

Chapter 2

Distributed Information Systems Architecture



Middleware for Heterogenous and Distributed Information Systems - WS05/06

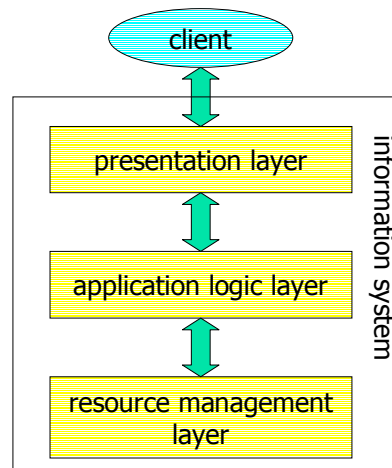
Chapter Outline

- Layers of an information system
 - presentation
 - application logic
 - resource management
- Design strategies
 - top-down, bottom-up
- Architectures
 - 1-tier, 2-tier, 3-tier, n-tier
- Distribution alternatives
- Communication
 - synchronous, asynchronous



Layers of an Information System

- Separation of functionality into three **conceptual** layers
 - presentation
 - interacts with client
 - present information
 - accept requests
 - graphical user interface, or module that formats/transforms data, or ...
 - application logic
 - programs that implement the services offered by the IS
 - often retrieves/modifies data
 - resource management
 - manages the data sources of the IS
 - DBMSs
 - file system
 - any "external" system
- In an IS implementation, these layers might not be clearly distinguishable



Top-Down Information System Design

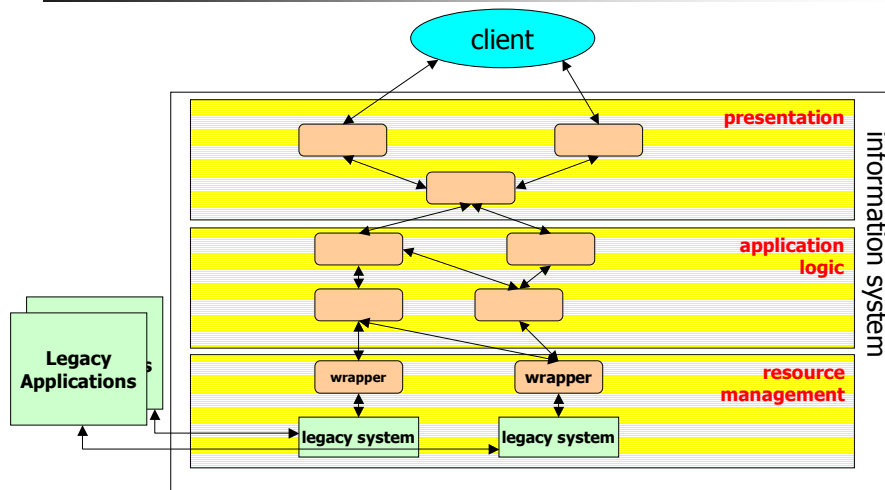
- Steps
 - 1) define access channels and client platforms
 - 2) define presentation formats and protocols
 - 3) define functionality (application logic) necessary to deliver the content and formats
 - 4) define the data sources and data organization needed
- Design involves specification of system distribution across different computing nodes
 - distribution possible at every layer
 - homogenous environment, **tightly-coupled** components
- **Pro:** focus on high-level goals, addresses both functional and non-functional requirements
- **Con:** can only be applied if IS is developed from scratch

Bottom-up Information System Design

- Steps
 - 1) define access channels and client platforms
 - 2) examine existing resources and their functionality (RM layer)
 - 3) wrap existing resources, integrate them into consistent interface (AL layer)
 - 4) adapt output of AL for client (P layer)
- Design focuses on integration/reuse of existing (legacy) systems/applications
 - functionality of components is already (pre-)defined
 - modification or re-implementation is often not a choice
 - driven by characteristics of lower layers
 - start with high-level goals, then determine how it can be achieved using existing components
 - often starts with thorough analysis of existing applications and systems to determine which high-level objectives can be achieved
 - results in **loosely-coupled** systems
 - components can mostly be used stand-alone
 - underlying systems often remain autonomous
- Not an advantage, but a necessity



Bottom-Up Design Example



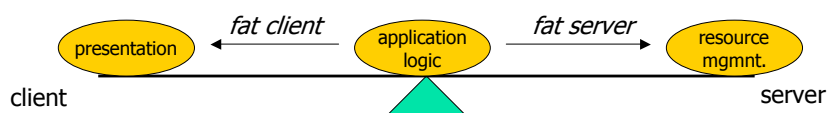
1-Tier Architecture

- All layers are combined in a single tier
- Predominant on mainframe-based computer architectures
 - client is usually a "dumb terminal"
 - focus on efficient utilization of CPU, system resources
- "Monolithic" system
 - no entry points (APIs) from outside, other than the channel to the dumb terminals
 - have to be treated as black boxes
 - integration requires "screen scraping"
 - program that simulates user, parses the "screens" produced by the system
 - the prototype of a legacy system
- Advantages
 - optimizes performance by merging the layers as necessary
 - client development, deployment, maintenance is not an issue
- Disadvantages
 - difficult and expensive to maintain
 - further increased by lack of documentation and qualified programmers



2-Tier Architecture

- Pushed by emergence of PC, workstations (replacing dumb terminals)
 - (parts of the) presentation layer is moved to the PC
 - exploit the processing power of PC
 - free up resources for application logic/resource management layers
 - possibility to tailor presentation layer for different purposes
 - e.g., end-user presentation vs. administrator presentation modules
 - typically realized as client/server system
 - one (popular) approach: client corresponds to presentation layer, server includes the application logic and resource management layers
 - another approach (more traditional C/S): client includes presentation and application logic layer, server provides resource management services
 - where does the client end and the server begin?
 - thin client/fat server vs. fat client/thin server

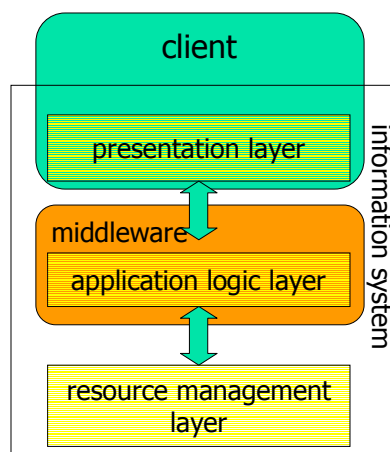


Properties of 2-Tier Architecture

- Pro
 - emphasis on "services" provided by server, requested consumed by client
 - definition of application programming interfaces (APIs) as published server interfaces
 - portability, stability
 - multiple types of clients can utilize the same server API
 - server can support multiple clients at the same time
 - sufficient scalability for departmental applications
- Con
 - scalability is often limited (esp. for thin clients)
 - requires to move to very powerful server machines
 - especially fat clients require increased software maintenance/deployment on client side
 - client is often turned into an integration engine interacting with multiple types of servers
 - extra application layer appears in thin clients

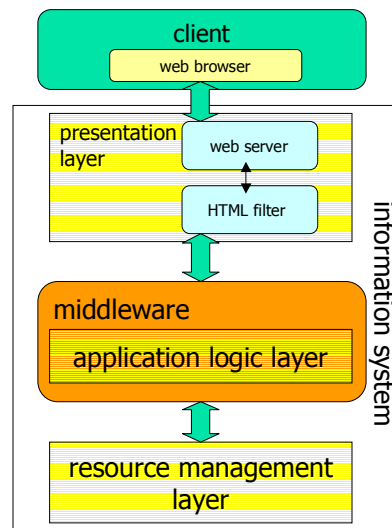
3-Tier Architecture

- Usually based on a clear separation between the three layers
 - client tier implements presentation layer
 - middle tier realizes application logic
 - employs middleware
 - resource management layer composed of a (set of) servers (e.g., DBS)
- Addresses scalability
 - application layer can be distributed across nodes (in a cluster)
- Portability of application logic
- Supports integration of multiple resource managers
- Disadvantages
 - increased communication



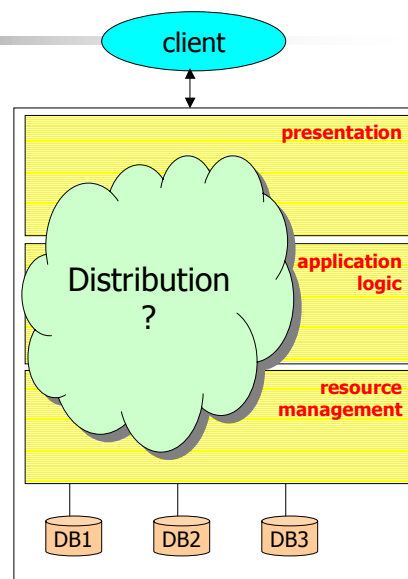
N-Tier Architecture

- Further generalizes 3-tier architecture
 - Resource layer may include 1-, 2-, 3-, N-tiered systems
 - focus on linking, integration of different systems
 - Presentation layer may be realized in separate tiers
 - especially important for supporting internet connectivity
 - client using browser
 - server-side presentation done by web server, dynamic HTML generation (HTML filter)
 - usually results in 4-tier architecture
- Going from N to N+1 in general
 - adds flexibility, functionality, distribution options
 - introduces performance, complexity, management, tuning issues



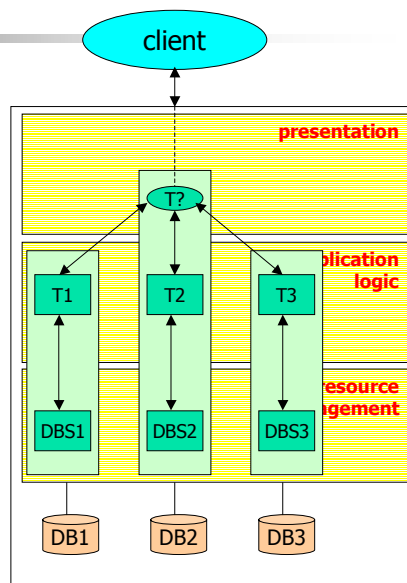
Distributed IS

- Why distribution?
 - economic reasons
 - e.g., reduced hardware cost
 - organizational reasons
 - local support of org. structures
 - integration of existing (legacy) data sources or application systems
 - local autonomy
 - technical reasons
 - increase performance (locality of processing, exploit parallelism)
 - high availability and reliability (replication)
 - scalability
- Client view
 - distribution transparency
 - single system image
- Different realization alternatives
 - often used in combination



Alternative 1

- Transaction as the unit of distribution
 - transaction routing
 - only local transaction processing (within a node)
 - no cooperation among DBMS
- Pros
 - simple solution, easy to support
 - works in heterogeneous environments (e.g., with HTTP)
- Cons
 - inflexible, limited scope
 - transactions restricted to single node (i.e., no distributed transactions)



Middleware for Heterogenous and Distributed Information Systems - WS05/06

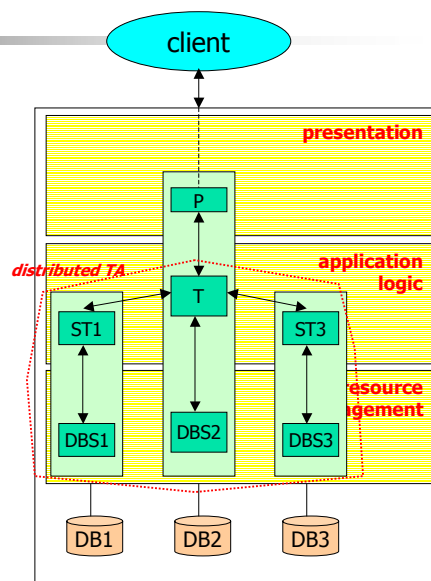


© Prof. Dr.-Ing. Stefan Dießloch

13

Alternative 2

- Application program/component as the unit of distribution
 - invocation of (remote) program components through RPC/RMI-based mechanisms
 - RPC, CORBA/EJB-RMI, Stored Procedures, ...
 - "programmed" distribution
 - middleware can help to achieve location transparency
 - each program (component) accesses local DB only
 - distributed transaction processing
 - coordinated by TP-monitor/application server
 - supported by (local) application server and DBMSs
- Pros
 - locality of processing (low communication overhead)
 - supports application reuse, heterogeneous data sources
- Cons
 - inflexibility regarding data access operations
 - potential programming model complexity (distribution, error handling, ...)
 - DB access operation cannot reach across multiple nodes



Middleware for Heterogenous and Distributed Information Systems - WS05/06

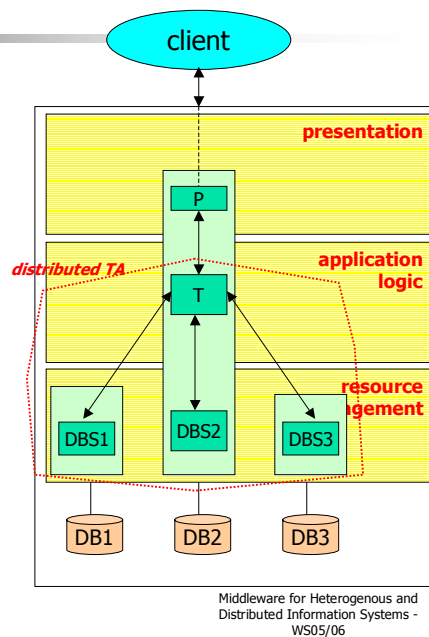


© Prof. Dr.-Ing. Stefan Dießloch

14

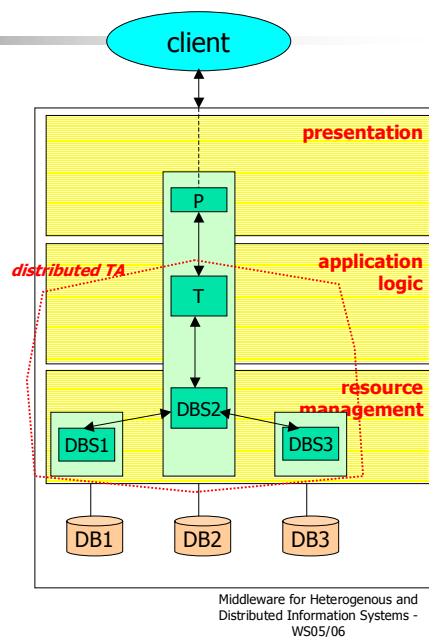
Alternative 3

- DB operation as the unit of distribution
 - Application can access remote data sources
 - function request shipping, data access services
 - (proprietary) DBMS client software
 - DB-gateways
 - Programmer aware of multiple databases
 - multiple schemas
 - each DB operation restricted to a single DB/schema
 - Distributed transaction processing
 - similar to alternative 2
- Pros
 - high flexibility for data access
- Cons
 - potentially increased communication overhead
 - programming model complexity
 - multiple DBs, schemas
 - heterogeneity of data sources, access APIs, ...



Alternative 4

- Distribution controlled by DBMS (middleware)
 - single logical DB and DB-schema for application programmer
 - distributed transaction processing
 - see alternatives 2 and 3
 - DB-operation may span across multiple data sources
 - homogeneous: parallel and distributed DBMS
 - DB-partitioning, DB-sharing
 - heterogeneous: federated DBMS middleware
 - schema integration
 - wrapper-based cooperation, governed by federated DBMS server

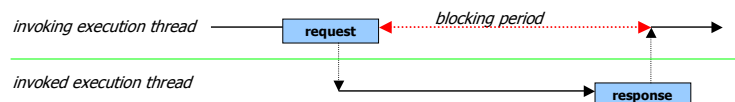


Communication in an Information System

- Blocking and non-blocking interactions
 - "synchronous" and "asynchronous" are accepted synonyms in our context
 - formal definition of synchronous involves additional aspects (transmission time), which we are ignoring here
 - interactions is
 - synchronous/blocking, if the involved parties must wait for interaction to conclude before doing anything else
 - asynchronous/non-blocking, otherwise

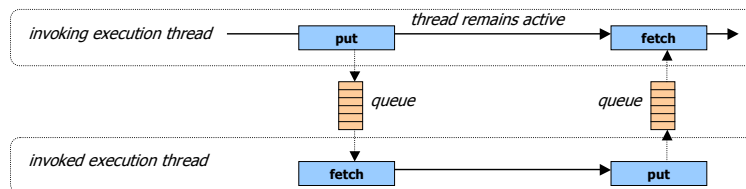
Synchronous or Blocking Calls

- Thread of execution at the requestor side must wait until response comes back
- Advantage: Easier to understand for the programmer
 - state of calling thread will not change before response comes back
 - code for invoking a service and processing the response are next to each other
- Disadvantage: Calling thread must wait, even if a response is not needed (right away) for further processing steps
 - waste of time, resources
 - blocking process may be swapped out of memory
 - running out of available connections
 - tight coupling of components/tiers
 - fault tolerance: both parties must be online, work properly for the entire duration of call
 - system maintenance: server maintenance forces client downtime



Asynchronous or Non-Blocking Calls

- Thread of execution at requestor side is not blocked
 - can continue working to perform other tasks
 - check for a response message at a later point, if needed
- Message queues
 - intermediate storage for messages until receiver is ready to retrieve them
 - more detail: chapters on message-oriented middleware
- Can be used in request-response interactions
 - requester "actively waits"
 - handle load peaks
- Supports other types of interaction
 - information dissemination, publish/subscribe



Middleware

- Middleware
 - supports the development, deployment, and execution of complex information systems
 - facilitates **interaction** between and **integration** of applications across multiple distributed, heterogenous platforms and data sources
- Wide range of middleware, at every IS layer
 - integrating databases on a LAN
 - integrating complete 3-tier systems within a company
 - linking business partners across company boundaries
 - ...

Two major aspects

- Middleware as a programming abstraction
 - hide complexities of building IS
 - distribution
 - communication
 - data access, persistence
 - error/failure handling
 - transaction support
- Middleware as infrastructure
 - realizes complex software infrastructure that implements programming abstractions
 - development
 - deployment
 - code generation, application "assembly"
 - runtime execution

