ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

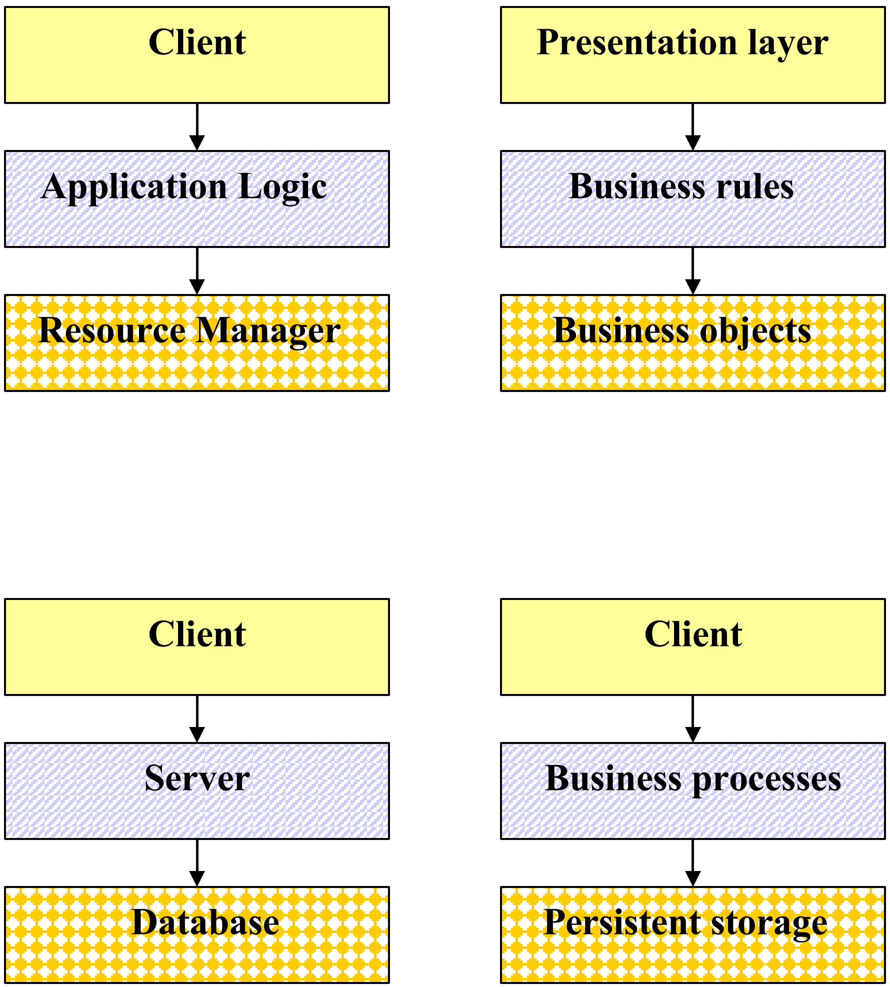
Chapter 1:
Distributed Information Systems

Gustavo Alonso
Computer Science Department
Swiss Federal Institute of Technology (ETHZ)
alonso@inf.ethz.ch
http://www.iks.inf.ethz.ch/

Contents - Chapter 1

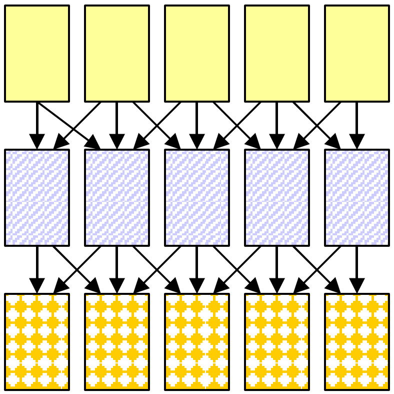
- Design of an information system
  - Layers and tiers
  - Bottom up design
  - Top down design
- Architecture of an information system
  - One tier
  - Two tier (client/server)
  - Three tier (middleware)
  - N-tier architectures
  - Clusters and tier distribution
- Communication in an information system
  - Blocking or synchronous interactions
  - Non-blocking or asynchronous interactions

Layers and tiers



- Client is any user or program that wants to perform an operation over the system. Clients interact with the system through a presentation layer
- The application logic determines what the system actually does. It takes care of enforcing the business rules and establish the business processes. The application logic can take many forms: programs, constraints, business processes, etc.
- The resource manager deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application logic. This is typically a database but it can also be a text retrieval system or any other data management system providing querying capabilities and persistence.

A game of boxes and arrows

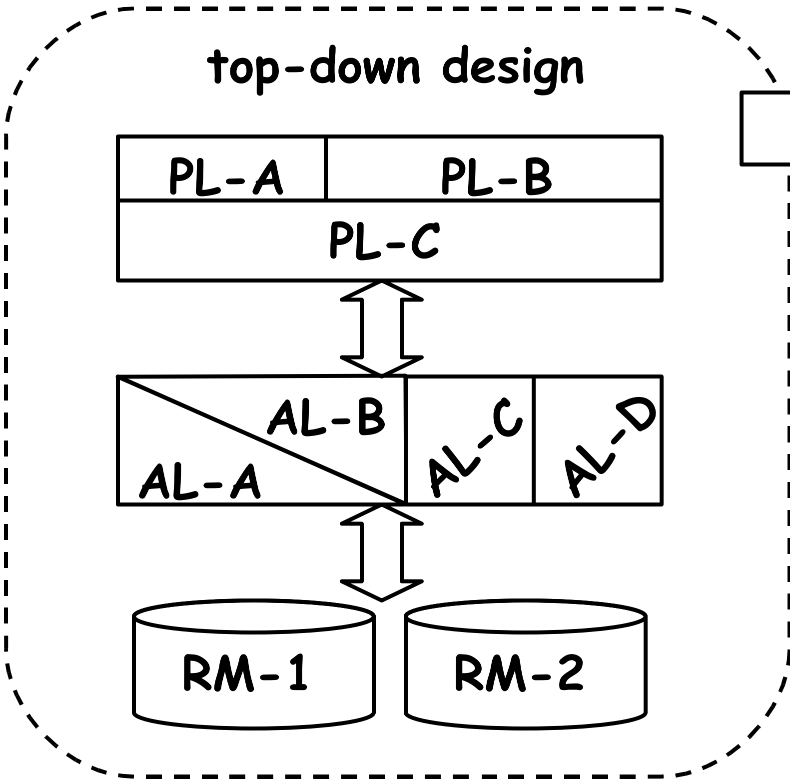


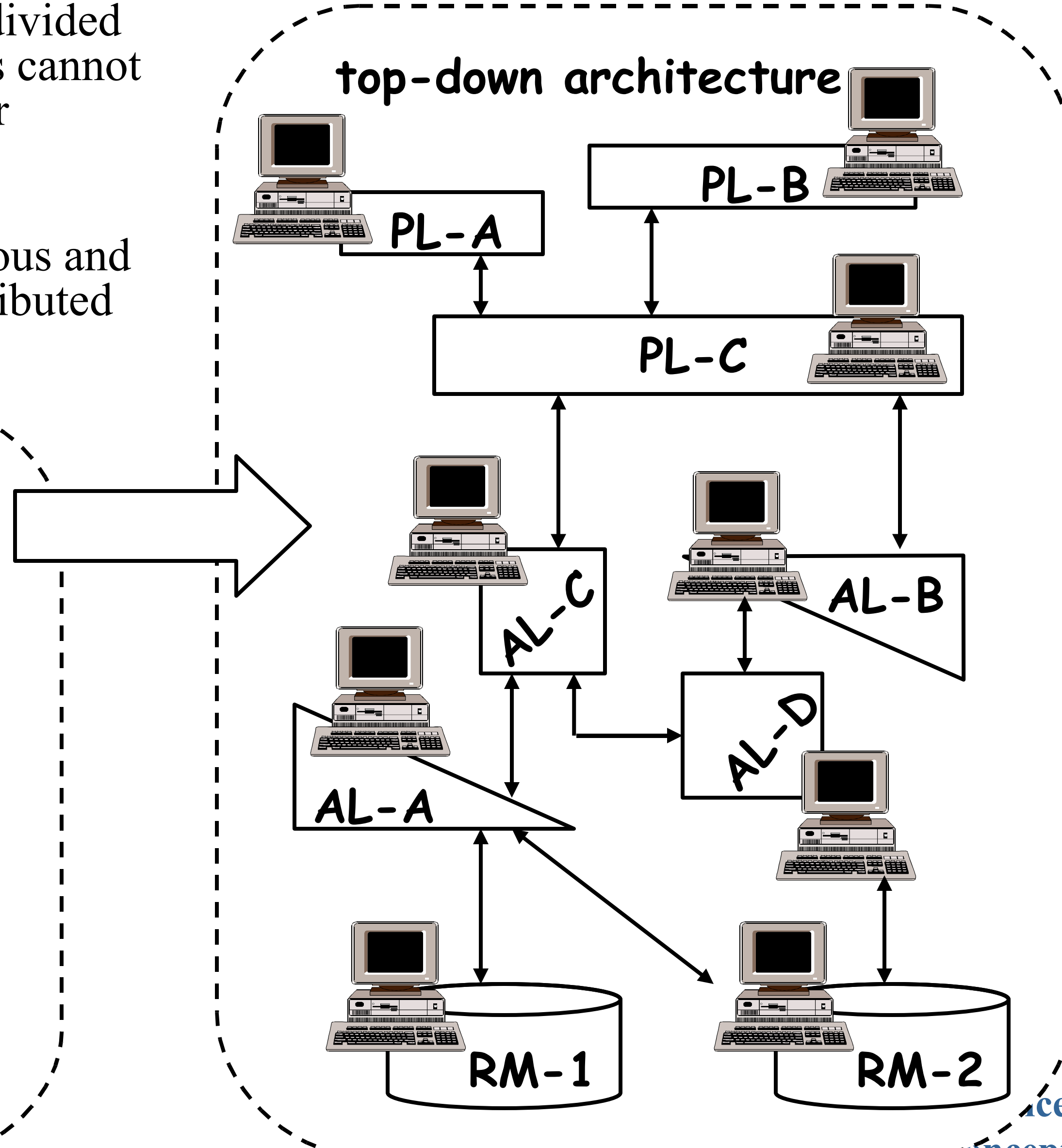
There is no problem in system design that cannot be solved by adding a level of indirection. There is no performance problem that cannot be solved by removing a level of indirection.

- Each box represents a part of the system.
- Each arrow represents a connection between two parts of the system.
- The more boxes, the more modular the system: more opportunities for distribution and parallelism. This allows encapsulation, component based design, reuse.
- The more boxes, the more arrows: more sessions (connections) need to be maintained, more coordination is necessary. The system becomes more complex to monitor and manage.
- The more boxes, the greater the number of context switches and intermediate steps to go through before one gets to the data. Performance suffers considerably.
- System designers try to balance the flexibility of modular design with the performance demands of real applications. Once a layer is established, it tends to migrate down and merge with lower layers.

Top down design

- The functionality of a system is divided among several modules. Modules cannot act as a separate component, their functionality depends on the functionality of other modules.
- Hardware is typically homogeneous and the system is designed to be distributed from the beginning.

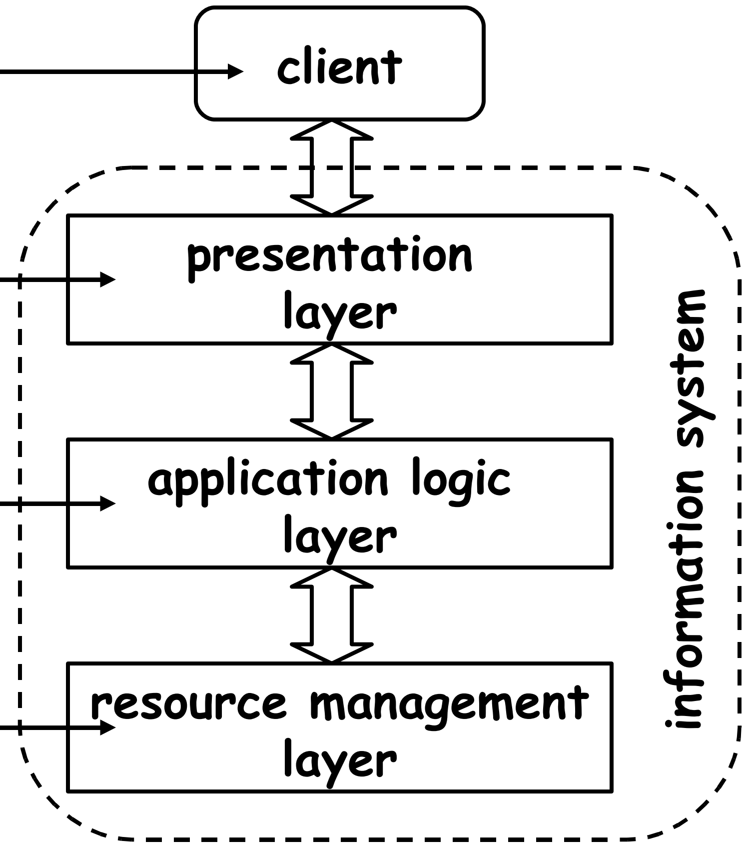




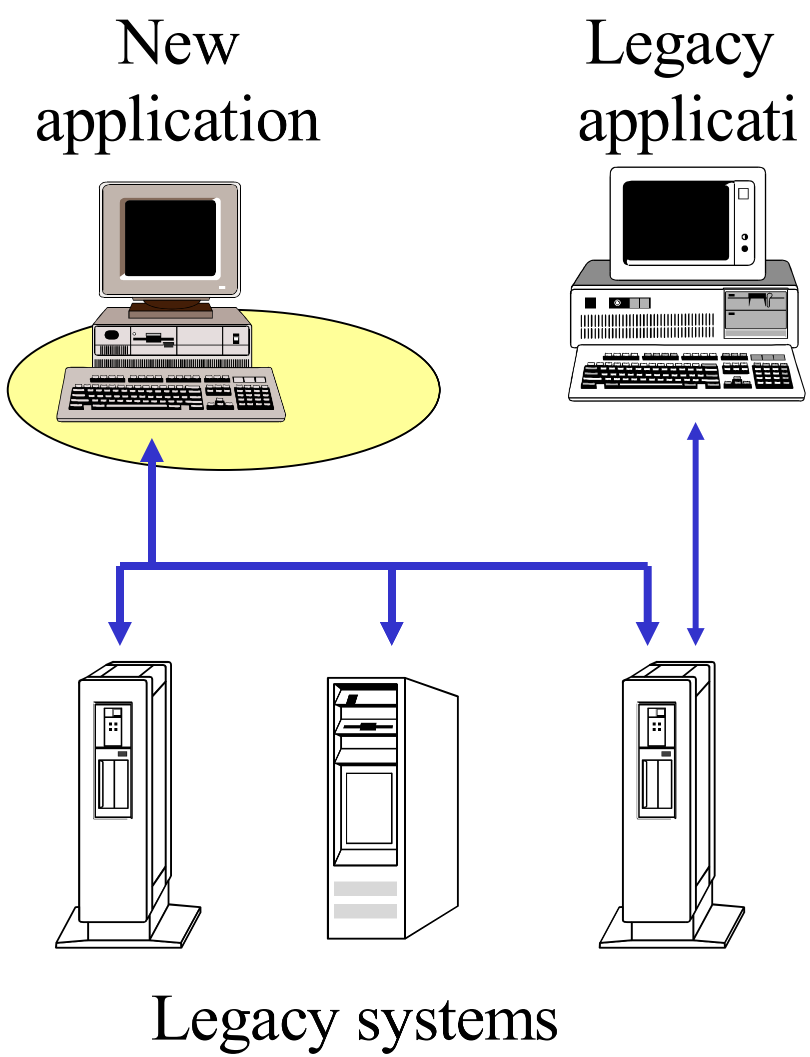
Top down design

top-down design

1. define access channels and client platforms
2. define presentation formats and protocols for the selected clients and protocols
3. define the functionality necessary to deliver the contents and formats needed at the presentation layer
4. define the data sources and data organization needed to implement the application logic

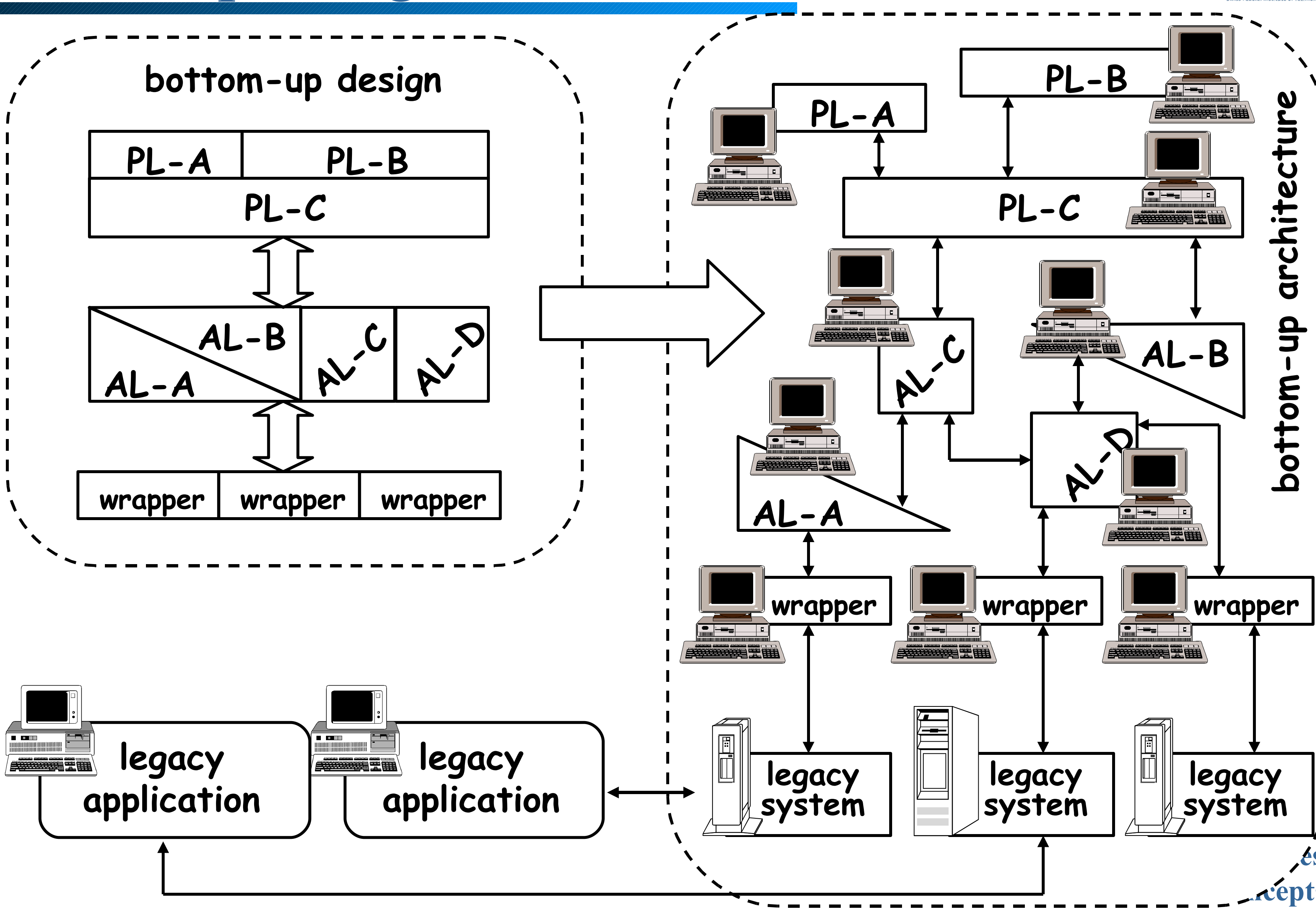


Bottom up design



- In a bottom up design, many of the basic components already exist. These are stand alone systems which need to be integrated into new systems.
- The components do not necessarily cease to work as stand alone components. Often old applications continue running at the same time as new applications.
- This approach has a wide application because the underlying systems already exist and cannot be easily replaced.
- Much of the work and products in this area are related to middleware, the intermediate layer used to provide a common interface, bridge heterogeneity, and cope with distribution.

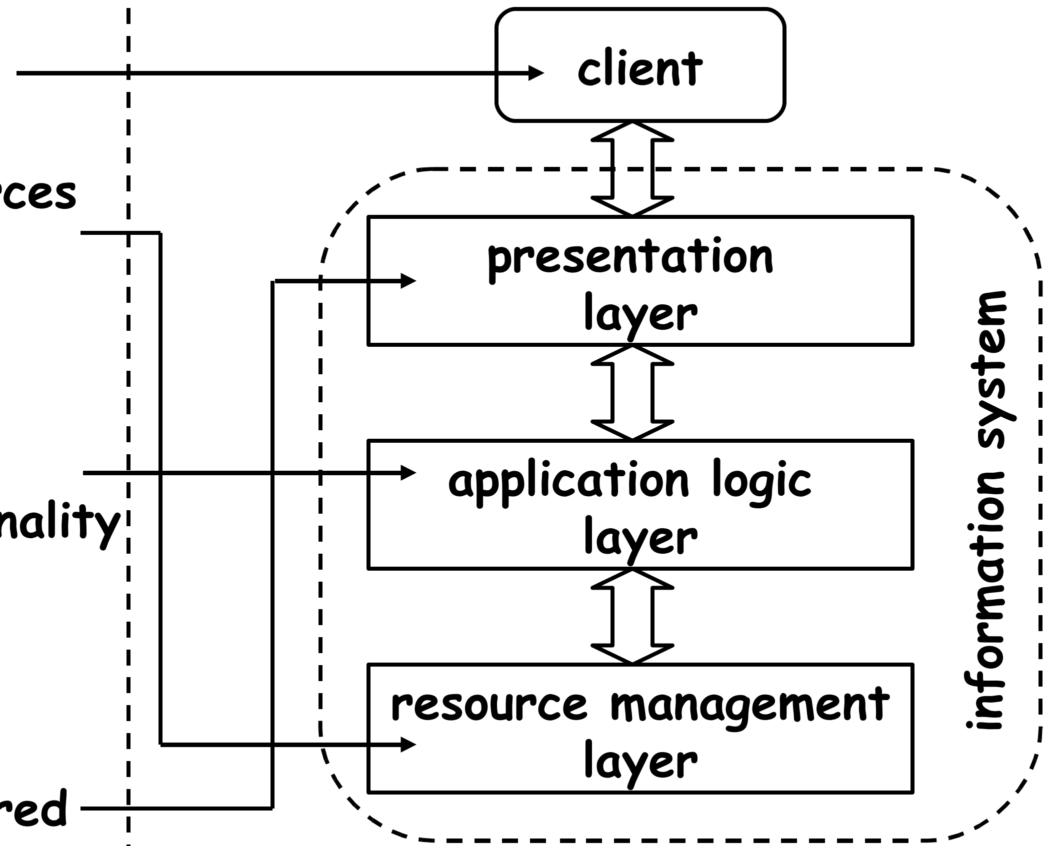
Bottom up design



Bottom up design

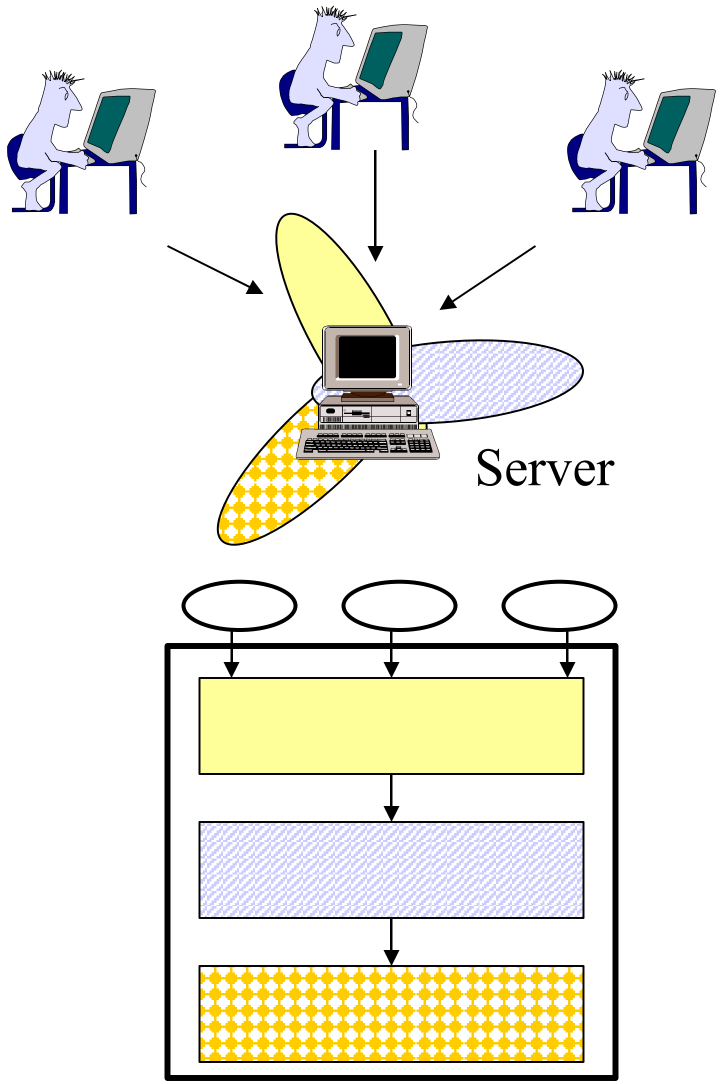
bottom-up design

1. define access channels and client platforms
2. examine existing resources and the functionality they offer
3. wrap existing resources and integrate their functionality into a consistent interface
4. adapt the output of the application logic so that it can be used with the required access channels and client protocols



One tier: fully centralized

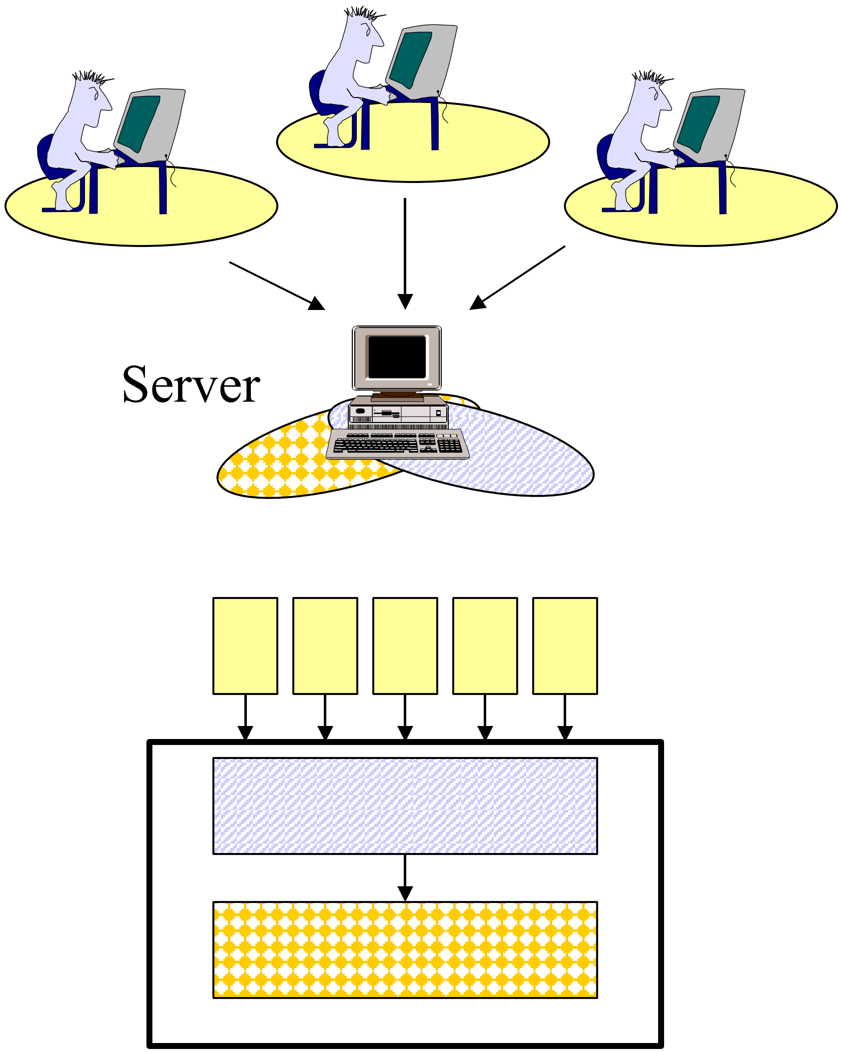
1-tier architecture



- The presentation layer, application logic and resource manager are built as a monolithic entity.
- Users/programs access the system through display terminals but what is displayed and how it appears is controlled by the server. (These are "dumb" terminals).
- This was the typical architecture of mainframes, offering several advantages:
  - → no forced context switches in the control flow (everything happens within the system),
  - → all is centralized, managing and controlling resources is easier,
  - → the design can be highly optimized by blurring the separation between layers.

Two tier: client/server

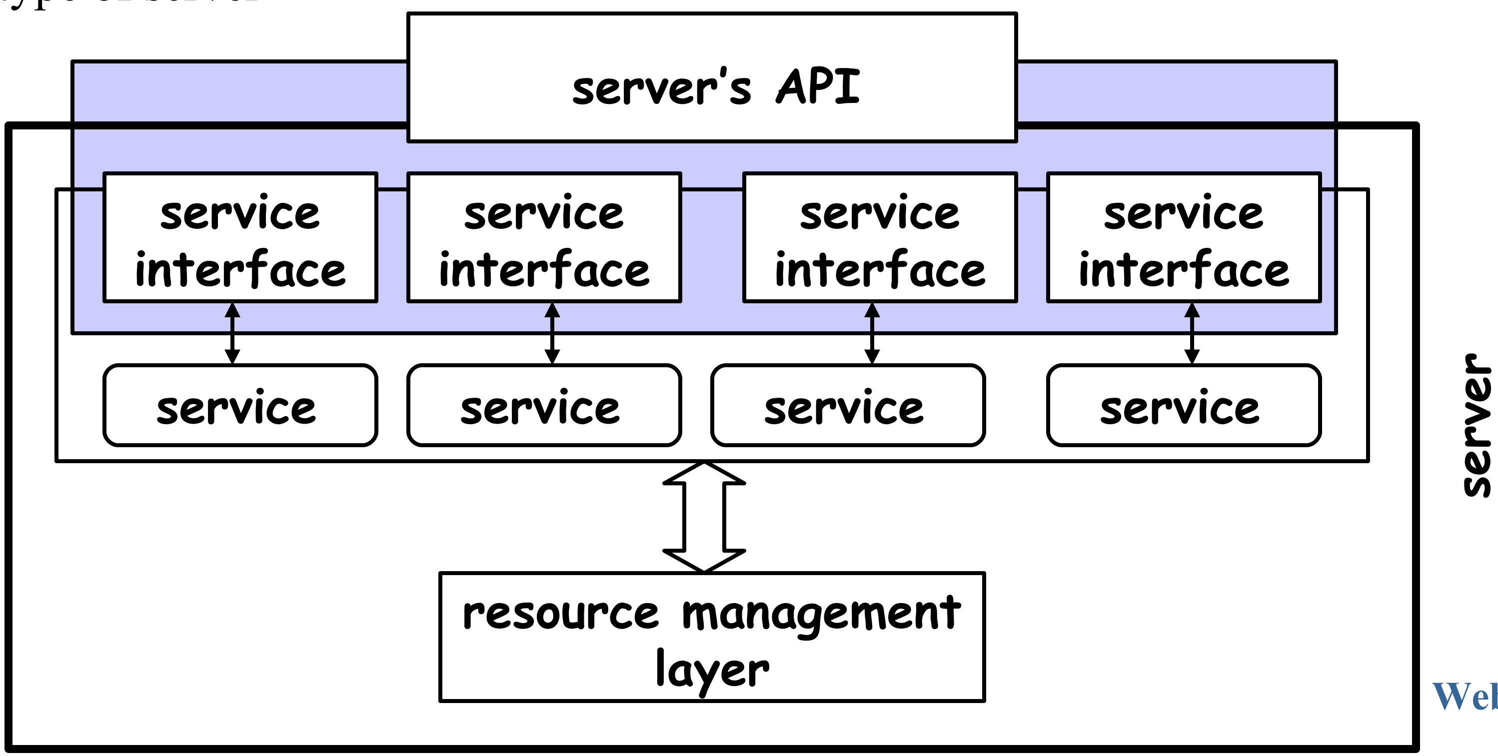
2-tier architecture



- As computers became more powerful, it was possible to move the presentation layer to the client. This has several advantages:
  - Clients are independent of each other: one could have several presentation layers depending on what each client wants to do.
  - One can take advantage of the computing power at the client machine to have more sophisticated presentation layers. This also saves computer resources at the server machine.
  - It introduces the concept of API (Application Program Interface). An interface to invoke the system from the outside. It also allows designers to think about federating the systems into a single system.
  - The resource manager only sees one client: the application logic. This greatly helps with performance since there are no client connections/sessions to maintain.

API in client/server

- Client/server systems introduced the notion of service (the client invokes a service implemented by the server)
- Together with the notion of service, client/server introduced the notion of service interface (how the client can invoke a given service)
- Taken all together, the interfaces to all the services provided by a server (whether there are application or system specific) define the server's Application Program Interface (API) that describes how to interact with the server from the outside
- Many standardization efforts were triggered by the need to agree to common APIs for each type of server

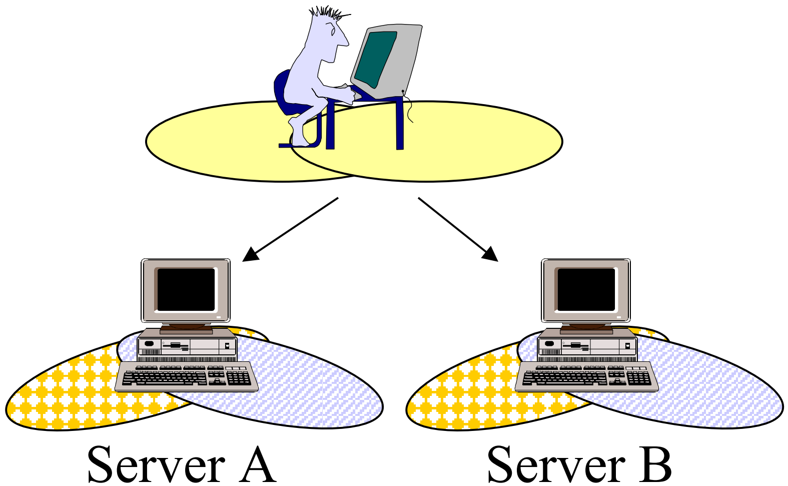


Technical aspects of the 2 tier architecture

- There are clear technical advantages when going from one tier to two tier architectures:
  - take advantage of client capacity to off-load work to the clients
  - work within the server takes place within one scope (almost as in 1 tier),
  - the server design is still tightly coupled and can be optimized by ignoring presentation issues
  - still relatively easy to manage and control from a software engineering point of view

- However, two tier systems have disadvantages:
  - The server has to deal with all possible client connections. The maximum number of clients is given by the number of connections supported by the server.
  - Clients are "tied" to the system since there is no standard presentation layer. If one wants to connect to two systems, then the client needs two presentation layers.
  - There is no failure or load encapsulation. If the server fails, nobody can work. Similarly, the load created by a client will directly affect the work of others since they are all competing for the same resources.

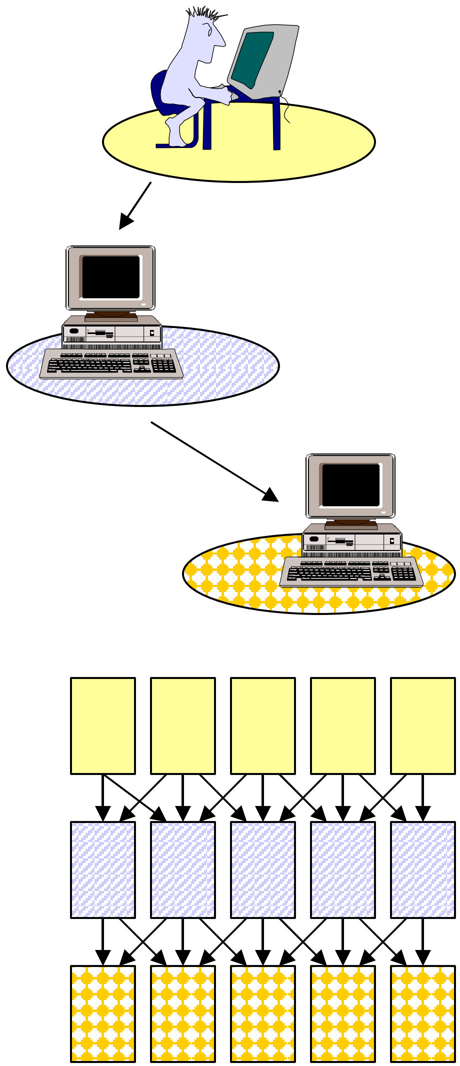
The main limitation of client/server



- If clients want to access two or more servers, a 2-tier architecture causes several problems:
  - the underlying systems don't know about each other
  - there is no common business logic
  - the client is the point of integration (increasingly fat clients)
    - The responsibility of dealing with heterogeneous systems is shifted to the client.
    - The client becomes responsible for knowing where things are, how to get to them, and how to ensure consistency
- This is tremendously inefficient from all points of view (software design, portability, code reuse, performance since the client capacity is limited, etc.).
- There is very little that can be done to solve this problems if staying within the 2 tier model.

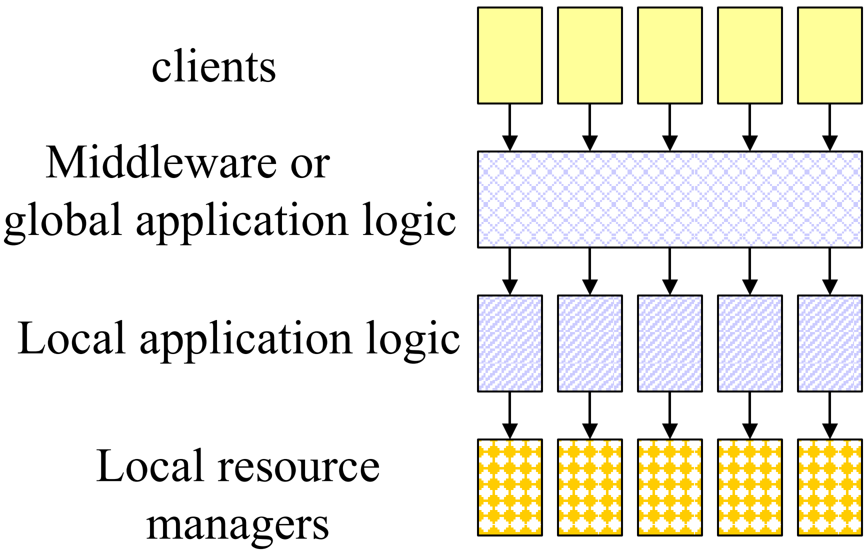
Three tier: middleware

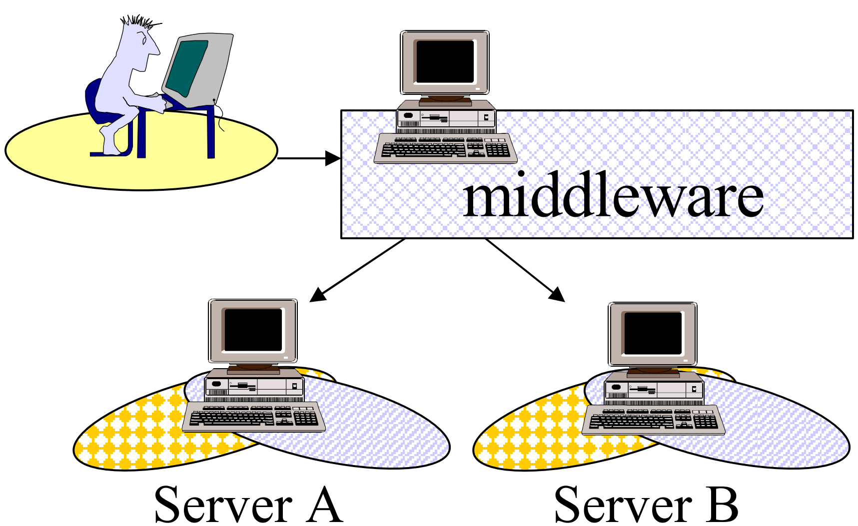
3-tier architecture



- In a 3 tier system, the three layers are fully separated.
- The layers are also typically distributed taking advantage of the complete modularity of the design (in two tier systems, the server is typically centralized)
- A middleware based system is a 3 tier architecture. This is a bit oversimplified but conceptually correct since the underlying systems can be treated as black boxes. In fact, 3 tier makes only sense in the context of middleware systems (otherwise the client has the same problems as in a 2 tier system).

Middleware



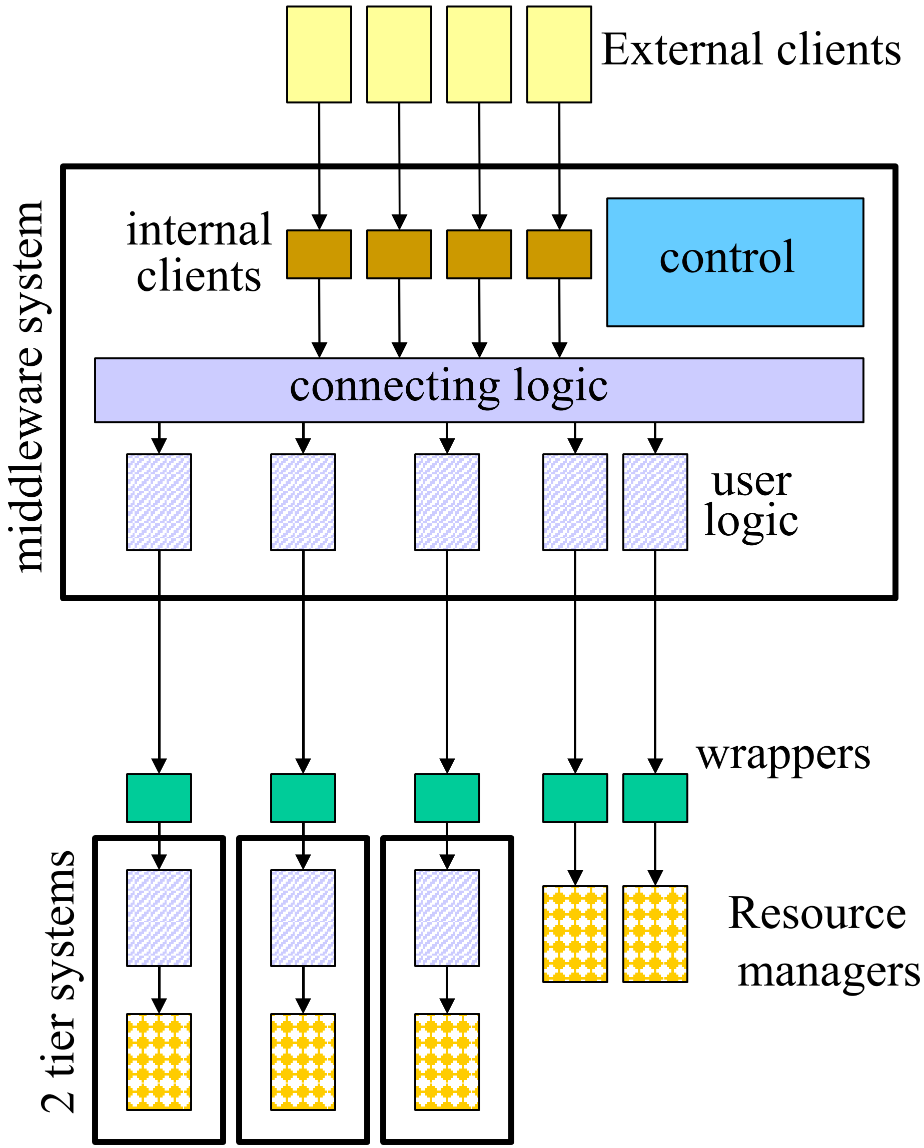


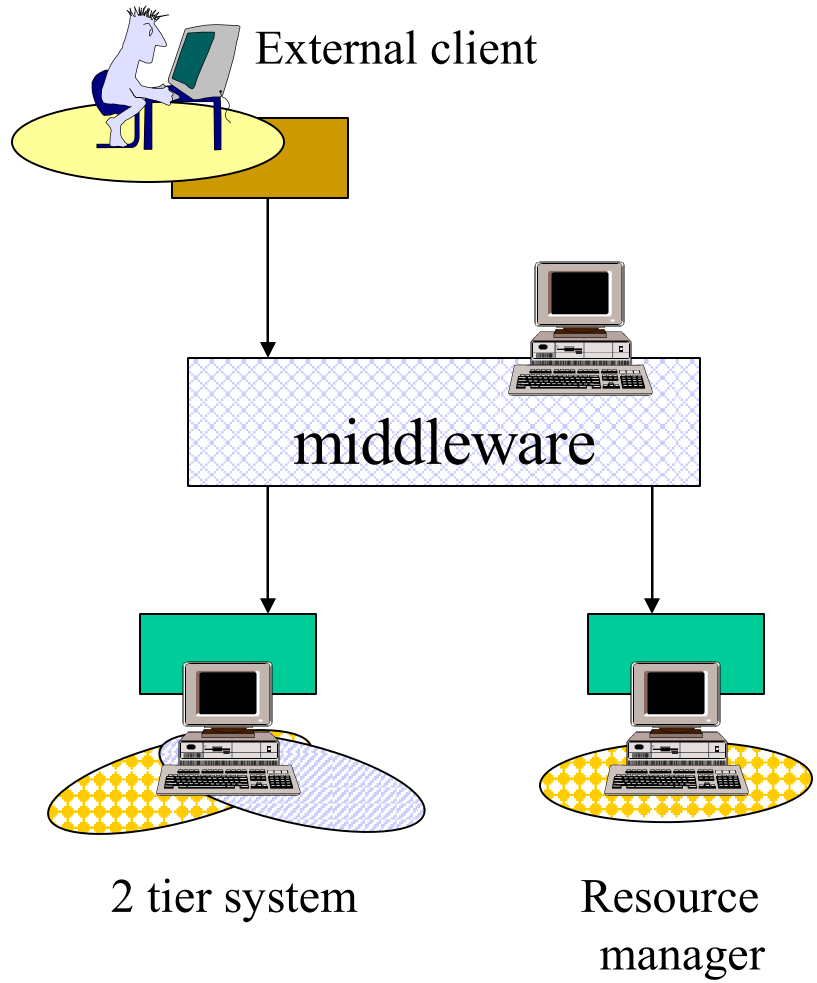
- Middleware is just a level of indirection between clients and other layers of the system.
- It introduces an additional layer of business logic encompassing all underlying systems.
- By doing this, a middleware system:
  - simplifies the design of the clients by reducing the number of interfaces,
  - provides transparent access to the underlying systems,
  - acts as the platform for inter-system functionality and high level application logic, and
  - takes care of locating resources, accessing them, and gathering results.
- But a middleware system is just a system like any other! It can also be 1 tier, 2 tier, 3 tier ...

Technical aspects of middleware

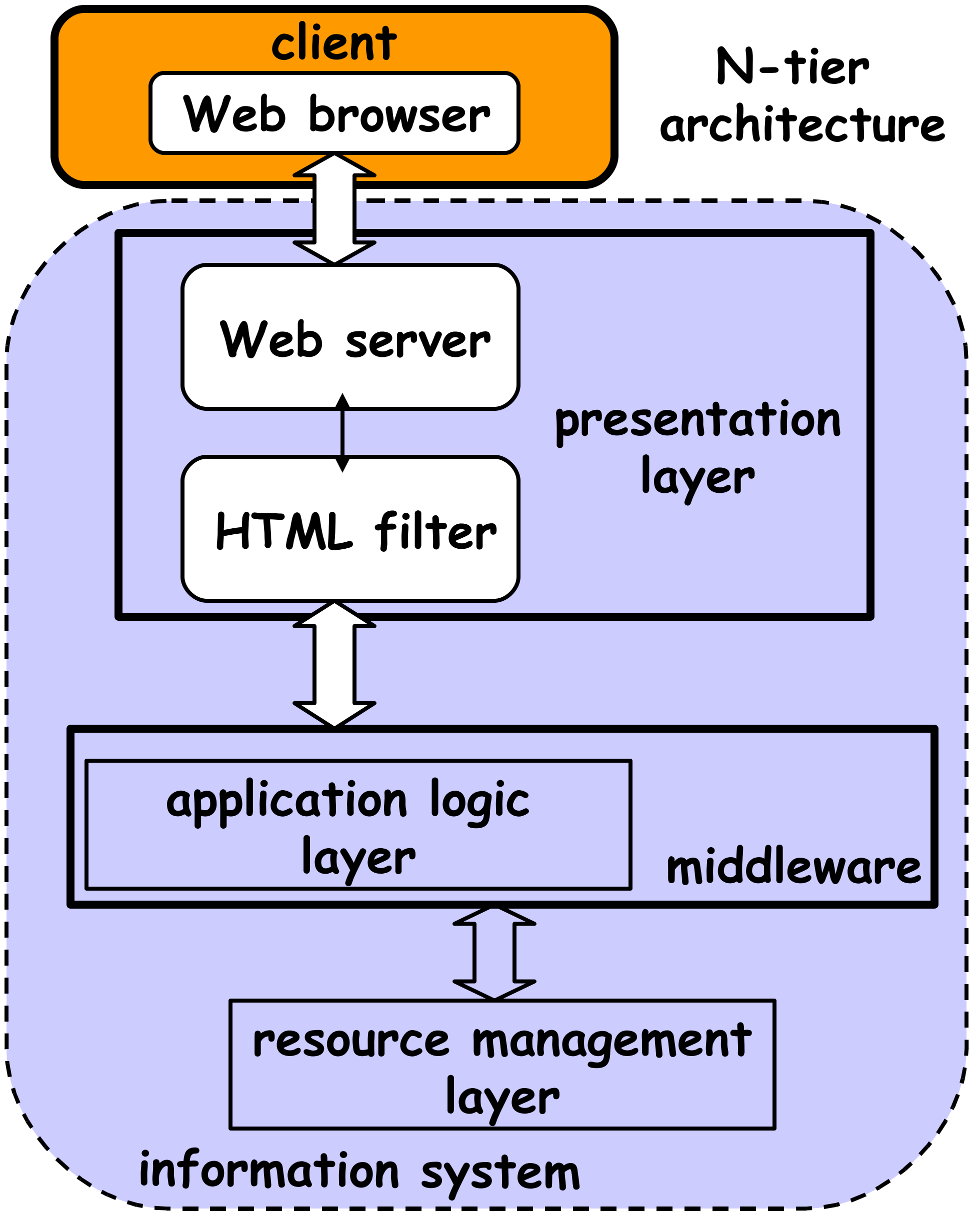
- The introduction of a middleware layer helps in that:
  - → the number of necessary interfaces is greatly reduced:
    - clients see only one system (the middleware),
    - local applications see only one system (the middleware),
  - → it centralizes control (middleware systems themselves are usually 2 tier),
  - → it makes necessary functionality widely available to all clients,
  - → it allows to implement functionality that otherwise would be very difficult to provide, and
  - → it is a first step towards dealing with application heterogeneity (some forms of it).
- The middleware layer does not help in that:
  - → it is another indirection level,
  - → it is complex software,
  - → it is a development platform, not a complete system

A three tier middleware based system ...



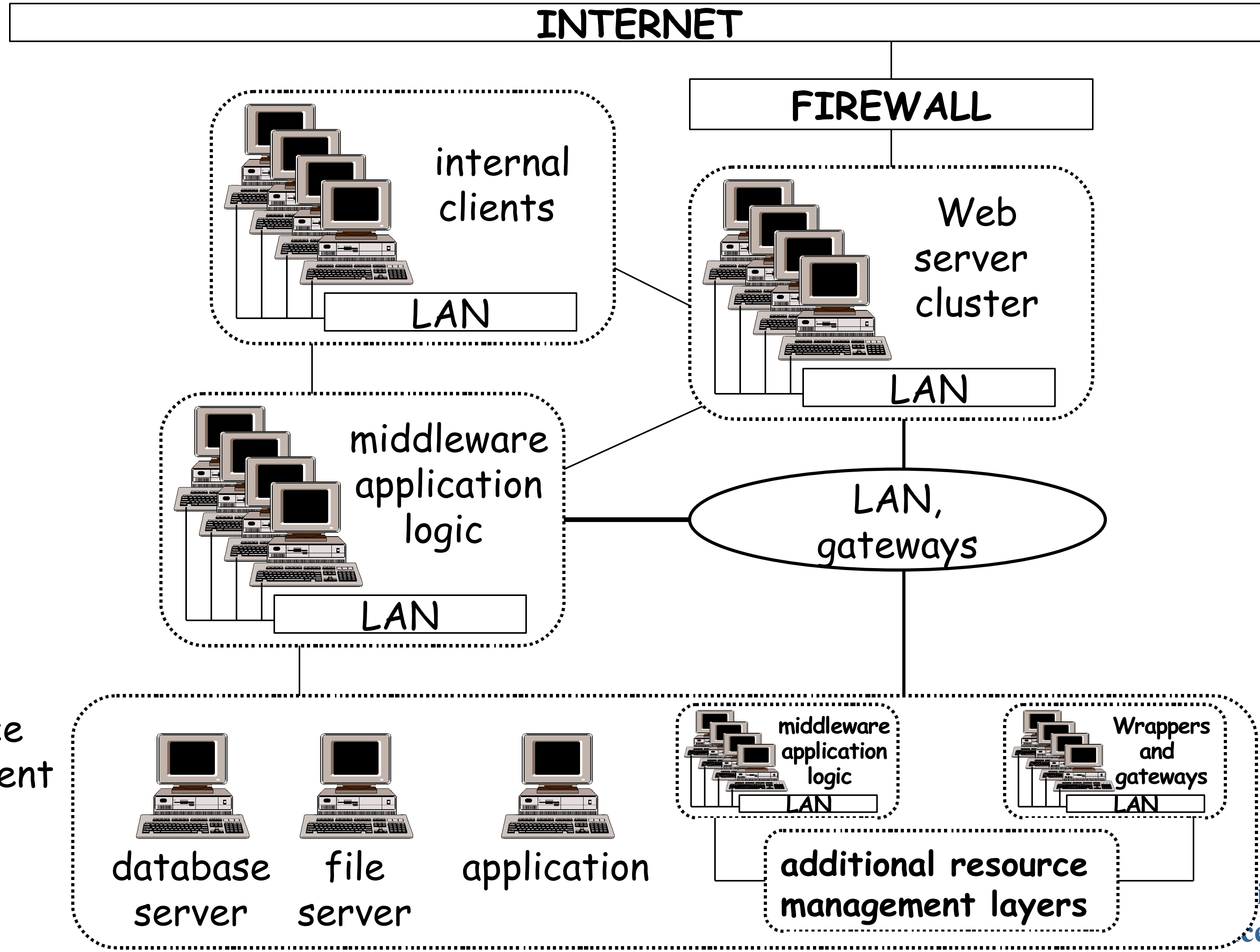


N-tier: connecting to the Web



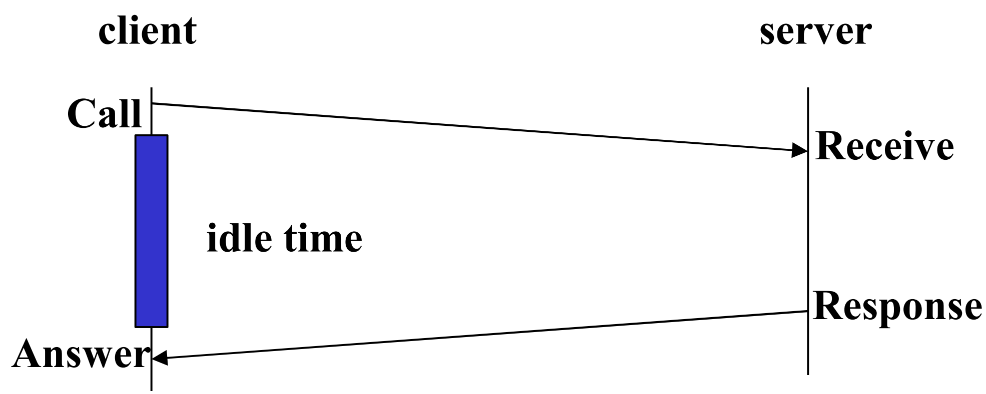
- N-tier architectures result from connecting several three tier systems to each other and/or by adding an additional layer to allow clients to access the system through a Web server
- The Web layer was initially external to the system (a true additional layer); today, it is slowly being incorporated into a presentation layer that resides on the server side (part of the middleware infrastructure in a three tier system, or part of the server directly in a two tier system)
- The addition of the Web layer led to the notion of "application servers", which was used to refer to middleware platforms supporting access through the Web

N-tier systems in reality



Blocking or synchronous interaction

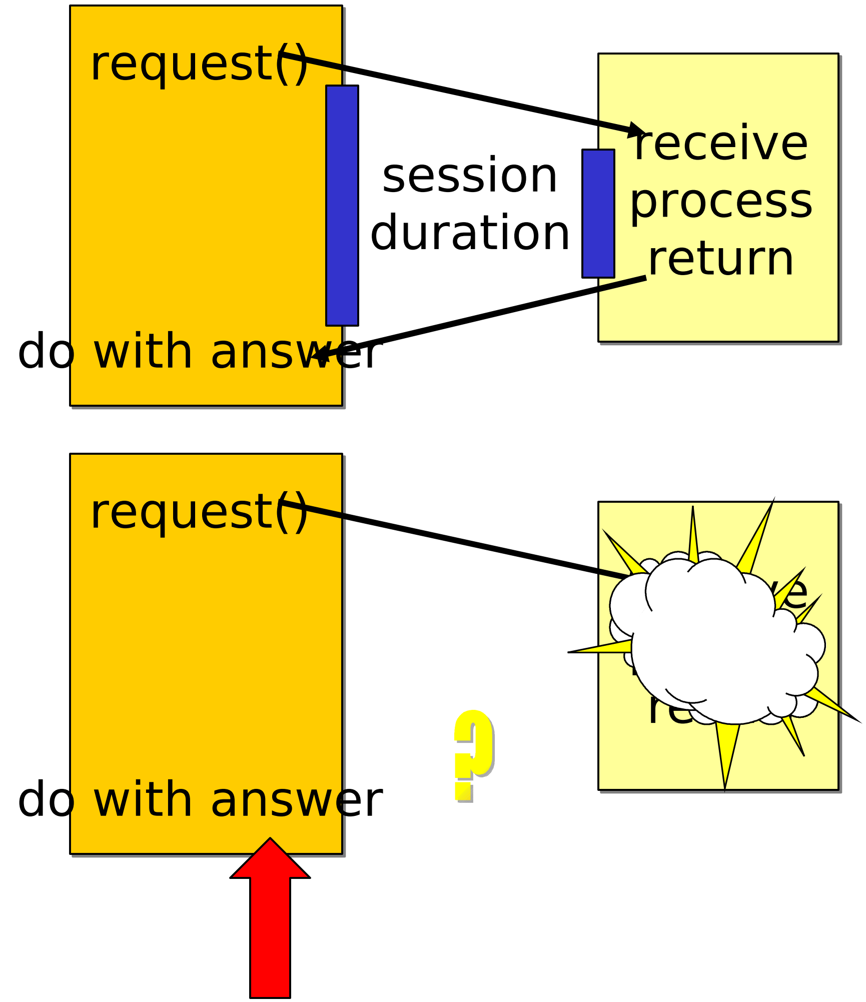
- Traditionally, information systems use blocking calls (the client sends a request to a service and waits for a response of the service to come back before continuing doing its work)
- Synchronous interaction requires both parties to be "on-line": the caller makes a request, the receiver gets the request, processes the request, sends a response, the caller receives the response.
- The caller must wait until the response comes back. The receiver does not need to exist at the time of the call (TP-Monitors, CORBA or DCOM create an instance of the service/server /object when called if it does not exist already) but the interaction requires both client and server to be "alive" at the same time



- Because it synchronizes client and server, this mode of operation has several disadvantages:
  - → connection overhead
  - → higher probability of failures
  - → difficult to identify and react to failures
  - → it is a one-to-one system; it is not really practical for nested calls and complex interactions (the problems becomes even more acute)

Overhead of synchronism

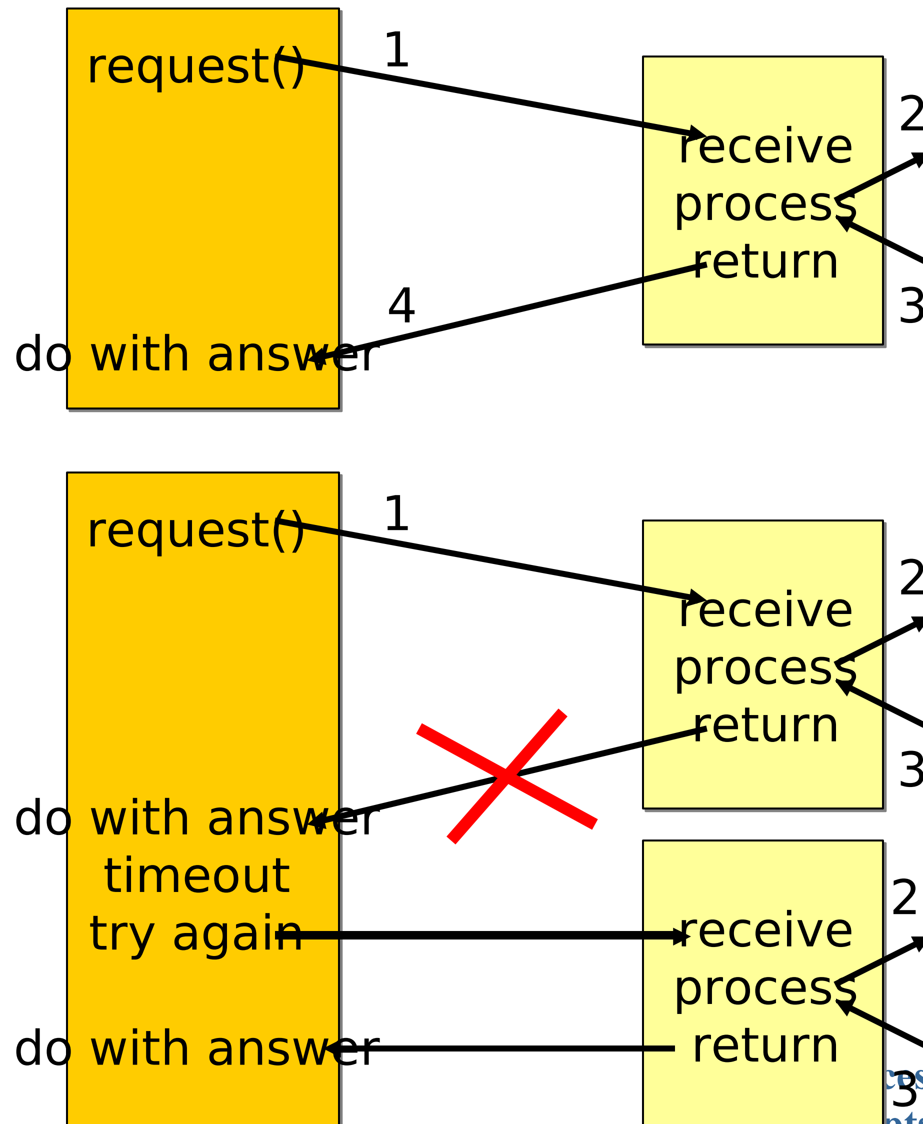
- Synchronous invocations require to maintain a session between the caller and the receiver.
- Maintaining sessions is expensive and consumes CPU resources. There is also a limit on how many sessions can be active at the same time (thus limiting the number of concurrent clients connected to a server)
- For this reason, client/server systems often resort to connection pooling to optimize resource utilization
  - → have a pool of open connections
  - → associate a thread with each connection
  - → allocate connections as needed
- Synchronous interaction requires a context for each call and a context management system for all incoming calls. The context needs to be passed around with each call as it identifies the session, the client, and the nature of the interaction.



Context is lost
Needs to be restarted!!

Failures in synchronous calls

- If the client or the server fail, the context is lost and resynchronization might be difficult.
  - ➞ If the failure occurred before 1, nothing has happened
  - ➞ If the failure occurs after 1 but before 2 (receiver crashes), then the request is lost
  - ➞ If the failure happens after 2 but before 3, side effects may cause inconsistencies
  - ➞ If the failure occurs after 3 but before 4, the response is lost but the action has been performed (do it again?)
- Who is responsible for finding out what happened?
- Finding out when the failure took place may not be easy. Worse still, if there is a chain of invocations (e.g., a client calls a server that calls another server) the failure can occur anywhere along the chain.



Two solutions

ENHANCED SUPPORT

- Client/Server systems and middleware platforms provide a number of mechanisms to deal with the problems created by synchronous interaction:
  - Transactional interaction: to enforce exactly once execution semantics and enable more complex interactions with some execution guarantees
  - Service replication and load balancing: to prevent the service from becoming unavailable when there is a failure (however, the recovery at the client side is still a problem of the client)

ASYNCHRONOUS INTERACTION

- Using asynchronous interaction, the caller sends a message that gets stored somewhere until the receiver reads it and sends a response. The response is sent in a similar manner
- Asynchronous interaction can take place in two forms:
  - non-blocking invocation (a service invocation but the call returns immediately without waiting for a response, similar to batch jobs)
  - persistent queues (the call and the response are actually persistently stored until they are accessed by the client and the server)

Message queuing

- Reliable queuing turned out to be a very good idea and an excellent complement to synchronous interactions:
  - → Suitable to modular design: the code for making a request can be in a different module (even a different machine!) than the code for dealing with the response
  - → It is easier to design sophisticated distribution modes (multicast, transfers, replication, coalescing messages) an it also helps to handle communication sessions in a more abstract way
  - → More natural way to implement complex interactions between heterogeneous systems