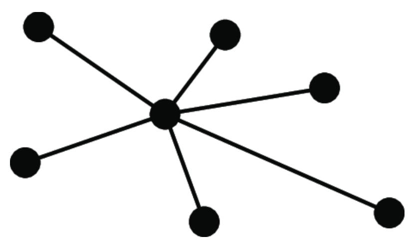
Distributed Systems Overview

Thomas Schwarz, SJ

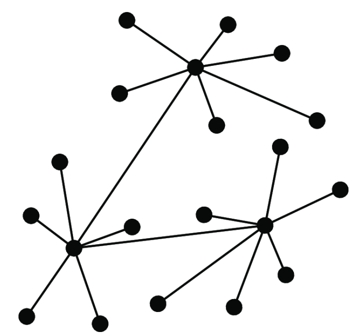
Origins

- 1980:
  - Computers are big and expensive
  - No communication between computers
- 1980s:
  - System on a chip: cheap, small computers
  - High-speed computer networks
- 2010 onward:
  - Multi-core systems
- 2020 onward:
  - Ubiquitous computing, high interconnection between systems

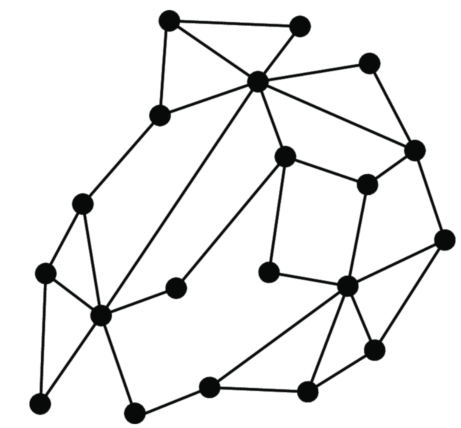
What does “Distributed” mean?



Centralized System



Decentralized System



Distributed System

- Distributed vs. decentralized system
- Two views on realizing distributed systems
  - Integrative view: connecting existing networked computer systems into a larger system.
  - Expansive view: an existing networked computer systems is extended with additional computers

What does “Distributed” mean?

- Two definitions
  - A decentralized system is a networked computer system in which processes and resources are **necessarily** spread across multiple computers.
  - A distributed system is a networked computer system in which processes and resources are **sufficiently** spread across multiple computers

What does “Distributed” mean?

- Examples of decentralized systems:
  - Federated Learning
    - Machine learning with data from several companies
    - Company data is not revealed to other companies
    - By necessity decentralized

What does “Distributed” mean?

- Examples of decentralized systems:
  - Blockchain - Distributed Ledger
    - Participants do not trust each other
      - but want to collaborate
    - Participants validate transactions by others

What does “Distributed” mean?

- Examples of distributed systems:
  - Email service: Log in via the name url, but get directed to one of many different servers
  - **C**ontent **D**elivery **N**etworks e.g. Akamai
  - **N**etwork **A**ttached **S**torage: various storage devices connected by a (often dedicated) network

Common Misconceptions

- Centralized solutions do not scale
  - Make distinction between logically and physically centralized. The root of the Domain Name System:
    - logically centralized
    - physically (massively) distributed
    - decentralized across several organizations

Common Misconceptions

- Centralized solutions have a single point of failure
  - Generally not true (e.g., the root of DNS).
  - A single point of failure is often:
    - easier to manage
    - easier to make more robust

Common Misconceptions

- In general:
  - There are many, poorly founded, misconceptions regarding scalability, fault tolerance, security, etc.
  - We need to develop skills by which distributed systems can be readily understood so as to judge such misconceptions.

Perspectives in Studying Distributed Systems

- Architecture: common organizations
- Process: what kind of processes, and their relationships
- Communication: facilities for exchanging data
- Coordination: application-independent algorithms
- Naming: how do you identify resources?
- Consistency and replication: performance requires of data, which need to be the same
- Fault tolerance: keep running in the presence of partial failures
- Security: ensure authorized access to resources

Design Goals

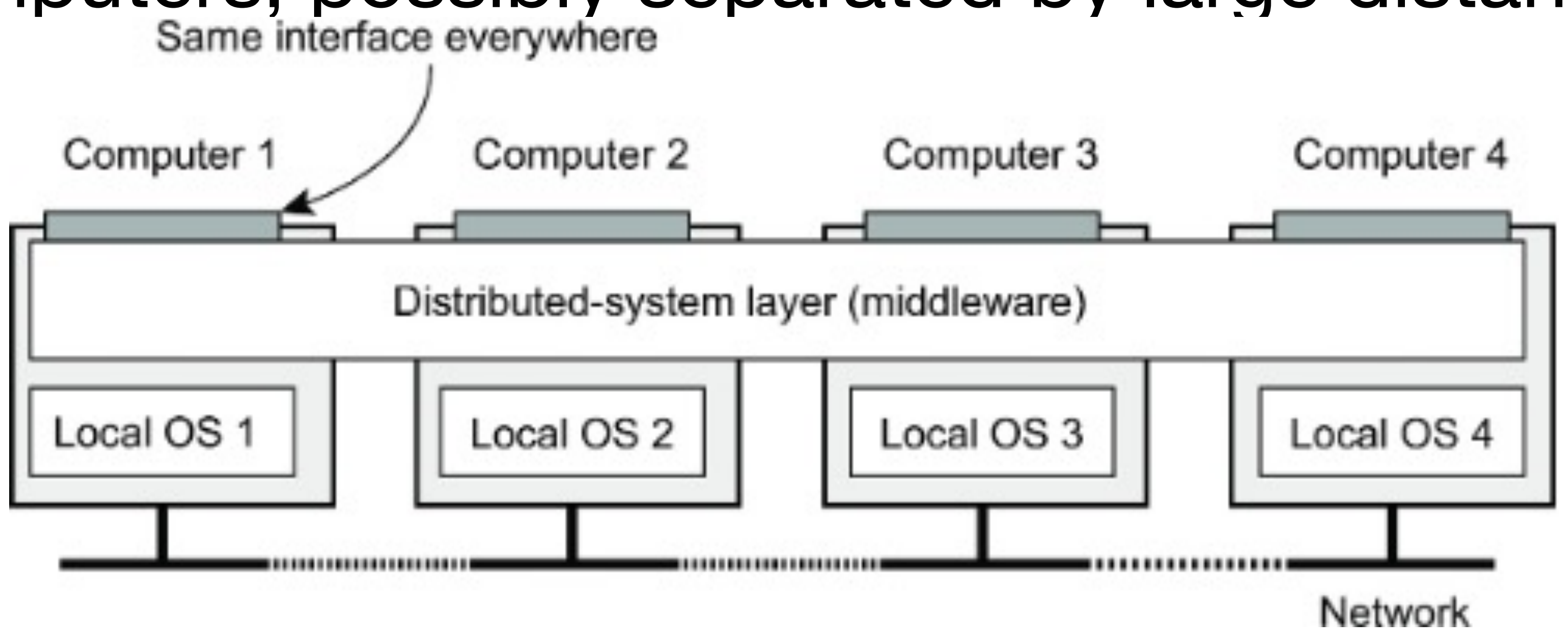
- Overall design goals
  - Support sharing of resources
  - Distribution transparency
  - Openness
  - Scalability

Sharing Resources

- Cloud-based shared storage and files
- Peer-to-peer assisted multimedia streaming
- Shared mail services
  - e.g. outsourced mail system
- Shared Web hosting
  - Content Distribution Networks)

Distribution Transparency

- What is transparency?
  - The phenomenon by which a distributed system attempts to hide the fact that its processes and resources are physically distributed across multiple computers, possibly separated by large distances.



- Layer between applications and operating systems: a middleware layer

Distribution Transparency

Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Distribution Transparency

- Transparency is limited by:
  - There are communication latencies that cannot be hidden
  - Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash
  - Full transparency will cost performance, exposing distribution of the system
  - Keeping replicas exactly up-to-date with the master takes time
  - Immediately flushing write operations to disk for fault tolerance

Distribution Transparency

- Exposing distribution may be good
  - Making use of location-based services (finding your nearby friends)
  - When dealing with users in different time zones
  - When it makes it easier for a user to understand what's going on (when e.g., a server does not respond for a long time, report it as failing).

- Conclusion
  - Distribution transparency is a nice goal, but achieving it is a different story, and it should often not even be aimed at.

Openness of distributed systems

- Open distributed system
  - A system that offers components that can easily be used by, or integrated into other systems. An open distributed system itself will often consist of components that originate from elsewhere.
- What are we talking about?
  - Be able to interact with services from other open systems, irrespective of the underlying environment:
  - Systems should conform to well-defined interfaces
  - Systems should easily interoperate
  - Systems should support portability of applications
  - Systems should be easily extensible

Openness of distributed systems

- **Implementing openness: policies**
  - What level of consistency do we require for client-cached data
  - Which operations do we allow downloaded code to perform?
  - Which QoS requirements do we adjust in the face of varying bandwidth?
  - What level of secrecy do we require for communication?
- **Implementing openness: mechanisms**
  - Allow (dynamic) setting of caching policies
  - Support different levels of trust for mobile code
  - Provide adjustable QoS parameters per data stream
  - Offer different encryption algorithms

Separation of Policies and Mechanisms

- Observation
  - The stricter the separation between policy and mechanism, the more we need to ensure proper mechanisms, potentially leading to many configuration parameters and complex management.
- Finding a balance
  - Hard-coding policies often simplifies management, and reduces complexity at the price of less flexibility. There is no obvious solution.

Dependability

- Basics
  - A component provides services to clients.
  - To provide services, the component may require the services from other components
  - $\Rightarrow$ a component may depend on some other component.
- Specifically
  - A component C depends on C_* if the correctness of C's behavior depends on the correctness of C_*'s behavior. (Components are processes or channels.)

Requirements for Dependability

Requirement	Description
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

Reliability vs Dependability

- Reliability R(t) of component C
  - Conditional probability that C has been functioning correctly during [0, t) given C was functioning correctly at the time T = 0.
- Traditional metrics
  - Mean Time To Failure (MTTF): The average time until a component fails.
  - Mean Time To Repair (MTTR): The average time needed to repair a component.
  - Mean Time Between Failures (MTBF): Simply MTTF + MTTR.

Terminology

Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

Terminology

Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don’t hire sloppy programmers
Fault tolerance	Build a component and make it mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

Security

- Observation:
  - A distributed system that is not secure, is not dependable
- What we need
  - Confidentiality: information is disclosed only to authorized parties
  - Integrity: Ensure that alterations to assets of a system can be made only in an authorized way
-

Security

- Authorization, Authentication, Trust
    - Authentication: verifying the correctness of a claimed identity
    - Authorization: does an identified entity has proper access rights?
    - Trust: one entity can be assured that another will perform particular actions according to a specific expectation

Security

- Keeping it simple
  - It’s all about encrypting and decrypting data using security keys.
  - Notation:
    - K (data) denotes that we use key K to encrypt/decrypt data.

Security Mechanisms

- Symmetric cryptosystem
  - With encryption key EK (data) and decryption key DK (data):
    - if data = DK (EK (data)) then DK = EK . Note: encryption and descryption key are the same and should be kept secret.

Security Mechanisms

- Asymmetric cryptosystem
  - Distinguish a public key PK (data) and a private (secret) key SK (data).
  - Encrypt message from Alice to Bob:
    - data $= SK_{\text{Bob}}\left(PK_{\text{Bob}}(\text{data}\right)$
  - Sign message from Alice to Bob:
    - Alice sends [data, $SK_{\text{Alice}}(\text{data})$]
    - Bob checks that data $= PK_{\text{Alice}}\left(SK_{\text{Alice}}(\text{data})\right)$

Security Mechanisms

- Secure hashing
  - In practice, use hash functions
    $h : \text{Object} \mapsto h(\text{Object})$
    - $h(\text{Object})$ is a small byte string
  - Any change in the object changes with high probability $h(\text{Object})$.
  - If two hashes are different, objects are different
  - It two hashes are equal, objects are likely to be the same

Scale

- “scalable” is a buzz-word
- At least three components
    - Number of users or processes (size scalability)
    - Maximum distance between nodes (geographical scalability)
    - Number of administrative domains (administrative scalability)

Scale

- Observation
  - Most systems account only, to a certain extent, for size scalability.
  - Often a solution: multiple powerful servers operating independently in parallel.
  - Today, the challenge still lies in geographical and administrative scalability.

Size Scalability

- Root causes for scalability problems with centralized solutions
  - The computational capacity, limited by the CPUs
  - The storage capacity, including the transfer rate between CPUs and disks
  - The network between the user and the centralized service

Geographical Scalability

- Cannot simply go from LAN to WAN: many distributed systems assume synchronous client-server interactions: client sends request and waits for an answer. Latency may easily prohibit this scheme.
- WAN links are often inherently unreliable: simply moving streaming video from LAN to WAN is bound to fail.
- Lack of multipoint communication, so that a simple search broadcast cannot be deployed. Solution is to develop separate naming and directory services (having their own scalability problems).

Administrative Scalability

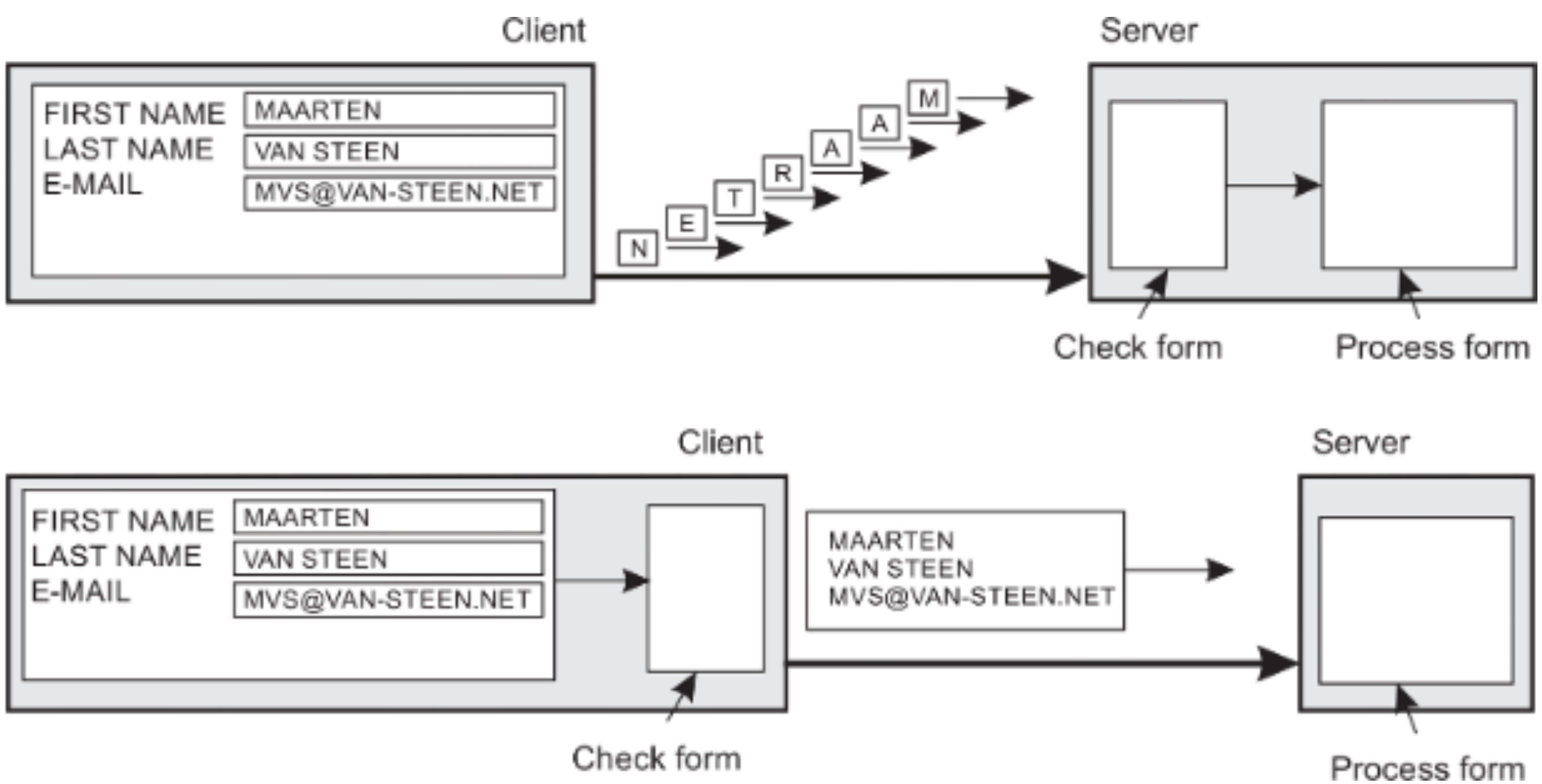
- Essence
  - Conflicting policies concerning usage (and thus payment), management, and security
  - Examples
    - Computational grids: share expensive resources between different domains.
    - Shared equipment: how to control, manage, and use a shared radio telescope constructed as large-scale shared sensor network?
- Exception: several peer-to-peer networks
- File-sharing systems (based, e.g., on BitTorrent)
- Peer-to-peer telephony (early versions of Skype)
- Peer-assisted audio streaming (Spotify)
- Note: end users collaborate and not administrative entities

Techniques for Scaling

- Hide communication latencies
  - Make use of asynchronous communication
  - Have separate handler for incoming response
  - Problem: not every application fits this model

Techniques for Scaling

- Facilitate solution by moving computations to client



Techniques for Scaling

- Partition data and computations across multiple machines
  - Move computations to clients (Java applets and scripts)
  - Decentralized naming services (DNS)
  - Decentralized information systems (WWW)

Techniques for Scaling

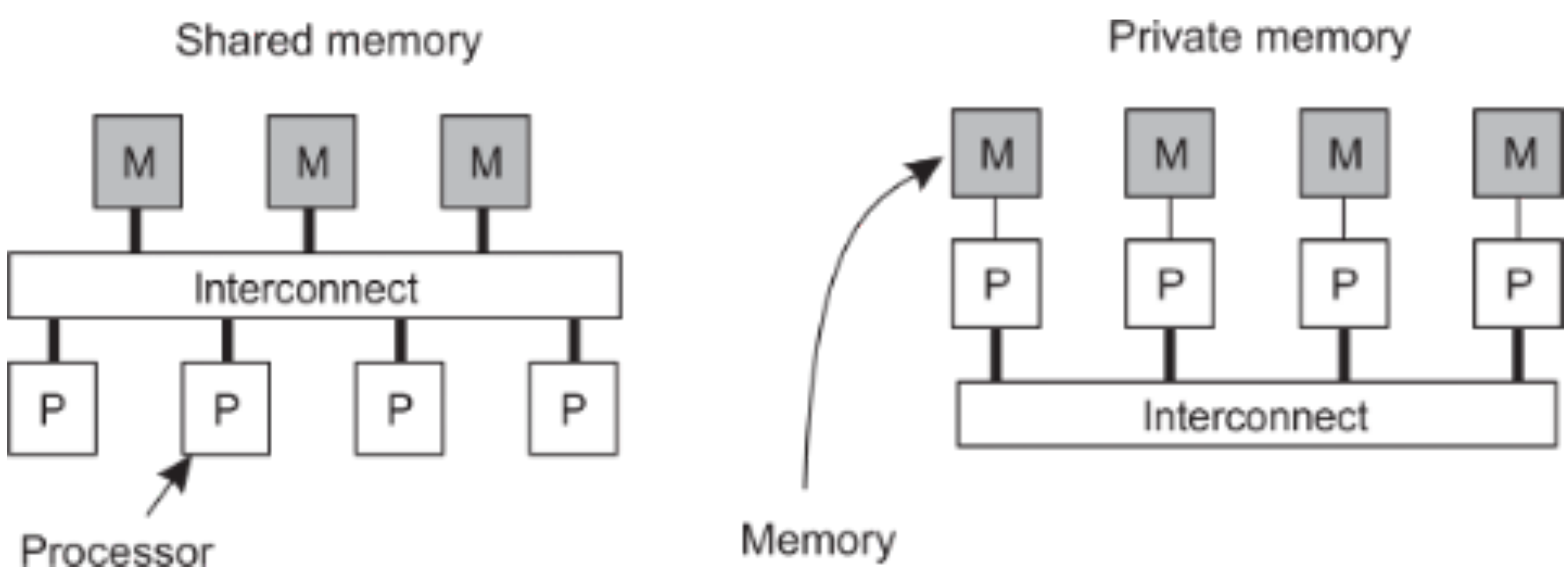
- Replication and caching: Make copies of data available at different machines
  - Replicated file servers and databases
  - Mirrored Websites
  - Web caches (in browsers and proxies)
  - File caching (at server and client)

Techniques for Scaling: Replication

- Applying replication is easy, except for one thing
  - Having multiple copies (cached or replicated), leads to inconsistencies: modifying one copy makes that copy different from the rest.
  - Always keeping copies consistent and in a general way requires global synchronization on each modification.
  - Global synchronization precludes large-scale solutions.
  - Observation
    - If we can tolerate inconsistencies, we may reduce the need for global synchronization, but tolerating inconsistencies is application dependent

Parallel Computing

- Observation
  - High-performance distributed computing started with parallel computing
  - Multiprocessor and multicore versus multicomputer

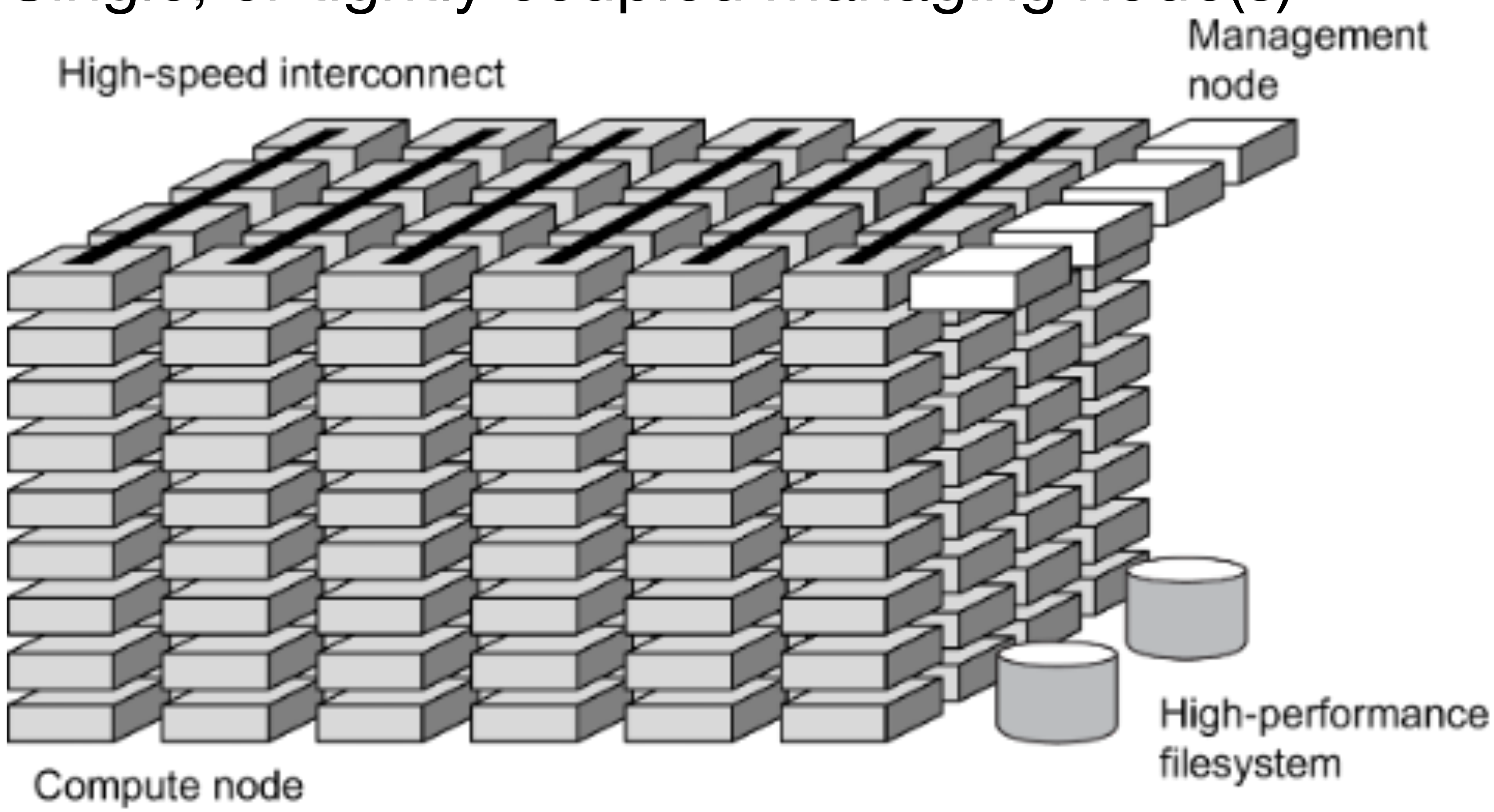


Distributed Shared Memory Systems

- Observation
  - Multiprocessors are relatively easy to program in comparison to multicomputers, yet have problems when increasing the number of processors (or cores). Solution: Try to implement a shared-memory model on top of a multicomputer.
- Example through virtual-memory techniques
  - Map all main-memory pages (from different processors) into one single virtual address space. If a process at processor A addresses a page P located at processor B, the OS at A traps and fetches P from B, just as it would if P had been located on local disk.
- Problem
  - Performance of distributed shared memory could never compete with that of multiprocessors, and failed to meet the expectations of programmers. It has been widely abandoned by now.

Cluster Computing

- Essentially a group of high-end systems connected through a LAN
  - Homogeneous: same OS, near-identical hardware
  - Single, or tightly coupled managing node(s)

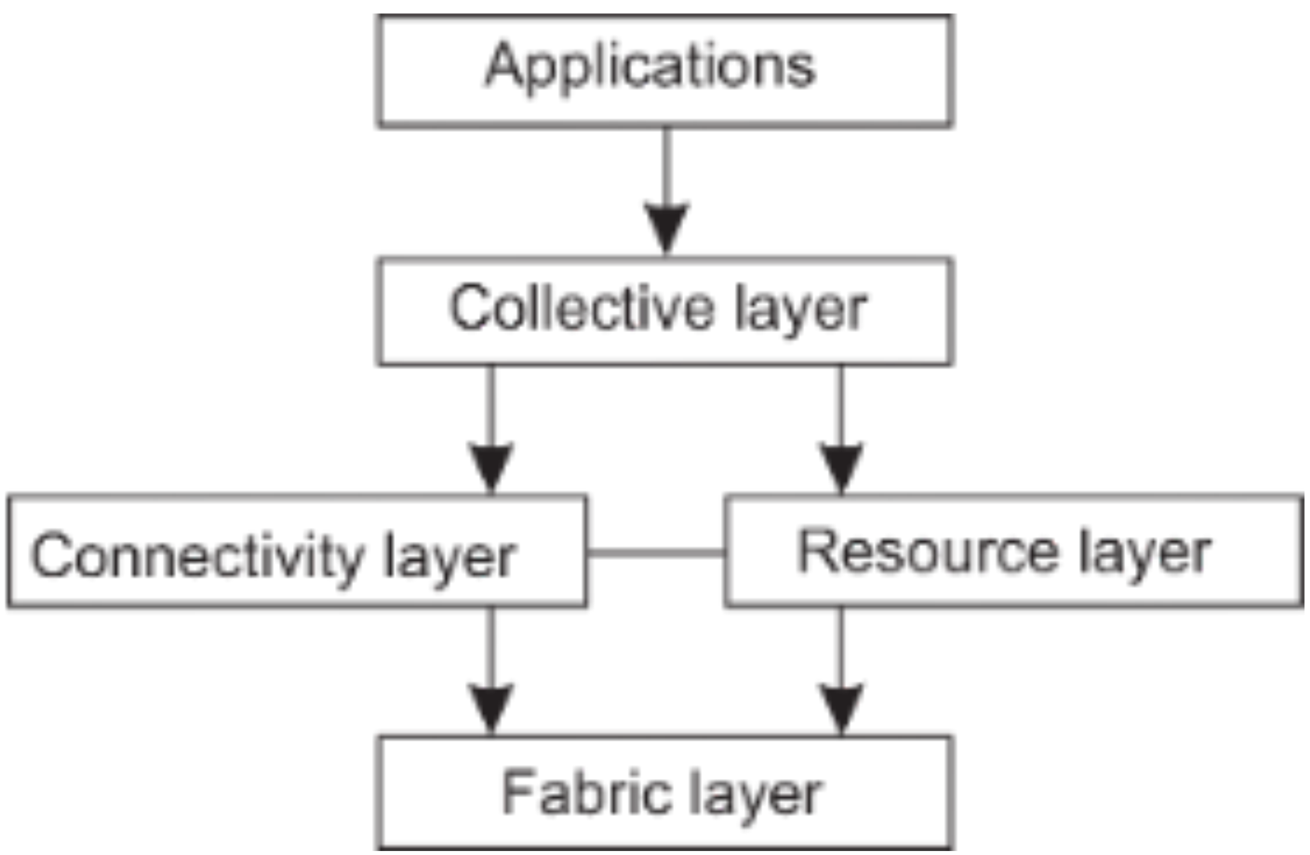


Grid Computing

- The next step: plenty of nodes from everywhere
  - Heterogeneous
  - Dispersed across several organizations
  - Can easily span a wide-area network

  - Note
    - To allow for collaborations, grids generally use virtual organizations. In essence, this is a grouping of users (or better: their IDs) that allows for authorization on resource allocation.

Architecture of Grid computing



- The layers
- Fabric: Provides interfaces to local resources (for querying state and capabilities, locking, etc.)
- Connectivity: Communication/ transaction protocols, e.g., for moving data between resources. Also various authentication protocols.
- Resource: Manages a single resource, such as creating processes or reading data.
- Collective: Handles access to multiple resources: discovery, scheduling, replication.
- Application: Contains actual grid applications in a single organization.

Integration

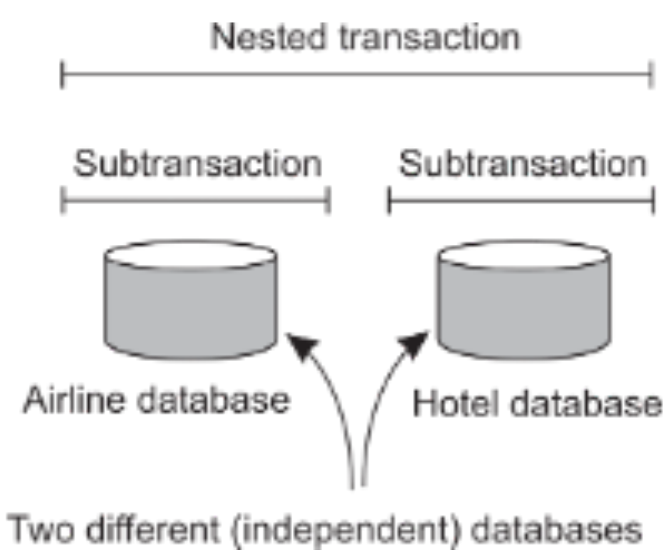
- Situation
  - Organizations confronted with many networked applications, but achieving interoperability was painful.
  - Basic approach
  - A networked application is one that runs on a server making its services available to remote clients. Simple integration: clients combine requests for (different) applications; send that off; collect responses, and present a coherent result to the user.
- Next step
  - Allow direct application-to-application communication, leading to Enterprise Application Integration.

Example: Enterprise Application Integration (EAI)

- Transaction

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
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

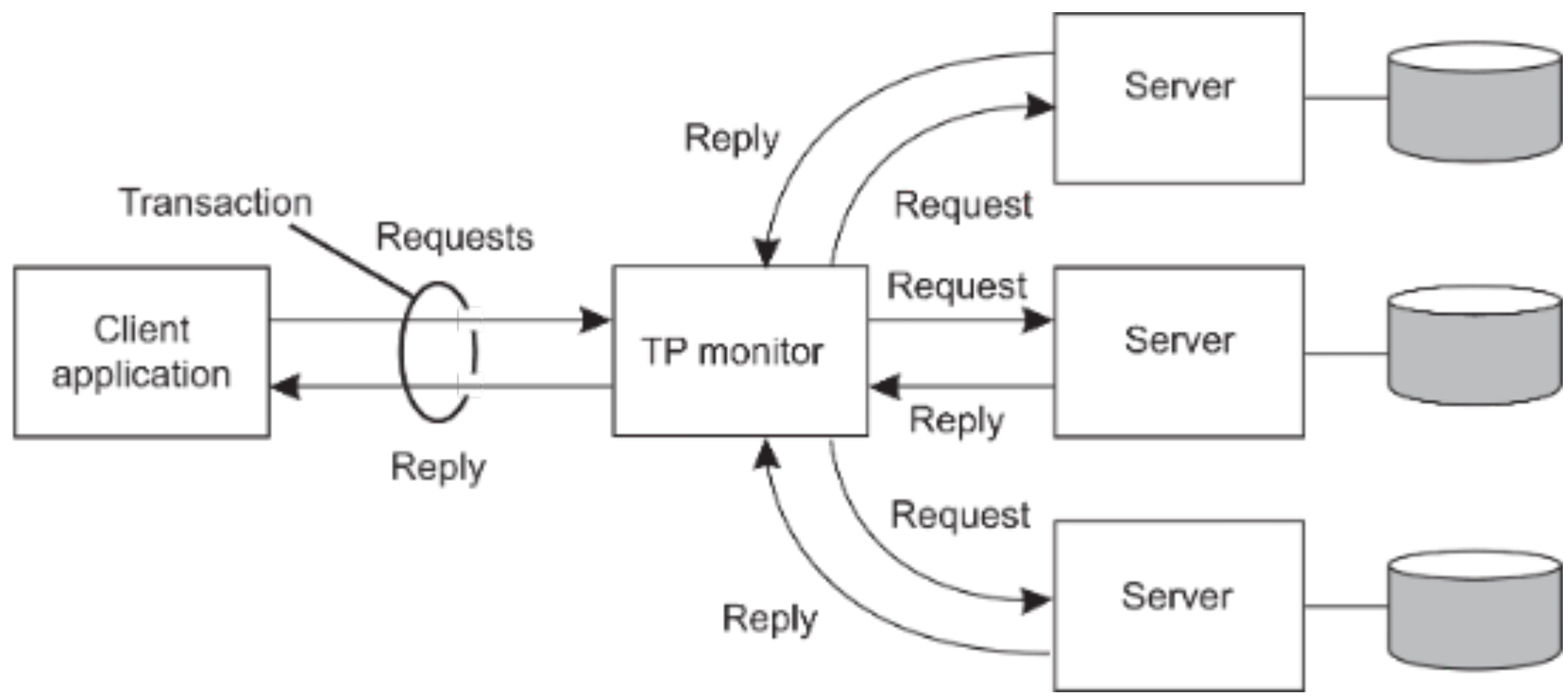
Issue: All or Nothing



- Atomic: happens indivisibly (seemingly)
- Consistent: does not violate system invariants
- Isolated: not mutual interference
- Durable: commit means changes are permanent

Example: Enterprise Application Integration (EAI)

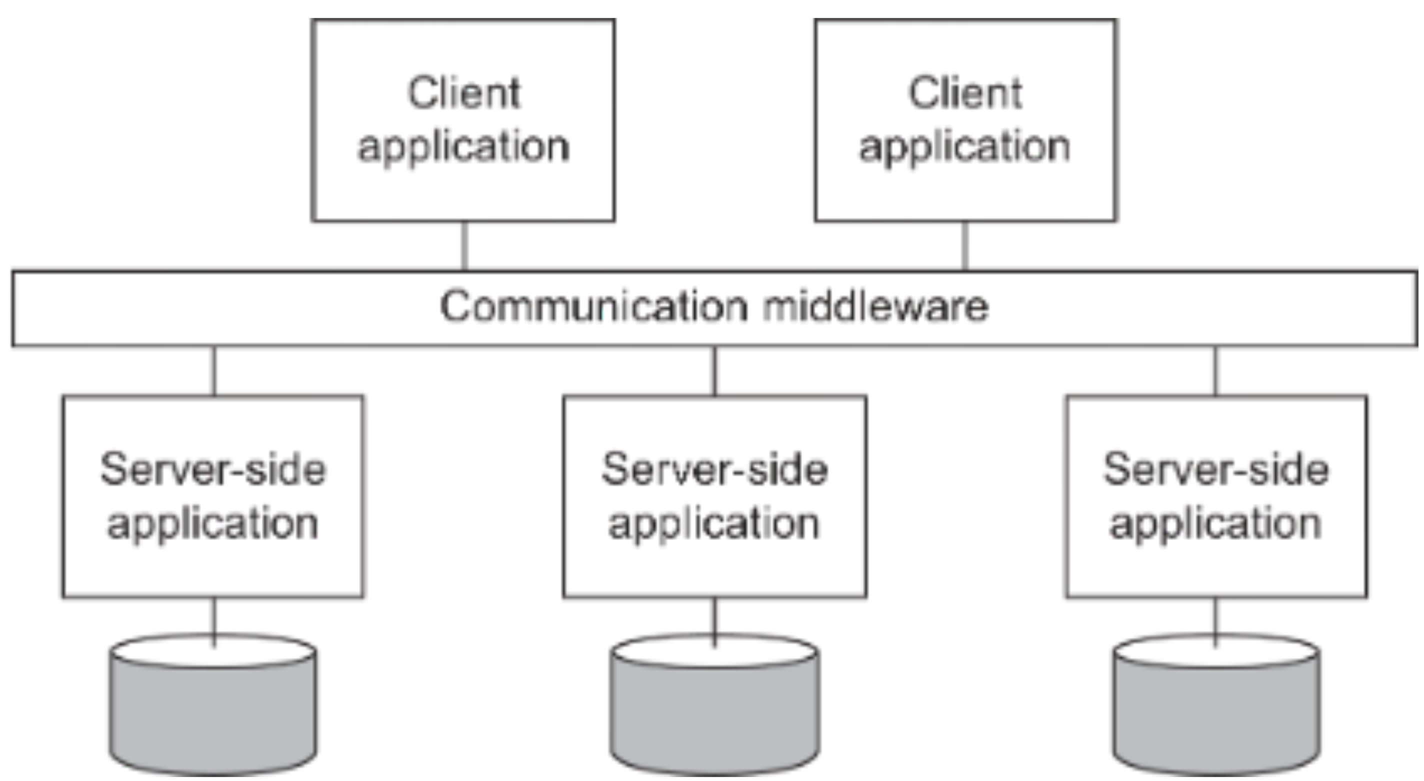
- Transaction Processing Monitor



- Observation: Often, the data involved in a transaction is distributed across several servers. A TP Monitor is responsible for coordinating the execution of a transaction.

Middleware and EAI

- Middleware offers communication facilities for integration
  - Remote Procedure Call (RPC): Requests are sent through local procedure call, packaged as message, processed, responded through message, and result returned as return from call.
  - Message Oriented Middleware (MOM): Messages are sent to logical contact point (published), and forwarded to subscribed applications.



How to integrate

- File transfer: Technically simple, but not flexible:
  - Figure out file format and layout
  - Figure out file management
  - Update propagation, and update notifications.
- Shared database: Much more flexible, but still requires common data scheme next to risk of bottleneck.
- Remote procedure call: Effective when execution of a series of actions is needed.
- Messaging: RPCs require caller and callee to be up and running at the same time. Messaging allows decoupling in time and space.

Distributive Pervasive Systems

- Observation
  - Emerging next-generation of distributed systems in which nodes are small, mobile, and often embedded in a larger system, characterized by the fact that the system naturally blends into the user's environment.
  - Three (overlapping) subtypes
    - Ubiquitous computing systems: pervasive and continuously present, i.e., there is a continuous interaction between system and user.
    - Mobile computing systems: pervasive, but emphasis is on the fact that devices are inherently mobile.
    - Sensor (and actuator) networks: pervasive, with emphasis on the actual (collaborative) sensing and actuation of the environment.

Ubiquitous Systems

- Core elements

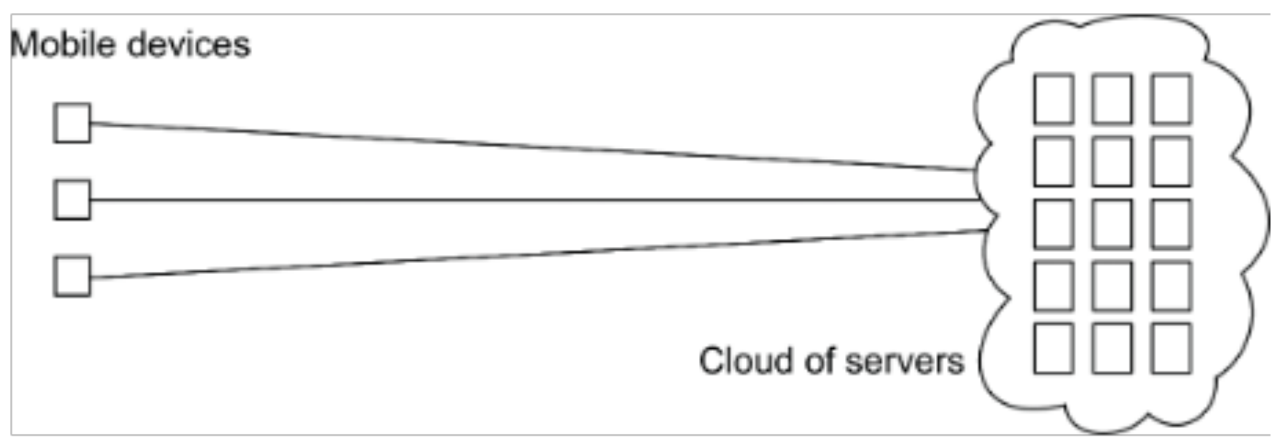
1. (Distribution) Devices are networked, distributed, and accessible transparently
2. (Interaction) Interaction between users and devices is highly unobtrusive
3. (Context awareness) The system is aware of a user's context to optimize interaction
4. (Autonomy) Devices operate autonomously without human intervention, and are thus highly self-managed
5. (Intelligence) The system as a whole can handle a wide range of dynamic actions and interaction.

Mobile Computing

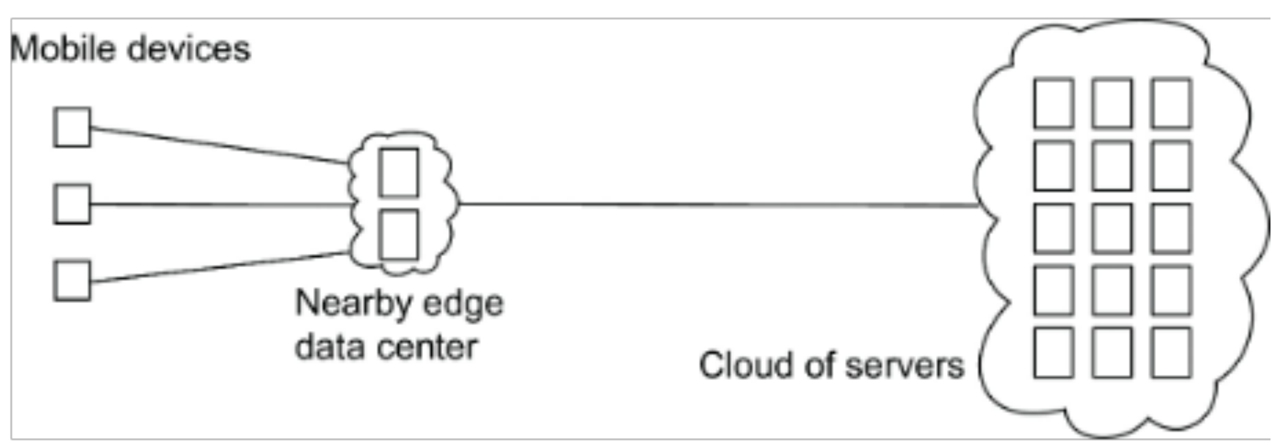
- Distinctive features
  - A myriad of different mobile devices (smartphones, tablets, GPS devices, remote controls, active badges).
  - Mobile implies that a device's location is expected to change over time
    - $\Rightarrow$
    - change of local services, reachability, etc. Keyword: discovery.
- Maintaining stable communication can introduce serious problems.
- For a long time, research has focused on directly sharing resources between mobile devices. It never became popular and is by now considered to be a fruitless path for research.

Mobile Computing

- **Bottomline:** Mobile devices set up connections to stationary servers, essentially bringing mobile computing in the position of clients of cloud-based services.



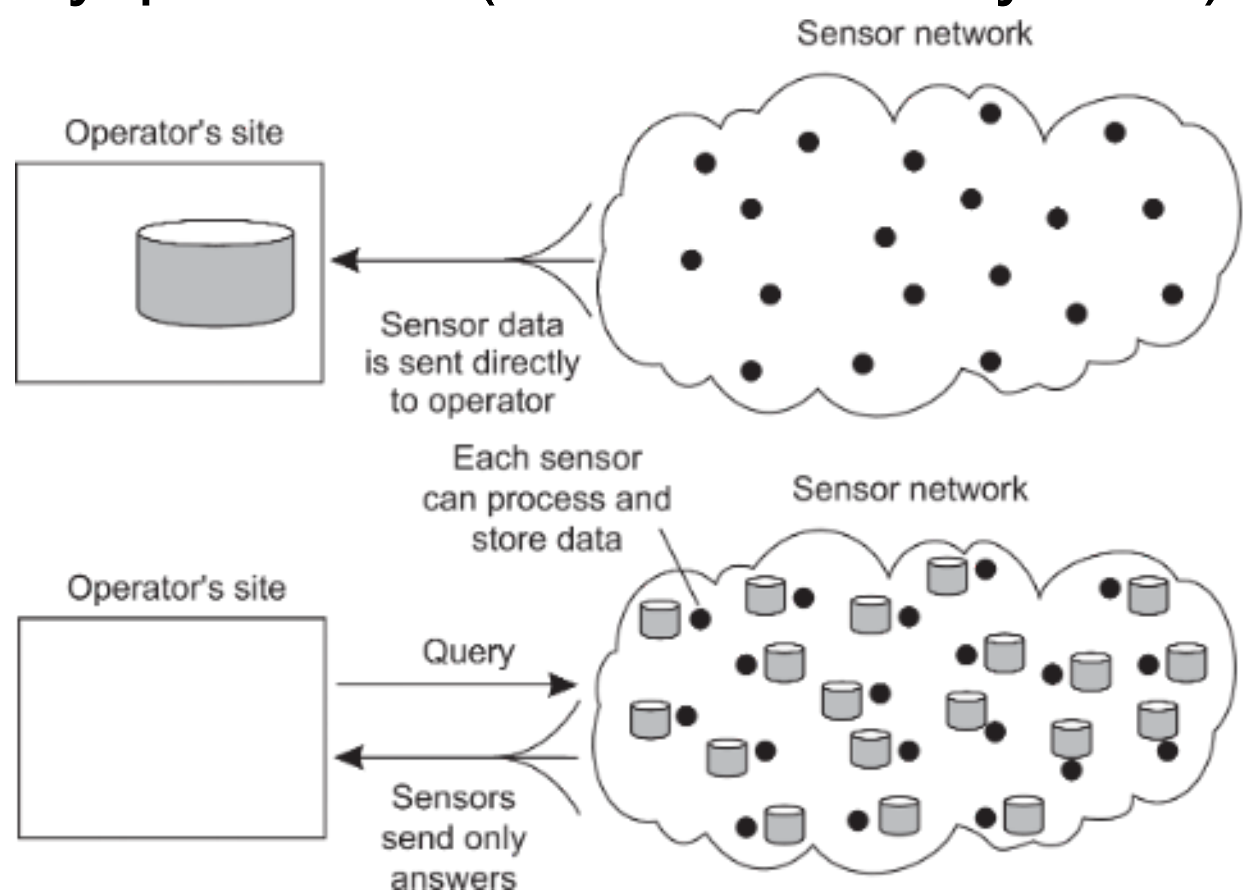
Mobile cloud computing



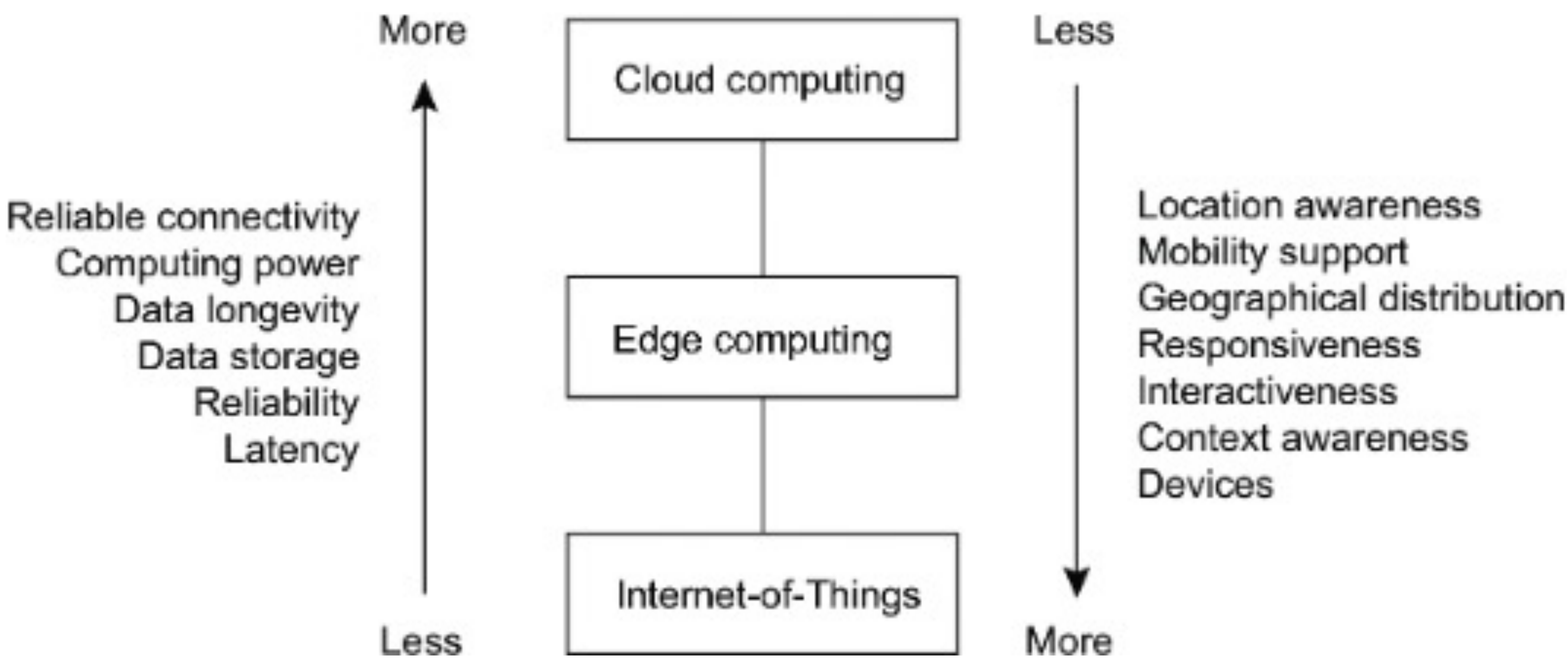
Mobile edge computing

Sensor Networks

- Characteristics: The nodes to which sensors are attached are:
    - Many (10s-1000s)
    - Simple (small memory/compute/communication capacity)
    - Often battery-powered (or even battery-less)



Cloud Edge Continuum



Pitfalls

- Observation: Many distributed systems are needlessly complex, caused by mistakes that required patching later on. Many false assumptions are often made.
- False (and often hidden) assumptions
- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator