

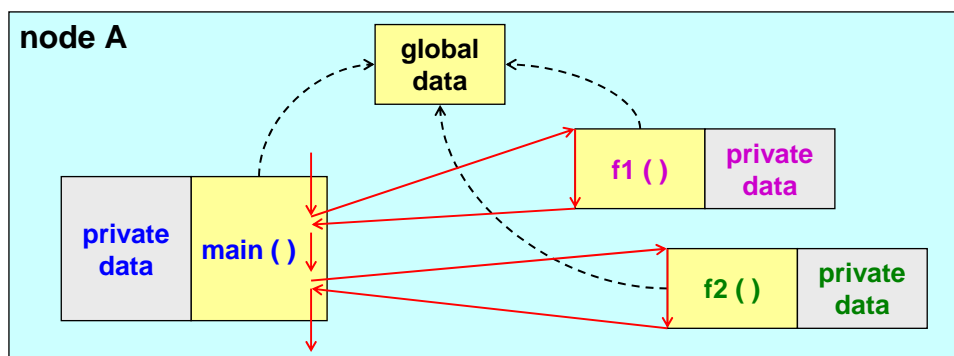
Distributed Systems Architectures

Antonio Lioy
<lioy@polito.it >

Politecnico di Torino
Dip. Automatica e Informatica

"Classical" processing

- local data (shared / private)
- unique address space
- sequential processing on a single CPU
- unambiguous processing flow (exception: interrupt)



“Classical” processing: advantages

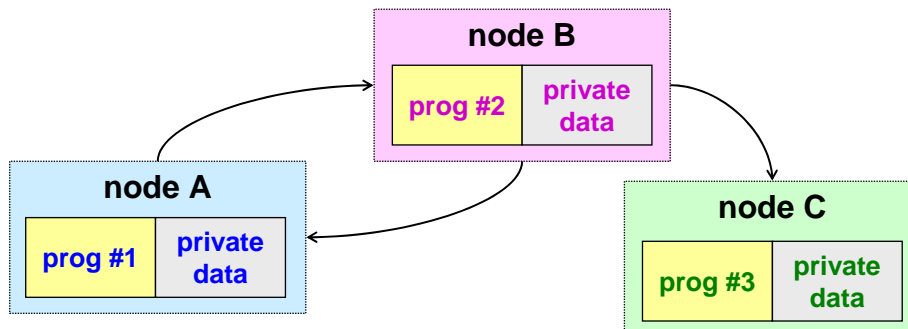
- **simplicity of programming**
- **robustness**
- **good possibility to optimize code**

“Classical” processing: problems

- **data protection from illegal operations**
 - operations executed on global data
 - private data is also accessible (!)
 - improved with OOP
- **low performance**
 - unique CPU, sequential processing
 - improved with multi-CPU systems and concurrent programming (thread, processes)
- **use only by means of physical access at the system**
 - terminals or “console”
 - improved with connection via modem

Distributed processing

- only local data (private)
- many address spaces
- concurrent processing on different CPUs
- many processing flows



Distributed processing: advantages

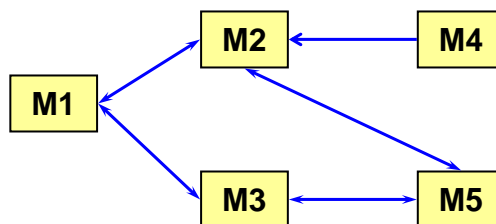
- high performance
 - many CPU
- good scalability
 - easier to increase the no. of CPU than the processing power of a single CPU
- data protection from illegal operations
 - separate address spaces, accessible only by the respective programs
- remote access
 - is not required the physical presence of the user

Distributed processing: problems

- **complexity of programming:**
 - how the various programs communicate ?
 - data format on various network nodes ?
 - necessity to define (application) protocols
- **poor scalability**
 - increased possibility of error / incorrect functionality
- **difficult to optimize code**
 - lack of a global view

Software architecture

- **collection of software modules (or components)**
- **... interacting through a well defined communication paradigm (or connectors)**
- **note: communication not necessarily occurring among network nodes (e.g. IPC on a single node)**



Client-server model

- **most widely used method to create distributed applications**
- **client and server are two separate processes:**
 - the server provides a generic service
 - the client requires the service
- **also on the same system**

The server

- **ideally is executed “continuously”:**
 - started at boot
 - started explicitly by the system administrator
- **accepts requests from or more nodes:**
 - TCP or UDP port (analogous to OSI concept of SAP)
 - fixed ports and usually already established
- **sends responses regarding a service**
- **ideally never exits:**
 - at shutdown
 - explicit action of the system administrator

The client

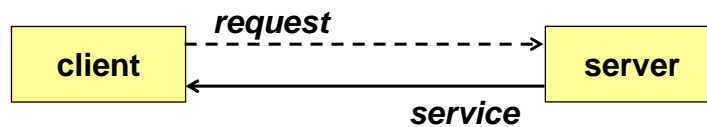
- started on request of a “user”
- sends request to a server
- wait for the response on a port dynamically allocated (cannot be a fixed port because there could exist many “users” that operate simultaneously, e.g. two windows of a web browser)
- executes a finite number of requests and then exits

Architectures

- using the concepts of client and server it is possible to construct various architectures
- **client-server architecture (C/S)**
 - asymmetric architecture
 - the location of the server is determined a priori
- **peer-to-peer architecture (P2P)**
 - symmetric architecture
 - each node can have the role of client and of server (simultaneously or at different times)

Client-server architecture (C/S)

- **architecture where client processes require the services provided by server processes**
- **advantages:**
 - simplicity of development
 - simplification of client
- **disadvantages:**
 - overload of server
 - overload of communication channel



2-tier C/S architecture

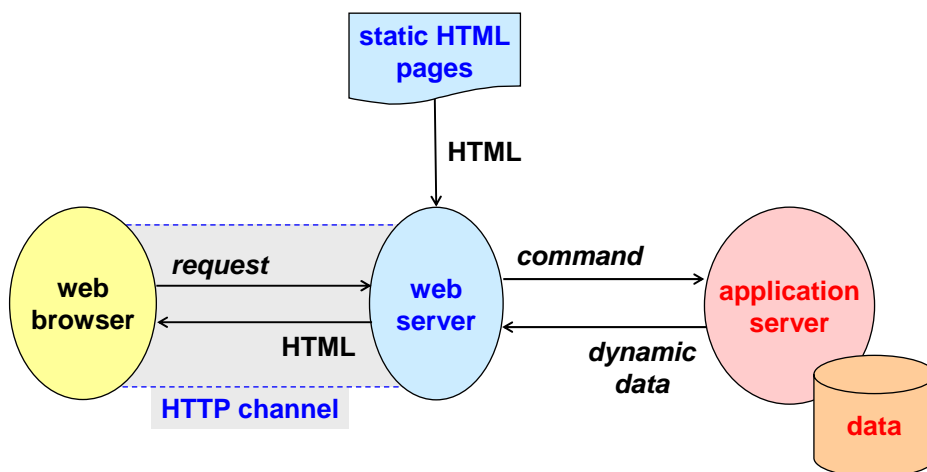
- **is the classical C/S, original (e.g. NFS)**
- **the client interacts directly with the server without intermediary steps**
- **architecture typically distributed both locally and geographically**
- **used in environments of small dimensions (50-100 simultaneous clients)**
- **disadvantages**
 - low scalability (e.g. as the number of users increases, the performance of server decreases)

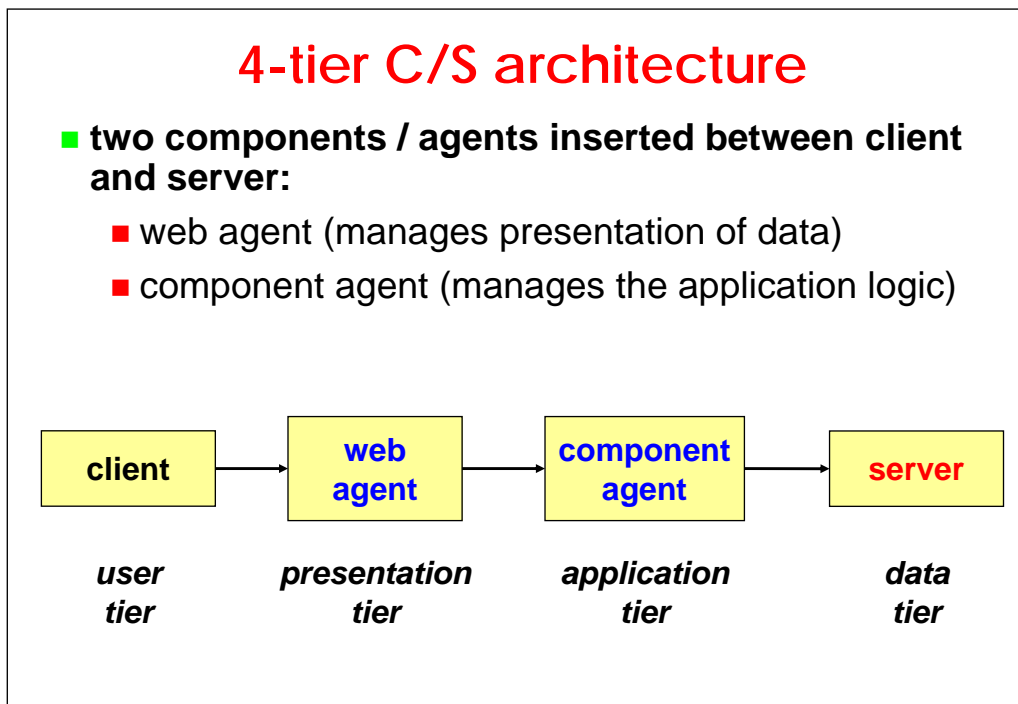
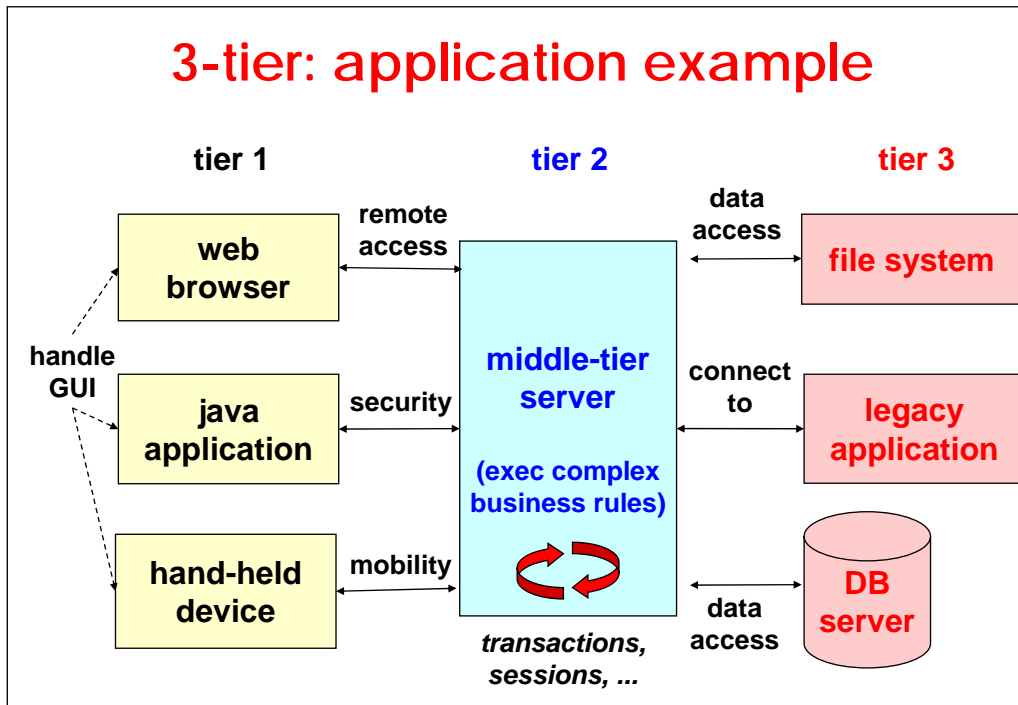
3-tier C/S architecture

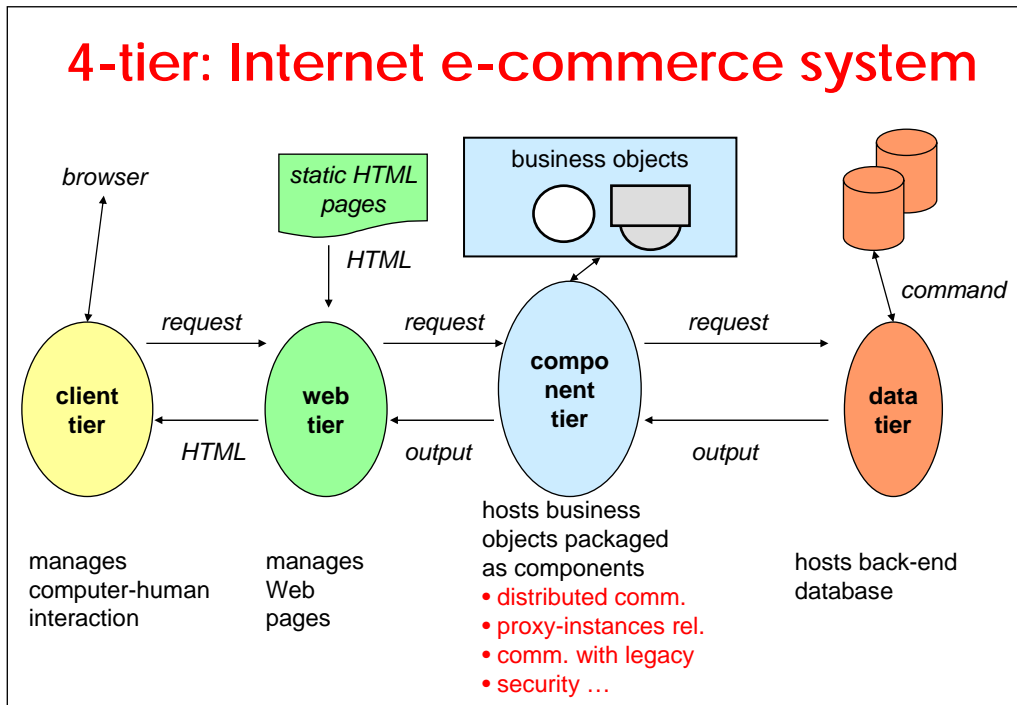
- a component (or agent) is inserted between the client and server, to carry out various roles:
 - filter (e.g. adapt a legacy system on mainframe to a C-S environment)
 - workload balancing on the server(s) (e.g. transaction monitor to limit the number of simultaneous requests)
 - intelligent services (e.g. distribute a request to several servers, collect the results and send them back to the client as a single response)



3-tier: web model







Client tier: browser o application?

■ web browser:

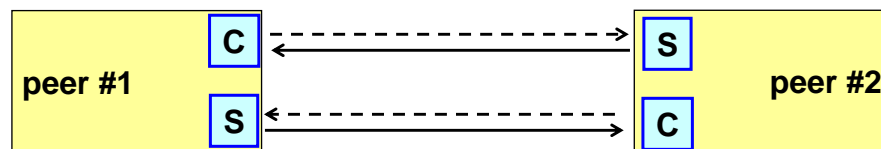
- (A) known by the users and managed by them
- (D) limited functionality
- (D) extensions not always supported:
 - applet (Java, Active-X)
 - script client-side (JavaScript, Vbscript)
 - plugin (Flash, ...)

■ client application custom / ad-hoc:

- (A) enhanced functionality (=requested by the server)
- (D) supported platforms
- (D) deployment / update / user assistance

Peer-to-peer (P2P) architecture

- architectures where processes can act simultaneously as client and as server
- advantages:
 - workload and the communication load is distributed among all processes
- disadvantages:
 - difficulty to coordinate / control



P2P computing

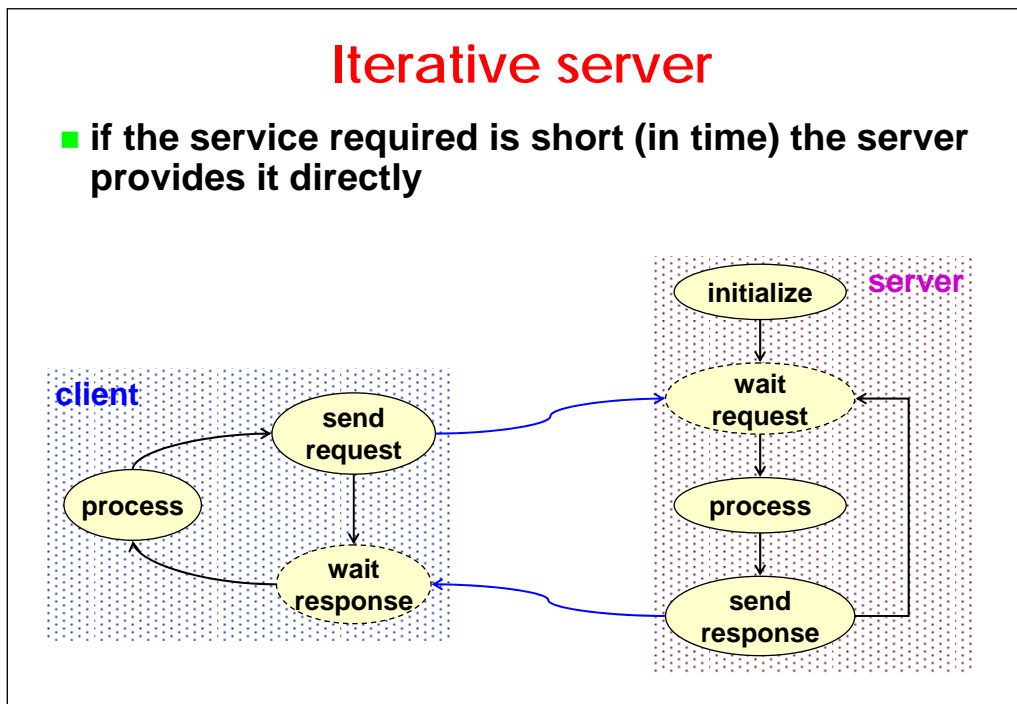
- the clients evolve from service users to autonomous service providers
- useful to share resources and exploit collaborative services
- computation capacity of single nodes is exploited better (consequently the servers are less loaded)
- networks are used better, with direct communication among nodes (consequently the link congestion towards the servers is avoided)

P2P architectures

- **collaborative computing**
 - network community for distributed tasks (e.g. grid computing)
- **edge services**
 - orthogonal(?) services as “qualifying factors” for the creation of a P2P community (e.g. Internet fax)
- **file sharing**
 - to exchange information in network without loading them on a server, but leaving them where they are placed
 - es. Gnutella (gnutella.wego.com), WinMX, Kazaa

Server types

- **internal architecture of the server influences significantly the performance of the overall system**
- **necessary to choose the model most appropriate to the application problem**
- **does not exist a solution suitable for ALL uses (there is the risk that it is too complicated)**



Examples of iterative servers

- standard TCP/IP services of short period:
 - daytime (tcp/13 o udp/13) RFC-867
 - qotd (tcp/17 o udp/17) RFC-865
 - time (tcp/37 o udp/37) RFC-868
- in general, services where is strongly required to limit the server load (only one user at a time)
- advantages:
 - simplicity of programming
 - quickness of response (when it is able to connect!)
- disadvantages:
 - limit of server load

Performance of an iterative server

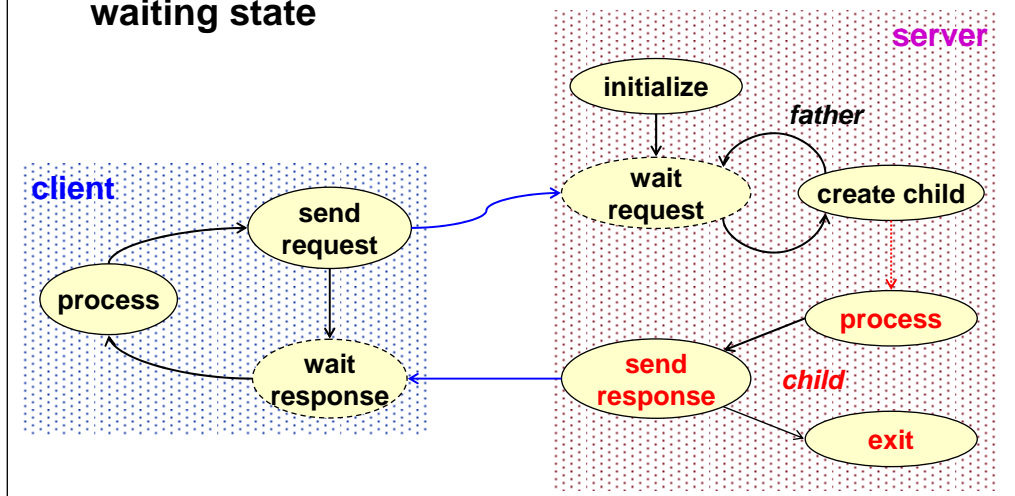
- the number of CPUs doesn't influence performance
- denote T_E the CPU time of processing a request at server [s]
- maximum performance (in optimal conditions):

$$P = 1 / T_E \text{ services / s}$$
- in case of simultaneous requests from more clients, the ones not served enter in competition successively (unless the queue of requests has size > 1)
- the latency of service depend on the load $W \geq 1$ of the node that hosts the server:

$$L = T_E \times W \text{ s}$$

Concurrent server

- when the service has long duration, the server starts a "child" subprocess and goes back to the waiting state



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
 - ...
- **in general, services with complex processing or that require long time period and/or not predictable in advance**

Concurrent server: analysis

- **advantages:**
 - load on server theoretically/ideally unlimited
- **disadvantages:**
 - complexity of programming (concurrent)
 - latency of response (create a child subprocess, T_F)
 - max load on server is limited (each child subprocess requires RAM, CPU cycles, access cycles to disk, ...)

Performance of a concurrent server

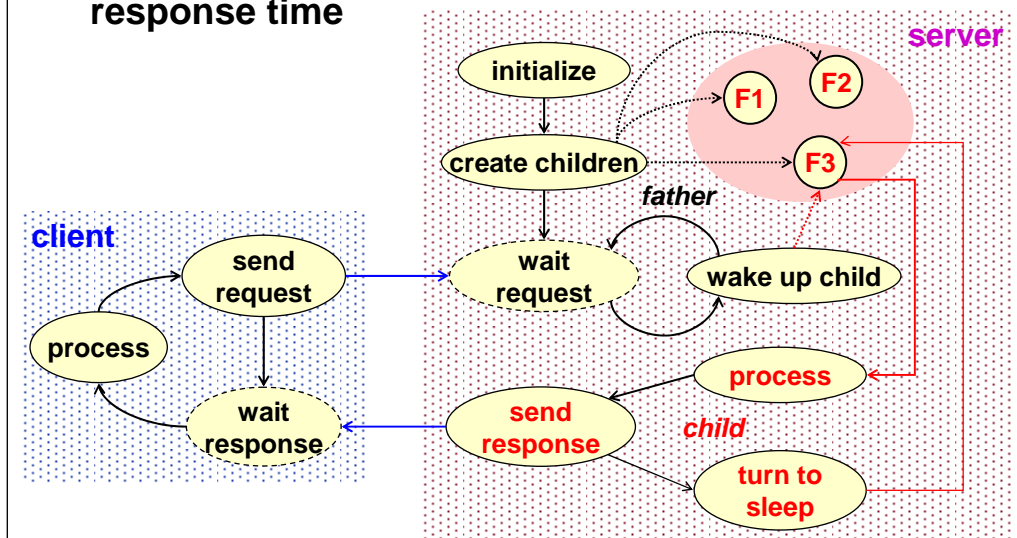
- influenced by the number of CPU (denoted with C)
- denote T_F the CPU time to create a child subprocess [s]
- maximum performance (in optimal conditions):

$$P = C / (T_F + T_E) \text{ services / s}$$
- in case of simultaneous requests from more clients, the ones not served enter in competition successively (unless the queue of requests has size > 1)
- the latency of service depends on the load W of the node that hosts the server:

$$(T_F + T_E) \leq L \leq (T_F + T_E) \times W / C \text{ s}$$

"Crew" server

- pre-activation of child subprocesses to improve response time



Examples of "crew" server

- **all concurrent services can be implemented with a crew server**
- **high performance network services:**
 - subject to high load (=no. of simultaneous users)
 - with low response delay (latency)
- **typical examples:**
 - web server for e-commerce
 - DBMS server

"Crew" server: analysis

- **advantages:**
 - load theoretically/ideally unlimited (additional children can be generated depending on the server load)
 - quickness of response (waking up a child takes less time than creating it)
- **disadvantages:**
 - complexity of programming (concurrent)
 - management of children pool
 - synchronisation and concurrency of accesses of various children subprocesses to the shared resources of server

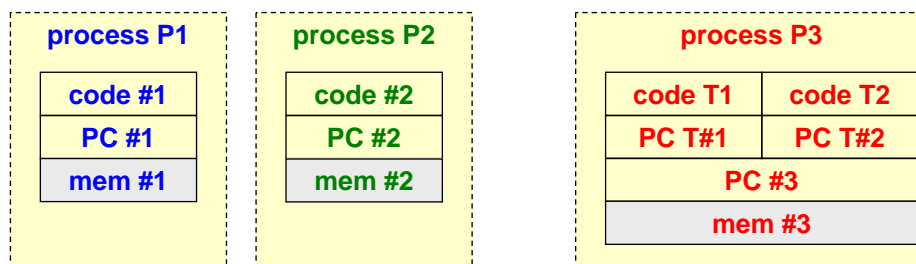
Performance of a "crew" server

- analogous to that of a concurrent server, with T_F replaced with the time required to activate a child subprocess T_A (usually negligible)
- if the server generates other children subprocesses because the existent ones finished, then the performance P is a combination weighted with the probability G to generate new children:

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

Concurrent programming

- simultaneous execution of several processing modules on the same CPU
- two main models:
 - processes
 - threads



Processes vs. thread

■ activation of a module

- [P] slow
- [T] fast

■ communication among modules

- [P] difficult (requires IPC, e.g. pipe, shared memory)
- [T] easy (the same address space)

■ protection among modules

- [P] optimal, both of memory and of CPU cycles
- [T] very poor