# Distributed systems' architecture

**Antonio Lioy < lioy@polito.it >**

***english version created and modified by***
**Marco D. Aime < m.aime@polito.it >**

***Politecnico di Torino***
***Dip. Automatica e Informatica***

## Typical application model

- **user interface (UI)**
  - manages all I/O with the user
- **application logic**
  - computations required to provide service to the user
- **data (rough)**
  - information required by the application

## Example ("classic" app.)

```
#include <stdio.h>
int main ( )
{
    double vat_percentage = 20;
    double price;
char buf[100];
printf ("cost? ");
gets (buf);
sscanf (buf, "%lf", &cost);
    price = cost * (1 + vat_percentage / 100);
printf ("selling price = %.2lf\n", price);
return 0;
}
```

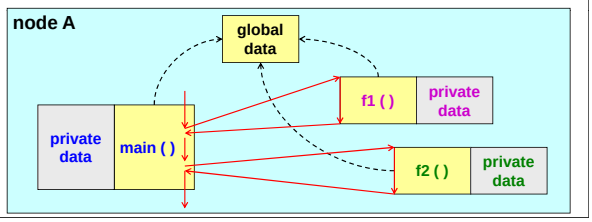
application data

application logic

user interface

## "Classic" computing

- **local data (shared / private)**
- **single address space**
- **sequential computing over a single CPU**
- **unique computation flow (exception: interrupt)**




## "Classic" computing: advantages

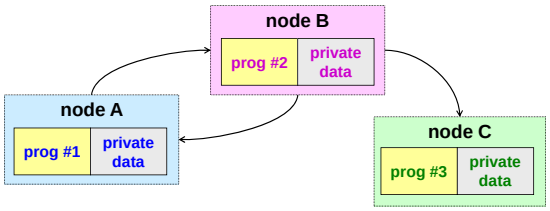
- **ease of programming**
- **robustness**
- **good chance of optimisation**

## "Classic" computing: problems

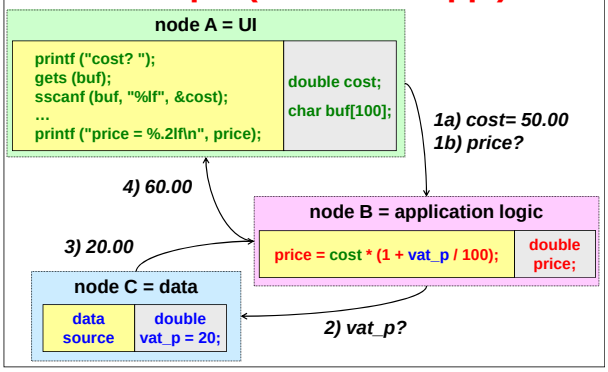
- **data protection from illegal operations**
  - operations over global data
  - also private data are accessible (!)
  - partially mitigated with OOP
- **low performance**
  - single CPU, sequential computation
  - mitigated with multi-CPU systems and concurrent programming (thread, processes)
- **requires physical access to the system (for usage)**
  - terminals or "console"
  - mitigated with modem / network connections

## Distributed computing

- only local data (private)
- multiple address spaces
- concurrent computation over different CPUs
- multiple computation flows




## Example (distributed app.)




## Distributed computing: advantages

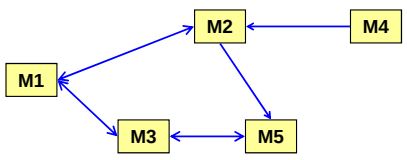
- **high performance**
  - several CPUs
- **good scalability**
  - increasing the number of CPUs is easier then increasing the performance of a single CPU
- **data protected from illegal operations**
  - disjoint memory spaces, accessible only by their respective programs
- **network access**
  - user physical presence not required

## Distributed computing: problems

- **programming complexity:**
  - how the various programs communicate together?
  - which data format on the various network nodes?
  - need to define (application) protocols
  - operations synchronisation may lead to delay and slowing down
- **scarce robustness**
  - higher chance of errors / faults
- **hard optimisation**
  - lack of global view

## Software architecture

- **collection of software modules (or components)**
- **… interacting through a well-defined communication paradigm (or connectors)**
- **note: communication not always via network (e.g. IPC within the same node)**




## Client-server model

- **the most diffused method to create distributed applications**
- **client and server are two different processes:**
  - the server provides a generic service
  - the client requests the service
- **also on the same system**

**Consider the difference between client and server:**
**– as hw elements of a computing system**
**– as processes of a distributed architecture**

## The server

- **ideally it has been executing ‘‘for ever’’:**
  - activated at boot time
  - activated explicitly by the system administrator
- **accepts requests from one or more points:**
  - TCP o UDP port (concept similar to OSI’s SAP)
  - fixed ports, usually pre-determined
- **sends responses relative to the service**
- **ideally it never stops:**
  - at shutdown
  - explicit action by the system administrator

## The client

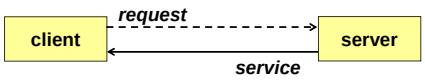
- **activated upon request of a “user”**
- **sends requests to a server**
- **waits for the response on a dynamically allocated port (the port cannot be fixed since there can be many simultaneous “users”, e.g. two windows of a web browser)**
- **executes a finite number of requests and then stops**

## Architecture

- **several architecture types can be built by using the client and server concepts**
- **client-server (C/S) architecture**
  - asymmetric architecture
  - server position is determined a priori
- **peer-to-peer (P2P) architecture**
  - symmetric architecture
  - every node can play both the client and server roles (simultaneously or at different times)

## Client-server architecture (C/S)

- **architecture in which client processes request services offered by server processes**
- **advantages:**
  - simple to design
  - simplification of the client
- **disadvantages:**
  - server overload
  - communication channel overload




## C/S 2-tier architecture

- **the original, classic C/S (e.g. NFS)**
- **the client interacts directly with the server without any intermediate step**
- **typically distributed over either a local or a geographic scale**
- **used in small environments (50-100 simultaneous clients)**
- **disadvantages:**
  - low scalability (e.g. as the number of clients increases, the server performance decreases)

## C/S 2-tier: heavy or light client?

- **three components (UI, application logic, data) … to be distributed over two elements (client & server)**
- **solution 1 = fat client / thin server**
  - client = UI + application logic
  - server = data
  - traditional scheme, difficulties with development (ad-hoc sw) and management (installation, updates), less security
- **solution 2 = thin client / fat server:**
  - client = UI
  - server = application logic + data
  - (e.g. the web) heavy on servers, higher security

## C/S 3-tier architecture

- **an additional component (or agent) is placed between client and server, playing various roles:**
  - filter (e.g. to adapt a legacy mainframe system to a C-S environment)
  - balance workload on the server(s) (e.g. load balancer with several equivalent servers)
  - intelligent services (e.g. to distribute a request to several different servers, collect results, and reply to client as a single response)

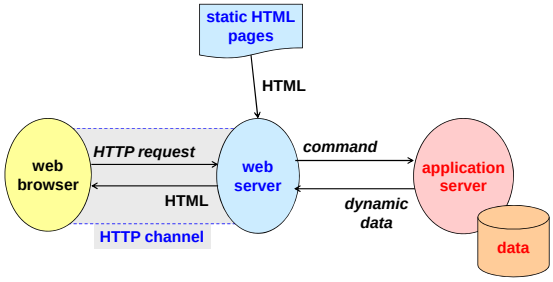
## Examples of 3-tier systems

- **to improve (computing) performance:**
  - agent = load balancer
  - server = server farm of homogeneous or equivalent servers
- **to adapt the server to client's capabilities:**
  - agent = mediator / broker
  - server = set of heterogeneous / not equivalent servers

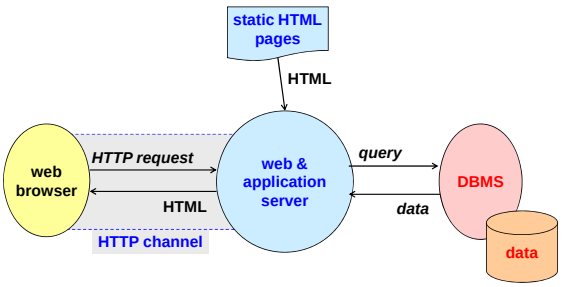
## The UI and the web

- **"traditional" custom UI:**
  - difficult development, deployment and maintenance
  - difficult user training
- **the "modern" web UI is split in two:**
  - standard client-side UI (=browser)
  - standard server-side UI (=server web), easily programmable

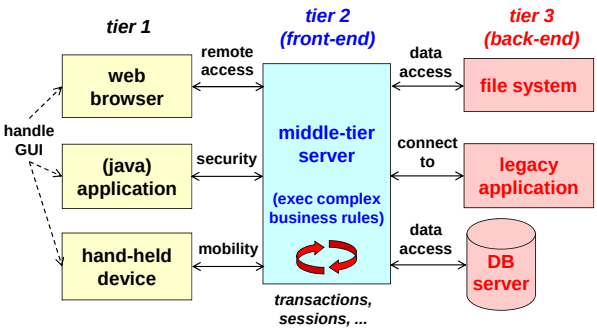
## 3-tier: web model (I)




## 3-tier: web model (II)

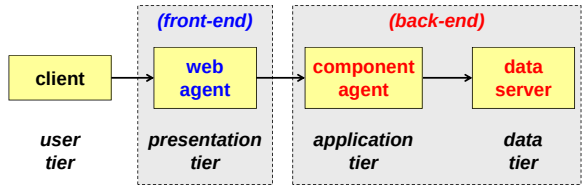



## 3-tier: application example



## C/S 4-tier architecture

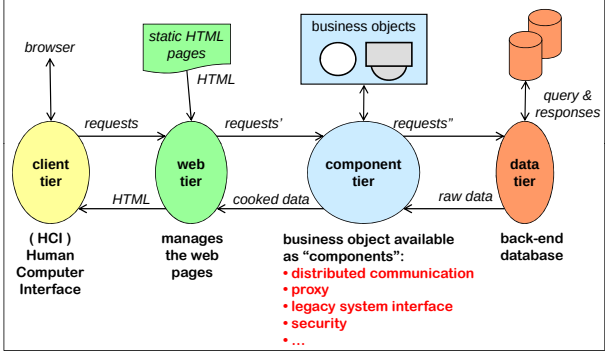
- **two components / agents placed between client and server:**
  - web agent (manages data presentation)
  - component agent (manages application logic)




## How to improve network performance?

- **3/4-tier architectures improve computing performance … but the front-end is a bottle neck**
- **how to improve? the service provider has no control over the network segment between clients and the front-end**
- **improvement attempts:**
  - statistics on the clients' sources
  - replicate the front-end (one per each network which clients come from)
  - how to direct clients towards the right front-end?
    - based on language/domain (e.g. srv.it, srv.fr)
    - based on routing (e.g. Akamai modified DNS)

## 4-tier: Internet e-commerce system




## Client tier: browser or application?

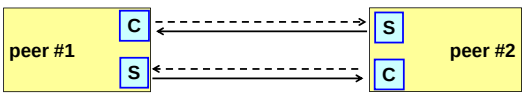
- **web browser:**
  - (P) known to users and managed by users
  - (P) standard data & communication (HTTP, HTML)
  - (C) uncertain version of protocol and data (shared minimum?)
  - (C) limited performance (interpreter)
  - (C) limited functionality (simple graphic interface)
  - (C) extensions not always supported:
    - applets (Java, Active-X)
    - client-side scripts (JavaScript, VBscript)
    - plug-ins (Flash, …)

## Client tier: browser or application?

- **custom / ad-hoc client application:**
  - (P) rich functionality (=required by the server)
  - (P) high performance
  - (C) user training
  - (C) supported platforms
  - (C) deployment & maintenance
  - (C) support to the users

## Peer-to-peer architecture (P2P)

- **architecture in which processes may act simultaneously as client and server**
- **advantages:**
  - computing and communication load distributed among all the processes
- **disadvantages:**
  - difficulties in coordination / control



## P2P computing

- **clients evolve from mere service users to autonomous service providers**
- **to share resources and run collaborative services**
- **better exploitation of capabilities at every node (to decrease the load on servers)**
- **better exploitation of networks, with direct node communication (to avoid congestions on the server uplinks)**

## P2P architectures

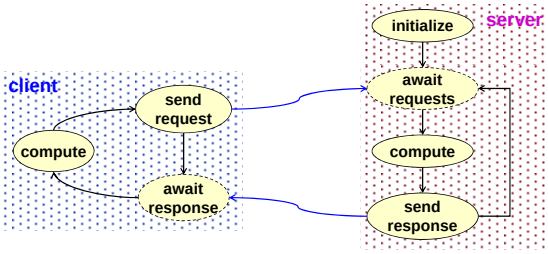
- **collaborative computing**
  - network community for distributed tasks (e.g. grid computing; open or closed; also for confidential data or for fixed deadline computation?)
- **edge service**
  - orthogonal services as “enabling factors” to form P2P communities (e.g. TOIP, Internet fax)
- **file sharing**
  - to share information over the network without uploading to a server, and leaving it where it is (problem: the index)
  - e.g. Gnutella (gnutella.wego.com), WinMX, Kazaa

## Server models

- **the internal server architecture heavily influences the overall performance of the system**
- **the best model should be selected based on the specific application problem**
- **no solution is good for every use (the risk is choosing an over-complicated one)**

## Iterative server

- **if the requested service requires short time, the server directly provides it**




## Examples of iterative servers

- **standard TCP/IP service with short duration:**
  - daytime (tcp/13 o udp/13) RFC-867
  - qotd (tcp/17 o udp/17) RFC-865
  - time (tcp/37 o udp/37) RFC-868
- **generally, for services with strong load limitations (one user per time)**
- **advantages:**
  - simple to programme
  - response speed (when the connection succeeds!)
- **disadvantages:**
  - limited workload

## Performance of iterative servers

- **does not depend on the number of CPUs**
- **with $T_E$ the CPU time required by server computation [s] (hypothesis: $T_E >> T_R$)**
- **maximum performance (optimal conditions):**

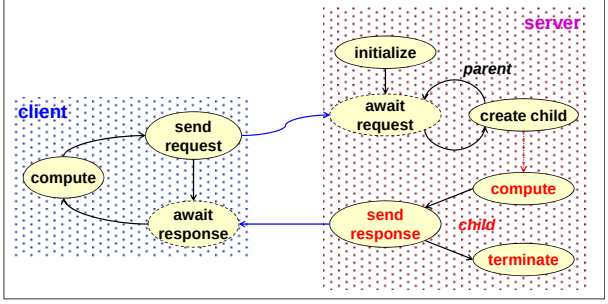
$$P = 1 / T_E \quad \text{services / s}$$

- **in case of multiple simultaneous requests, the ones not served compete again later (but not in case the request queue has width > 1)**
- **latency [ s ] of the service depends on the workload W>=1 of the node hosting the server:**

$$T_E \le L \le T_E \times W \quad \text{that is} \quad L \sim T_E \times E(W)$$

## Concurrent server

- **when the service requires long time, the server starts a "child" sub-process and returns on waiting**




## Examples of concurrent servers

- **the majority of standard TCP/IP services:**
  - echo (tcp/7 o udp/7) RFC-862
  - discard (tcp/9 o udp/9) RFC-863
  - chargen (tcp/19 o udp/19) RFC-864
  - telnet (tcp/23) RFC-854
  - smtp (tcp/25) RFC-2821
  - …
- **generally, for services with complex computation or with long and/or unpredictable duration**

## Concurrent server: analysis

- **advantages:**
  - load ideally unlimited
- **disadvantages:**
  - complex to programme (concurrent programming)
  - slow response (child creation, $T_F$)
  - limited maximum load (every child requires RAM, CPU cycles, disk access cycles, …)

## Performance of concurrent servers

- **depends on the number of CPUs (let it be C)**
- **with $T_F$ the CPU time to create a new child [s]**
- **maximum performance (optimal conditions):**

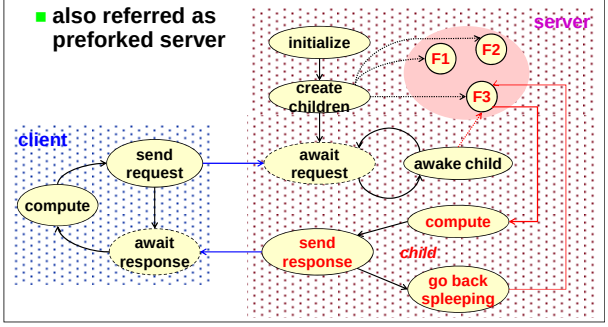
$$P = C / (T_F + T_E) \text{ services / s}$$

- **in case of multiple simultaneous requests, the ones not served compete again later (but not in case the request queue has width > 1)**
- **the service latency depends on the workload W of the node hosting the server:**

$$(T_F + T_E) \le L \le (T_F + T_E) \times W / C \quad s$$

## "Crew" server

- **pre-activation of children to improve response time**
- **also referred as preforked server**




## Examples of "crew" servers

- **every concurrent services can be implemented with "crew" server**
- **for high performance network services:**
  - with high load (=number of simultaneous clients)
  - with low response delay (latency)
- **typical examples:**
  - web server for e-commerce
  - DBMS server

## "Crew" server: analysis

- **advantages:**
  - workload ideally unlimited (additional children can be spawned depending on the load)
  - response speed (awake a child faster than create)
  - chance to limit the maximum load (only pre-generated children)
- **disadvantages:**
  - complex to programme (concurrent programming)
  - children management (children pool)
  - synchronization and concurrency of children access to shared server resources

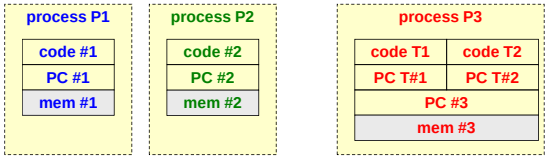
## Performances of "crew" servers

- **similar to concurrent server's ones, with $T_F$ replaced with the time $T_A$ required to activate a child (usually negligible)**
- **if additional children can be created (after exhausting the original pool), the performance is a combination weighted with the probability G of requiring additional children:**

$$P = (1 - G) \times [\ C / (T_A + T_E)\ ] + G \times [\ C / (T_F + T_E)\ ]$$

## Concurrent programming

- **simultaneous work of multiple computing modules on the same CPU**
- **two main models:**
  - processes
  - threads




## Processes vs. threads (I)

- **module activation**
  - [ P ] slow
  - [ T ] fast
- **module communication**
  - [ P ] difficult (requires IPC, e.g. pipes, shared memory)
  - [ T ] easy (same address space)

## Processes vs. threads (II)

- **module protection:**
  - [ P ] optimal, both of memory and of CPU cycles
  - [ T ] worst (and the access to shared memory requires synchronisation and may cause deadlock)
- **debug:**
  - [ P ] not trivial but possible
  - [ T ] very difficult (cannot replicate the scheduling)