Distributed Systems

Summer Term 2020

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: March 12, 2020

Distributed Systems

Summer Term 2020

1 Introduction

Contents

- What makes a distributed system?
- Software architecture
- Architecture models
- Cluster

Literature

- Hammerschall: 1
- Tanenbaum, van Steen: 1
- Colouris, Dollimore, Kindberg: 1, 2
- Stallings: 13.4

1.1 What makes a distributed system?

In a distributed system, components located on different computers work together to coordinate their actions by exchanging messages.

G. Coulouris

A distributed system is a set of independent computers that appear to the user as a single, coherent system.

A. Tanenbaum

A distributed system is a collection of processors that neither share main memory nor a clock.

A. Silberschatz

A distributed system is one on which I can't do any work because some machine I've never heard of has crashed.

L. Lamport

1.1 What makes a distributed system? ...

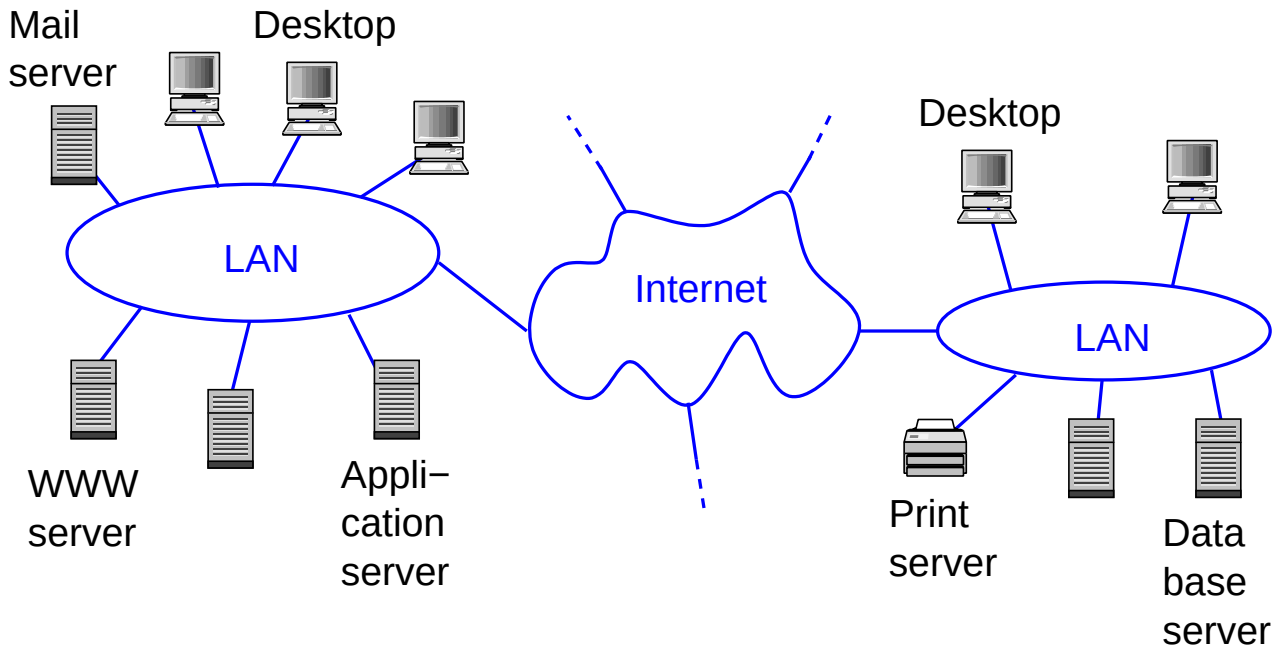
- A distributed system is **a system**
  - in which **hardware and software components** are based on **networked computers**, and
  - communicate and coordinate their actions only via the **exchange of messages**.
- The boundaries of the distributed system are defined by a common application
- Best known example: Internet
  - communication via the standardized Internet protocols
    - IP and TCP / UDP (☞ lecture Computer Networks)
  - users can use services / applications, regardless of the present location

1.1 What makes a distributed system? ...

What is a distributed application?

- Application that uses a distributed system to create a self-contained functionality
- Application logic distributed among several, largely independent components
- Components often executed on different machines
- Examples:
  - simple internet applications (e.g. WWW, FTP, email)
  - distributed information systems (e.g. flight booking)
    - SW intensive, data centered, interactive, highly concurrent
  - distributed embedded systems (e.g. in the car)
  - distributed mobile applications (e.g. for handhelds)

A typical distributed system



Why distribution?

- Central, non-distributed applications are
  - generally safer and more reliable
  - generally more performant
- Main reason for distribution: sharing of resources
  - Hardware resources (printer, scanner, ...)
    - cost saving
  - Data and information (file server, database, ...)
    - information exchange, data consistency
  - Functionality (centralization)
    - error avoidance, reuse

- Resources (e.g. computers, data, users, ...) are distributed
  - sometimes worldwide
- Cooperation via message exchange
- Concurrency
  - but: parallel processing of **a single** request is not the primary goal
- No global clock (more precisely: no global time)
- Distributed status information
  - no uniquely determined global state
- Partial errors are possible (independent failures)

Parallel vs. distributed systems

- Parallel system:
  - motivation: higher performance through parallel execution
  - multiple tasks (processes/threads) working on one job
  - tasks are fine-grained: frequent communication
  - tasks work simultaneously (parallel)
  - homogeneous hardware / OSs, regular network structure
- Distributed system:
  - motivation: distributed resources (computers, data, users)
  - multiple tasks (processes/threads) working on one or many jobs
  - tasks are coarse grained: communication less frequent
  - tasks work synchronized (usually one after the other)
  - inhomogeneous (processors, networks, OSs, ...)

1.3 Challenges and Goals of Distributed Systems

- **Heterogeneity**: computer hardware, networks, OSs, programming languages, implementations by different developers, ...
  - solution: **middleware**
    - software layer that hides heterogeneity by providing a unified programming model
    - e.g. CORBA: distributed objects, remote method invocation
    - e.g. web services: remote procedure calls (services)
- **Openness**: easy extensibility (with new services)
  - requirements:
    - key interfaces are published / standardized
    - uniform communication mechanisms / protocols
    - components must conform to standards

[Coulouris, 1.4]

1.3 Challenges and Goals of Distributed Systems ...

- **Security**
  - information: confidentiality, integrity, availability
    - esp. with mobile code
  - users: authentication, authorization
- **Scalability**: number of resources or users can grow without negative impact on performance and cost
- **Error handling** (partial errors)
  - error detection (e.g. checksums)
  - error masking (e.g. retransmission of a message)
  - error tolerance (e.g. browser: “server not available”)
  - recovery (of data) after errors
  - redundancy (of hardware and software)

Notes for slide 25:

There are the following problems with the realization of scalability:

- Cost control: the system hardware should be extensible; the effort should be (at most) proportional to the number of users.
- Performance loss control: the algorithms used should scale well with the number n of nodes, i.e. with $\mathcal{O}(n \log n)$ or better.
- Prevent exhaustion of software resources: as an example, think of the 32-bit IPv4 addresses.
- Avoid performance bottlenecks: decentralized algorithms without bottlenecks.

Ideally, a system should be able to scale without adapting the application and system software.

Techniques that support scalability include replication and caching.

1.3 Challenges and Goals of Distributed Systems ...

- **Concurrency**
  - synchronization, consistency of replicated data
- **Transparency**
  - access∼: local and remote accesses identical
  - location∼: no need to know the location

  } network∼
  - mobility∼: transparent relocation of resources
  - replication∼: transparent replication of resources
  - concurrency∼: shared use of resources without disruptions
  - error∼: hiding errors due to component failure
  - performance∼: performance is largely independent of the load
  - scaling∼: system scales without negative impact on users

Notes for slide 26:

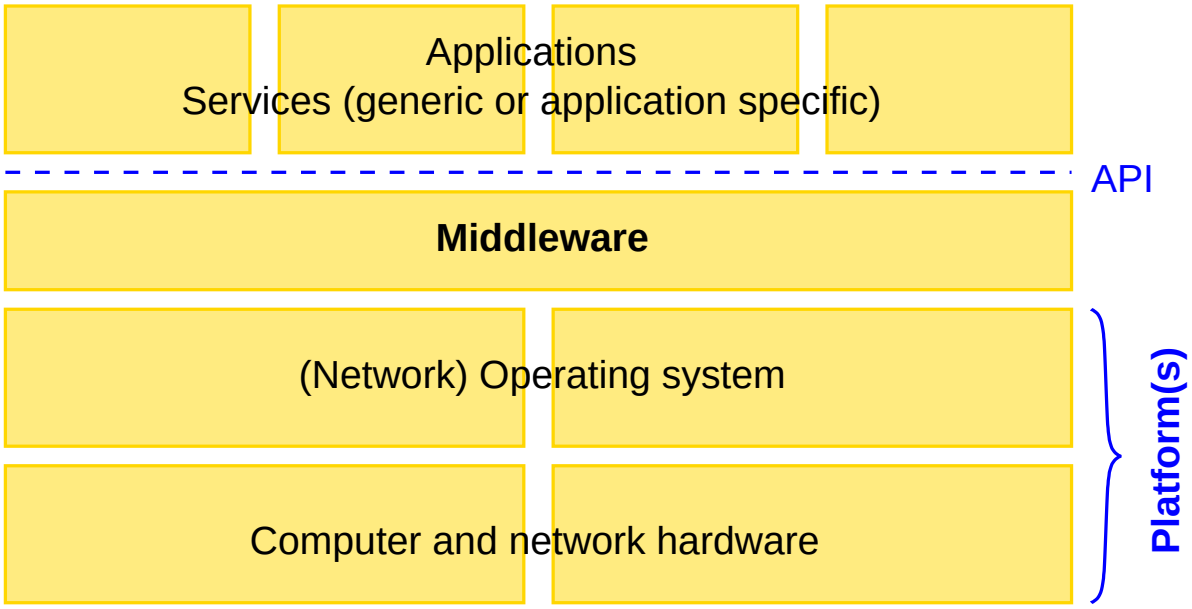
The concurrency transparency corresponds to the concept of isolation in the context of database systems.

1.4 Software Architecture

Types of Operating Systems for Distributed Systems

- Network operating system:
  - traditional OS, extended by support for network applications (API for sockets, RPC, ...)
  - each computer has its own OS, but can use services of other computers (file system, email, `ssh`, ...)
  - the existence of the other computers is visible
- Distributed operating system:
  - uniform OS for a network of computers
  - transparent for the user
  - requires cooperation of the OS kernels
  - so far mainly research projects

Typical layers in a distributed system



[Coulouris, 2.2.1]

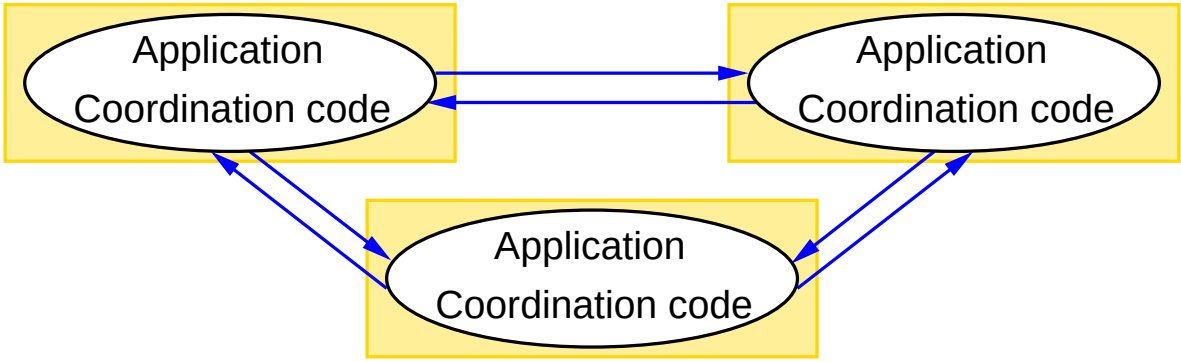
Middleware

- Tasks:
  - hiding of distribution and heterogeneity
  - providing a common programming model / API
  - provision of general services
- Functions e.g:
  - communication services: remote method calls, group communication, event notifications
  - replication of shared data
  - security services
- Examples: CORBA, EJB, .NET, Axis2 (Web Services), ...
  (☞ Lecture Client/Server Programming)

- An architecture model characterizes:
  - roles of an application component within the distributed application
  - relationships between application components
- Role defined by the type of process the component is running in:
  - client process
    - short-lived (for the duration of use by the user)
    - acts as initiator of interprocess communication (IPC)
  - server process
    - lives 'unlimited'
    - acts as a service provider for an IPC
  - peer process
    - short-lived (for the duration of use by the user)
    - acts as initiator and service provider

Peer-to-Peer Model

- Collaboration of peer processes for a distributed activity
  - each process manages a local part of the resources
  - distributed coordination and synchronization of actions at application level

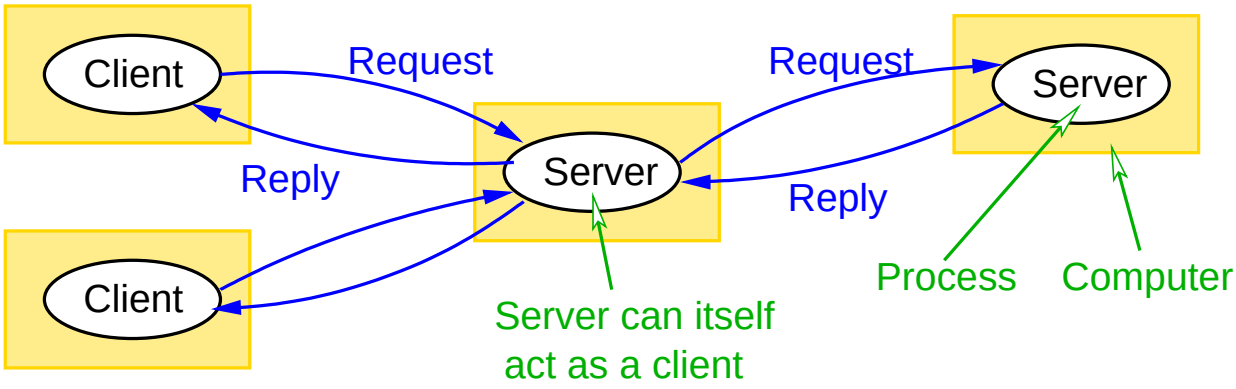


- E.g.: file sharing applications, routers, video conferences, ...

[Coulouris, 2.2.2]

Client/Server Model

- Asymmetric model: Servers provide services that can be used by (multiple) clients.
  - servers usually manage resources (centralized)

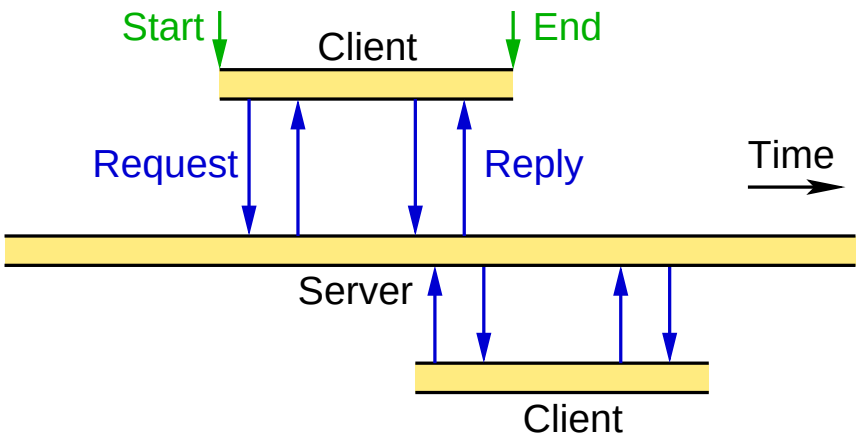


- Most common model for distributed applications (ca. 80 %)

(Animated slide)

Client/Server Model ...

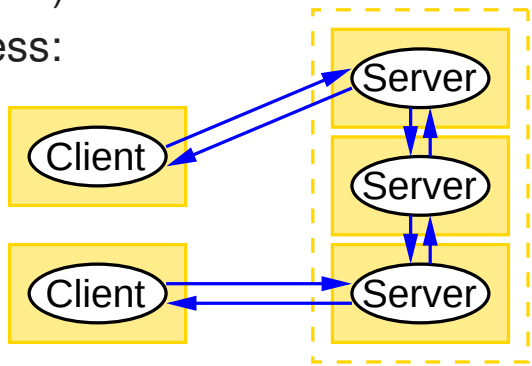
- Usually concurrent requests from several client processes to the server process



- Examples: file server, web server, database server, DNS server, ...

Variants of the client/server model

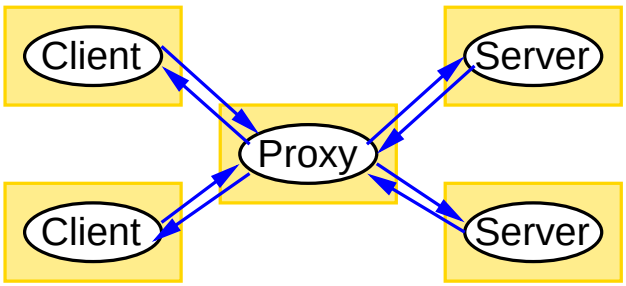
- Cooperating servers
  - Network of servers transparently processes a request
  - Example: Domain Name Server (DNS)
    - if server cannot determine address: request is transparently forwarded to another server



- Replicated servers
  - replicas of server processes are provided
    - transparent replicas (often in clusters)
      - requests are automatically distributed to the servers
    - public replicas (e.g. mirror servers)
  - goals: better performance, reliability

Variants of the client/server model ...

- Proxy-Server / Caches
  - proxy is a delegate for the server
  - task often is caching of data / results
  - e.g. web proxy



- Mobile code
  - executable server code migrates to client on request
  - code is executed by the client
  - best-known example: JavaScript / Java applets in the WWW
- Mobile agents
  - agent contains code and data, moves through the network and performs actions on local resources

n-Tier Architectures

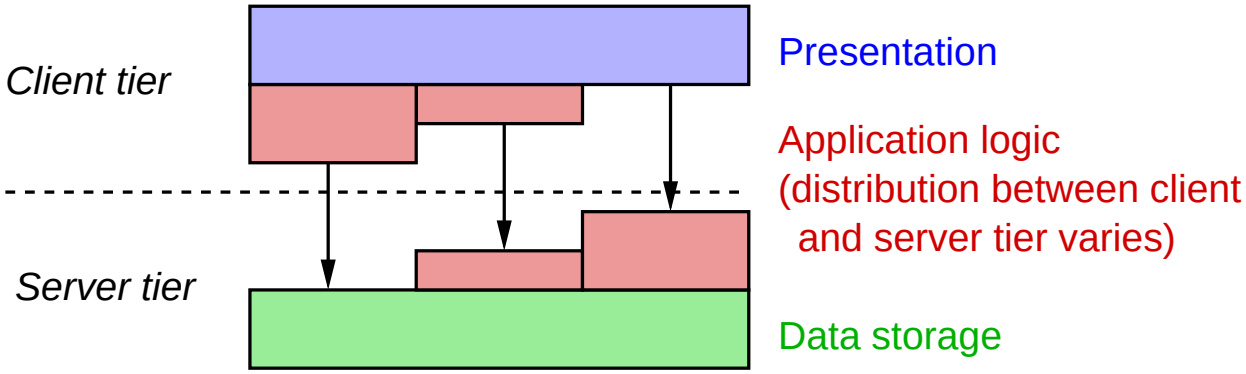
- Refinements of Client/Server Architecture
- Models for distributing an application to the nodes of a distributed system
- Mainly used in information systems
- **Tier** (german: Schicht / Stufe) denotes an independent process space within a distributed application
  - process space can, but does not have to, correspond to a physical host
  - several process spaces on one computer are possible

The Tier Model

- Typical tasks in an information system:
  - presentation – interface to the user
  - application logic – actual functionality
  - data storage – storage of data in a database
- The tier model determines:
  - assignment of tasks to application components
  - distribution of application components on tiers
- Architectures:
  - 2-tier architectures
  - 3-tier architectures
  - 4-or-more-tier architectures

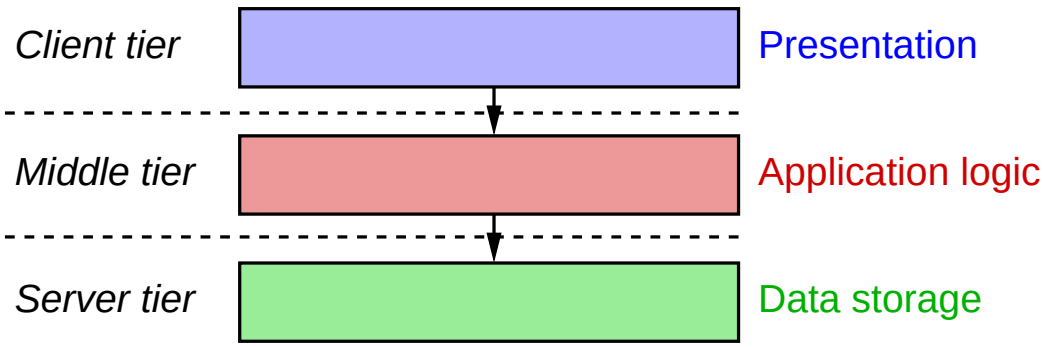
2-Tier Architecture

- Client and server tier
- No own tier for the application logic



- Advantage: simple, high performance
- Disadvantage: difficult to maintain, poorly scalable

3-Tier Architecture



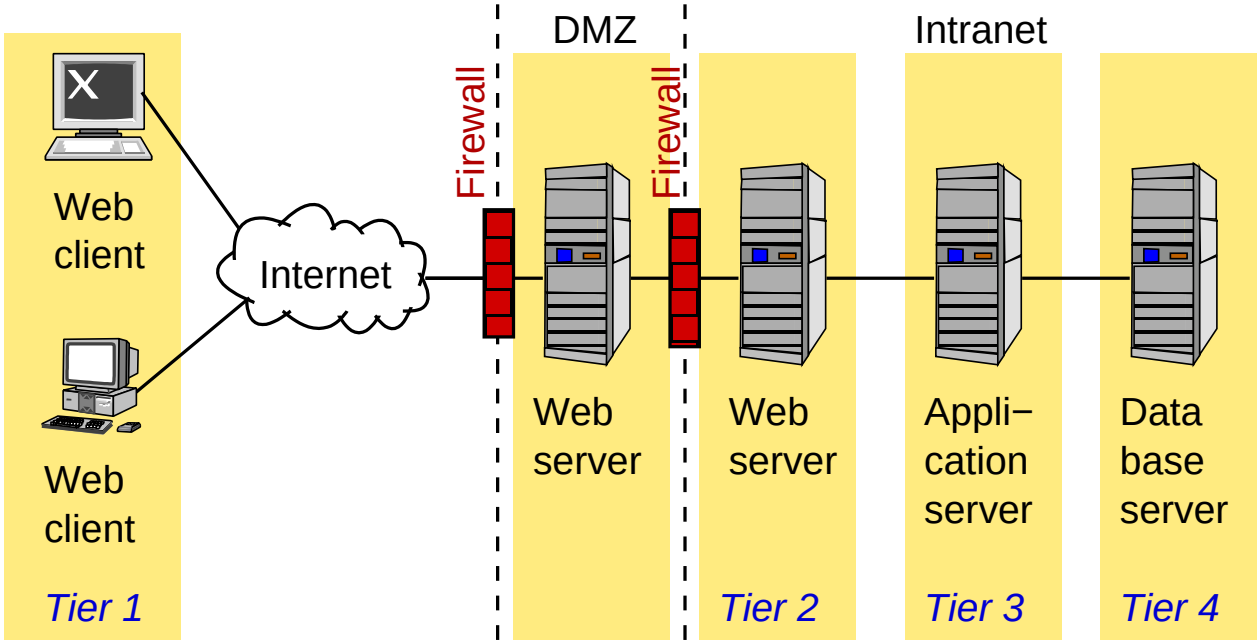
- Standard distribution model for simple web applications:
  - client tier: web browser for display
  - middle tier: web server with servlets / JSP / ASP
  - server tier: database server
- Advantages: Application logic centrally administrable, scalable

4-or-more-Tier Architectures

- Difference to 3-tier architecture:
  - application logic distributed across multiple tiers
- Motivation:
  - minimization of complexity (divide and conquer)
  - better protection of individual application parts
  - reusability of components
- Many distributed information systems have 4-or-more-tier architectures

(Animated slide)

Example: Typical Internet Application



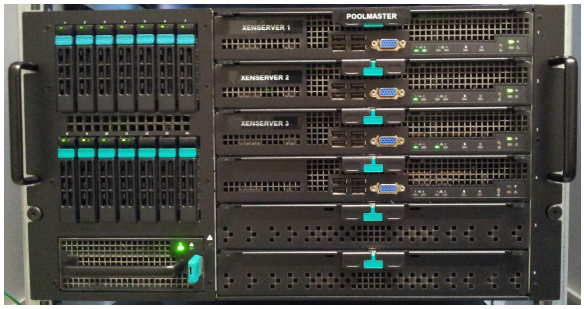
Thin and fat clients

- Characterizes complexity of the application component on the client tier
- Ultra-thin client
  - client tier only for presentation: pure display of dialogs
  - presentation component: web browser
  - only possible with 3-or-more-tier architectures
- Thin client
  - client tier for presentation only: display of dialogs, preparation of data for display
- Fat client
  - parts of the application logic on the client tier
  - usually with 2-tier architectures

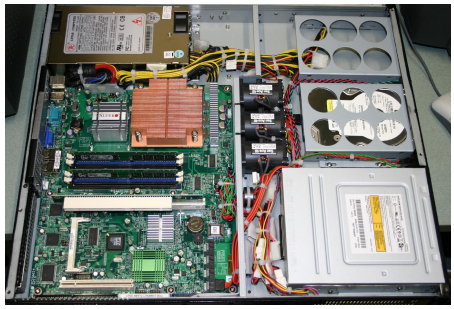
Distinction from Enterprise Application Integration (EAI)

- EAI: integration of different applications
  - communication, exchange of data
- Goals similar to distributed applications / middleware
  - middleware is often used for EAI as well
- Differences:
  - distributed applications: application components, high degree of coupling, usually little heterogeneity
  - EAI: complete applications, low degree of coupling, mostly great heterogeneity (different technologies, systems, programming languages, ...)

1.6 Cluster



- Cluster: group of networked computers that acts as a unified computing resource
  - i.e. multicomputer system
  - nodes usually standard PCs or blade server
- Application mainly as high performance server
- Motivation:
  - (step-by-step) scalability
  - high availability
  - good price/performance ratio



[Stallings, 13.4]

1.6 Cluster ...

Uses for Clusters

- High availability (HA) clusters
  - improved reliability
  - when a node is faulty: services are migrated to other nodes (failover)
- Load balancing cluster
  - incoming requests are distributed to different nodes of the cluster
    - usually by a (redundant) central instance
  - frequently with WWW or email servers
- High performance computing cluster
  - cluster as parallel computer

Cluster configurations

- Passive standby (no actual cluster)
  - processing of all requests by primary server
  - secondary server takes over tasks (only) in case of failure
- Active standby
  - all servers process requests
  - enables load balancing and improved reliability
  - problem: access to data of other / failed server
  - alternatives:
    - replication of data (a lot of communication)
    - shared hard disk system (usually mirrored disks or RAID system for fail-safe operation)

Active Standby Configurations

- Separate servers with data replication
  - separate disks, data is continuously copied to secondary servers
- Server with shared hard disks
  - shared nothing cluster
    - separate partitions for each server
    - in case of server failure: reconfiguration of the partitions
  - shared disc cluster
    - simultaneous use by all servers
    - requires lock manager software to lock files or records

1.7 Summary

- Distributed system
  - HW and SW components on networked computers
  - no shared memory, no global time
  - motivation: use of distributed resources
- Challenges
  - heterogeneity, openness, security, scalability
  - error handling, concurrency, transparency
- Software architecture: middleware
- Architectural models:
  - peer-to-peer, client/server
  - n-tier models
- Cluster: high availability, load balancing