk to each other
 1. The browser on your machine
 2. The web server it happens to connect with... which has a database "behind" it



What about the Web?



Web servers are kept current by the database but usually don't talk to it when your request comes in

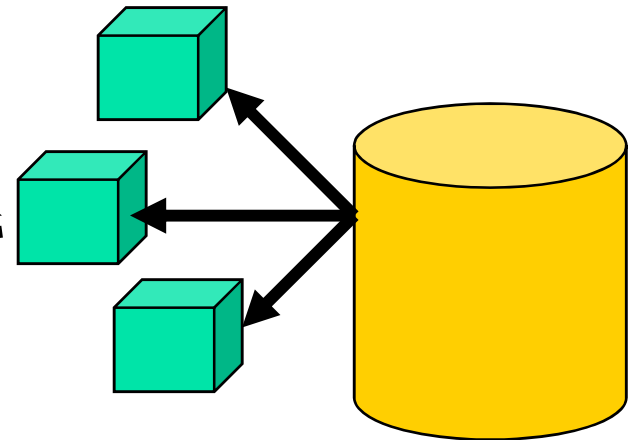


What about the Web?

Web servers immediately forget the interaction



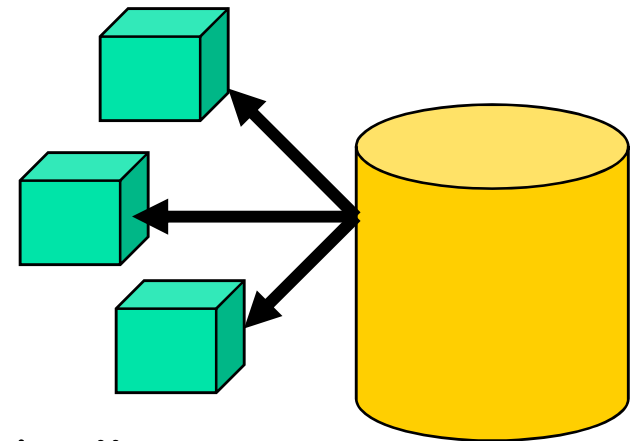
Reply updates cookie





What about the Web?

Web servers have no
memory of the interaction



Purchase is a “transaction”
on the database



What about the Web?

- But... the data center that serves your request may be a complex DS
 - Many servers and perhaps multiple physical sites
 - Opinions about which clients should talk to which servers
 - Data replicated for load balancing and high availability
 - Complex security and administration policies
- So: we have a “networked application” talking to a DS



Other Examples of DSs

- Air traffic control system with workstations for the controllers
- Banking/brokerage trading system that coordinates trading (risk management) at multiple locations
- Factory floor control system that monitors devices and reschedules work as they go on/offline



Is the Web “reliable”?

- We want to build DSs that can be relied upon to do the **correct thing** and to **provide services** according to the **user’s expectations**
- Not all systems need reliability
 - If a web site doesn’t respond, you just try again later
 - If you end up with two wheels of brie, well, throw a party!
- Reliability is a growing requirement in “critical” settings but these remain a small percentage of the overall market for networked computers
- And as we’ve mentioned, it entails satisfying multiple properties...



Reliability is a Broad Term (1)

- **Fault-Tolerance:** remains correct despite failures
- High or continuous **availability:** resumes service after failures, doesn't wait for repairs
- **Performance:** provides desired responsiveness
- **Recoverability:** can restart failed components
- **Consistency:** coordinates actions by multiple components, so they mimic a single one
- **Security:** authenticates access to data, services
- **Privacy:** protects identity, locations of users



Reliability (2)

- **Correct specification:** the assurance that the system solves the intended problem
- **Correct implementation:** the assurance that the system correctly implements its specification
- **Predictable performance:** the guarantee that a DS achieves desired levels of performance, e.g. data throughput from source to destination, latencies measured for critical paths, requests processed per second, etc.
- **Timeliness:** in systems subject to real-time constraints, the assurance that actions are taken within the specified time bounds, or are performed with a desired degree of temporal synchronization between components



“Failure” also has Many Meanings

- Halting failures: component simply stops
- **Fail-stop**: halting failures with notifications
- Omission failures: failure with send/receive message
- Network failures: network link breaks
- **Network partition**: network fragments into two or more disjoint sub-networks
- **Timing failures**: action early/late; clock fails, etc.
- **Byzantine failures**: arbitrary malicious behavior



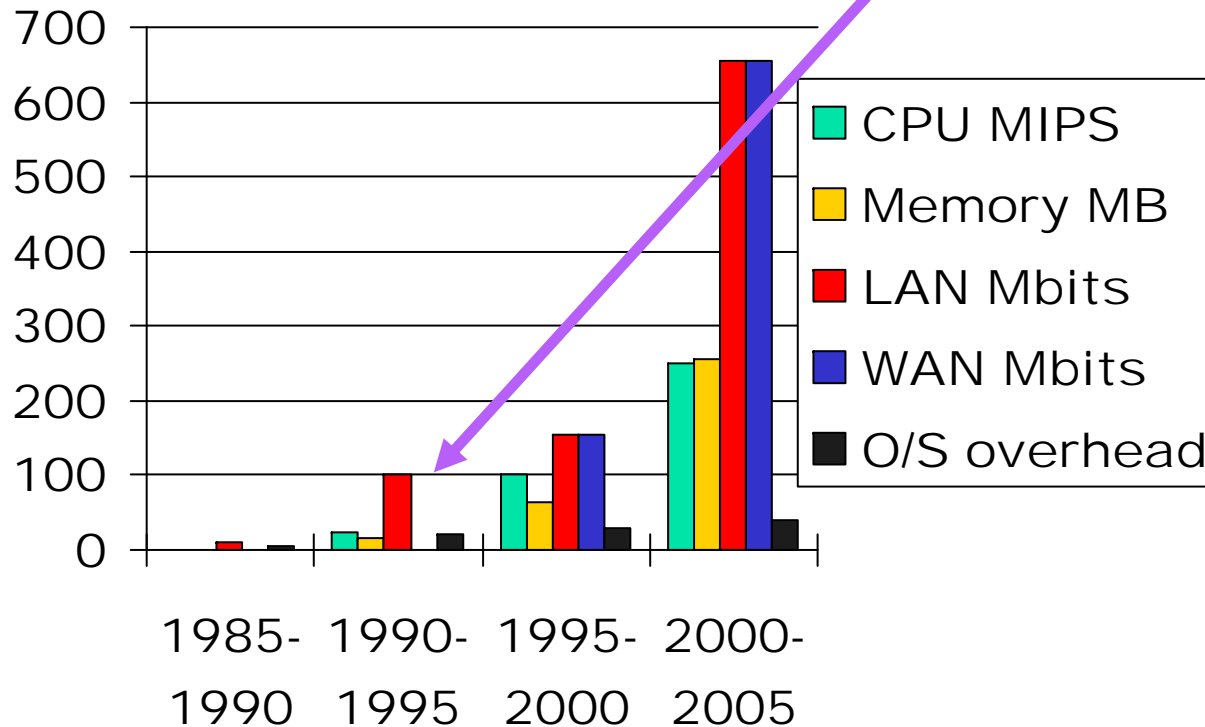
Examples of Failures

- My PC suddenly freezes up while running a text processing program. No severe damage is done. This is a halting failure
- A network **file server** tells its clients that it is about to **shut down**, then goes offline. This is a fail-stop failure. (The notification can be trusted)
- An **intruder hacks** the network and replaces some parts with fakes. This is a Byzantine failure.



Technology Trends

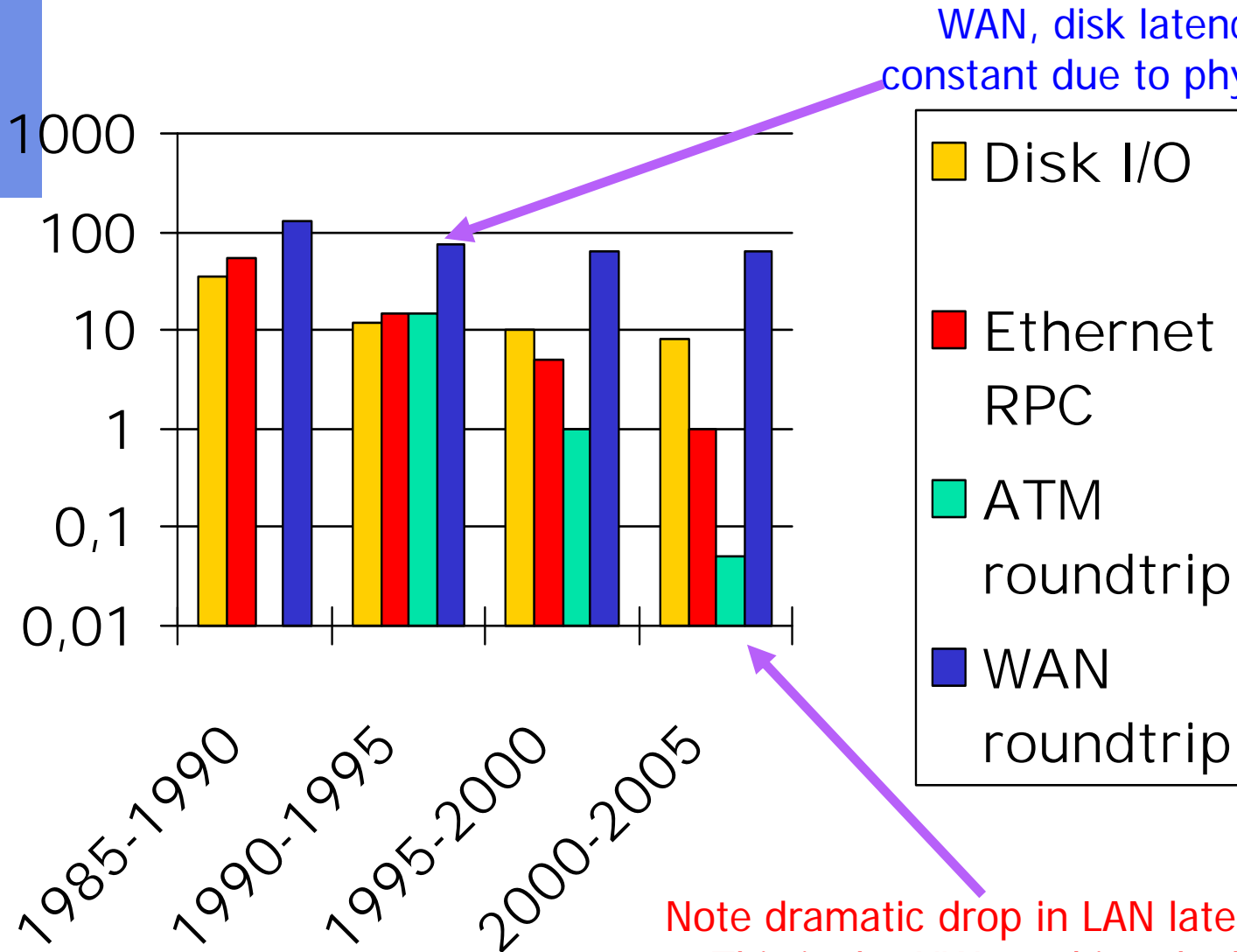
Did the sudden growth in LAN speed give us the Web?



Source: Scientific American, Sept. 1995

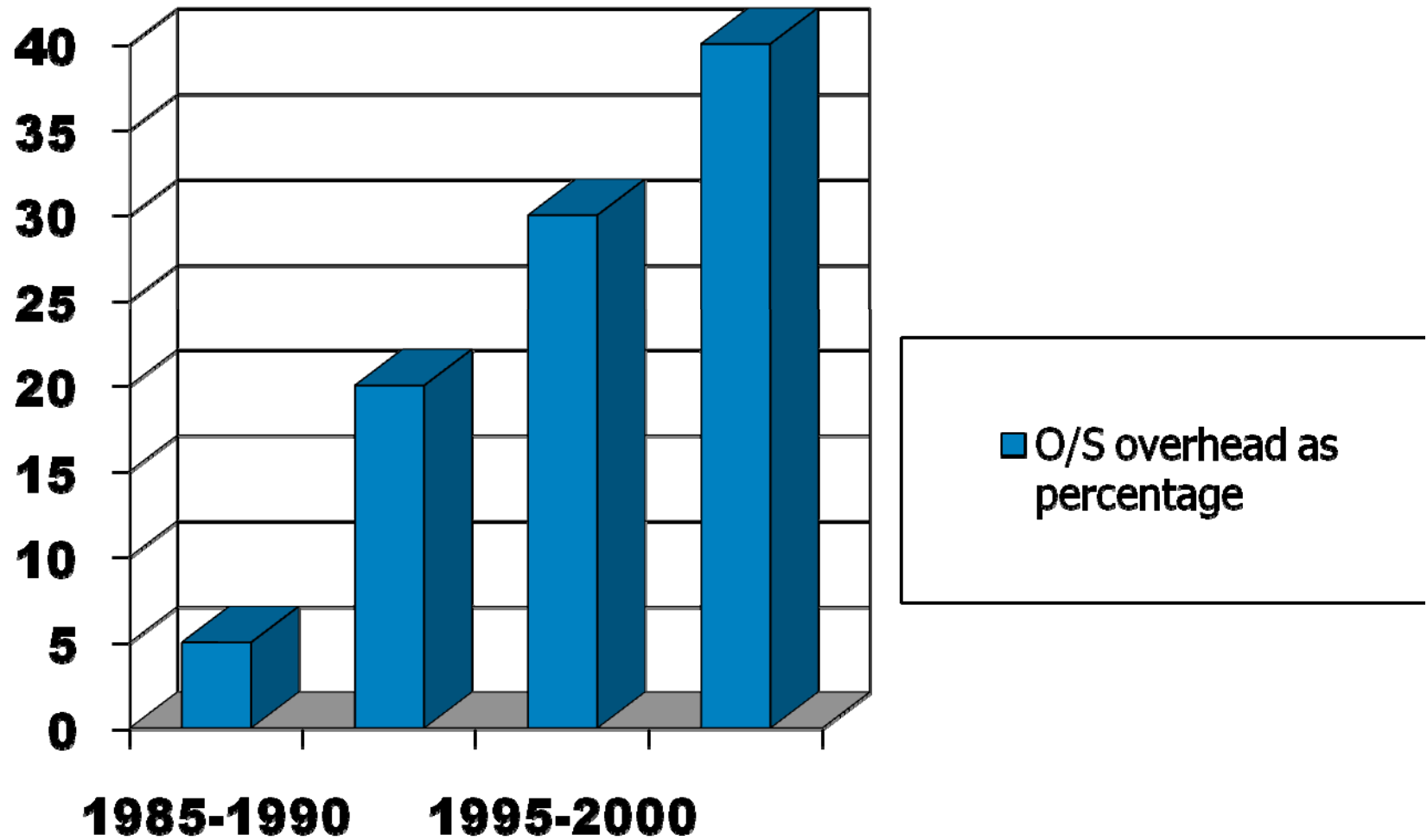


Typical Latencies (milliseconds)





OS Latency: Expensive overhead on LAN





Reliability versus Performance

- Some think that more reliable means “slower”
 - Indeed, it usually costs time to overcome failure
 - For example, if a packet is lost, you probably need to resend it, and may need to solicit the retransmission
- But for many applications, performance is a big part of the application itself: **too slow** means “**not reliable**”
- Reliable systems thus must look for highest possible performance
- ... but unlike unreliable systems, they can't cut corners in ways that make them flakey but faster



Discovery

- Consider the problem of discovering the right server to connect with
 - Your computer needs current map data for some place, perhaps an amusement park
 - Can think of it in terms of layers – the basic park layout, overlaid with extra data from various services, such as “length of the line for the Cyclone Coaster” or “options for vegetarian dining near here”



Why is Discovery hard?

- Client has opinions
 - You happen to like vegetarian food, but not spicy food. So your search is partly controlled by client goals
 - But a given service might have multiple servers (e.g. Amazon might have data centers in Europe and in the US...) and may want your request to go to a particular one
 - Once we find the server name we need to map it to an IP address
 - And the Internet itself has routing “opinions” too



Other Things we might need

- Standard ways to handle
 - Reliability, in all the senses we listed
 - Life cycle management
 - Automated startup of services, if someone asks for one and it isn't running; backup; etc...
 - Automated migration and load-balancing, monitoring, parameter adaptation, self-diagnosis and repair...
 - Tools for integrating legacy applications with new, modern ones

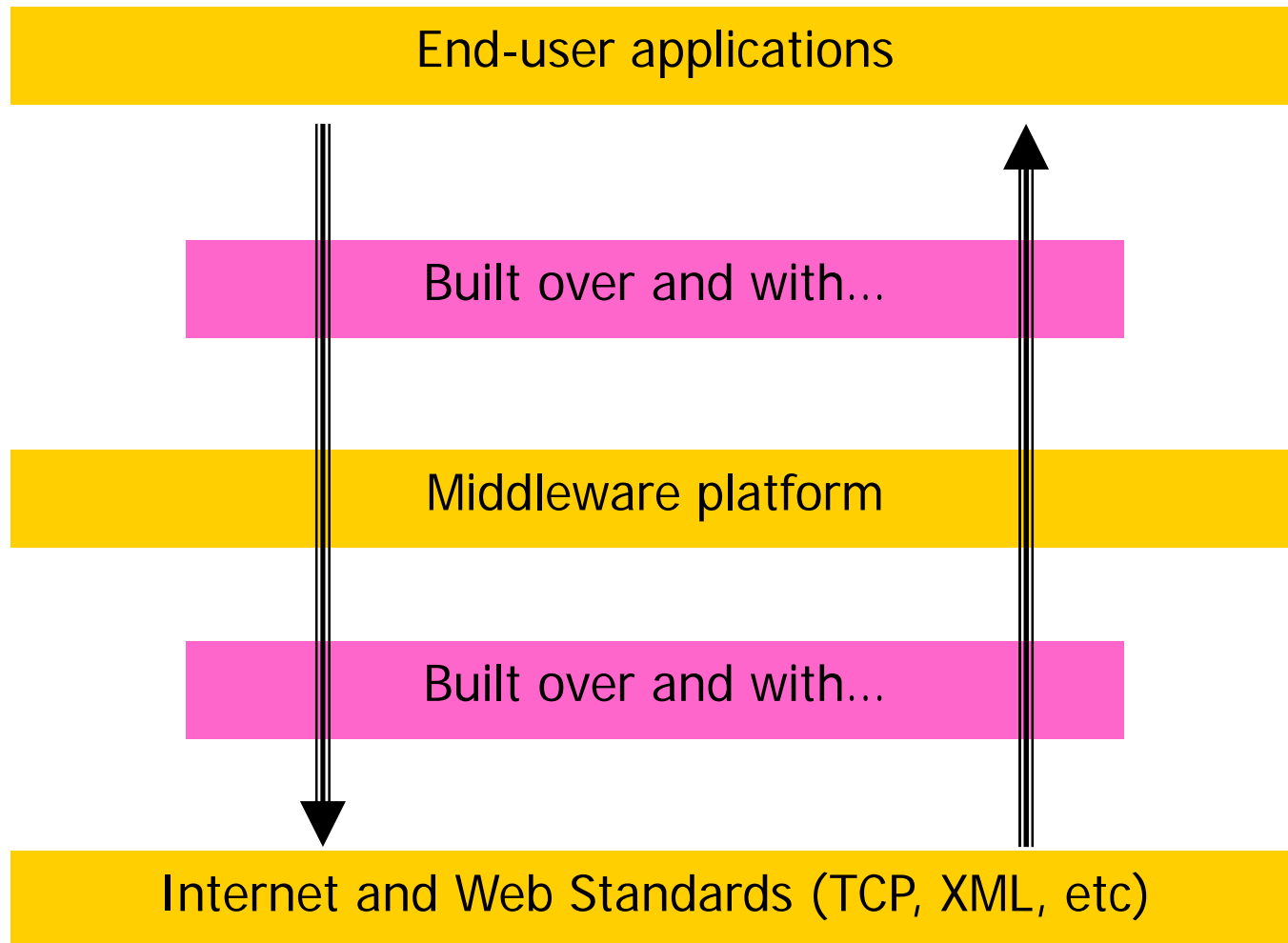


Concept of a Middleware Platform

- These are big software systems that automate many aspects of application management and development
- In this course we will **not discuss in detail**
 - CORBA: a stable and slightly outmoded platform focused on “objects”
 - Web Services: the hot new “service oriented architecture”
- However, we want to find **conceptual solutions**, useful for applications as well as middleware applications



Layers: Modern Perspective





For Example

- Imagine a banking system with many programs, one at each branch
- And suppose that only some can talk to others due to firewalls and other restrictions, e.g.
 - A can talk to B, B can talk to C, but A can't talk to C



How to handle this?

- In the distant past, people cooked up all sorts of weird hacks
- Today, a standard approach is to build a routing layer
 - Inside the application, it would automatically forward messages towards their destinations
 - Thus A can talk to C (via B)



Once we have this...

- Now we can split our brains, in a good way:
 - Above this routing layer, we write code as if routing from anyone to anyone was automatic
 - Inside the routing layer, we implement this functionality
 - Below the routing layer we just do point-to-point messaging where the bank permits it and we never end up trying to send messages over links not available to us



This Layering looks elegant!

- It lets us focus attention on issues in one place and simplifies code as a result
- Also helpful when debugging...
- Platform architectures simply take the same approach further



Inherent Characteristics of DS

- Physical Distribution
- Logical Distribution
- Sharing of Resources
- Heterogeneity
- Real Parallelism
- Failure Tolerance
- Layered Software



Physical Distribution

- HW distribution (devices, computers)
 - “Network” of autonomous computers
 - Geographic distribution matches physical world
- Interconnected via
 - physical communication links, e.g.
 - Fiber optic
 - (fast) Ethernet
 - wireless interconnection, e.g. WLAN



Physical Distribution

- SW distribution:
 - Tasks or processes with specific services enabling
 - Decentralized computing
 - Can reduce turnaround times
 - Shared “expensive” resources
 - Improved availability of whole DS
 - Information/Data
 - Distributed data base
 - Replication enabling worldwide collaboration



Sharing Resources

- Sharing often done *without clients' knowledge*
 - sharing printer hidden by a spooler
 - sharing files and/or directories hidden by the FS

- *Why sharing?*
 - To reduce cost for HW resources
 - High quality printer
 - High quality scanner
 - ...

- *How to design sharing?*
 - Dependent on
 - Range of validity
 - Intensity of collaboration
 - ... others?



Sharing Services

- Service := software component that manages a set of related resources and offers a set of hopefully comfortable functions
 - File service: store and retrieve **persistently stored, named data containers**
 - Print service: print documents, photos etc.
 - E-Commerce: sell or buy products via Internet
 - amazon
 - ebay
 - ...



Clients & Servers

- Client := user/process/task requesting a service
- Server := process/task on one or multiple machines offering a specific service, e.g. file service

How do clients and servers typically interact?

- Clients synchronously request something via a
 - Remote procedure call (*RPC*) or
 - Remote method invocation (*RMI*) or
 - "Internode"-IPC
- Typical client/server application
 - Web browser
 - Web server



Heterogeneity

- **Heterogeneous HW & SW**
 - **Networks**
 - Type of connection
 - Technology
 - Topography
 - **Processors**
 - Data representation
 - 32-bit, 64-bit
 - Instruction Set
 - **OS**
 - API & execution environment
 - Linux, Vista, ...



Real Parallelism

- **Concurrency** occurs on different levels
 - Multiple clients use a file server concurrently
 - A file-server can be multi-threaded

⇒ Increased synchronization requirement

- To solve concurrency problems in DS we need
 - either a **single** instance with a
 - **centralized** algorithm
 - **global** state

} bottleneck
single point of failure
 - or **distributed** instances with a
 - **distributed** algorithm
 - **set of local** states

} consistency problem



Failures & Failure Handling

- \exists failures in (almost) all technical systems
- In DS we have to face different failure types, e.g.
 - Partial failures
 - some nodes still work whilst others are down
 - Transient failures
- Handling of failures:
 - detect, mask, tolerate
 - recover after failures
 - avoid, by providing enough redundancy



Detection of Failure

- Some failures are detectable, e.g.
 - via checksums (integrity of data)
 - sequence numbers (missing messages)
- Others are not (e.g. server-breakdown)
 - *How long do you wait until you **assume** that your favorite server is down?*
 - *Can you improve your client-server protocol a little bit?*

Challenge:

- Learn to handle or live with undetectable, but assumed failures



Masking & Tolerance of Failures

- Some detectable failures can be masked, i.e. the application has not to deal with
 - **Lost message** → just resend it at a lower layer
 - How to decide that a message has been lost?
 - **Lost file** → take the file object from a replicated file server
- Failures, that cannot be masked should be reported to the application ⇒ **distributed applications must be aware of error reports**



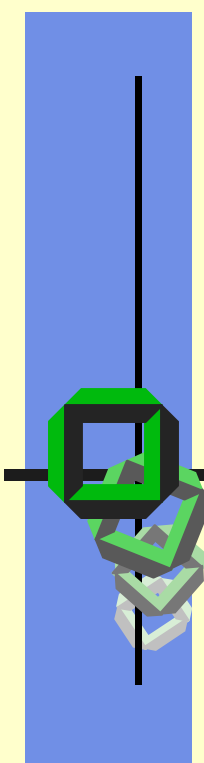
Recovery

- Node break downs or losses of components often show typical error symptoms:
 - Computations are incomplete
 - Permanent data stay inconsistent
 - No periodic alive messages
- Two types of failures
 - Transient failures
 - Reinstall a consistent system state, e.g. via checkpoints, forward/backward recovery, transaction models
 - Permanent failures
 - Replace or repair the incorrect component
- Recovery has to be taken into account already in the **design phase** of a DS



Uncertainty Principle

- At the same instance of time two processes in a DS do not always have the same view of the system's state
- Typically a process in a DS has either
 - an incomplete system state or it has
 - a complete, but potentially outdated system state
- Due to the lack of a global physical time, it is hard to determine, if two events occur at the same time, thus we need algorithms that deliver a consistent snapshot of the DS



DS Goals & Challenges

Transparency
Openness
Flexibility
Scalability
Security
Reliability
Performance



Distribution Transparency

Transparency	Description: ... hides ...
Access	differences in data representation & resource access
Location	where a resource is located
Migration	that a resource (object) moves to another location
Relocation	that a resource is moved to another location
Replication	that a resource is located at multiple nodes
Concurrency	that a resource is shared by several competing users
Failure	that failure and recovery of a resource might occur
Scaling	the reconfiguration of the system with growing load

Different forms of transparency in a DS



Degree of Transparency

- High degree of transparency is often preferable
- However, sometimes \exists drawbacks:
 - In a WAN you cannot hide latency completely due to many intermediate routers & switches
 - You want to decide how often a Web browser tries to contact a broken Web server before switching to another replicated web server
 - In a DS that requires a high degree of consistent replicas, updates on replicated data will take some time
 - An employee of Siemens ([Munich](#)) that wants to print a document prefers an overloaded printer nearby to a lazy printer at Siemens ([Nuremberg](#))



Openness

- In standard networks, specific rules are formalized as network protocols
- Standardized interfaces and mechanisms
 - Message types
 - Interface definition language (IDL)
- Proper and complete specification
 - Interoperability
 - Portability
 - Maintenance



Flexibility & Adaptability

- To achieve flexibility in an open DS \Rightarrow use a **component based** system architecture
 - Add new system components (on the fly!)
 - Update or replace old ones
 - Install different versions of a system component to be adaptable
- Clean interfaces not restricted to top-most layer enabling better adaptability, e.g.
 - Clients of a web browser want to determine their private caching policy
 - How long should data be cached depending on
 - Data type
 - Session time ...



Scalability

- Performance does not decrease significantly with more and/or newer nodes in the DS
 - ⇒ consequence for a system architect:
 - Avoid any form of a centralized approach unless you cannot provide efficient distributed substitutes
 - Central resources might become a bottleneck:
 - Components (single server)
 - Tables (directories in DFS)
 - Algorithms (deadlock detection)

Jochen Liedtke's remarks:

- Symmetric systems tend to be scalable
- Simple, yet elegant systems tend to be efficient



Design Rules improving Scalability

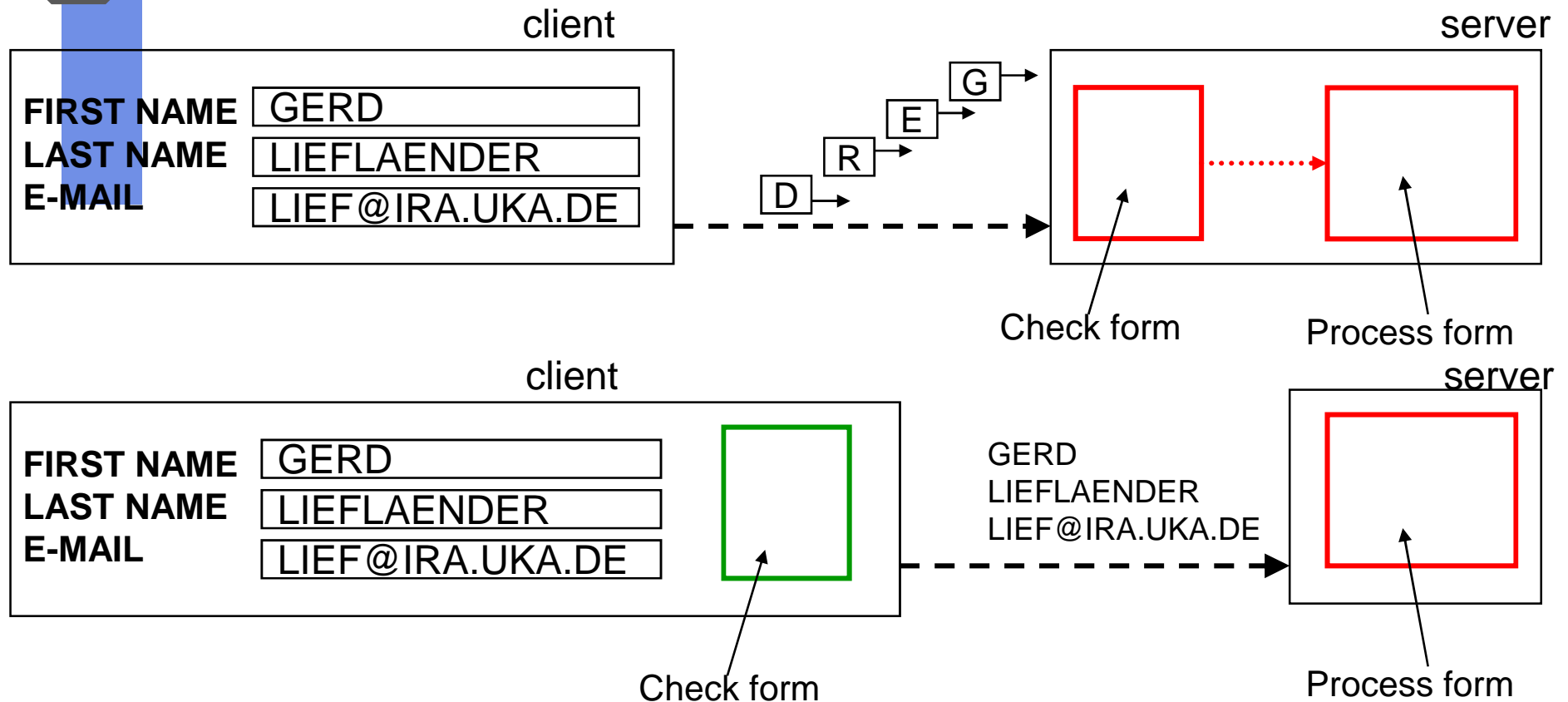
1. Do not require any node within the DS to hold the complete system state
2. Allow nodes to make decisions based only upon local information
3. Design algorithms that can survive failures of individual nodes
4. Make no assumptions about a global clock



Additional Scalability Problems

- DS designed for LANs often use synchronous IPC
 - Delays due to message transfer ~ 100 μ sec
 - LAN is reliable & with efficient broadcast
- In global DS no efficient synchronous IPC
 - Delays due to message transfer ~ 100 msec
 - WAN is unreliable & point-to-point
- Example: Locate a server in a network
 - LAN: broadcast a server lookup, collect all positive replies, take the best fitting one
 - WAN: broadcasting is too inefficient (see Internet with its billion users)

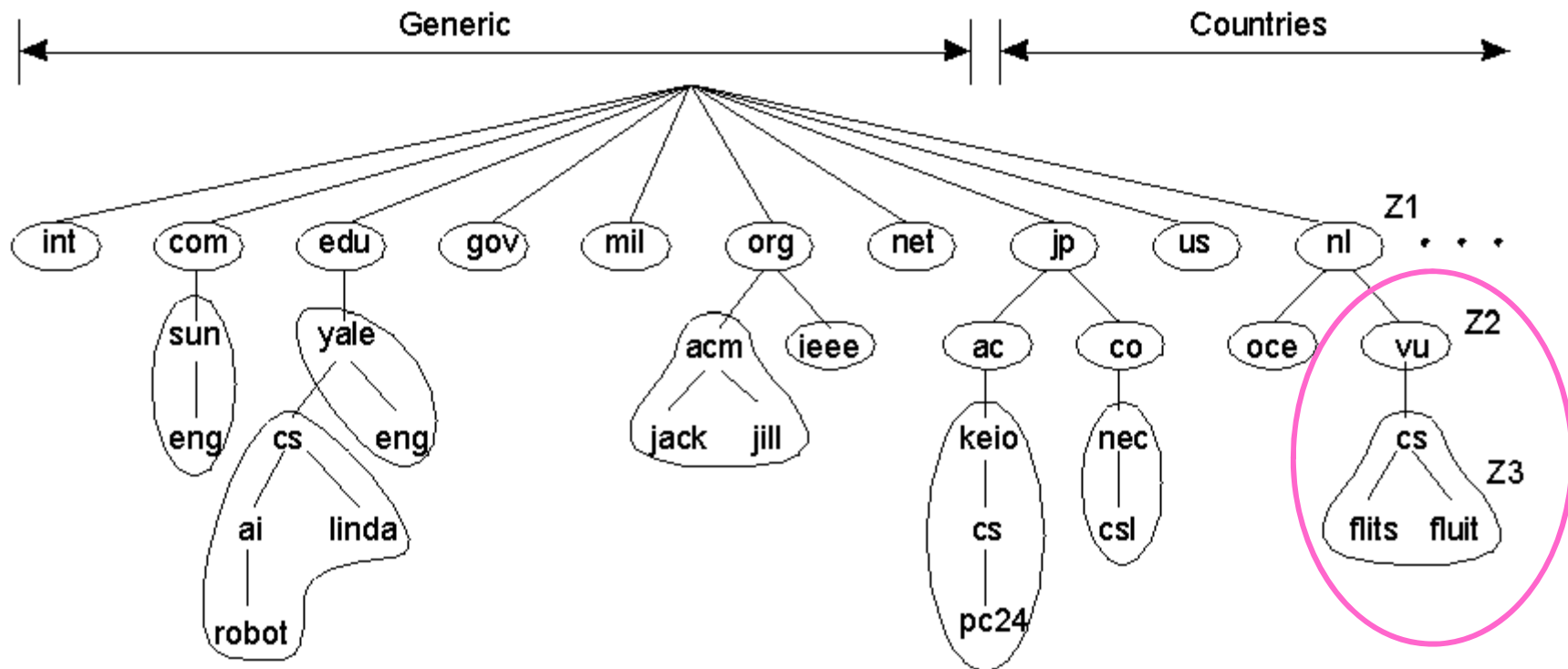
Techniques for Scalability (1)



- Difference between having the server or the client check whether the form has been filled completely ...
- Security check done on the server side



Techniques for Scalability (2)



- Dividing the DNS name space into **disjoint zones**, **domains** in order to achieve improved performance



Further Techniques for Scalability

■ Replication

- Web servers are replicated
- URLs contain the name of the corresponding replicated server
- Replication is planned by server administration

■ Caching

- Local copy of a webpage at the client's site
 - Caching happens on demand & client can specify when content is outdated
- Both techniques can lead to consistency problems



Security

Challenge: face **intelligent, malicious attackers**
all around the world

What has to be done?

- Maintain integrity of state
- Maintain privacy of data
- Prevent unauthorized use of services

- Maintain **availability** of services, i.e.
implement robust systems

see other courses



Security (2)

Guiding principles:

- Design for security from the very first start
- Separate security policies from mechanisms
- Use strong authentication and encryption
- Provide tight resource management
- Install a small trusted computing base (TCB)
- Rely on diversity

Ann N. Sovarel et al.: "Where's the FEEB? The Effectiveness of Instruction Set Randomization"



Reliability

- DS should improve **availability** (= fraction of time during which the system is **usable**)
 - Ideally: Boolean OR of component availabilities
 - Worst case: Boolean AND

Whenever we present a solution, check its reliability!

- Main techniques:
 - Avoid simultaneous failure of critical components
 - HW/SW-redundancy \Rightarrow complicates consistency
 - **Avoid single points of failure whenever possible**



Reliability (2)

Fault tolerance: **recover properly** from failures

- Minimize loss of data and state
- Minimize impact on running applications
- Retain the system's consistency

In DS, fault tolerance is more complex due to

- Partial failure (distributed state)
- Both network and node failure
- Complex failure modes (Byzantine)



Performance

- High performance in a DS is not that easy
- Other requirements can conflict with, e.g.
 - Transparency
 - Extra overhead needed
 - Self Management & Migration
 - Additional delay can occur (not occurred yesterday)
 - Reliability
 - Send additional messages to replicated server
 - Install alive messages



User's Performance Requirements

- **Response time:**
Users need fast, predictable responses, \Rightarrow
 - as few components as possible
 - whenever possible, local IPC
 - exchange only necessary information
- **Throughput:**
Number of completed applications/time-unit \Rightarrow
 - the weakest system component dominates throughput
 - prepare DS for future extensions
- **Load balancing:**
Automatically migrate load to nodes that
 - are currently free or
 - do less important work



Quality of Service

- If the functionality of a service is provided \Rightarrow
how to guarantee its quality of service?
- Tackle the following problems:
 - Performance
 - Reliability
 - Security
 - e.g. no **DOS (Denial Of Service)** attacks
 - Latency ... and other real-time requirements



Replication and Caching

When replication or caching is used:

*How to guarantee that users get the **new(est) version** in case of proxy or client caching?*

- 1. approach:
Whenever the server is updated, invalidate all caches, e.g. you have to know them, i.e. some kind of a statefull server
- 2. approach:
Estimate, when cached information might be outdated
⇒ you cannot always expect “up to date” data



Dependability¹

Dependability includes:

- Correctness:
 - DS should act as specified

- Security:
 - what location inside DS is the best protected one

- Failure tolerance:
 - describe how system still runs in case of failures

¹ Dependability = Verlässlichkeit



Typical DS Design Pitfalls¹

False assumptions made by novice DS developers:

- Network is reliable
- Network is secure
- Network is homogeneous
- Topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- Only one administrator

¹ According to L. Peter Deutsch

Read: <http://devlinux.org/deutsch-interview.html>



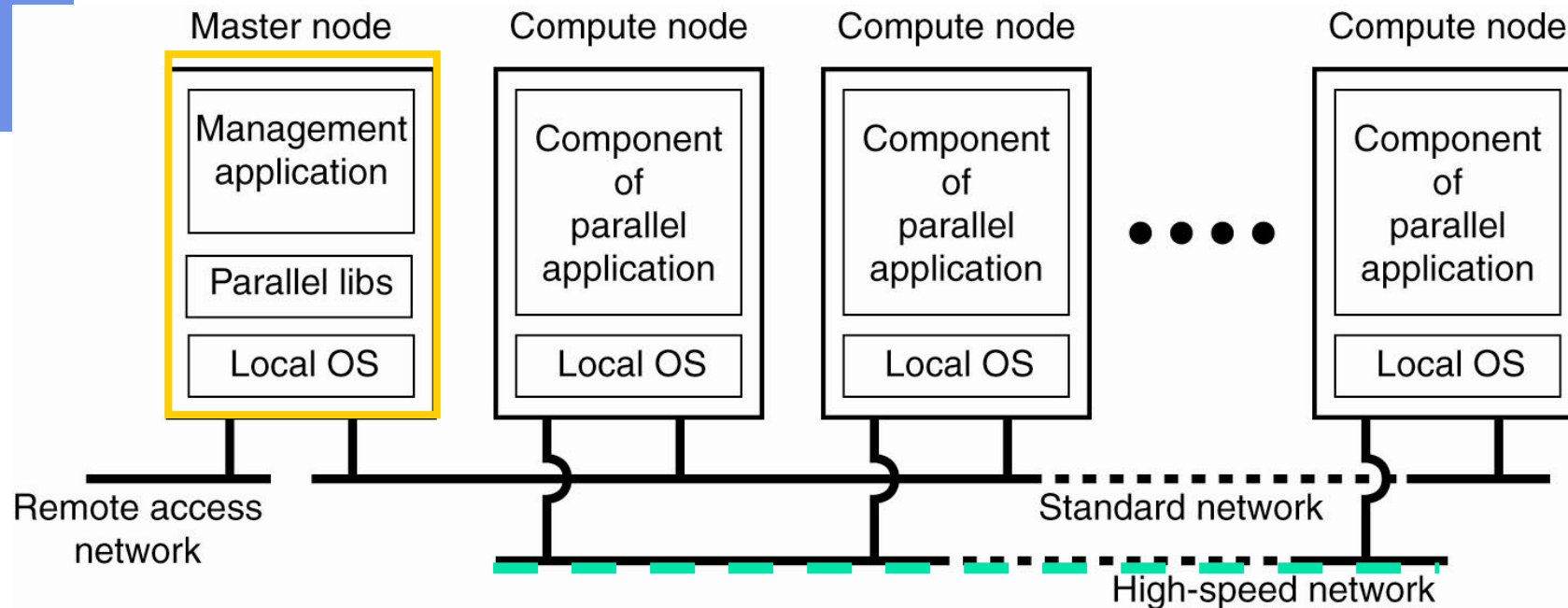
Types of DS

Distributed Computing Systems

Distributed Information Systems

Distributed Pervasive Systems

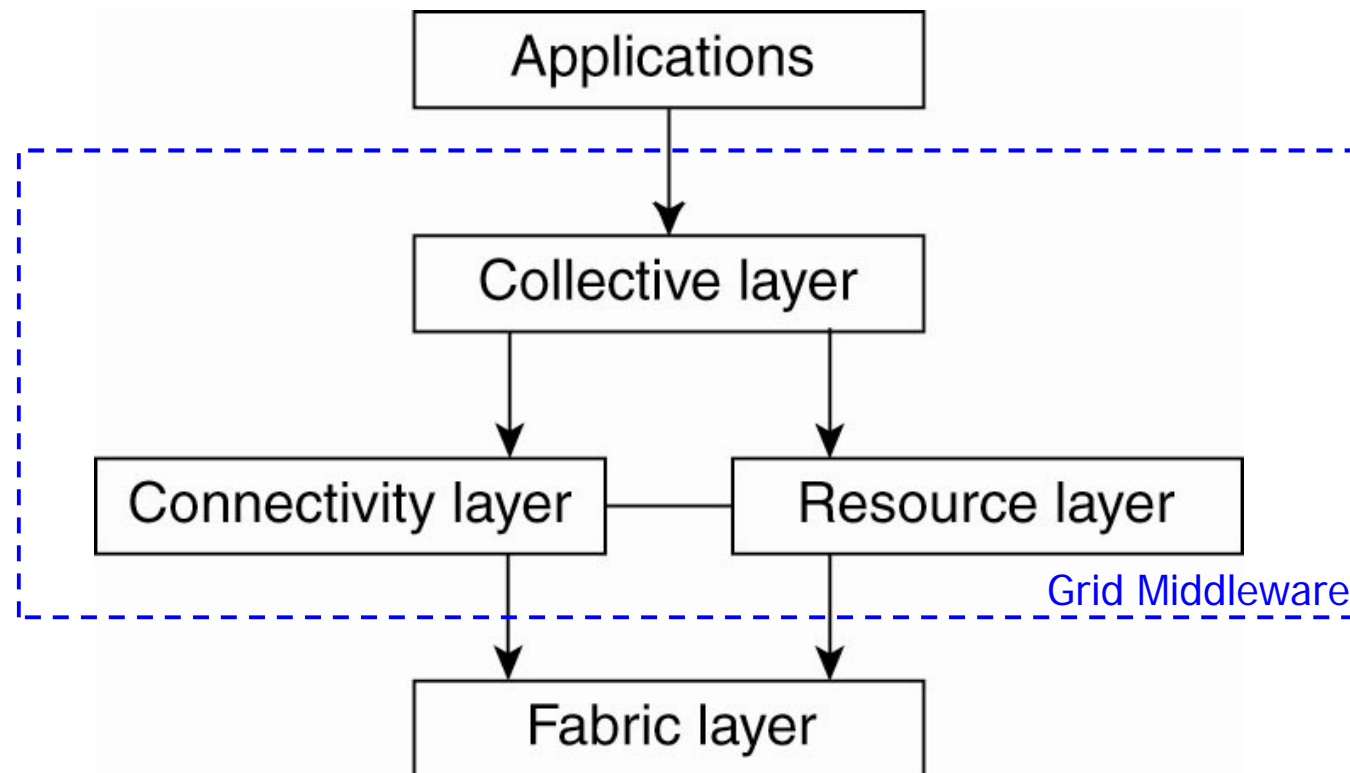
Cluster Computing Systems



- Example of a cluster computing system
- Clusters tend to be **homogeneous** (HW & SW)



Grid Computing Systems



- Layered architecture for grid computing systems (see: Foster et al: The Anatomy of the Grid, enabling Scalable Virtual Organizations)



Transaction Processing Systems (1)

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

- Example primitives for transactions



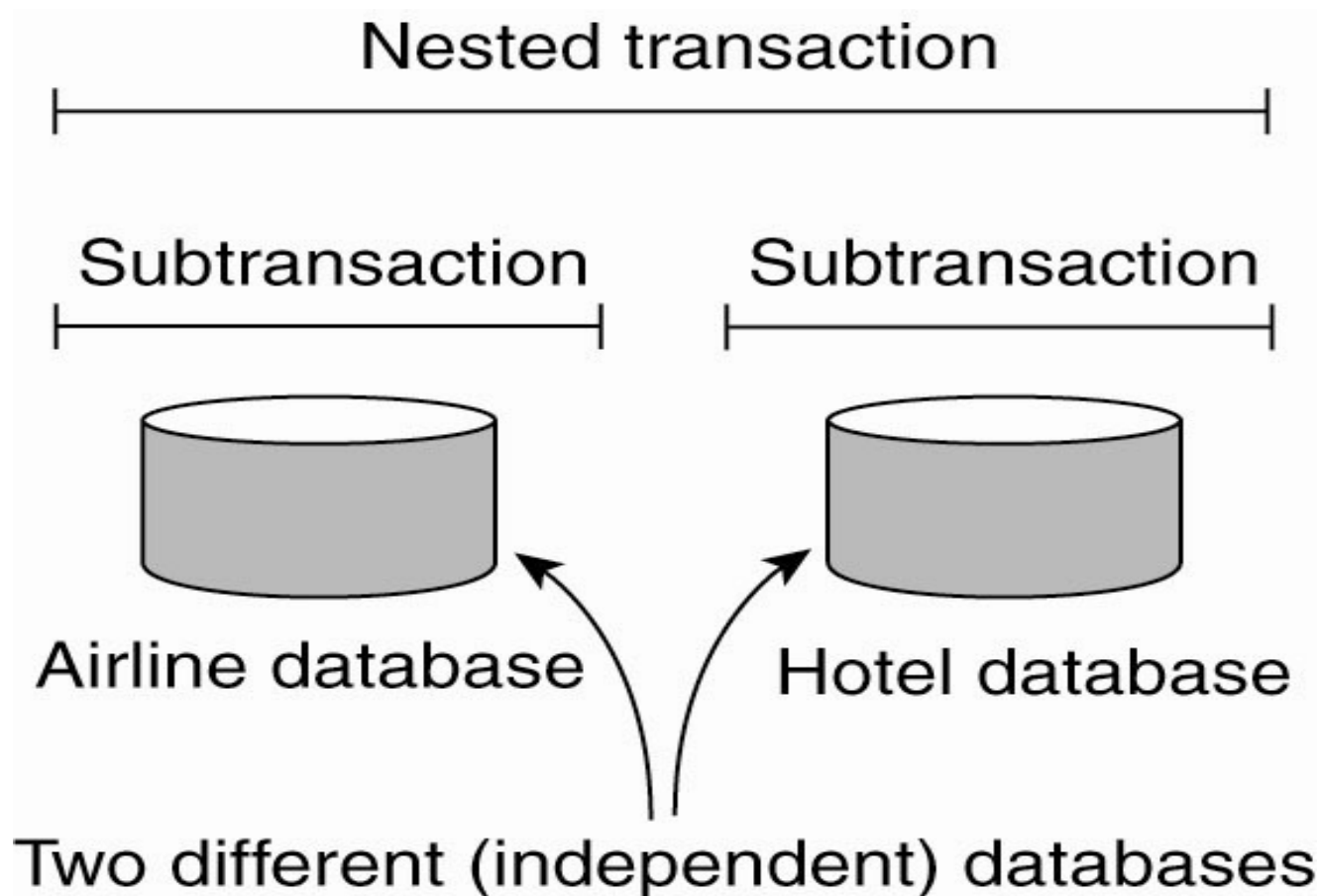
Transaction Processing Systems (2)

Characteristic properties of transactions:

- **Atomic**: To the outside world, the transaction happens **indivisibly**
- **Consistent**: The transaction **does not violate** system invariants.
- **Isolated**: Concurrent transactions **do not interfere** with each other.
- **Durable**: Once a transaction commits, the changes are permanent.



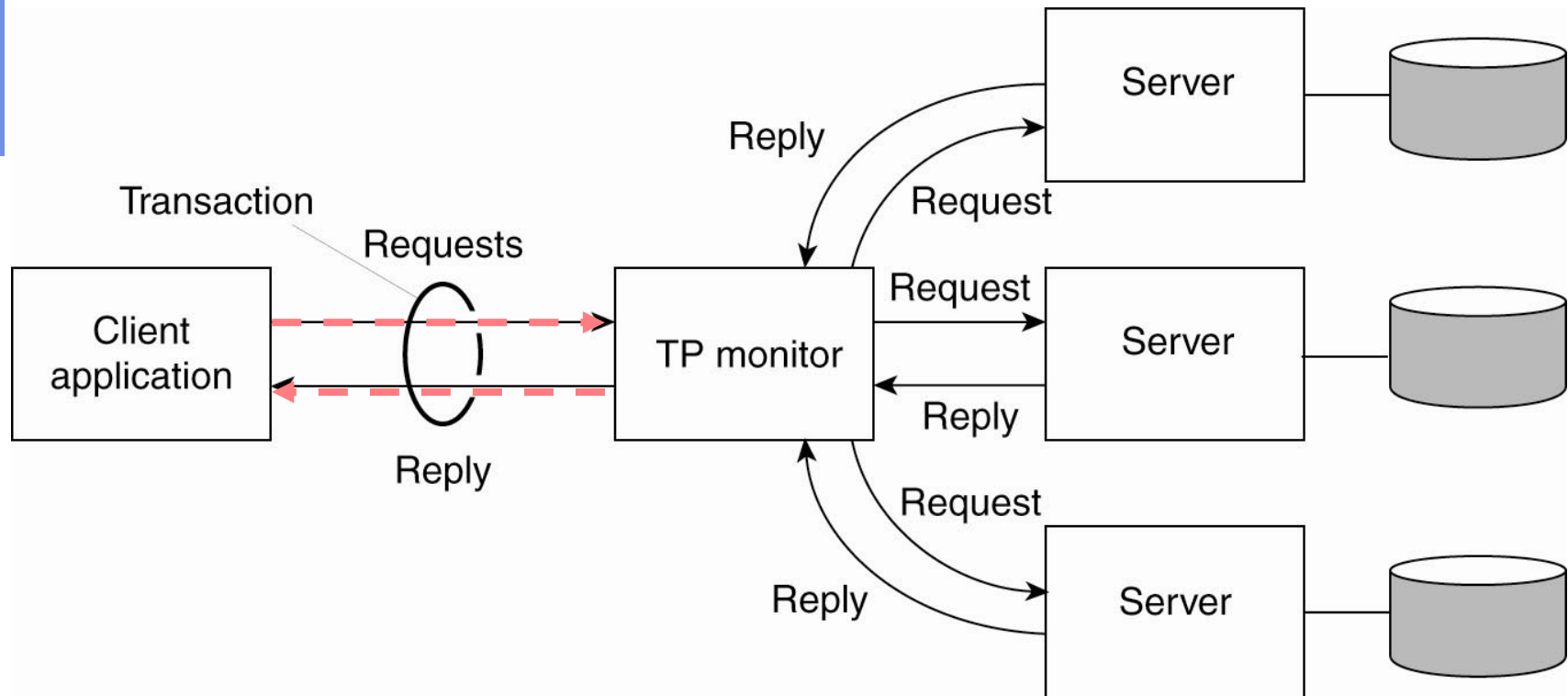
Transaction Processing Systems (3)



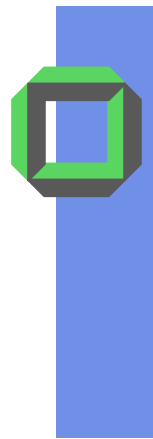
- Simple example of a nested transaction



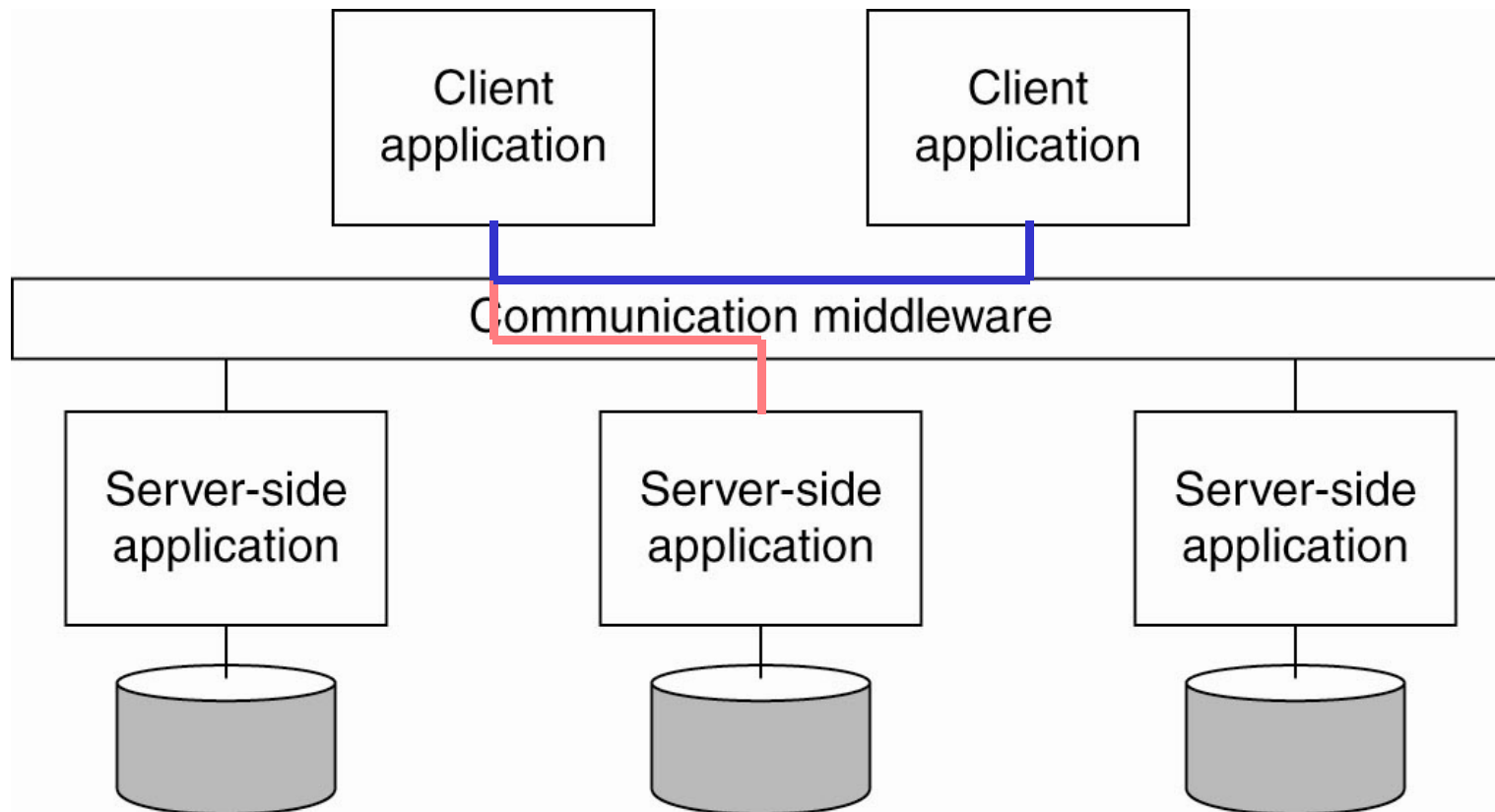
Transaction Processing Systems (4)



- Role of a TP monitor in DSs



Enterprise Application Integration



- Middleware as a communication facilitator in enterprise application integration



Distributed Pervasive Systems

Additional requirements for pervasive DS

- Embrace contextual changes
 - Mobile & embedded small computing devices
 - Battery powered, only wireless connection
 - Need to discover their environment
- Encourage ad hoc composition
- Recognize sharing as the default

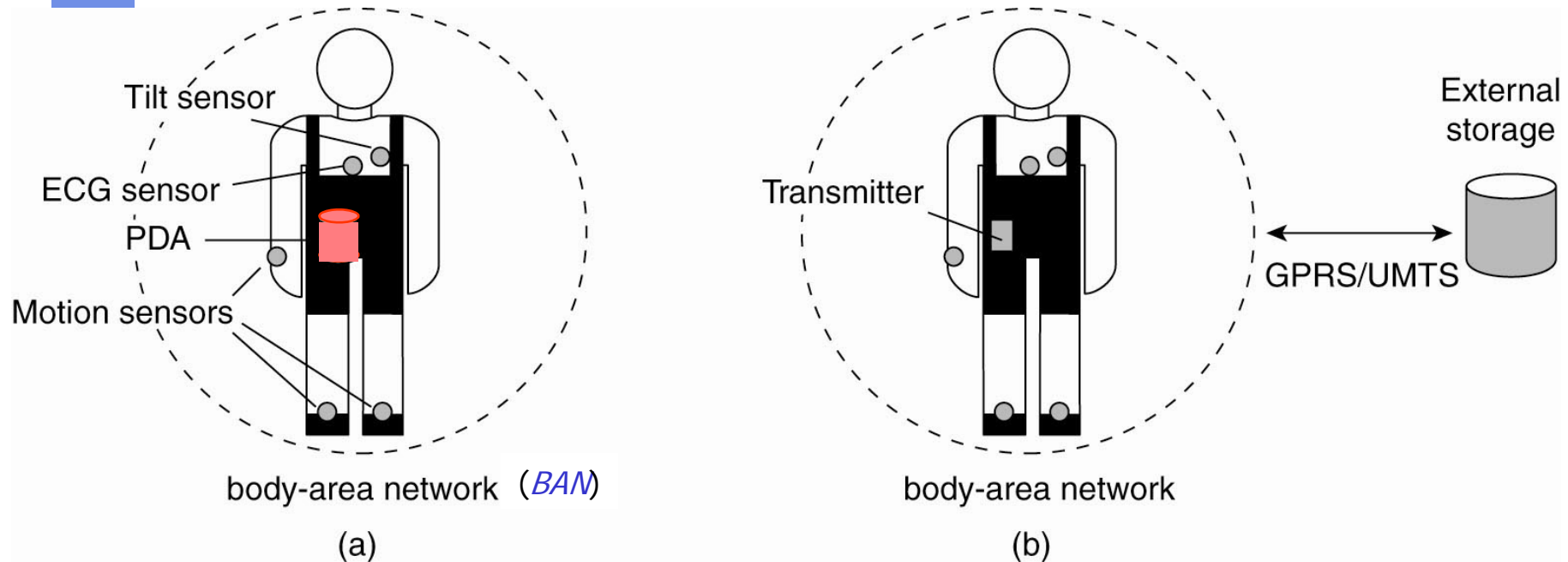


Home Systems

- Home networks with
 - PCs, TV, video, game boys, ...
 - Smart phones, PDAs
 - Kitchen ~ or cleaning robots, ...
 - Surveillance camera
 - Control units for lights, sun protection, ...
- Need for self configuration & management
 - See Universal Plug and Play standard (UPnP)
 - How to update without manual intervention
 - Personal space supported by recommenders



Electronic Health Care Systems (1)



- Monitoring a person in a pervasive electronic health care system, using
 - (a) a local **hub collecting data** that are offloaded from time to time to a larger storage device (hub can also manage the BAN)
 - (b) a continuous wireless connection, BAN is hooked up to an external network



Electronic Health Care Systems (2)

Prevent people from being hospitalized, yet still monitored

Personal HCS are often body-area network (BAN)

Problems of health care systems:

- *Where and how should monitored data be stored?*
- *How can we prevent loss of crucial data?*
- *What infrastructure is needed to generate & propagate alerts?*
- *How can physicians provide online feedback?*
- *How can you install robustness of the monitoring system?*
- *What are security issues & how can proper policies be enforced?*



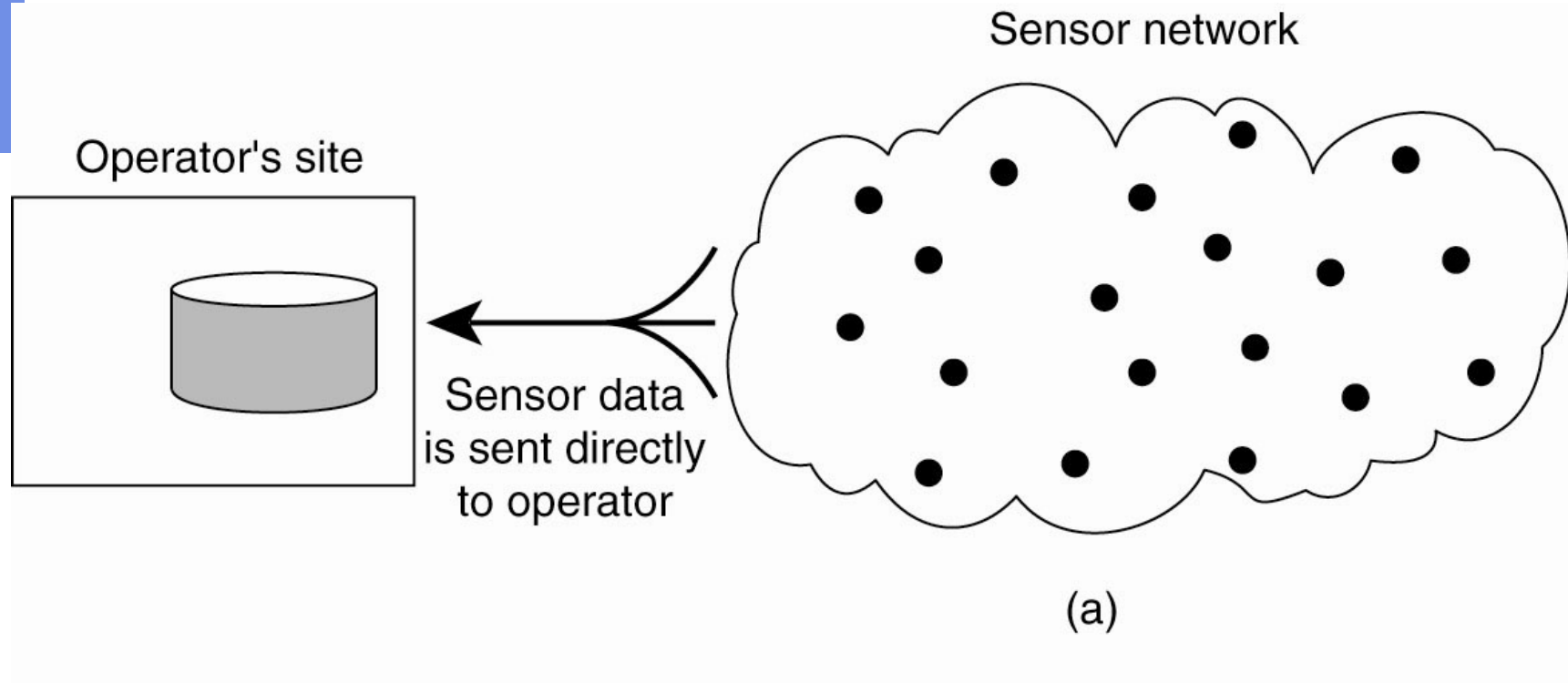
Sensor Networks (1)

Questions concerning sensor networks:

- *How do we (dynamically) set up an efficient tree in a sensor network?*
- *How does aggregation of results take place? Can it be controlled?*
- *What happens when network links fail?*
- *How can we increase the lifetime of the sensors whilst decreasing their energy amount?*



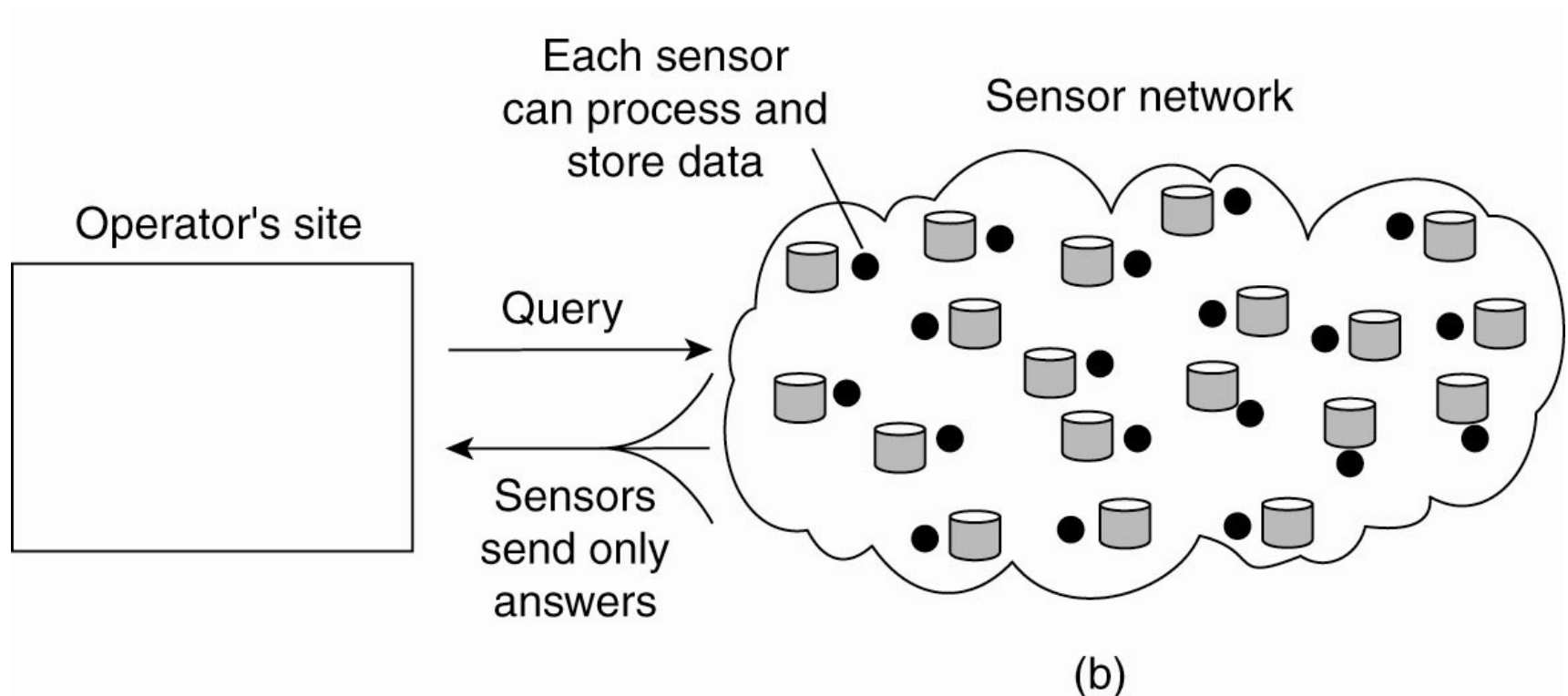
Sensor Networks (2)



- Organizing a sensor network database, while storing and processing data (a) only at the operator's site or ...



Sensor Networks (3)



- Organizing a sensor network database, while storing and processing data ... or (b) only at the sensors.